ALGORITHMS
Design and Analysis

Harsh Bhasin

Assistant Professor
Department of Computer Science
Jamia Hamdard
New Delhi

OXFORD
UNIVERSITY PRESS

To

My Mother

About the Author

Harsh Bhasin is currently an Assistant Professor in the Department of Computer Science, Jamia Hamdard, New Delhi. Prior to this, he has taught as visiting faculty in many colleges including Delhi Technological University and also has a rich industrial experience as a programmer. He was also the proprietor of S.S. Developers, a firm based in Faridabad, Haryana.

Prof. Bhasin is a B. Tech and M. Tech in Computer Science as also a UGC NET qualified. He has been actively involved in research and had also received the Young Researcher's Award by ErNet in 2012. His areas of interest include genetic algorithms, cellular automata, big data, theory of computation, C#, and algorithms.

Prof. Bhasin has been involved in the development of a number of Enterprise Resource Planning Systems. He has published around 60 research papers in various national and international journals of repute and is the author of "Programming in C#" published by OUP, India. He has also reviewed papers, journals, and books for renowned publishers. He has been the Editor-in-Chief of the special issue on 'Applicability of Soft Computing Techniques in NP Problems,' SciEp, USA.

Mr Bhasin can be reached at his Facebook page DTUComputation and via e-mail at i_harsh_bhasin@yahoo.com or at thevibrantindian@blogspot.com.

Preface

There are only two ways to live your life. One is as though nothing is a miracle. The other is as though everything is a miracle.

— Albert Einstein

An algorithm is a step-by-step process of analysing and solving the given problem in a logical manner. A well-designed algorithm is required to develop efficient program codes as well as minimize the usage of computer resources. Algorithms are implemented using a programming language. Mere knowledge of a programming language without the fundamental understanding of algorithms may make one a competent coder but not a programmer. Thus, algorithms form the building blocks of computer programming.

Design and analysis of algorithms is one of the key subjects offered in computer science and information technology streams. Knowledge of basic data structures and mathematics is a prerequisite for studying this subject. This course makes the students learn the standard design techniques, such as divide and conquer, greedy approach, and dynamic programming, as also analyse the applicability of a technique in a given problem. Another important goal of this course is to help students develop the ability to study an algorithm and find its complexity.

Thus, this book discusses the know-hows of the paradigms used for designing an algorithm and illustrates the standard procedures for accomplishing the task. It provides a sound understanding of the asymptotic notations required to analyse the effect of the increase in input size of an algorithm on its space and time requirements. It also discusses the various artificial intelligence techniques which would help the readers handle intractable problems.

ABOUT THE BOOK

Algorithms: *Design and Analysis* is a textbook designed for the undergraduate and postgraduate students of computer science engineering, information technology, and computer applications. It will help the students understand the fundamentals and applications

of algorithms. The book will serve as a useful reference for researchers and practising programmers who intend to pursue a career in algorithm designing as well as students preparing for interviews and exams such as GATE and UGC NET.

The book offers an adequate mix of both theoretical and mathematical treatment of the concepts. It covers the basics, design techniques, advanced topics, and applications of algorithms. The concepts and algorithms are explained with the help of examples. However, every attempt has been made to keep the text as precise as possible. Moreover, some advanced topics in the book have been included considering the fact that they would be used or implemented in research.

Each chapter of the book includes a variety of chapter-end exercises in the form of MCQs (with answers), review questions, and programming exercises to help readers test their knowledge. In the book, many problems have been solved using more than one method for better understanding.

KEY FEATURES

The following are the salient features of the book:
- Offers in-depth treatment of topics such as complexity analysis, design paradigms, data structures, and machine learning algorithms
- Introduces topics such as decrease and conquer, transform and conquer, and PSpace along with standards paradigms
- Explains numerical methods including Euclid's theorem and Chinese Remainder Theorem, and also reviews essential mathematical concepts
- Provides points to remember and a list of key terms at the end of each chapter that will help readers quickly recapitulate the important concepts
- Includes exercises given at the end of each chapter and Appendix A10 (Problems) to help students prepare for their examinations and job interviews

ORGANIZATION OF THE BOOK

The book has been divided into four sections. The first section introduces the fundamental concepts and complexity analysis of algorithms. The second section deals with basics of data structures. The third section introduces the various design techniques and the fourth section deals with advanced topics. The sections are followed by ten appendices.

Section I: Basic Concepts of Algorithms

The first section of the book defines algorithms and discusses the application of algorithms and the techniques to analyse them. It includes four chapters.

Chapter 1 introduces the subject by tracing the evolution of algorithms and highlighting the importance and applications of algorithms in recent times. The chapter also explains the different ways of writing an algorithm.

Chapter 2 presents basic mathematical techniques such as logarithms, arithmetic, and geometric progression along with an involved discussion on the asymptotic notations. The chapter also provides a comparison of the asymptomatic notations.

Chapter 3 introduces the concept of recursion and deals with different methods of solving recursive equations.

Chapter 4 discusses ways to find complexities along with the proving techniques. It also covers amortized and probabilistic analysis.

Section II: Data Structures

Knowledge about data structures is a prerequisite for understanding the concepts discussed in the following sections. Therefore, this section has been added to aid the readers in refreshing the basics of data structures while studying the design techniques. The rest of the book relies heavily on the terminology used in these chapters. This section has four chapters.

Chapter 5 discusses stacks, queues, linked lists, and arrays along with their implementations, while *Chapter 6* presents an overview of trees and the algorithms involved therein. *Chapter 7* revisits the concepts of graphs and its applications and *Chapter 8* presents various linear and quadratic sorting algorithms.

Section III: Design Techniques

This section forms the core of the course on design and analysis of algorithms. It covers the various design paradigms and their applications and implementation. This section includes six chapters.

Chapter 9 discusses the divide and conquer technique and its applications in solving problems such as quick sort, merge sort, selection sort, convex hull, Strassen's matrix multiplication, defective chessboard, and finding minimum distance between N points. Master theorem and its proof is also explained in this chapter.

Chapter 10 introduces the concept of greedy approach. This concept has been used to solve problems such as job scheduling, knapsack, coin changing, and minimum cost spanning tree. Other problems that can be solved using the greedy approach are quick sort, merge sort, binary search, and Strassen's matrix multiplication.

Chapter 11 explains dynamic programming. Some of the problems, namely, subset sum problem, 0/1 knapsack, matrix chain multiplication, longest common sub-sequence, optimal binary search, and travelling salesman problem that can be solved using dynamic programming have been illustrated in this book.

Chapters 12 and *13* discuss backtracking and branch and bound techniques, respectively. In spite of being similar to backtracking, branch and bound technique is more effective and efficient. Some of the problems that can be solved by these two techniques are the maze problem, subset sum problem, the *N*-Queens problem, m-colouring problem, travelling salesman problem, and Hamiltonian cycle.

Chapter 14 introduces the concept of randomization, which would help in analysing and understanding algorithms in an altered way. The chapter discusses the Monte Carlo and Las Vegas algorithms and applications of randomized algorithms through the book problem, load balancing, and quick sort problems.

Most of the problems discussed in the book can be solved in linear or quadratic time. That is, for these problems the running time is $O(n)$ or $O(n^2)$. Such problems which can be solved in polynomial time, that is, problems which have complexity $O(n^k)$, where k is an integral constant, are referred to as P-type problems. The problems that cannot be verified in polynomial time are called NP problems. These concepts are also discussed in this chapter.

Section IV: Advanced Topics

The fourth section of the book introduces some advanced topics and includes 10 chapters.

Chapter 15 explains the transform and conquer technique. The technique is used to solve problems such as Gauss elimination method for finding the solution of a set of linear equation; the LU decomposition for solving a set of equations; the Horner's method for finding the lowest common multiple, etc.

Chapter 16 introduces decrease and conquer approach. The method can be used to solve many problems notably permutation generation, which is used to handle NP-hard problems like the travelling salesman problem.

Chapter 17 deals with the number theoretic algorithms and covers topics such as GCD and Euclid theorem. The Chinese remainder theorem has also been discussed in this chapter. These topics are widely used in cryptography and cryptanalysis.

Chapter 18 introduces an immensely important topic, called string matching, which finds applications in many areas including computational biology. It includes Naive string matching algorithm, Rabin–Karp algorithm, deterministic and non-deterministic finite automata, and Knuth–Morris–Pratt Automata. The chapter also discusses tries and suffix trees along with the most common methods used for accomplishing the task of string matching.

Chapter 19 discusses the complexity classes—P and NP problems. It explains Cook's theorem, the concept of reducibility, and NP hard problems.

The book not only discusses the time complexity but also the space complexity. *Chapter 20* introduces the space complexity of an algorithm, PSpace, along with its applications.

Chapter 21 presents the concept of approximation algorithms. These are one of the most important tools for handling the NP complete problems. The chapter also covers ρ-approximation and its applications.

Chapter 22 discusses the concept of parallel algorithms. The chapter discusses basic concepts such as the generations of computers and parallel computers before introducing the parallel random access machine (PRAM) model. Hypercube algorithms have also been covered in the chapter.

Artificial intelligence techniques are now widely used to handle intractable problems. In order to equip the readers with such techniques, the book introduces the concepts of genetic algorithms and machine learning in *Chapter 23*. Applications of genetic algorithms in solving knapsack, subset sum, travelling salesman, vertex cover, and maximum clique problems have been presented in this chapter.

Chapter 24 of this book introduces computational biology and bioinformatics. This is essential as the readers would be able to apply the algorithms techniques studied earlier in these areas.

The book also has 10 appendices which carry forward the concepts studied in the previous sections. Topics such as probability, matrix operations, Red-black tress, linear programming, DFT, scheduling, and a reprise of sorting, searching, and amortized analysis have been discussed in the appendices (A1 to A9).

The last Appendix A10 includes some interesting problems based on almost all the topics discussed in the book.

ONLINE RESOURCES

To aid teachers and students, the book is accompanied with online resources which are available at http://oupinheonline.com/book/bhasin-algorithms/9780199456666. The contents of online resources include:

For Faculty

- Chapter-wise PowerPoint slides
- Solution manual for select chapter-end problems
- Assignment questions with answers

For Students

- Additional MCQs for test generator (with answers) for each chapter
- C language implementation of algorithms
- Interview questions with answers

ACKNOWLEDGMENTS

I have been lucky enough to get the motivation and guidance from various people during the journey of developing this book. First of all, I would like to thank Professor Moin Uddin, Dean, Faculty of Management and Information Technology, former Pro-Vice Chancellor, Delhi Technological University for showing faith in me and inspiring me to achieve my goals. I am also thankful to Professor A.K. Sharma, former Dean and Chairperson, Department of Computer Science, YMCA, Faridabad, for his constant encouragement while working on this book, research papers, and other projects.

Dr Michael Wing's ACM Sigsoft Software Engineering Notes has helped me improve my writing skills and inspired me to be more articulate. Moreover, I have learnt the most important lesson of my life from Dr Wing: 'If you know a thing but you cannot express it then it is of no use.'

I would like to thank esteemed Professor Ranjit Biswas, Head, Department of Computer Science, Jamia Hamdard, Professor Naresh Chauhan, Head and Chairperson, Department of Computer Sceince, YMCA University of Science and Technology, Professor Daya Gupta, Department of Computer Science, Delhi Technological University, and Dr S.K. Pal, Scientist, Department of Defence and Research Organization, Government of India, for their valuable suggestions regarding the book. I would also like to acknowledge my colleagues, Dr Farheen Siddique and Dr G.D. Panda, for their contributions in the chapter on computational biology and my students, Faisal Naved, Mohd. Haider, Sourav Gupta, Yogesh Kumar, Chirag Ahuja, and Subham Kumar, for their critical reviews and contributions in developing the web resources of this book.

I would like to express my sincere gratitude to my mother, sister, and rest of the family including my pets, Zoe and Xena, and friend Mr. Naved Alam and others for extending their unconditional support to me.

I am also thankful to the editorial team of Oxford University Press for providing valuable assistance.

I would be glad to receive comments or suggestions from the readers and users of this book for further improvement of the future editions of the book. You can reach me at i_harsh_bhasin@yahoo.com.

Harsh Bhasin

Brief Contents

Detailed Contents

SECTION I

BASIC CONCEPTS OF ALGORITHMS

Some infinities are bigger than other infinities.
— John Green

Introduction to Algorithms

OBJECTIVES

After studying this chapter, the reader will be able to

- Understand the importance of algorithms
- Trace the origin of algorithm
- Define an algorithm
- Learn the various ways of writing an algorithm
- Understand what the future has in store for us vis-à-vis algorithms
- Understand the concept of designing an algorithm

1.1 INTRODUCTION

A computer engineer is expected to tackle any given problem in the most efficient manner. This efficiency can be in terms of memory or time or both. However, efficiency becomes important only if the solution, proposed by the person, solves the problem. The steps followed to do so constitute an algorithm. This chapter introduces the concept of algorithm, discusses the ways of writing an algorithm, and explores the basics of algorithmic designing. We start with the informal definition of an algorithm.

Definition Algorithm refers to the steps to be carried out in order to accomplish a particular task.

Algorithms are used everywhere, from a coffee machine to a nuclear power plant. A good algorithm should use the resources such as the CPU usage, time, and memory judiciously. It should also be unambiguous and understandable. The output produced by an algorithm lies in a set called *range*. The input, is taken from a set 'domain' (input constraints). From the domain only the values satisfying given constraints can be taken. These constraints are referred to as input constraints. Input constraints determine the values of x_i, i.e., input. The inputs are related to each other as governed by relation corresponding to the task that is to be accomplished. This is referred to as explicit constraint.

The formal definition and characteristics of an algorithm are discussed later in this chapter. However, the above discussion introduces the concept.

1.2 IMPORTANCE OF ALGORITHMS

To understand the importance of algorithms, let us take an example. Most of us must have found solutions to a lot of our problems using Google. When we type a query in Google, we are presented with the ordered set of results that are more or less relevant. However, this ranking is done via an algorithm, which ranks the pages in accordance with the query entered. This ranking algorithm not only checks the textual similarity of the query with the web page, but also calculates the inlinks (number of pages pointing to that page) and the outlinks (number of pages that the page is pointing to) of that page. This algorithm has helped Google in achieving its present status. Most of us would agree that Google has changed our life. Therefore, the credit goes to the page rank algorithm that Google uses.

Another example that can be cited here is that of 'Google Maps'. Most of us must have used it to find the route from one location to another. 'Google Maps' helps us to get driving directions by using the shortest path algorithms explained in this book. So, even Google Maps is based on algorithms. The application fascinates and at times annoys owing to the incorrect results displayed. However, one must appreciate the fact that it is a computer program, which is an implementation of some algorithm. The algorithm is still being refined but the fact is that it presents us with a thing which, at one point of time, was the sole prerogative of man.

Nowadays, algorithms have become important even for biological endeavours. Governments across the world want to make a global database of DNAs to combat the menace of terrorism. This would be possible only if sorting and searching algorithms are developed, which can extract information from billions of DNAs. In order to accomplish this task, nature-based algorithms are being developed. The subsequent chapters of this book examine the searching algorithms and their complexities. One of the nature-based search procedures called genetic algorithms have been explained in Chapter 23.

Let us take another example to demonstrate the importance of algorithms. Optimization problems are one of the most important problems not only in computer science but also in economics. The study of algorithms helps us to solve optimization problems as well. All of us know the fact that, in order to make a business profitable, the total cost should be minimized and the profit should be maximized. Algorithmic designing techniques like 'greedy approach', introduced in Section III of the book, help in achieving the task of optimization.

Summarizing the importance of algorithms discussed earlier, we can say the following:
- It helps in enhancing the thinking process. They are like brain stimulants that will give a boost to our thinking process.
- It helps in solving many problems in computer science, computational biology, and economics.
- Without the knowledge of algorithms we can become a coder but not a programmer.

- A good understanding of algorithms will help us to get a job. There is an immense demand of good programmers in the software industry who can analyse the problem well.
- The fourth section of the book that introduces genetic algorithms and randomized approach will help us to retain that job.

Having discussed the importance of the subject, let us move on to the history of the subject. It is important to know the history of the subject as it helps in creating an interest in the subject. The problems faced in the past pave way for deducing the possible solutions in the future.

1.3 HISTORY OF ALGORITHM

The origin of the word algorithm is indirectly linked to India. A scholar in the, 'House of Wisdom' in Baghdad, Abu Abdullah Muhammad Ibn Musa Al Khwarizmi, wrote a book about Indian numerals in which rules of performing arithmetic with such numerals were discussed. These rules were referred to as 'algorism' from which the word algorithm was derived. His book was translated into Latin in the 18th century. This was followed by the invention of Boolean algebra by George Boole and the creation of language in special symbols by Frege. The concept of algorithms, given its present form by a genius named Alan Turing, helped in the inception of artificial intelligence.

Algorithms have been used for long in mathematics and computer science. Euclid's theorem and the algorithm of Archimedes to calculate the value of 'Pi' are classic examples of algorithms. These events reinforced the belief that if a task is to be performed, then it must be performed with a predefined sequence of steps that are unambiguous and efficient. However, this belief would be challenged in the late 20th century with the introduction of *non-deterministic algorithms*.

Not only in mathematics and computer science are there algorithms, they are part of our daily lives. When a person is taught how to make tea, even then an algorithm is edified. Algorithms are camouflaged as directions and rules, which form the basis of our existence. The challenge, however, is to make these algorithms efficient and robust.

1.4 ALGORITHM: DEFINITION

An algorithm is a sequence of steps that must be carried out in order to accomplish a particular task. Three things are to be considered while writing an algorithm: input, process, and output. The input that we give to an algorithm is processed with the help of the procedure and finally, the algorithm returns the output. It may be stated at this point that an algorithm may not even have an input. An example of such an algorithm is pseudorandom number generator (PRNG). Some random number generators generate

a number without a seed. In such cases, the algorithm does not require any input. The processing of the inputs generates an output. This processing is the most important part of the algorithm. This book intends to examine the various methodologies used to write a good algorithm. It may be noted though that algorithm writing is more of an art. We can be taught the basics, but the art of writing an algorithm will have to be developed by practicing more and more algorithms.

While writing an algorithm, the time taken to accomplish the task and the memory usage must also be considered. The prime motto is to solve the problem but efficiency of the process followed should not be compromised.

There is a distinction between natural language and algorithmic writing. While speaking or writing a letter, we may use ambiguous terms unknowingly or deliberately. However, the intelligence of the reader or the listener disambiguates the whole thing. For example, if we have reservations about a person with regard to his/her knowledge of the subject, then we might camouflage our unwillingness to work with him/her as a personal or social problem. This may not be the case when algorithms are concerned. While writing an algorithm we will have to be clear and unambiguous about our objectives. Moreover, any statement in an algorithm should be strictly deterministic. However, in non-deterministic algorithms, this condition does not hold.

To summarize the discussion
- An algorithm is a sequence of steps in order to carry out a particular task.
- It can have zero or more inputs.
- It must have at least one output.
- It should be efficient both in terms of memory and time.
- It should be finite.
- Every statement should be unambiguous.

The meaning of finite is that the algorithm should have countable number of steps. It may be stated that a program can run infinitely but an algorithm is always finite. For example, an operating system of a server, in spite of being a program runs 24×7 but an algorithm cannot be infinite.

1.5 WAYS OF WRITING AN ALGORITHM

There are various ways of writing an algorithm. In this section, three ways have been explained and exemplified taking requisite examples. However, the chapters that follow explain the problems introduced in this section in detail.

1.5.1 English-Like Algorithm

An algorithm can be written in many ways. It can be written in simple English but this methodology also has some demerits. Natural languages can be ambiguous and therefore lack the characteristic of being definite. Since each step of an algorithm should be clear and should not have more than one meaning, English language-like algorithms

are not considered good for most of the tasks. However, an example of linear search, in which an element is searched at every position of the array and the position is printed if an element is found, is given below. In this algorithm, 'A' is the array in which elements are stored and 'item' is the value which is to be searched. The algorithm assumes that all the elements in 'A' are distinct. Algorithm 1.1 depicts the above process.

Algorithm 1.1   English-like algorithm of linear search

Step 1. Compare 'item' with the first element of the array, A.
Step 2. If the two are same, then print the position of the element and exit.
Step 3. Else repeat the above process with the rest of the elements.
Step 4. If the item is not found at any position, then print 'not found' and exit.

However, Algorithm 1.1, in spite of being simple, is not commonly used. The flow-chart or a pseudocode is more common as compared to 'English-like algorithms', which is used in some chapters such as Chapters 23 and 24 of this book.

1.5.2 Flowchart

Flowcharts pictorially depict a process. They are easy to understand and are commonly used in the case of simple problems. The process of linear search, explained in the previous subsection, is depicted in the flowchart illustrated in Fig. 1.1. The conventions of flowcharts are depicted in Table 1.1.
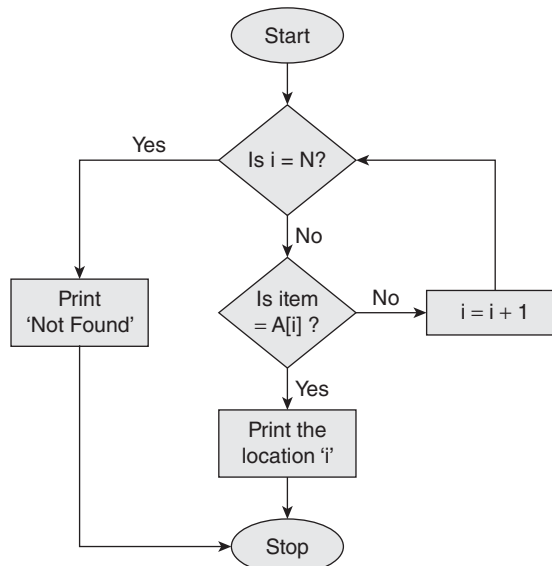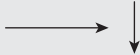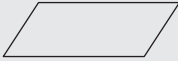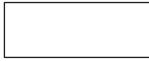
Figure 1.1   Flowchart of linear search

Table 1.1 Flowchart conventions

S. No.	Name	Element Representation	Meaning
1.	Start/End		An oval is used to indicate the beginning and end of an algorithm.
2.	Arrows		An arrow indicates the direction of flow of the algorithm.
3.	Connectors		Circles with arrows connect the disconnected flowchart.
4.	Input/Output		A parallelogram indicates the input or output.
5.	Process		A rectangle indicates a computation.
6.	Decision		A diamond indicates a point where a decision is made.

In the flowchart, shown in Fig 1.1, A[ ] is an array containing *N* elements. The index of the first element is *O* which is also the initial value of *i*. Such depictions, though easy to comprehend, are used only for simple straightforward problems. Hence, this book neither recommends nor uses the above two types for writing algorithms, except for some cases.

1.5.3 Pseudocode

The pseudocode has an advantage of being easily converted into any programming language. This way of writing algorithm is most acceptable and most widely used. In order to be able to write a pseudocode, one must be familiar with the conventions of writing it. Table 1.2 shows the pseudocode conventions.

Table 1.2 Pseudocode conventions

S. No.	Construct	Meaning
1.	`// Comment`	Single line comments start with //

2.	`/* Comment Line 1`
	`   Comment Line 2`
	`   ○`
	`   ○`
	`   Comment Line n`
	`*/`
		Multi-line comments occur between /* and */

3.	`{`
	`statements`
	`}`
		Blocks are represented using { and }. Blocks can be used to represent compound statements (collection of simple statements) or the procedures.

4.	`;`	Statements are delimited by ;

(Contd)

Table 1.2 (Contd)

S. No.	Construct	Meaning
5.	`<variable> = <Expression>`	This is an assignment statement. The statement indicates that the result of evaluation of the expression will be stored in the variable.
6.	`a > b`	a and b are expressions, and `>` is a relational operator 'greater than'. The Boolean expression `a > b` returns true if a is greater than b, else returns false.
7.	`a < b`	a and b are expressions, and `<` is a relational operator 'less than'. The Boolean expression `a < b` returns true if a is less than b, else returns false.
8.	`a <= b`	a and b are expressions, and `<=` is a relational operator 'less than or equal to'. The Boolean expression `a <= b` returns true if a is less than or equal to b, else returns false.
9.	`a >= b`	a and b are expressions, and `>=` is a relational operator 'greater than or equal to'. The Boolean expression `a >= b` returns true if a is greater than or equal to b, else returns false.
10.	`a != b`	a and b are expressions, and `!=` is a relational operator 'not equal to'. The Boolean expression `a != b` returns true if a is not equal to b, else returns false.
11.	`a == b`	a and b are expressions, and `==` is a relational operator 'equal to'. The Boolean expression `a == b` returns true if a is equal to b, else returns false.
12.	`a AND b`	a and b are expressions, and `AND` is a logical operator. The Boolean expression `a AND b` returns true if both the conditions are true, else it returns false.
13.	`a OR B`	a and b are expressions, and `OR` is a logical operator. The Boolean expression `a OR b` returns true if any of the condition is true, else it returns false.
14.	`NOT a`	a is an expression, and `NOT` is a logical operator. The Boolean expression 'NOT a' returns true if the result of a evaluates to False, else returns False.
15.	`if<condition>then<statement>`	The statement indicates the conditional operator `if`.
16.	`if<condition>then<statement1>` `else<statement2>`	The statement is an enhancement of the above `if` statement. It can also handle the case wherein the condition is not satisfied.

17.	`Case`
	`{`
	`    :<condition 1>: <statement 1>`
	`    ○`
	`    ○`
	`    :<condition n>: <statement n>`
	`    :default: <statement n+1>`
	`}`
		The statement is a depiction of `switch case` used in C or C++.

(Contd)

Table 1.2 (Contd)

S. No.	Construct	Meaning

18.	```while<condition>do```
	```{```
	``` statements```
	```}```
		The statement depicts a `while` loop

19.	```repeat```
	``` statements```
	```until<condition>```
		The statement depicts a `do–while` loop

20.	```for variable = value1 to value2```
	```{```
	``` statements```
	```}```
		The statement depicts a `for` loop

21.	```Read```	Input instruction
22.	```Print```	Output instruction
23.	```Algorithm<name> (<parameter list>)```	The name of the algorithm is `<name>` and the arguments are stored in the `<parameter list>`

Algorithm 1.2 depicts the process of linear search. The name of the algorithm is 'Linear Search'. The element 'item' is to be searched in the array 'A'. The algorithm uses the conventions stated in Table 1.2.

Algorithm 1.2   Linear search

```
Algorithm Linear_Search(A, n, item)
{
    for i = 1 to n step 1 do
    {
        if(A[i] == item)
        {
            print i;
            exit();
        }
    }
    print "Not Found"
}
```

Two approaches of writing an algorithm are described in the following chapters, the first approach, followed in Chapters 9–11 explicitly states the input, output constraints, etc. The algorithms stated in this fashion require least effort to implement. However, at times, when the details of the implementation are not to be included in the algorithm, the English-like algorithms come to our rescue. The algorithm broadly describes what is to be done, not exactly how it is to be done. As stated earlier, some chapters such as Chapters 23 and 24 follow this approach.

1.6 DESIGN AND ANALYSIS vs ANALYSIS AND DESIGN

In order to accomplish a task, a solution needs to be developed. This is called designing of an algorithm. For example, if an array 'A' of length n is given and our requirement is to find out the maximum element of the array, then we can take a variable 'Max' whose initial value is A[1], which is the first element of the array. Now, start traversing the array, compare the value of Max with each element, if we are able to find any element greater than Max, then we can set Max to the value of that element, else continue. The process is depicted in Algorithm 1.3.

Algorithm 1.3 Finding maximum element from an array

```
Algorithm Max(A, n)
{
    Max = A[1];
    for i = 2 to n step 1 do
    {
        if(A[i]>Max) then
        {
            Max=A[i];
        }
    }
    Print "The maximum element is A[i]"
}
```

The next step would be to analyse the time complexity of the algorithm. Table 1.3 shows the number of times each statement is executed.

The above analysis gives an idea of maximum amount of resources (in this case time) required to run the algorithm. This is referred to as algorithm design and analysis (ADA) (see Fig. 1.2).

However, this may not be the case most of the times. Often, we have to develop software for the client. The client has some set-up and will not want to upgrade his systems in order to install the software. In such cases, we must analyse the hardware and

Table 1.3 Number of times each statement is executed in Algorithm 1.3

`Max = A[1];`	1
`for i := 2 to n step 1 do{`	n
`if(A[i]>max) {`	n−1
`Max=A[i];}}`	less than or equal to (n−1)
`print "The maximum element is A[i]"`	1
	Maximum: (n)

Design Analyse

Figure 1.2 Design and analysis

Analyse Design

Figure 1.3 Analysis and design

the set-up of the client and then decide on the algorithms we would be using in order to accomplish the tasks. Here, we cannot apply techniques like diploid genetic algorithm on a system that uses a P4, similarly there is no point in using algorithms that are time efficient but probably use extensive resources in a very advanced set-up. The process is referred to as analysis and design. The process is depicted in Fig. 1.3.

The general approach being used is design and analysis; however, analysis and design is far more practical and hence implementable.

1.7 PRESENT AND FUTURE

The algorithms developed at present focus more on efficiency and optimization. When we develop an algorithm, the first and foremost task is to solve the problem at hand. So, correctness comes first even if it is at the expense of time and memory. When we are able to solve the problem, then we try to make our algorithm efficient. That is to say, we try to use constructs that require less time and perhaps less memory too. Most of the algorithms developed also cater to the requirement that each statement should be unambiguous and definite. So to summarize, in developing an algorithm, the following things are taken care of:
1. Make sure that the solution is correct.
2. Try to make sure that the time consumption is least.
3. Try to make sure that the memory consumption is also least.
4. Try to keep each statement unambiguous and definite.
    However, doing so can be a problem in the following cases:
1. When the search space is so large that it is not possible to obtain an exact solution.
2. The algorithm implements a non-deterministic machine.
    The above two points have become common owing to the stress laid on the non-deterministic algorithms in recent years and the need to process huge amount of data in web mining.

Consuming least time and least memory would also be irrelevant in future because processes have become so fast that even if our algorithm produces better results in spite of taking more time, it will be considered good.

As far as memory is concerned, consuming a little more memory and giving better results is considered acceptable in the present scenario and become more acceptable owing to the decrease in the cost of memory. For example, a 20 GB hard disk would have cost ₹3000 in 2001–2002, however, now we would get a 500 GB hard disk at the same

price. So, 25 times increase in the memory and no increase in the cost in a decade points to the fact that memory is becoming cheap. Therefore, it would not be advisable to spend a lot of time devising algorithms which would save a few bytes of memory.

It may also be noted that the algorithms will now be based on artificial intelligence techniques rather than determinism. The development of Deep Blue, a computer which could play chess and was able to challenge Garry Kasparov, is an example of such algorithms.

So, the future algorithms will depend on

1. Artificial intelligence techniques, which would help in optimization.
2. Ability to utilize hardware capabilities and process power as well.
3. Non-determinism which will have to be integrated so as to solve real-time problems and parallel processes.

1.8 FLOW OF THE BOOK

This book has been divided into four sections: Basic Concepts of Algorithms, Data Structures, Design Techniques, and Advanced Topics. The first section lays stress on the importance of algorithms and the complexity measurements and the second section deals with the basic data structures. This section have been included in this book so that the readers who have not studied data structures should not find it difficult to comprehend the rest of the chapters. The section forms the basis of the concepts explained in Sections III and IV. The readers who have done a basic course of data structures may jump to the third section. However, it would be beneficial to at least go through the sections to be able to implement the strategies examined in the sections that follow.

The third section focuses on the paradigms such as divide and conquer, dynamic approach, backtracking, and branch and bound to solve various problems. The web resources of the book also include the codes of some of the standard problems such as knapsack and job sequencing. The section is the most important of the three sections and it is important to understand the section to be able to become an accomplished algorithm designer.

The fourth section explores some of the yet unexplored areas such as artificial intelligence and computational biology. The section is essential for those doing or intending to do an advanced course in algorithms. Figure 1.4 depicts the organization of the book.

1.9 CONCLUSION

This chapter introduces the concept of algorithms. It may be stated at this point that before proceeding any further, it is essential to be able to write an algorithm. In order to do so the conventions of algorithm writing must be clearly understood. Moreover, it is also important to understand the difference between analyses of algorithms and followed by design and the concept of analysis and design. The chapter examines the concept, and it will become clear as we proceed further. Although an algorithm can be written in any of the three ways explained in the chapter, following the convention of

Decoding algorithms			
I. Basic concepts of algorithms	II. Data structures	III. Design techniques	IV. Advanced topics
Introduction to algorithms, complexity measurement, etc.	Essential data structure and recursion, sorting in linear and quadratic time and introduction to graphs	Techniques of designing, divide and conquer, greedy algorithms, dynamic programming, backtracking, branch and bound, and randomized algorithms, etc.	Transform and conquer, NP-completeness, approximation algorithms, number theoretic, decrease and conquer approach, topological sorting, space-time trade offs, limitations of algorithmic power, parallel algorithms, hypercube, artificial intelligence approaches, and miscellaneous topics

Figure 1.4   Organization of the book

Algorithms 1.2 and 1.3 is advisable. The chapters that follow discuss the various strategies of designing and analysing algorithms. Hence, this chapter serves as a foundation stone of the text that follows. In order to become an accomplished programmer, the know-how of algorithms is essential. So, it would be better to understand and implement the concepts given in the chapters that follow.

Points to Remember

- An algorithm is different from a program. An algorithm is finite; a program can be infinite.
- An algorithm can be pictorially depicted by a flowchart.
- The analysis of an algorithm is essential in order to judge whether it can be implemented in the given conditions.
- The analysis of an algorithm may consider time or space or both.
- The design of an algorithm can follow the analysis of the requirements. This approach is referred to as analysis and design.
- The algorithm can be designed in order to accomplish a task, and then can be analysed. This approach is referred to as design and analysis.

KEY TERMS

Algorithm It is a sequence of steps to accomplish a particular task efficiently and effectively.

Constraint The conditions that control the selection of elements in backtracking.

Explicit Constraint The conditions that determine how should various x_i's are related to each other.

Implicit Constraint An element x_i can take its values only from a legal set of values called domain.

EXERCISES

I. Multiple Choice Questions

1. Which of the following is a part of an algorithm?
   (a) Input
   (b) Output
   (c) Steps to be carried out in order to accomplish the task
   (d) All of the above

2. What is the most desirable characteristic of an algorithm?
   (a) Usability
   (b) Documentation
   (c) Ability to accomplish task
   (d) None of the above

3. Which of the following disciplines make use of ADA?
   (a) Automation
   (b) Computer science
   (c) Biology
   (d) All of the above

4. Who gave the concept of algorithms in its present form?
   (a) George Boole
   (b) Frege
   (c) Al Khwarizmi
   (d) Alan Turing

5. Who invented Boolean algebra?
   (a) George Boole
   (b) Frege
   (c) Al Khwarizmi
   (d) Alan Turing

6. Who created a language in special symbols?
   (a) George Boole
   (b) Frege
   (c) Al Khwarizmi
   (d) Alan Turing

7. Which is not an essential characteristic of algorithm?
   (a) Definiteness
   (b) Finiteness
   (c) Efficiency
   (d) Effectiveness

8. Definiteness property of algorithm means
   - (a) Considering the time taken to accomplish the task and the memory usage
   - (b) Each step of an algorithm must be precisely defined, unambiguously
   - (c) The number of steps in an algorithm must be finite and further each step must be executable in finite amount of time
   - (d) Each step must be sufficiently basic so that it can be done exactly by a person using pencil and paper
9. Effectiveness property of algorithm means
   - (a) Considering the time taken to accomplish the task and the memory usage
   - (b) Each step of an algorithm must be precisely defined, unambiguously
   - (c) The number of steps in an algorithm must be finite and further each step must be executable in finite amount of time
   - (d) Each step must be sufficiently basic so that it can be done exactly by a person using pencil and paper
10. Algorithm must be
    - (a) Programming language dependent
    - (b) Programming language independent
    - (c) Either of the above
    - (d) None of the above

II. Review Questions

1. What are algorithms? What are the characteristics of an algorithm?
2. What is meant by time complexity and memory complexity?
3. Briefly trace the history of algorithms.
4. What are the various ways of writing an algorithm?
5. Why do you think that each instruction of an algorithm must be definite?
6. Give an example of an algorithm which does not take any input.
7. Can there be an algorithm that does not have an output?
8. What are the differences between an algorithm and a program?
9. Explain the various approaches of writing an algorithm.
10. Explain with the help of an example time memory trade-off.

III. Application-based Questions

1. Write an algorithm to find the smallest number from amongst three numbers.
2. Write an algorithm to find the greatest common divisor of two numbers.
3. Write an algorithm to find the square root of a number.
4. Given an array write an algorithm to search an element from the array.
5. Given an array write an algorithm to find the maximum element from the array.
6. Given an array write an algorithm to find the minimum element from the array.

7. Write an algorithm to find second maximum element from an array.
8. Write an algorithm to sort an array.
9. Write an algorithm to find out the maximum element from a matrix.
10. Write an algorithm to find the trace of a matrix.

Answers to MCQs

1. (d)	3. (d)	5. (a)	7. (d)	9. (d)
2. (c)	4. (d)	6. (b)	8. (b)	10. (b)

CHAPTER 2

Growth of Functions

OBJECTIVES

After studying this chapter, the reader will be able to

- Understand the concept and importance of asymptotic notations
- Understand basic mathematical concepts such as arithmetic progression, geometric progression, and logarithms
- Explain the properties of asymptotic functions
- Compare algorithms on the basis of asymptotic complexity

2.1 INTRODUCTION

In order to accomplish a task, the most important thing is to design a correct algorithm. An algorithm can be called correct if it accomplishes the required task. However, sometimes in spite of being correct, an algorithm may not be of much use, in the case where it takes a lot of time. For example, applying linear search in order to find out an element is correct, but what if the array contains more than 10^{10} elements?

Even if one element is processed in 10^{-6} seconds, it will take 10,000 seconds or around 3 hours to search an element. Now imagine that the same task is to be accomplished in an array that contains the roll numbers of all the students of a university. In that case this procedure will require a lot of time. So, it is important that the algorithm should be correct as well as efficient. The understanding of running time is also important in order to compare the efficiency of two algorithms. This chapter deals with the analysis of algorithms. The analysis is aimed at finding out the running time of an algorithm.

It is difficult to find the exact running time of an algorithm. It requires rigorous mathematical analysis. The calculation of exact running time also requires the knowledge of sequences and series and logarithms among others. Moreover, the exact analysis provides no additional advantage compared to an approximate analysis. The exact analysis gives the exact polynomial function that relates the input size with the running time, whereas the approximate analysis gives the power of input size on which the running time depends. For example, the exact running time of an algorithm may be $3 \times n^2 + 2 \times n + 3$. In this case, the approximate running time would be $f(n^2)$. So, the highest power of

n is what matters while calculating the approximate running time of an algorithm. Even the constants that are there with the term containing the highest power do not matter.

It may also be stated that the number of inputs to an algorithm may not always be the number of variables that are given as an input to the algorithm. For example, if an algorithm takes an array as an input, then the input size is generally taken as *n* and not 1. So, the idea is that since an array contains *n* elements, the number of inputs to the algorithm must be taken as the number of elements in that array. The algorithm will most probably deal with most, if not all, of the elements of the array.

> **Tip:** In an array, the number of input elements is the length of the array, not 1.

The argument can be extended to a two-dimensional array as well. The number of inputs of an algorithm that manipulates an array having *n* rows and *m* columns is taken as $n \times m$, and not 1. This is because the number of elements in the data structure is $n \times m$.

Section 2.3 introduces the asymptotic notations and the procedure to find the asymptotic complexity of an algorithm. However, in order to understand the mathematics of the asymptotic notations, basic mathematical concepts are needed, which have been presented in Section 2.2. The topics discussed in Section 2.2 will help the readers to understand the complexity analysis of the algorithms given in the subsequent chapters.

2.2 BASIC MATHEMATICAL CONCEPTS

This section deals with the basic topics such as an arithmetic progression, geometric progression, and logarithms. The definition of the general sequence and the sum of *n* terms of arithmetic and geometric progressions have been dealt with in the present section. This section also throws light on logarithms, so that the idea of complexity can be understood clearly.

2.2.1 Logarithms

Logarithm is one of the most important concepts in mathematics. The analysis of algorithms also requires the concept of logarithms. The concept can be used in O notation and for making calculations simpler. The definition of logarithm is as follows:

> **Definition** If $a^b = c$, then $\log_a c = b$, that is, log *c* to the base *a* is *b*.

For example, since $5^3 = 125$, $\log_5 125 = 3$

The standard notations are as follows:

$$\log_{10} x = \log x$$
$$\log_e x = \ln x$$
$$\log_2 x = \lg x$$

Moreover, the important properties of logarithms are as follows:

$$\log a + \log b = \log ab \qquad\qquad \text{Formula 1}$$

$$\log a - \log b = \log a / b \qquad\qquad \text{Formula 2}$$

$$\log a^b = b \times \log a \qquad\qquad \text{Formula 3}$$

$$\log_a b = \frac{\log a}{\log b} \qquad\qquad \text{Formula 4}$$

$$\log_a b = \frac{1}{\log_b a} \qquad\qquad \text{Formula 5}$$

For example,

$$\log 5 + \log 2 = \log 10$$

$$\log 15 - \log 3 = \log 5$$

$$\log 2^3 = 3 \times \log 2$$

$$\log_2 7 = \frac{\log 7}{\log 2}$$

$$\log_2 3 = \frac{1}{\log_3 2}$$

The above properties can be used to simplify and solve the equations involving log. For example, in order to express $\log_2 1000$ in terms of $\log_2 5$, 1000 needs to be factorized. Since

$$1000 = 5^3 \times 2^3$$

$$\log_2 1000 = \log_2 2^3 + \log_2 5^3 = 3\log_2 2 + 3\log_2 5 = 3 + 3\log_2 5$$

It may be noted at this point that $\ln(1 + x)$ can be evaluated with the help of the following formula:

$$\ln(1 + x) = x - \frac{x^2}{2!} + \frac{x^3}{3!} - \frac{x^4}{4!} + \cdots$$

Log to the base 10 can be calculated by calculating ln and then applying the base change formula (Formula 4). For example, in order to calculate log 5 to the base 10, the following steps must be followed:

$$\ln(1+4) = 4 - \frac{4^2}{2!} + \frac{4^3}{3!} - \frac{4^4}{4!} + \cdots = 1.395, \text{ so } \log 5 = \frac{1.395}{2.31} = 0.60206, \text{ since } \log_e 10 = 2.31.$$

It may be stated at this point that the function log grows at a very slow rate. Figure 2.1 shows the variation of x and $\log_e x$.
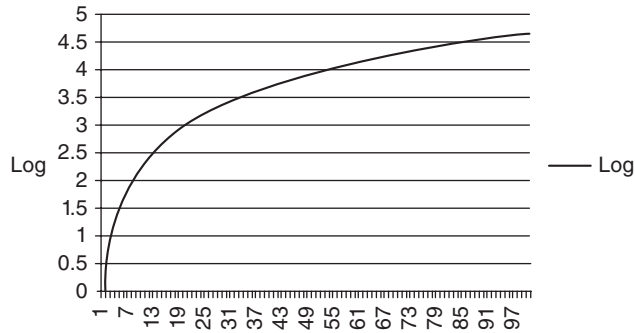
Figure 2.1 The log function

The composition of log is defined as

$$\ln \ln x = (\ln(\ln x))$$

There is a difference between $\ln_a^k x = $ taking $\ln$ of $\ln_a x\,(k-1)$ times and $(\ln_a x)^k$, which is multiplying $(\ln_a x)\,k$ times.

2.2.2 Arithmetic Progression

An arithmetic progression (AP) is one in which the difference between any two terms is constant. The first term of the sequence is generally denoted by 'a' and the constant difference is denoted by 'd'. The terms of an AP are therefore as follows:

$$a, (a + d), (a + 2 \times d), (a + 3 \times d), \ldots \qquad \text{Formula 6}$$

The nth term T_n of the sequence is given by Formula 7:

$$T_n = a + (n - 1) \times d \qquad \text{Formula 7}$$

For example, the sequence 2, 7, 12, 17, 22, … is an AP since the difference between any two terms is 5. However, the following sequence is not an AP since the difference between the consecutive terms is not constant,

$$2, 8, 12, 18, 22, \ldots$$

In order to understand the concept, let there be an AP having the first term = 23 and common difference 12. The ninth term of the AP would be

$$T_n = 23 + (9 - 1) \times 12 = 23 + 96 = 119$$

In order to find the mth term from the end, the following formula can be used:

$$m\text{th term from the end} = (n - m + 1)\text{th term from the beginning} \quad \text{Formula 8}(a)$$

In order to find out the number of terms of an AP, whose 'a', 'd', and 'T_n' are given, apply Formula 7 and equate it to the given value of T_n, in order to obtain the value of n. The value of n so obtained would be the number of terms.

For example, in the sequence

$$213, 247, \ldots, 519$$

the value of 'a' is 213, the value of 'd' is 34, and that of T_n is 519. Since $T_n = 213 + (n-1) \times 34 = 519$, the value of n comes out to be 10. However, if the value of n comes out to be a non-integer, then the given last term does not form the part of the sequence.

The sum of the terms of an AP is given by

$$S_n = \frac{n}{2}(2 \times a + (n-1) \times d) \qquad \text{Formula 8}(b)$$

where a is the first term, d is the common difference, and n is the number of terms of the AP.

2.2.3 Geometric Progression

A geometric progression (GP) is one in which the ratio of any two terms is constant. The first term of the sequence is generally denoted by 'a' and the common ratio is denoted by 'r'. The terms of GP are, therefore, as follows:

$$a, (a \times r), (a \times r^2), (a \times r^3), \ldots$$

The nth term T_n of the sequence is given by Formula 9

$$T_n = a \times r^{n-1} \qquad \text{Formula 9}$$

For example, the sequence

$$2, 10, 50, 250, \ldots$$

is a GP since the ratio of any two terms is 5. However, the following sequence is not a GP since the ratio of the consecutive terms is not constant.

$$2, 8, 12, 18, 22, \ldots$$

In order to understand the concept, let there be a GP having the first term = 23 and common ratio 12. The ninth term of the GP would be

$$T_n = 23 \times 12^8 = 9889579008$$

In order to find the mth term from the end, the following formula can be used.

$$m\text{th term from the end} = (n - m + 1)\text{th term from the beginning.}$$

In order to find out the number of terms of a GP, whose 'a', 'r', and 'T_n' are given, apply Formula 9, and equate it to the given value of T_n, in order to obtain the value of n. The value of n so obtained would be the number of terms.

For example, in the sequence

$$245, 735, \ldots, 59535$$

the value of a is 245, the value of r is 3, and that of T_n is 59535. Since

$$T_n = 245 \times 3^{n-1} = 59535$$

the value of n comes out to be 6. However, if the value of n comes out to be a non-integer, then the given last term does not form the part of the sequence.

The sum of the terms of a GP is given by Formula 10 (a)

$$S_n = \frac{a(r^n - 1)}{(r - 1)} \qquad \text{Formula } 10(a)$$

where a is the first term, r is the common ratio, and n is the number of terms of the GP, if the value $r > 1$.

If $r < 1$, then the formula becomes $S_n = \dfrac{a(1 - r^n)}{(1 - r)}$. In this case, the value of sum upto infinity is given by the following formula:

$$S_n = \frac{a}{(1 - r)} \qquad \text{Formula } 10(b)$$

2.3 ASYMPTOTIC NOTATION

The word 'asymptotic' is made up of three words: 'a', 'sym', and 'totic'. The meaning of 'a' is not, and 'sym' means touch. Asymptote, therefore, means a line that approaches the curve of the polynomial approximately.

Asymptotic notation finds the upper bound of the polynomial as in the case of 'big Oh' notation; or the lower bound as in the case of ω notation; or containment as in the case of θ notation. This section throws some light on the three notations and illustrates the procedure to find asymptotic notations.

2.3.1 O Notation: Big Oh Notation

The big Oh notation is used when the upper bound of a polynomial is to be found. The notation is helpful in finding out the maximum amount of resources an algorithm requires, in order to run. This is important as pre-empting the maximum time (or resources) requirement can help us to schedule the task accordingly. It is also helpful to compare the best-suited algorithm amongst the set of algorithms, if more than one algorithm can accomplish a given task. Figure 2.2 shows the relation between $g(n)$ and $O(g(n))$.

Definition $f(n) = O(g(n))$ if $f(n) \leq C \times g(n)$, $n \geq n_0$, C and n_0 are constants.
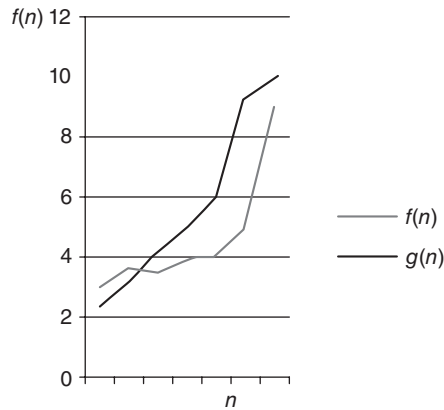
Figure 2.2 The big Oh
notation: $f(n) = O(g(n))$

In order to understand the above point, let us take an example of an algorithm whose running time varies according to the function: $4 \times n^2 + 5 \times n + 3$, n being the number of inputs. The value of the function is less than or equal to $5 \times n^2$, if the value of n is ≥ 6. Table 2.1 shows the variation of values of the polynomial and $5n^2$.

Table 2.1 Comparison of $f(n)$ and $g(n)$

n	4*n*n + 5*n + 3	5*n*n
1	12	5
2	29	20
3	54	45
4	87	80
5	128	125
6	177	180

Hence, it becomes evident from the table that the value of n for which $5 \times n^2$ becomes greater than $4 \times n^2 + 5 \times n + 3$ is 6.

It can, therefore, be stated that

$$g(n) = 5 \times n^2 \text{, for } n \geq 6$$

The examples that follow this section examine the concept in more detail. It may be noted that an algorithm that takes $O(n)$ time is better than the one that takes $O(n^2)$ time.

If $O(n)$ is the upper bound of an algorithm, then $O(n^2)$, $O(n^3)$, $O(n^4)$, etc., would also be the upper bounds.

2.3.2 Ω Notation: Omega Notation

The omega notation is used when the lower bound of a polynomial is to be found. The notation is helpful in finding out the minimum amount of resources, an algorithm

requires, in order to run. Finding out the minimum amount of resources is important as this time can help us to schedule the task accordingly. It is also helpful to compare the best-suited algorithm amongst the set of algorithms, if more than one algorithm can accomplish a given task. Figure 2.3 shows the relation between $g(n)$ and $\Omega(g(n))$.
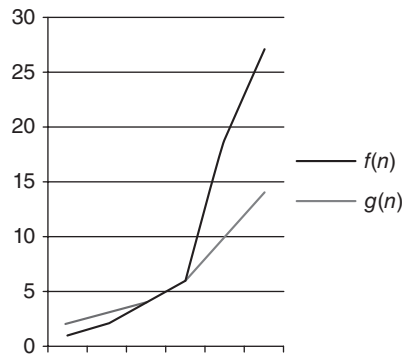
Figure 2.3 $f(n) = \Omega(g(n))$

Definition $f(n) = \Omega(g(n))$, if $f(n) \geq C \times g(n)$, $n \geq n_0$, C and n_0 are constants.

Table 2.2 Comparison of $f(n)$ and $g(n)$

N	5*n*n + 2*n + 5	4*n*n
1	12	4
2	29	16
3	56	36
4	93	64
5	140	100
6	197	144

The grey line depicts the lower bound of the function. In order to understand the above point, let us take an example of an algorithm whose running time varies according to the function $5 \times n^2 + 2 \times n + 7$. The function is greater than or equal to $4 \times n^2$, if the value of n is ≥ 1. Table 2.2 shows the variation of values of the polynomial and $4n^2$. Hence, it becomes evident from the table that $4 \times n^2$ is less than $5 \times n^2 + 2 \times n + 7$ for all values of $n \geq 1$. So it can be stated that $g(n) = 4 \times n^2$, for $n \geq 1$.

If $\Omega(n^3)$ is the lower bound of an algorithm, then $\Omega(n^2)$, $\Omega(n)$, $\Omega(1)$, etc., would also be the lower bounds. For example, if the minimum time taken by an algorithm is $2n + 5$ and the maximum is $4n + 34$, then we can say that the time taken by the algorithm is $c_1 n \leq T(n)$ and hence $T(n) = \Omega(n)$

The examples that follow this section examine the concept in more detail.

2.3.3 θ Notation: Theta Notation

The theta notation is used when the bounds of a polynomial are to be found. The notation is helpful in finding out the minimum and the maximum amount of resources, an algorithm requires, in order to run. Finding out the bounds of resources (or time) is important as this can help us to schedule the task accordingly. The notation is also helpful in finding

the best-suited algorithm amongst the set of algorithms, if more than one algorithm can accomplish a given task. Figure 2.4 shows the relation between $g(n)$ and $\theta(g(n))$.
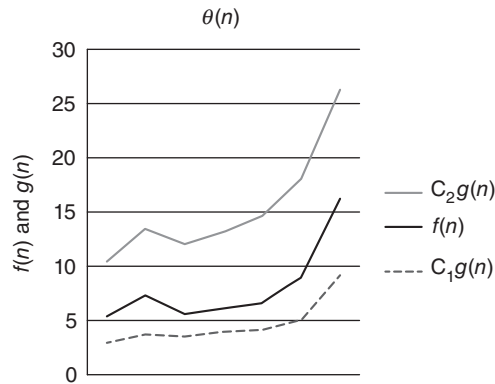
Figure 2.4 $f(n) = \theta\ (g(n))$

Definition $f(n)=\theta(g(n))$, if $C_1 g(n) \leq f(n) \leq C_2 g(n)$, $n \geq n_0$, C and n_0 are constants.

The grey and the dashed lines depict $C_2 g(n)$ and $C_1 g(n)$, accordingly. In order to understand the above point, let us take an example of an algorithm whose running time varies according to the function $3 \times n^2 + 2 \times n + 1$, where n is the input size. The function is greater than or equal to $2 \times n^2$ and less than or equal to $4n^2$, if the value of n is ≥ 3. Table 2.3 shows the variation of values of the polynomial and $4n^2$ and $2 \times n^2$.

Table 2.3 Comparison of $g(n)$ and $\theta(g(n))$

n	3*n*n + 2*n + 1	4*n*n	2*n*n
1	6	4	2
2	17	16	8
3	34	36	18
4	57	64	32
5	86	100	50
6	121	144	72
7	162	196	98
8	209	256	128
9	262	324	162
10	321	400	200
11	386	484	242

Hence, it becomes evident from the table that the value of n for which $2 \times n^2$ is less than $3 \times n^2 + 2 \times n + 1$ is 1 and the value of n for which the function $4n^2$ becomes greater than $3 \times n^2 + 2 \times n + 1$ is 3.

Therefore, $g(n) = 4 \times n^2$, for $n \geq 1$.

The examples that follow this section examine the concept in more detail.

Illustration 2.1 Two algorithms A_1 and A_2 run on the same machine. The running time of A_1 is $100n^2$ and the running time of A_2 is 2^n. For what value of n, A_1 runs faster than A_2?

Solution Table 2.4 shows the variation of values of $100n^2$ and 2^n with n. It may be noted that for $n \geq 15$, 2^n exceeds $100n^2$. So, till $n = 14$, A_2 runs faster.

Table 2.4 Variation of $100n^2$ and 2^n with n

n	$100n^2$	$2n$
1	100	2
2	400	4
3	900	8
4	1600	16
5	2500	32
6	3600	64
7	4900	128
8	6400	256
9	8100	512
10	10,000	1024
11	12,100	2048
12	14,400	4096
13	16,900	8192
14	19,600	16,384
15	22,500	32,768

2^n is greater than $100n^2$ for all $n \geq 15$

In order to make the concept clear, let us also analyse the graph of $100n^2$ and 2^n with n. Figure 2.5 shows the graph. It may be noted that after $n = 15$, 2^n is way ahead of $100n^2$.
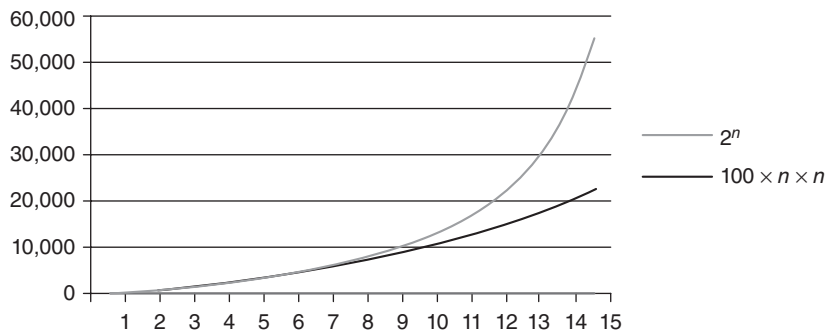
Figure 2.5 Variation of $100n^2$ and $2n$ with n

Now, most of you must be wondering as to why a question which you could have solved in the sixth grade is given to you in this book. The reason being this question portrays the gist of the chapter. It may be noted that in most of the algorithms, the number of inputs is generally greater than 15. The number can be in 100 s or even in 1000 s. The large number of inputs forms the basis of growth of functions which forms the basis of this chapter. Another example that demonstrates the concept is as follows:

Illustration 2.2 Two algorithms A_1 and A_2 run on the same machine. The running time of A_1 is $100n^{30}$ and the running time of A_2 is 2^n. Can A_1 run faster than A_2?

Solution The question is similar to the first illustration, except for the fact that the power of n in this case is 30. Now from a quick observation, it appears that A_1 will always take more time as compared to A_2, but Fig. 2.6 shows the variation of values of $100n^{30}$ and 2^n with n. It may be noted that for $n \geq 245$, 2^n exceeds $100n^{30}$.
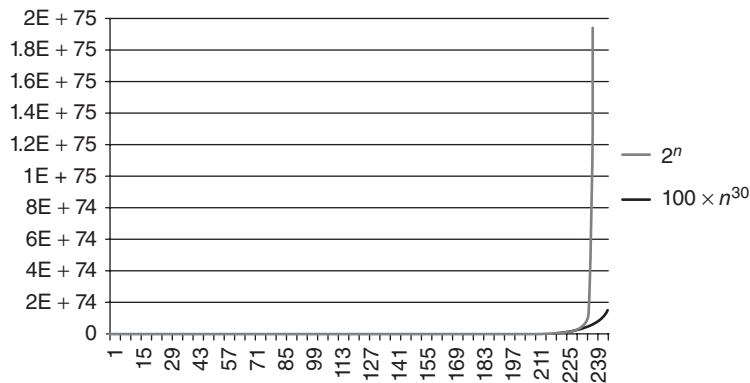
Figure 2.6 Variation of $100n^2$ and 2^n with n

So, it is worth remembering that for large values of n, 2^n will always be greater than $c \times n^m$, n being the number of inputs and m is any integer.

Illustration 2.3 Find omega notation for $g(n) = 3 \times n^2 + 2 \times n + 5$.

Solution As per the definition of 'Ω' notation, the function $f(n)$ such that

$$3 \times n^2 + 2 \times n + 5 \geq c \times f(n), \, n \geq n_0$$

will be the $\Omega(g(n))$.

It may be noted that $3 \times n^2 + 2 \times n + 5 \geq 2 \times n^2, n \geq n_0$. Table 2.5 shows the values of $3 \times n^2 + 2 \times n + 5$ and $2 \times n^2$. Hence, for $\forall n \geq 1$, the above inequality holds

$$3 \times n^2 + 2 \times n + 5 \geq 2 \times n^2$$

Therefore, $3 \times n^2 + 2 \times n + 5 = \Omega(n^2)$.

Table 2.5 Variation of $3n^2 + 2n + 5$
and $2n^2$ with n

n	$3n^2 + 2n + 5$	$2n^2$
1	10	2
2	21	8
3	38	18
4	61	32
5	90	50
6	125	72
7	166	98
8	213	128
9	266	162
10	325	200
11	390	242
12	461	288
13	538	338
14	621	392
15	710	450

Illustration 2.4 Find big Oh notation for $g(n) = 3 \times n^2 + 2 \times n + 5$.

Solution As per the definition of 'O' notation, the function $f(n)$ such that

$$c_1 \times f(n) \geq 3 \times n^2 + 2 \times n + 5, \, n \geq n_0$$

will be the $O(g(n))$.

It may be noted that $3 \times n^2 + 2 \times n + 5 \leq 4 \times n^2, \, n \geq n_0$. Table 2.6 shows the values of $3 \times n^2 + 2 \times n + 5$ and $4 \times n^2$. Hence, for $\forall n \geq 4$, the above inequality holds.

Therefore, $$3 \times n^2 + 2 \times n + 5 = O(n^2)$$

Table 2.6 Variation of $3n^2 + 2n + 5$ and $4n^2$ with n

n	$3n^2 + 2n + 5$	$4n^2$	
1	10	4	for $n \geq 4$, $4n^2$ becomes greater than $3n^2 + 2n + 5$
2	21	16	
3	38	36	
4	61	64	
5	90	100	
6	125	144	
7	166	196	
8	213	256	

(Contd)

Table 2.6   (Contd)

n	$3n^2 + 2n + 5$	$4n^2$
9	266	324
10	325	400
11	390	484
12	461	576
13	538	676
14	621	784
15	710	900

Illustration 2.5 Find theta notation for $g(n) = 3 \times n^2 + 2 \times n + 5$.

Solution As per the definition of 'θ' notation, the function $f(n)$ such that

$$c_1 \times f(n) \leq 3 \times n^2 + 2 \times n + 5 \leq c_2 \times f(n), \ n \geq n_0$$

will be the $\theta(g(n))$. The above two illustrations confirm the fact that if $f(n) = n^2$, then both

$$c_1 \times f(n) \leq 3 \times n^2 + 2 \times n + 5, \ n \geq n_0$$

$$3 \times n^2 + 2 \times n + 5 \leq c_2 \times f(n), \ n \geq n_0$$

Therefore, $3 \times n^2 + 2 \times n + 5 = \theta(n^2)$.

Illustration 2.6 Find omega notation for $g(n) = n \times \log n + 5$.

Solution As per the definition of 'ω' notation, the function $f(n)$ such that

$$c_1 \times f(n) \leq n \times \log n + 5, \ n \geq n_0$$

will be the $\omega(g(n))$.

Now, if the value of $f(n) = \log(n)$, then the above equation is satisfied. Table 2.7 shows the variation of the given function, $g(n)$ with $f(n)$. The graph is depicted in Fig. 2.7.

Table 2.7   Variation of $n\log(n) + 5$ with $\log(n)$

n	$n\log(n) + 5$	$\log(n)$
1	5	0
2	5.60206	0.30103
3	6.431364	0.477121
4	7.40824	0.60206
5	8.49485	0.69897
6	9.668908	0.778151
7	10.91569	0.845098
8	12.22472	0.90309
9	13.58818	0.954243
10	15	1
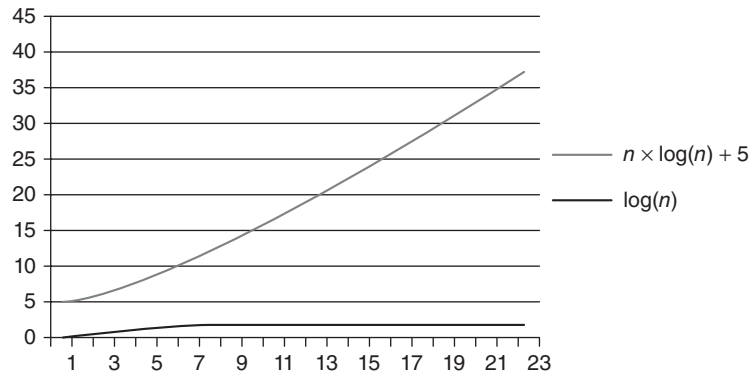
Figure 2.7 Variation of $n\log n + 5$ and $\log n$ with n

It is evident from the graph that $n \times \log n + 5 \geq \log n,\ n \geq 1$. Therefore, $\omega(n \times \log n + 5) = \log n$.

Illustration 2.7 Find 'O notation' for $g(n) = n\log n + 5$.

Solution As per the definition of 'O' notation, the function $f(n)$ such that

$$c_1 \times f(n) \geq n \times \log n + 5,\ n \geq n_0$$

will be the $O(g(n))$.

Now, if the value of $f(n) = 2n$, then the above equation is satisfied. Table 2.8 shows the variation of the given function, $g(n)$ with $f(n)$. The graph is depicted in Fig. 2.8. Since 2 is a constant, therefore, $f(n) = n$ qualifies as $O(f(n))$.

Table 2.8 Variation of $n\log(n) + 5$ and $2n$ with n

n	$n\log(n) + 5$	$2n$
1	5	4
2	5.60206	8
3	6.431364	12
4	7.40824	16
5	8.49485	20
6	9.668908	24
7	10.91569	28
8	12.22472	32
9	13.58818	36
10	15	40
11	16.45532	44
12	17.95017	48
13	19.48126	52
14	21.04579	56
15	22.64137	60
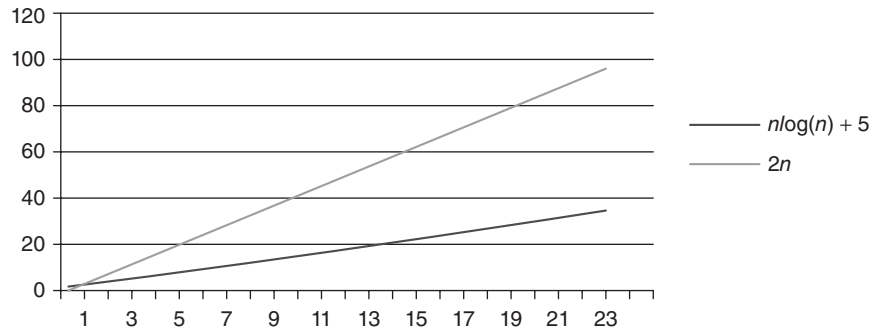
Figure 2.8   Variation of $n\log n + 5$ and $2n$ with n

It is evident from the graph that $n \times \log n + 5 \leq 2n$, $n \geq 1$. Therefore, $n \times \log n + 5 = O(n)$.

2.3.4 ω Notation: Small Omega Notation

The ω notation is defined as follows:

$$f(n) = \omega(g(n)), \text{ iff } \lim_{n \to \infty} \frac{g(n)}{f(n)} = 0$$

For instance, if $f(n) = 2 \times n + 3$, then $g(n) = 1$, since $\lim_{n \to \infty} \dfrac{1}{2 \times n + 3} = 0$.

In most of the cases, the degree of $g(n)$ is one less than $f(n)$. However, the premise does not always hold good.

If the value of $f(n) = 2 \times n^2 + 3 \times n + 7$, then $g(n)$ would be n, which has degree one less than $f(n)$. However, if $f(n) = \log n$, then $g(n) = 1$.

2.3.5 o Notation: Small oh Notation

The o notation is defined as follows:

$$f(n) = o(g(n)), \text{ iff } \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

For instance, if $f(n) = 2 \times n + 3$, then $g(n) = n^2$, since $\lim_{n \to \infty} \dfrac{2 \times n + 3}{n^2} = 0$.

It may be noted that, in most of the cases, the degree of $g(n)$ is one more than $f(n)$. If the value is $f(n) = 2 \times n^2 + 3 \times n + 7$, then $g(n)$ would be n^3, which has degree one more than $f(n)$.

2.3.6 Comparison of Functions

It was stated earlier that 2^n surpasses any other function. However, the statement is not true for all the values of n. The present section compares the values of n, $\log n$, n^2, n^3, and 2^n. The values are given in Table 2.9 and the corresponding graph is shown in Fig. 2.9.

Table 2.9 Comparison of functions

Log n	n	n^2	n^3	2^n
0	1	1	1	2
0.30103	2	4	8	4
0.477121	3	9	27	8
0.60206	4	16	64	16
0.69897	5	25	125	32
0.778151	6	36	216	64
0.845098	7	49	343	128
0.90309	8	64	512	256
0.954243	9	81	729	512
1	10	100	1000	1024
1.041393	11	121	1331	2048
1.079181	12	144	1728	4096
1.113943	13	169	2197	8192
1.146128	14	196	2744	16384
1.176091	15	225	3375	32768
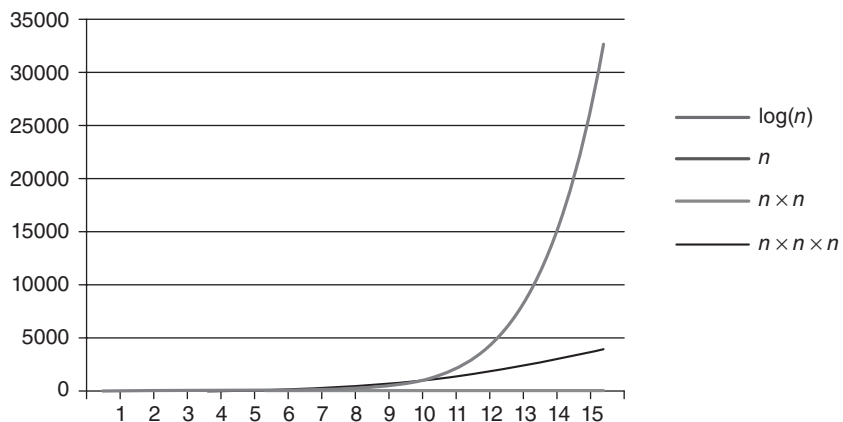
Figure 2.9 Variation of various functions with n

Note that 2^n is always greater than any function, for larger values of n. It may be noted that the lines depicting 2^n and n seem to overlapp because of the scale of y-axis. The students are advised to plot the values of n and 2^n for smaller scale.

2.4 PROPERTIES OF ASYMPTOTIC COMPARISONS

The first property that is being discussed is reflexivity. Reflexivity, in general, is defined as $f(a) = a$, where $f(x)$ is a function and a belongs to its domain.

In the case of asymptotic notations, the *reflexivity* is defined as follows:

$$f(n) = \Omega(f(n))$$
$$f(n) = O(f(n))$$
and
$$f(n) = \theta(f(n))$$

The symmetry property in general is defined as follows:

For a function $f(x, y)$, $f(a, b) = f(b, a)$.

In the case of asymptotic functions, *symmetry* can be stated as $f(n) = \theta(g(n))$ if $g(n) = \theta(f(n))$. However, as per the definitions of O and Ω, if $f(n) = O(g(n))$, then $g(n) = \Omega(f(n))$. This property is called *transpose symmetry*.

The transpose symmetry is also valid for o and ω, that is, if $f(n) = o(g(n))$, then $g(n) = \omega(f(n))$. The *transitivity*, in general, is defined as

$$f(a) = g(b) \text{ and } g(b) = c, \text{ then } f(a) = c$$

In the case of asymptotic notations, the following relations hold:

$$f(n) = \theta(g(n)) \text{ and } g(n) = \theta(j(n)), \text{ then } f(n) = \theta(j(n))$$
$$f(n) = O(g(n)) \text{ and } g(n) = O(j(n)), \text{ then } f(n) = O(j(n))$$
$$f(n) = \Omega(g(n)) \text{ and } g(n) = \Omega(j(n)), \text{ then } f(n) = \Omega(j(n))$$
$$f(n) = o(g(n)) \text{ and } g(n) = o(j(n)), \text{ then } f(n) = o(j(n))$$
$$f(n) = \omega(g(n)) \text{ and } g(n) = \omega(j(n)), \text{ then } f(n) = \omega(j(n))$$

However, it may be stated at this point that *trichotomy* does not hold in the case of asymptotic notations.

2.5 THEOREMS RELATED TO ASYMPTOTIC NOTATIONS

This section presents some basic theorems related to the asymptotic notations introduced in the earlier sections. The theorem proofing, in the case of asymptotic functions, requires the understanding of basic definitions. Theorem 2.1 has been proved. The proofs of Theorems 2.2 and 2.3 are left as an exercise for the readers.

Theorem 2.1 If $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$, then $f(n) = \theta(g(n))$.

Proof If $f(n) = O(g(n))$, then there exists c_1 such that

$$f(n) \leq c_1 (g(n))$$

Moreover, $f(n) = \Omega(g(n))$, therefore, there exists c_2 such that

$$f(n) \geq c_2 g(n)$$

Combining the above two results it may be stated that

$$c_1 \times g(n) \le f(n) \le c_2 \times g(n)$$

which means that $f(n) = \theta(g(n))$.

The above theorem can be understood with the help of the following example. Let $f(n) = c_1 x^n + c_2 x^{n-1} + \cdots + c_n x^0$, then

$$f(n) = O(x^n)$$

also

$$f(n) = \Omega(g(n))$$

therefore

$$f(n) = \theta(g(n))$$

Theorem 2.2 If $f(n)$ and $g(n)$ are two non-negative functions, then

$$\max(f(n),\, g(n)) = \theta(f(n) + g(n))$$

Theorem 2.3 If $f(n)$ and $g(n)$ are two non-negative functions, then

$$\max(f(n),\, g(n)) = O(f(n) + g(n))$$

2.6 CONCLUSION

This chapter introduces the concept of asymptotic notations. It would help in determining the space and time complexity of the algorithms that follow. The chapter is the basis of the rest of the chapters. The knowledge of basic sequences and logarithms is also necessary in order to handle difficult mathematical tasks. The chapter, therefore, throws some light on the basic mathematical concepts as well. There are five basic asymptotic functions, each of which has been defined in Section 2.3. The properties of these functions have been dealt with in Section 2.4. It is highly recommended that the reader strives to find all the asymptotic notations of as many functions as he/she can.

Points to Remember

- The big Oh notation is used when the upper bound of a polynomial is to be found.
- The omega notation is used when the lower bound of a polynomial is to be found.
- The theta notation is used when the bounds of a polynomial are to be found.
- $f(n) = \Omega(f(n))$

- $f(n) = O(f(n))$ and $f(n) = \theta(f(n))$
- If $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$, then $f(n) = \theta(g(n))$.
- If $f(n)$ and $g(n)$ are two non-negative functions, then $\max(f(n), g(n)) = \theta(f(n) + g(n))$
- If $f(n)$ and $g(n)$ are two non-negative functions, then $\max(f(n), g(n)) = O(f(n) + g(n))$
- A $\theta(1)$ algorithm is better than $\theta(n)$, which in turn is better than $\theta(n^2)$ and so on. Same is the case with O and Ω.
- A $\theta(n^k)$ algorithm is better than $\theta(k^n)$

KEY TERMS

Arithmetic progression An arithmetic progression (AP) is one in which the difference between any two terms is constant.

Geometric progression A geometric progression (GP) is one in which the ratio of any two terms is constant.

O notation $f(n) = O(g(n))$, if $f(n) \leq C \times g(n)$, $n \geq n_0$, C and n_0, C and n_0, are constants.

o notation $f(n) = \omega(g(n))$, iff $\lim_{n \to \infty} \dfrac{f(n)}{g(n)} = 0$.

Ω notation $f(n) = \Omega(g(n))$, if $f(n) \geq C \times g(n)$, $n \geq n_0$, C and n_0, C and n_0 are constants.

θ notation $f(n) = \theta(g(n))$, if $c_1 g(n) \leq f(n) \leq c_2 g(n)$, $n \geq n_0$, C and n_0 are constants.

ω notation $f(n) = \omega(g(n))$, iff $\lim_{n \to \infty} \dfrac{g(n)}{f(n)} = 0$.

EXERCISES

I. Multiple Choice Questions

1. If $f(n) = 2 \times n + 5$, then $f(n)$ is
   - (a) $O(n)$
   - (b) $O(n^2)$
   - (c) $O(n^3)$
   - (d) All of the above

2. If $f(n) = 4 \times n + 3$, then $f(n)$ is
   - (a) $O(n)$
   - (b) $O(1)$
   - (c) $O(\log n)$
   - (d) All of the above

3. If $f(n) = 3 \times n + 7$, then $f(n)$ is
   - (a) $\Omega(n)$
   - (b) $\Omega(n^2)$
   - (c) $\Omega(n^3)$
   - (d) All of the above

4. If $f(n) = 3 \times n + 7$, then $f(n)$ is
   - (a) $\Omega(n)$
   - (b) $\Omega(\log n)$
   - (c) $\Omega(1)$
   - (d) All of the above

5. If $f(n) = 2 \times n^2 + 5 \times n + 3$, then $O(n)$
   - (a) $O(n)$
   - (b) $O(n^2)$
   - (c) $O(n^3)$
   - (d) All of the above

6. If $f(n) = 2 \times n^2 + 5 \times n + 3$, then $O(n)$
   - (a) $O(n)$
   - (b) $O(1)$
   - (c) $O(\log n)$
   - (d) None of the above

7. If $f(n) = 3 \times \log n + 7n + 3$, then $f(n)$ is
   - (a) $\Omega(n)$
   - (b) $\Omega(n^2)$
   - (c) $\Omega(\log n)$
   - (d) All of the above

8. If $f(n) = 3 \times \log n + 7n + 3$, then $f(n)$ is
   - (a) $O(n)$
   - (b) $O(\log n)$
   - (c) $\Omega(1)$
   - (d) All of the above

9. If $f(n) = 3 \times \log n + 7 \times 2^n + 3$, then $f(n)$ is
   - (a) $\Omega(2^n)$
   - (b) $\Omega(n^2)$
   - (c) $\Omega(\log n)$
   - (d) All of the above

10. If $f(n) = 3 \times \log n + 7n + 3$, then $f(n)$ is
    - (a) $O(2^n)$
    - (b) $O(\log n)$
    - (c) $O(n)$
    - (d) All of the above

II. Review Questions

1. Explain the big Oh notation.
2. Explain the significance of the omega notation.
3. Explain the importance of the theta notation.
4. What is the importance of the study of growth of functions?
5. Why do we need to know the maximum and minimum amount of resources required by an algorithm to run?

III. Numerical Problems

1. Find big Oh notation for the following:
   - (a) $f(n) = 3 \times n + 2$
   - (b) $f(n) = 3 \times n^2 + 5 \times n + 4$
   - (c) $f(n) = n^2 + 3 \times n + 1$
   - (d) $f(n) = 100 \times n^2 + 91 \times n + 4000$
   - (e) $f(n) = 3 \times n^3 + 2 \times n^2 + 5 \times n + 2$
   - (f) $f(n) = 3 \times n^3 + 2198 \times n^2 + 55 \times n + 27$

(g) $f(n) = 3 \times n^4 + 2 \times n^3 + 5 \times n + 2$

(h) $f(n) = 2 \times n^{33} + 2 \times n^{20} + 87 \times n + 19$

(i) $f(n) = 2^n + 3 \times n^3 + 2 \times n^2 + 5 \times n + 2$

(j) $f(n) = 2^{3n} + 2^n + 3 \times n^3 + 2 \times n^2 + 5 \times n + 2$

(k) $f(n) = 3 \times \log n + 2 \times n^2 + 5 \times n + 2$

$f(n) = 3 \times \log n + 2$

2. Find theta notation for the following:

(a) $f(n) = 5 \times n + 2$

(b) $f(n) = 2 \times n^2 + 4 \times n + 3$

(c) $f(n) = n^2 + 7 \times n + 9$

(d) $f(n) = 10 \times n^2 + 191 \times n + 4296$

(e) $f(n) = 6 \times n^3 + 12 \times n^2 + 57 \times n + 23$

(f) $f(n) = 3 \times n^3 + 19 \times n^2 + 50 \times n + 21$

(g) $f(n) = 30 \times n^4 + 2871 \times n^3 + 52 \times n + 2$

(h) $f(n) = 21 \times n^{33} + 12 \times n^{20} + 8 \times n + 119$

(i) $f(n) = 2^n + 90 \times n^3 + 22 \times n^2 + 25 \times n + 26$

(j) $f(n) = 2^{3n} + 2^n + 321 \times n^3 + 21212 \times n^2 + 5113 \times n + 24$

(k) $f(n) = 3 \times \log n + 234 \times n^2 + 523 + 22324$

(l) $f(n) = 0.3 \times \log n + 23112$

3. Find omega notation for the following:

(a) $f(n) = 5 \times n + 2$

(b) $f(n) = 4 \times n^2 + 3 \times n + 5$

(c) $f(n) = n^2 + 237 \times n + 9345$

(d) $f(n) = 1345 \times n^2 + 1435 \times n + 4245$

(e) $f(n) = 643 \times n^3 + 14352 \times n^2 + 5437 \times n + 236$

(f) $f(n) = 343 \times n^3 + 1459 \times n^2 + 50 \times n + 2167$

(g) $f(n) = 30 \times n^4 + 2871 \times n^3 + 52 \times n + 2567$

(h) $f(n) = 21 \times n^{33} + 152 \times n^{20} + 65 \times n + 9$

(i) $f(n) = 2^n + 90 \times n^3 + 22567 \times n^2 + 2565 \times n + 4$

(j) $f(n) = 2^{3n} + 2^n + 3 \times n^3 + 2 \times n^2 + 3 \times n + 4$

(k) $f(n) = 3 \times \log n + 3 \times n^2 + 2 \times n + 4$

(l) $f(n) = 3 \times \log n + 4$

4. Prove the following:

(a) $f(n) = 5 \times n + 2 = O(n^2)$

(b) $f(n) = 5 \times n + 2 = O(n)$

(c) $f(n) = n^2 + 237 \times n = \theta(n^2)$

(d) $f(n) = 14 \times n + 42 = O(n)$

(e) $f(n) = 6 \times n^3 + 4 \times n^2 + 5 \times n + 6 = O(n)$

(f) $f(n) = 3 \times n^3 + 2 \times n^2 + 1 \times n = \Omega(n^3)$

(g) $f(n) = 30 \times n^4 + 2871 \times n^3 + 52 \times n + 2567 = O(n^4)$

(h) $f(n) = 30 \times n^4 + 2871 \times n^3 + 52 \times n + 2567 = O(n^5)$

(i) $f(n) = 2^n + 7 \times n^2 + 6 \times n + 5 = O(2^n)$

(j) $f(n) = 2^{3n} + 2^n + 2^n = O(2^{3n})$

(k) $f(n) = \log n + 2 \times n + 4 = O(\log n)$

(l) $f(n) = 3 \times \log n + n = O(n)$

(m) $O(n) > O(\log n)$

(n) $O(2^n) > O(n^5)$

5. Arithmetic Progression
   (a) Find the T_n term for the following APs:

   (i) $a = \sqrt{5}$, $d = \sqrt{5}$ and $n = 17$.

   (ii) $a = \dfrac{2}{3}$, $d = \dfrac{7}{3}$ and $n = 18$.

(iii) $a = 2 + \sqrt{3}$, $d = 2 - \sqrt{3}$, and $n = 20$.

(iv) $a = 2 + 3i$, $d = 2 - 3i$, and $n = 23$, where $i = \sqrt{-1}$.

(b) In the above question, consider T_n to be the last term, find the 5th term from the end in each case.

(c) Find the number of terms in the following sequences:

(i)   213, 250, …, 546

(ii)   6580, 6817, …, 8713

(iii) i, 1, …, 10

(iv) $\dfrac{1}{7}, \dfrac{3}{7}, ..., 12$

(d) In question 20, consider T_n to be the last term, find the sum of the terms in each case.

6. Geometric Progression

(a) Find the T_n term for the following GPs:

(i) $a = \sqrt{5}$, $r = \sqrt{3}$, and $n = 17$.

(ii) $a = \dfrac{2}{3}$, $r = \dfrac{1}{3}$, and $n = 18$.

(iii) $a = 2 + \sqrt{3}$, $r = 2 - \sqrt{3}$, and $n = 20$.

(iv) $a = 2 + 3i$, $r = 1 - 3i$, and $n = 23$, where $i = \sqrt{-1}$.

(b) In the above question, consider T_n to be the last term, find the 5th term from the end in each case.

(c) Find the number of terms in the following sequences:

(i)   26, 69, …, 16767

(ii)   36, 216, …, 1679616

(iii)   0.21, 1.89, …, 111602.6

(iv)   3.124, 9.372, …, 2277.396

(d) In question c, consider T_n to be the last term, find the sum of the terms in each case.

7. Miscellaneous Problems

(a) If $\log 2 = 0.3010$ and $\log 5 = 0.6991$, then find the values of the logarithm for the following numbers:

(i) 2560		(vi) 10,000	
(ii) 320,000		(vii) 5000	
(iii) 64,000		(viii) 16,000	
(iv) 2560		(ix) 64	
(v) 1289		(x) 50	

8. Is $a^{n+1} = O(a^n)$ where a is an integer?

9. Is the statement 'The running time of an algorithm is maximum $\Omega(n^2)$' meaningful?

10. What is $O(f(n)) \cap \Omega(f(n))$?
11. Can you compare any two functions using asymptotic notations?
12. Which of the two is bigger as the value of n approaches ∞?

$$\sqrt{n} \text{ or } n^{\cos n}$$

13. Which of the two is bigger as the value of n approaches ∞?

$$n^{\log c} \text{ or } c^{\log n}$$

14. If $f(n) = O(f(n)^2)$?
15. Is $f(n) = \Omega(f(n/3))$?
16. If $f(n) = O(g(n))$, then $g(n) = O(f(n))$?
17. Find the O notation for $\log(n!)$.

Answers to MCQs				
1. (d)	3. (a)	5. (b), (c)	7. (c)	9. (d)
2. (a)	4. (d)	6. (d)	8. (a)	10. (a), (c)

CHAPTER 3

Recursion

OBJECTIVES

After studying this chapter, the reader will be able to

- Understand the importance and meaning of recursion
- Appreciate the importance of stacks in recursion
- Understand the rabbit problem
- Understand various methods for solving a recursive equation
- Apply substitution to solve the recursive equation
- Appreciate the concept of generating functions and their application in solving recursive equations

3.1 INTRODUCTION

Recursion means calling a function in itself. If a function invokes itself, then the phenomenon is referred to as *recursion*. However, in order to generate an answer, a terminating condition is must. In order to understand the concept, let us take an example. If the factorial of a number is to be calculated using the function fac(n) defined as follows:

$$\text{fac}(n) = n \times \text{fac}(n-1)$$

and fac(1) = 1, and if the value of n is 5, then the process of calculating fac(5) can be explained with the help of Fig. 3.1. fac(1) is calculated and its value is used to calculate
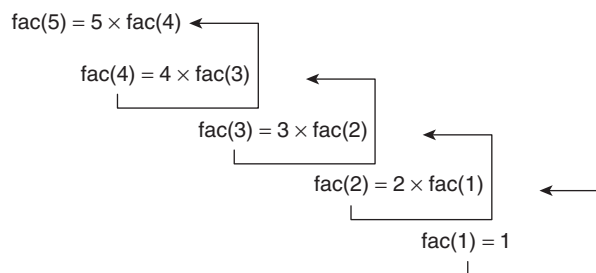
Last In First Out

Figure 3.1 Calculation of factorial of 5

fac(2), which in turn is used for calculating fac(3). fac(3) helps to calculate fac(4) and finally, fac(4) is used to calculate fac(5).

As is evident from Fig. 3.1, recursion uses the principle of last in first out and hence requires a stack. One can also see that had there been no fac(1), the evaluation would not have been possible. This was the reason for stating that recursion requires a terminating condition also.

3.2 RABBIT PROBLEM

The rabbit problem is one of the most famous problems in recursion. The problem is the source of Fibonacci series. The problem goes as follows. A newborn rabbit does not breed for the first 2 months. After which, each pair breeds a pair of rabbits each month. If initially there is a pair of rabbit, for the first 2 months, there will be a single pair, after which there would be two and three pairs in the next 2 months. However, in the fifth month, there would be five pairs. This number increases as shown in Fig. 3.2.
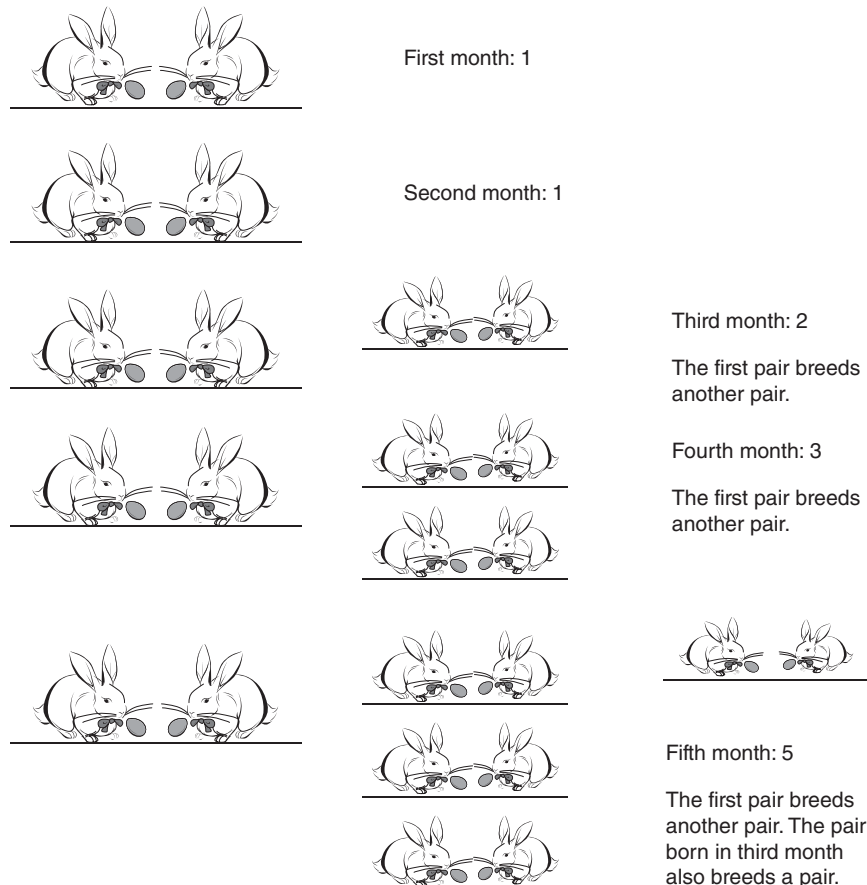
Figure 3.2 The rabbit problem

Interestingly, the sequence of the number of rabbit pairs formed is 1, 1, 2, 3, 5, 8, etc. Each term of this sequence is the sum of previous two terms. That is any term can be found by the following formula:

$$f(n) = f(n-1) + f(n-2)$$

where
$$f(1) = 1 \quad \text{and} \quad f(2) = 1$$

The above formula is easy to comprehend. It is, however, not that easy to find the explicit formula for the nth term of the above sequence. Section 3.3 would help us to derive an explicit formula of a given recursion relation.

3.3 DERIVING AN EXPLICIT FORMULA FROM RECURRENCE FORMULA

Though a recurrence formula gives an idea of how a particular term is related to the previous or the following term, it does not help us to directly find a particular term without having gone through all the intervening terms. For that, we need an explicit formula. There are three methods for finding an explicit formula from a recurrence relation. They are as follows:

- Substitution
- Generating functions
- Tree method

Substitution requires the substitution of a previous instance of the formula in the present relation. This method is discussed in Section 3.3.1. Generating functions are discussed in Section 3.6. The tree method requires finding the solution by determining the number of inputs processed at each level of the tree. The tree method is discussed in Section 4.3 of Chapter 4. The choice of the method, however, is a precarious issue. There is no thumb rule to determine which method to be used for a particular relation. However, many illustrations have been included in the following sections, which would help us to develop an insight into the complex process of deriving an explicit formula for a recurrence relation.

3.3.1 Substitution Method

The solution of a recurrence equation by substitution requires a previous instance of the formula to be substituted in the given equation. The process is continued till we are able to reach to the initial condition. Illustration 3.1 gives an example of the method.

Illustration 3.1 Solve the following recurrence relation by substitution:

$$a_n = 2 \times a_{n-1} + 3, \quad n \geq 2$$

$$a_1 = 2, \quad n = 2$$

Solution Since, $a_n = 2 \times a_{n-1} + 3$, $n \geq 2$, therefore

$$a_{n-1} = 2 \times a_{n-2} + 3, \quad n \geq 2 \tag{3.1}$$

Substituting the value of a_{n-1}, we get

$$a_n = 2 \times \left((2 \times a)_{n-2} + 3\right) + 3 \tag{3.2}$$

which is same as, $\qquad a_n = 4 \times a_{n-2} + 2 \times 3 + 3$

From the given equation, it can be inferred that

$$a_{n-2} = 2 \times a_{n-3} + 3 \tag{3.3}$$

By substituting Eq. (3.3) in Eq. (3.2), we get

$$a_n = 2 \times \left((2 \times (2 \times a)_{n-3} + 3) + 3\right) + 3, \text{ that is,}$$

$$a_n = 2^r \times a_{n-r} + 3 \times \left(1 + 2 + \cdots + 2^{r-1}\right)$$

or $\qquad a_n = 2^r \times a_{n-r} + 3 \times \left(2^r - 1\right) \tag{3.4}$

Putting $\qquad n - r = 2 \quad \text{or} \quad r = (n-2)$

we get $\qquad a_n = 2^{n-2} \times a_2 + 3 \times \left(2^{n-2} - 1\right)$

Since, $\qquad a_2 = 2$

Therefore, $\qquad a_n = 2^{n-2} \times 2 + 3 \times \left(2^{n-2} - 1\right)$

This implies, $\qquad a_n = 2^{n-1} + 3 \times \left(2^{n-2} - 1\right)$

or, $\qquad a_n = \dfrac{5}{2} 2^{n-1} - 3$

Illustration 3.2 A person wants to make an investment at the rate of 10% compounded annually. What will be the amount after n years if the initial amount is ₹10,000?

Solution As per the problem

$$a_n = a_{n-1} * 0.1, \; a_0 = 10,000 \tag{3.5}$$

Therefore, $a_{n-1} = a_{n-2} * 0.1$, substituting in Eq. (3.5), we get

$$a_n = a_{n-2} \times 0.1 \times 0.1 \tag{3.6}$$

Generalizing, we get

$$a_n = a_{n-r} \times 0.1^r \tag{3.7}$$

Putting $(n - r) = 1$, we get $r = n - 1$

$$a_n = a_0 \times (0.1)^{n-1}, \text{ where } a_0 = 10,000$$

Linear Recurrence Relation

A linear recurrence relation is of the form

$$a_n = k_1 \times a_{n-1} + k_2 \times a_{n-2} + \cdots + k_{n-1} \times a_1 \tag{3.8}$$

where k_1, k_2, k_3, etc., are constants.

Examples of linear recurrence relations are as follows:

- $a_n = 3 \times a_{n-1}$, where $a_1 = 1$
- $a_n = a_{n-1} + a_{n-2}$, $a_1 = 1$ and $a_2 = 1$
- $a_n = 2 \times a_{n-1} + 3 \times a_{n-2}$, $a_1 = 2$

The following equations are not linear:

- $a_n = n \times a_{n-1}$, where $a_1 = 1$
- $a_n = a_{n-1} + \left((a)_{n-2} \right)^2$, $a_1 = 1$ and $a_2 = 1$
- $a_n = 2^n \times a_{n-1} + 3 \times a_{n-2}$, $a_1 = 2$

Such equations can be solved by the method of generation functions described in the following discussion. First of all, the characteristic equation of the given relation is formed. In the characteristic equation formed, the order is the difference between the highest and the lowest subscripts of the equation.

For example, the equation corresponding to $a_n = a_{n-1} + a_{n-2}$ would be

$$s^2 = s + 1$$

That corresponding to $a_n = a_{n-1} + a_{n-2} + 2 \times a_{n-2}$ would be

$$s^3 = s^2 + s + 2$$

and so on.

The characteristic equation thus formed is solved. Suppose the roots are r_1, r_2, etc. (all different), then the solution of the given equation is

$$a_n = c_1 (r_1)^n + c_2 (r_2)^n + \cdots$$

However, if two of the roots are same $(= r_1, r_2 \ldots)$, then the solution would be

$$a_n = \left((c)_1 + rc_2 \right)(r_1)^n + c_2 (r_2)^n + \cdots$$

In this case, where three roots are same, the solution would be

$$a_n = \left((c)_1 + rc_2 + r^2 c_3 \right)(r_1)^n + c_2 (r_2)^n + \cdots$$

In the above cases, the values of constants can be found by the initial conditions. Illustrations 3.3, 3.4, and 3.5 depict the above conditions.

Illustration 3.3 Find the nth term of Fibonacci series.

Solution The general term of a Fibonacci series can be expressed as a recurrence relation as follows:

$$a_n = a_{n-1} + a_{n-2}, \quad a_1 = 1, \quad \text{and} \quad a_2 = 1$$

The characteristic equation of the above equation would be as follows:

$$s^2 = s + 1$$

Solving, we get

$$s = \frac{-(-1) \pm \sqrt{(-1)^2 - 4 \times 1 \times (-1)}}{2(1)}$$

which is

$$s = \frac{1 \pm \sqrt{5}}{2}$$

So, the solution would be

$$a_n = c_1 \left(\frac{1 + \sqrt{5}}{2} \right)^n + c_2 \left(\frac{1 - \sqrt{5}}{2} \right)^n$$

Putting $n = 1$ and $a_1 = 1$, we get

$$a_n = \left(\frac{1}{\sqrt{5}} \right) \left(\frac{1 + \sqrt{5}}{2} \right)^{n+1} - \left(\frac{1}{\sqrt{5}} \right) \left(\frac{1 - \sqrt{5}}{2} \right)^{n+1}$$

3.4 SOLVING LINEAR RECURRENCE EQUATION

A linear recurrence relation of order 'r' with constant coefficients is of the form

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_r a_{n-r}$$

where $c_r \neq 0$.
For example,

$a_n = 3a_{n-1}$ is a recurrence relation of order 1.

$a_n = a_{n-1} + a_{n-2}$ is also a recurrence relation, which depicts the Fibonacci series, of order 2.

$a_n = a_{n-1} + a_{n-2} + a_{n-3}$ is a recurrence relation of order 3.

The first step in solving a recursive relation is to form its characteristic equation. A characteristic equation is a polynomial equation formed by retaining the constants of the given equation and by replacing with the powers of s as shown in the following examples. As a matter of fact, the answer depends on the solution of the equation. So, it does not really make a difference, if one opts for other variables, except for s.

Examples of characteristic equations:

- For the equation $a_n = c_1 a_{n-1}$
  The characteristic equation would be $s = c_1 s^0$
- For the equation $a_n = c_1 a_{n-1} + c_2 a_{n-2}$
  The characteristic equation would be $s^2 = c_1 s^1 + c_2 s^0$
- For the equation $a_n = c_1 a_{n-1} + c_2 a_{n-2} + c_3 a_{n-3}$
  Then characteristic equation would be $s^3 = c_1 s^2 + c_2 s^1 + c_3 s^0$

The next step is to solve the characteristic equation. We must be familiar with the solution of a quadratic or a cubic equation. The contentious point is, therefore, to be able to find a_n from the roots of the characteristic equation. The following rules would help us to do so.

Solving the characteristic equation, we get roots $\alpha_1, \alpha_2, \alpha_3, \ldots$

- If $\alpha_1, \alpha_2, \alpha_3, \ldots$ are all distinct, then the solution is

$$a_n = c_1(\alpha_1)^n + c_2(\alpha_2)^n + c_3(\alpha_3)^n + \cdots$$

- If two roots α_1 and α_2 are same, the solution is of the form

$$a_n = (c_1 + nc_2)(\alpha_1)^n$$

- If the characteristic equation has 3 roots and all are equal, then

$$a_n = (c_1 + nc_2 + n^2 c_3)(\alpha_1)^n$$

The following illustrations would help us to understand the above concepts.

Illustration 3.4 Solve the following recurrence relation:

$$a_n = a_{n-1} + 6a_{n-2}$$

$$a_1 = 1, a_0 = 2$$

Solution For $a_n = a_{n-1} + 6a_{n-2}$, the characteristic equation would be

$$s^2 = s + 6s^0$$

By solving, we get
$$s^2 - s - 6s^0 = 0$$

or
$$s^2 - 3s + 2s - 6 = 0$$

or
$$(s-3)(s+2) = 0$$

Hence
$$s = 3, -2$$

So
$$a_n = c_1(\alpha_1)^n + c_2(\alpha_2)^n = c_1(-2)^n + c_2(3)^n \qquad (3.9)$$

Putting $n = 0$ in Eq. (3.9), we get

$$a_0 = c_1 + c_2 = 2$$

Putting $n = 1$ in Eq. (3.9), we get

$$a_1 = -2c_1 + 3c_2 = 1$$

i.e.
$$c_1 + c_2 = 2 \qquad (3.10)$$

or
$$-2c_1 + 3c_2 = 1 \qquad (3.11)$$

Solving the above two equations, we get $c_1 = c_2 = 1$

Putting the values in Eq. (3.9), we get

$$a_n = (-2)^n + (3)^n$$

Illustration 3.5 Solve $a_n = -6a_{n-1} - 12a_{n-2} - 8a_{n-3}$

Solution The order of a homogeneous equation is 3, so the characteristic equation is

$$s^3 = -6s^2 - 12s - 8$$

or
$$s^3 = +6s^2 + 12s + 8 = 0 \qquad (3.12)$$

$s = -2$ satisfies the equation, so $(s + 2)$ is a factor of Eq. (3.12).
Dividing Eq. (3.12) by $(s + 2)$, we get

$$(s + 2)(s^2 + 4s + 4) = 0$$

By solving, we get three identical roots $s = -2$

Therefore, the answer is

$$a_n = (c_1 + nc_2 + n^2 c_3)(-2)^n$$

where c_1, c_2, and c_3 are constants. Here, the values of a_0, a_1, etc., are not given, therefore, there is no way to find the values of the constants.

3.5 SOLVING NON-LINEAR RECURRENCE EQUATION

Section 3.4 explored the techniques of solving a linear recurrence equation. This section takes the concept forward and examines the solution of some special cases of a non-linear equation. Here, the linear part would be solved in the same way as we described earlier. However, the non-linear part requires substitution of a_n in the given equation (Table 3.1). The general form of the recurrence relation is $a_n = c_1 a_{n-1} + c_2 a_{n-2} + c_3 a_{n-3} \ldots = f(n)$. The value of $f(n)$ determines what is to be substituted in order to obtain the total solution.

In the case of a_n, if the characteristic equation also results in a_n, then we multiply the substitution by a_n till the solution of the characteristic equation and the substitution becomes different.

Table 3.1 Solving non-linear equation using recursion

$f(n)$	a_n
Cn	$c_1 n + c_2$
Cn^2	$c_1 n^2 + c_2 n + c_3$
a^n	$c_1 a^n$

Illustration 3.6 Solve $a_n - 5a_{n-1} + 6a_{n-2} = 3^n + n$

Solution The RHS of the given equation is $3^n + n$ and the characteristic equation is

$$s^2 - 5s + 6 = 0$$

By solving, we get $\qquad s = 3, 2$

So,
$$a_n = C_1 3^n + C_2 2^n$$

Since 3^n is common to both RHS and complementary function, we take the particular solution as

$$a_n = nC_1 + nC_2 3^n$$

Putting a_n in $a_n - 5a_{n-1} + 6a_{n-2} = 3^n + n$, we get

$$(nC_1 + C_2 + nC_3 3^n) - 5[(n-1)C_1 + C_2 + (n-1)C_3 3^{n-1}]$$
$$+ 6[(n-2)C_1 + C_2 + (n-2)C_3 3^{n-2}] = 3^n + n$$

So,
$$C_1 = 1/2$$
$$C_2 = 7/4$$
$$C_3 = 3$$

The particular solution, therefore, becomes

$$a_n = 1/2n + 7/4 + 3n3^n$$

Illustration 3.7 Solve $a_n = a_{n-1} + 2a_{n-2} + 2n^2$.

Solution The given equation can be written as

$$a_n - a_{n-1} - 2a_{n-2} = 2n^2 \qquad (3.13)$$

Since the RHS is $2n^2$, so let us take

$$a_n = C_1 n^2 + C_2 n + C_3$$

i.e.,
$$a_{n-1} = C_1 (n-1)^2 + C_2 (n-1) + C_3$$

and
$$a_{n-2} = C_1 (n-2)^2 + C_2 (n-2) + C_3$$

Putting in
$$a_n = a_{n-1} + 2a_{n-2} + 2n^2$$

$$(C_1 n^2 + C_2 n + C_3) = (C_1 (n-1)^2 + C_2 (n-1) + C_3) + 2(C_1 (n-2)^2 + C_2 (n-2) + C_3) + 2n^2$$

$$-2C_1 n^2 + n(10C_1 - 2C_2) + (-9C_1 + 5C_2 - 2C_3) = 2n^2$$

Comparing the coefficients, we get

$$-2C_1 = 2$$

or
$$C_1 = -1$$

or
$$10C_1 - 2C_2 = 0$$

$$C_2 = -5$$

$$-9C_1 + 5C_2 - 2C_3 = 0$$

i.e.,

$$C_3 = -8$$

Hence,

$$a_n = C_1 n^2 + C_2 n + C_3$$

The LHS of Eq. (3.13) is

$$a_n - a_{n-1} - 2a_{n-2}$$

The characteristic equation is $s^2 - s - 2 = 0$

or

$$s = \frac{1 \pm \sqrt{1+8}}{2}$$

$$= \frac{1 \pm 3}{2}$$

$$= 2, -1$$

i.e.,

$$a_n = C_1 2^n + C_2 (-1)^n$$

Combining the solution of the characteristic equation and the particular solution, we get

$$a_n = C_1 2^n + C_2 (-1)^n - n^2 - 5n - 8$$

3.6 GENERATING FUNCTIONS

The third method of solving a recurrence relation is using generating functions. To be able to solve a recurrence relation via a generating function, let us first of all learn to form a generating function of a recurrence relation.

An infinite series

$$a_0 + a_1 z + a_2 z^2 + a_3 z^3 + \cdots + a_n z^n$$

is called generating function of numeric function $(a_0, a_1, a_2, \ldots, a_n)$.

Case 1 If $(a_0 = a_1 = a_2, \ldots, = a_n = 1)$
Then, the generating function becomes

$$1 + z + z^2 + z^3 \cdots + z^n + \cdots$$

Please note that the above series is a GP and the sum $= \dfrac{a}{1-r} = \dfrac{1}{1-z}$

Hence, $A_{(z)} = \dfrac{1}{(1-z)}$

Case 2 For $2^0 + 2^1 z + 2^2 z^2 + 2^3 z^3 \cdots + 2^n z^n + \cdots$

$$A_{(z)} = \frac{1}{(1-2z)}$$

Generalization:

For $\qquad\qquad\qquad a^0 + a^1 z + a^2 z^2 + a^3 z^3 \cdots + a^n z^n + \cdots$

$$A_{(z)} = \frac{1}{(1-az)}$$

The following illustrations explore the concept and would help the reader to form a generating function for a recurrence relation.

Illustration 3.8 Find generating function for

$$a_n = 2.3^n + 4.5^n + 6.8^n$$

Solution Since for a^n, the generating function is

$$\frac{1}{(1-az)}$$

Therefore, for 3^n it becomes $\dfrac{1}{(1-3z)}$

$\qquad$ for 5^n it becomes $\dfrac{1}{(1-5z)}$, and

$\qquad$ for 8^n it becomes $\dfrac{1}{(1-8z)}$

The generating function for the given equation is

$$A_{(z)} = 2\frac{1}{(1-3z)} + 4\frac{1}{(1-5z)} + 6\frac{1}{(1-8z)}$$

Illustration 3.9 Find generating function for

$$a_n = 3^{n+4}$$

Solution

$$\begin{aligned} a_n &= 3^{n+4} \\ &= 3^n \cdot 3^4 \\ &= 3^n \cdot 81 \end{aligned}$$

The generating function for $\qquad\qquad 3^n = \dfrac{1}{1-3z}$

Therefore,

$$A_{(z)} = \frac{81}{1-3z}$$

Having seen the formation of a generating function for a recurrence relation, let us now see the method for finding the solution. In the following illustration, the value of $A(z)$ is given and the recurrence relation is to be solved.

Illustration 3.10 Find the numeric function corresponding to

$$A_{(z)} = \frac{3z}{(1-z)(1+2z)}$$

Solution First of all, the partial fraction for the given function is found

$$\frac{3z}{(1-z)(1+2z)} = \frac{A}{(1-z)} + \frac{B}{(1+2z)}$$

This is followed by the evaluation of the constants, in this case A and B.

$$A = \frac{3*1}{1+2*1} = \frac{3}{3} = 1$$

Therefore, $A = 1$
Similarly,

$$B = \frac{3*\left(-\dfrac{1}{2}\right)}{1-\left(-\dfrac{1}{2}\right)} = \frac{\dfrac{-3}{2}}{\dfrac{3}{2}} = -1$$

Table 3.2   Generating functions

Sequence	Generating function
1	$\dfrac{1}{(1-z)}$
a^n	$\dfrac{1}{(1-az)}$
$(n+1)$	$\dfrac{1}{(1-z)^2}$
N	$\dfrac{z}{(1-z)^2}$
$\dfrac{1}{n!}$	e^z
n^2	$\dfrac{z(1+z)}{(1-z^3)}$
a_n	$f(z)$
a_{n+1}	$\dfrac{f(z)-a_0}{(z)}$
a_{n+2}	$\dfrac{f(z)-a-a_z}{(z^2)}$

Hence, $B = -1$

$A(z)$ can therefore be written as $A_{(z)} = \dfrac{1}{(1-z)} - \dfrac{1}{(1+2z)}$.

The last step requires substituting the solution. Table 3.2 gives the values for a_n for various generating functions. On substituting the appropriate value, the value of a_n can be obtained,

$$a_n = (1)^n - (-2)^n$$

If $a = (a_0, a_1, a_2, \ldots, a_n)$ and $b = (b_0, b_1, b_2, \ldots, b_n)$ be two numeric functions then the corresponding generating functions are

$$A(z) = a_0 + a_1 z + a_2 z^2 \cdots$$

$$B(z) = b_z + b_1 z^1 + b_2 z^2 \cdots$$

The convolution $C = a*b$ is defined as

$$a_0 b_n + a_1 b_{n-1} + a_2 b_{n-2} \ldots + a_n b_0$$

Generating function of convolution C is the product of generating functions of the two sequences,

Hence, $$C(z) = A(z), B(z)$$

Now let us look at how to solve the recurrence relation using Table 3.2.

Illustration 3.11 Solve $a_n = 3a_{n-1} + 1, n \geq 2$

$$a_0 = 0 \quad a_1 = 1$$

Using generating equation.

Solution Putting $n = n + 1$ in the above equation, we get

$$a_{n+1} = 3\, a_n + 1$$

i.e. $$\frac{f(z) - a_0}{(z)} = 3f(z) + \frac{1}{(1-z)}$$

or $$f(z)\left(\frac{1}{z} - 3\right) = \frac{1}{(1-z)}$$

or $$f(z)\left(\frac{1-3z}{z}\right) = \frac{1}{(1-z)}$$

i.e. $$f(z) = \frac{z}{(1-z)(1-3z)}$$

Now by applying the partial fraction, we get

$$\frac{z}{(1-z)(1-3z)} = \frac{A}{(1-z)} + \frac{B}{(1-3z)}$$

$$A = \frac{1}{(1-3)} = -\frac{1}{2}$$

Similarly, B will be calculated as

$$B = \frac{1/3}{(1-1/3)} = \frac{1}{(2)}$$

$$f(z) = \frac{1/2}{(1-3z)} - \frac{1/2}{(1-z)}$$

Hence, $a_n = \dfrac{1}{2}3^n - \dfrac{1}{2}1^n = \dfrac{1}{2}3^n - \dfrac{1}{2}$

Illustration 3.12 Solve $a_n = a_{n-1} + a_{n-2}$ (i.e., find the explicit formulae for the Fibonacci sequence)

$$a_0 = 1, a_1 = 1$$

using generating equation.

Solution Since $a_n = a_{n-1} + a_{n-2}$

By putting $n = n + 2$, we get

$$a_{n+2} = a_{n+1} + a_n$$

$$\frac{f(z) - a_0 - a_1 z}{z^2} = \frac{f(z) - a_0}{z} + f(z)$$

$$f(z)(1/z^2 - 1/z - 1) = -\frac{1}{z} + \frac{1}{z^2} + \frac{1}{z}$$

$$f(z) = \frac{1 - z - z^2}{z^2}$$

Therefore,

$$f(z) = \frac{1}{1 - z - z^2}$$

By applying partial fraction, we get

$$\frac{\left(1/\sqrt{5}\right)\left(1/\sqrt{5}/2\right)}{1 - \left(1 + \sqrt{5}/2\right)z} - \frac{\left(-1/\sqrt{5}\right)\left(1/-\sqrt{5}/2\right)}{1 - \left(1 - \sqrt{5}/2\right)z}$$

So, the final answer is

$$\left(1/\sqrt{5}\right)\left(1 + \sqrt{5}/2\right)^{n+1} - \left(1/\sqrt{5}\right)\left(1 - \sqrt{5}/2\right)^{n+1} = O(\phi^n)$$

where ϕ is gold number.

3.7 CONCLUSION

The chapter explores the methods of solving a recurrence relation. The method of substitution, and that using generating functions have been examined in the chapter. These topics have also been exemplified. The topics will help the reader to analyse the backtracking algorithms and those using recursion effectively. The topics, though mathematical in nature, are essential for analysing algorithms also.

Points to Remember

- Recursion uses the principle of last in first out and hence requires a stack.
- In the Fibonacci series, the nth term can be found by taking the sum of the $(n - 1)$th and $(n - 2)$th term.
- Substitution, generating functions, and tree method are some of the methods to find the explicit formula for a recurrence equation.
- For the equation $a_n = c_1 a_{n-1}$, the characteristic equation would be $s = c_1 s^0$; for the equation $a_n = c_1 a_{n-1} + c_2 a_{n-2}$, the characteristic equation would be $s^2 = c_1 s^1 + c_2 s^0$; for the equation

$a_n = c_1 a_{n-1} + c_2 a_{n-2} + c_3 a_{n-3}$, the characteristic equation would be $s^3 = c_1 s^2 + c_2 s^1 + c_3 s^0$ and so on.

- If the roots of a characteristic equation are $\alpha_1, \alpha_2, \ldots$, then

  (a) If $\alpha_1, \alpha_2, \alpha_3, \ldots$ are all distinct, then the solution is

$$a_n = c_1 ((\alpha_1))^n + c_2 ((\alpha_2))^n + c_3 ((\alpha_3))^n + \cdots$$

  (b) If two roots α_1 and α_2 are same, the solution is of the form

$$a_n = (c_1 + nc_2)(\alpha_1)^n$$

  (c) If the characteristic equation has three roots and all are equal then the solution is

$$a_n = (c_1 + nc_2 + n^2 c_3)\alpha^n$$

KEY TERM

Recursion It means calling a function in itself. If a function invokes itself, then the phenomenon is referred to as recursion.

EXERCISES

I. Multiple Choice Questions

1. Which of the following is necessary to prevent stack overflow in recursion?
   - (a) An initial condition
   - (b) A recursion relation
   - (c) Range of variables
   - (d) None of the above

2. The following code evaluates the factorial of a number (in C language)

```
int fact(int x)
        {
        if(x==1)
                return 1;
        else
                    return (x*fac(x-1));
        }
```

   For which of the following the answer would not be correct
   - (a) 2
   - (b) 9
   - (c) 3
   - (d) 1

3. Which of the following cannot be solved by recursion?
   - (a) Fibonacci series
   - (b) Factorial of a number
   - (c) Power function
   - (d) All of the above can be solved by recursion

4. Which of the following strategies uses recursion extensively?
   - (a) Backtracking
   - (b) Greedy
   - (c) Both
   - (d) None of the above

5. Which of the following is not an example of a linear recursive equation?
   (a) $a_n = 7 \times a_{n-1}$, where $a_1 = 1$
   (b) $a_n = 2 \times a_{n-1} + 3 \times a_{n-2}$, $a_1 = 1$ and $a_2 = 1$
   (c) $a_n = 2 \times a_{n-1} + 3 \times a_{n-2}$, $a_1 = 2$
   (d) $a_n = n \times a_{n-1}$, where $a_1 = 1$

6. Which of the following is an example of a linear recursive equation?
   (a) $a_n = 3n \times a_{n-1}$, where $a_1 = 1$
   (b) $a_n = a_{n-1} + ((a)_{n-2})^7$, $a_1 = 1$ and $a_2 = 1$,
   (c) $a_n = 6^n \times a_{n-1} + 3 \times a_{n-2}$, $a_1 = 2$
   (d) $a_n = 7a_{n-1}$, where $a_1 = 1$

7. Which of the following methods can be used to solve a linear recursive equation?
   (a) Substitution
   (b) Generating function
   (c) Master theorem
   (d) All of the above

8. Which of the following algorithms does not generally use recursion?
   (a) Depth first search
   (b) Breadth first search
   (c) A*
   (d) Genetic algorithms

9. Which of the following is a general error while implementing recursion?
   (a) Stack overflow
   (b) Queue underflow
   (c) Underflow–overflow
   (d) None of the above

10. Which of the following can be used to solve the Tower of Hanoi problem?
    (a) Recursion
    (b) Divide and Conquer
    (c) Procedural
    (d) None of the above

II. Review Questions

1. What is recursion? What are the conditions for a recursive function to run?
2. Write a recursive function to implement the following:
   (a) Factorial of a number
   (b) Power function
   (c) Fibonacci series
   (d) Reversing a number

III. Numerical Problems:

1. Solve the following equations with the initial conditions:
   (a) $a_n = 2a_{n-1}$, $n \geq 1$, $a_0 = 3$
   (b) $a_n = 3a_{n-2}$, $n \geq 2$, $a_0 = 2$, $a_1 = 6$
   (c) $a_n = a_{n-1} + 3a_{n-2}$, $n \geq 2$, $a_0 = 3$, $a_1 = 6$
   (d) $a_n = a_{n-2}$, $n \geq 2$, $a_0 = 2$, $a_1 = -1$
   (e) $a_n = -6a_{n-1} - 9 a_{n-2}$, $n \geq 2$, $a_0 = 3$, $a_1 = -3$
   (f) $a_{n+2} = -4a_{n+1} + 5a_n$ $n \geq 2$ $a_0 = 2$, $a_1 = 8$
   (g) $a_n = 6a_{n-1} - 12a_{n-2} + 8a_{n-3}$ $a_0 = -5$, $a_1 = 4$, $a_2 = 88$

2. Find the general formula for solution of linear homogeneous recurrence relation. The characteristic equation has the roots: $-1, -1, -1, 2, 2, 5, 5, 7$.
3. If $a_n = 2a_{n-1} + 2^n$, prove that $a_n = n2^n$ using backtracking.
4. Calculate the previous problem using generating functions.
5. Find all the solutions of $a_n = 2a_{n-1} + 2n^2$.
6. In Numerical Problem 5, if $a_1 = 2$, find the solution.
7. Solve the simultaneous equations:

$a_n = 2a_{n-1} + 3b_{n-1}$

$b_n = a_{n-1} + 2b_{n-1}$

$a_0 = 0; b_0 = 2$

8. Suppose there are two rabbits, initially the number of rabbits double each month by natural reproduction and some rabbits are either added or removed each month. Construct a recurrence relation for the number of rabbits at the start of nth month given that during each month extra 10 rabbits are put on the island.
9. In Numerical Problem 8, find the number of rabbits at the start of nth month.

Answers to MCQs

1. (a)	3. (d)	5. (d)	7. (d)	9. (a)
2. (b)	4. (a)	6. (d)	8. (d)	10. (a)

CHAPTER 4

Analysis of Algorithms

OBJECTIVES

After studying this chapter, the reader will be able to

- Understand the method of finding the complexity of a recursive algorithm
- Appreciate the importance of the tree method in accomplishing the task
- Explain the basic proving techniques
- Understand Amortized Analysis
- Appreciate the importance of probabilistic analysis

4.1 INTRODUCTION

The second chapter introduced the concept of complexity, where the methods of finding the complexity of non-recursive algorithms were examined. The notations O, omega, and theta were also explained in Chapter 2. Though the chapter helps to find the complexity of simple algorithms, it would not be possible to find the complexity of recursive algorithms using those techniques. This chapter discusses the concept of complexity of recursive algorithms and presents various ways of finding it. The chapter briefly revisits the techniques of proving a premise. Section 4.4 discusses the method of contradiction and briefly explains the concept of proof by mathematical induction. The proving techniques would be helpful in the following chapters. The chapter also discusses Amortized Analysis and probabilistic analysis. There is another reason for inducting these topics in this book. The design of algorithms requires mathematical aptitude and at times, the concepts of discrete mathematics. If one wants to become a programmer, then it is imperative for him/her to understand and be able to design algorithms. In order to do so, one must have a basic, if not involved, knowledge of mathematics. However, for a basic course, this chapter may be skipped.

4.2 COMPLEXITY OF RECURSIVE ALGORITHMS

The following section explains the technique of finding the complexity of recursive algorithms. The methods to solve a recursive equation would help us to achieve the task.

The process requires the designing of a recursive equation relating the previous instances of the function with the present one and then solving the equation with substitution, tree method or Master method as the case may be.

Illustration 4.1 The factorial function can be related to its previous instance by the following function as the complexity of $T(n)$ depends on that of $T(n-1)$.

The factorial of a number n is the product of n and the factorial of the number preceding it,

$$T(n) = T(n-1) + T(1)$$

where $T(n)$ is the complexity of the algorithm having input 'n', and $T(n-1)$ is the complexity of the algorithm having input '$n-1$'.

Solution The above equation can easily be solved using the method of substitution.

The equation, in the next step, becomes $T(n) = (T(n-2) + T(1)) + T(1)$,

i.e., $$T(n) = T(n-2) + 2T(1)$$

On further substitution, we get

$$T(n) = T(n-(n-1)) + (n-1) \times T(1)$$
$$= n \times T(1) = O(n)$$

Listing 1

```
factorial(n)
       {
       if(n==1)
              {
              return 1;
              }
       else
              {
              return (n*factorial(n-1));
              }
       }
```

Complexity: $T(n) = O(n)$.

Illustration 4.2 Fibonacci series arises out of the rabbit problem, as discussed in Chapter 3 (Section 3.2). A Fibonacci term is the sum of the previous two Fibonacci terms (Listing 2). The complexity of a Fibonacci sequence can also be calculated using the method discussed in Illustration 4.1.

Solution The recursive equation of the Fibonacci sequence is as follows:

$$T(n) = T(n-1) + T(n-2), \; T(1) = 1 \text{ and } T(2) = 1$$

The solution of the above equation is

$$\left(\frac{1}{\sqrt{5}}\right)\left(1+\frac{\sqrt{5}}{2}\right)^{n+1} - \left(\frac{1}{\sqrt{5}}\right)\left(1-\frac{\sqrt{5}}{2}\right)^{n+1}$$

Listing 2

```
Fib(int n)
        {
        if(n==1)
                return 1;
        else if (n==2)
                return 1;
        else
                return (fib(n-1) + fib(n-2));
        }
```

Illustration 4.3 Tower of Hanoi problem has been discussed in Annexure. The number of moves in the nth iteration of the problem is one more than twice the number of moves in the previous instance.

Solution The recursive equation of the problem is

$$T(n) = 2 \times T(n-1) + 1, n > 1$$

$$T(n) = 1, n = 1$$

The characteristic equation of the above equation is $s - 2 = 0$. The solution of this equation is $s = 2$, i.e., $s = c(2)^n$. On putting the initial values, the value of c comes out to be 1.

For the particular solution, put $T(n) = c$ in the given equation, thus $c = 2c + 1$, so the value of c comes out to be -1. Therefore, the complete solution is $T(n) = 2^n - 1 = O(2^n)$.

4.3 FINDING COMPLEXITY BY TREE METHOD

The following discussion analyses the tree method of finding the complexity of a given algorithm. The tree method can be used in the case of recursive algorithms whose input size in various calls can be represented in a hierarchical structure, with the node representing the initial call and the sub-trees representing the trees formed in the subsequent calls.

The concept can be understood by taking the example of merge sort and the average case of quick sort. The algorithms will be discussed in Chapter 9. However, the point, as far as complexity is concerned, is that in both the cases, after each call, the number of items to be sorted gets reduced by half. The first call will split the array having n elements into two arrays having $n/2$ elements each. In the next step, there would be four arrays of $(n/4)$ elements. At the end of the divide step, there would be just one element (Fig. 4.1).

At the terminating condition, $\dfrac{n}{2^i} = 1$, that is, $i = \log_2 n$. The complexity of the two algorithms is, therefore, $O(\log(n))$. The tree method can also be used to solve the recursive algorithm having recursive equation $T(n) = T(n-1) + 1$. Such cases will be encountered in the worst case of quick sort, as discussed in Chapter 9.
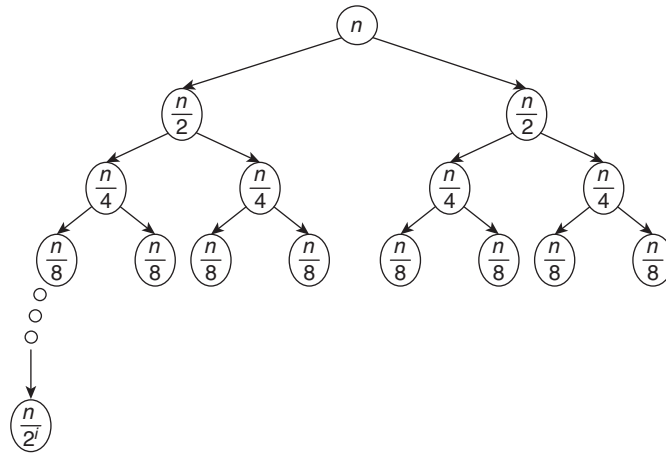
Figure 4.1 Complexity of merge sort

Tip: The complexity of the recursive equation of the type $(n) = aT\left(\dfrac{n}{b}\right) + f(n)$ can be found by Master method as discussed in Chapter 9.

4.4 PROVING TECHNIQUES

The discussion that follows explores some of the most basic methods of proving. These are proofs by contradiction and that by mathematical induction. The proof by contradiction can be used in the cases like proving a particular number is not rational. Mathematical induction is generally used in the cases wherein a mathematical equation or a premise is given. The concepts have been explained by taking easy examples. So even if the reader is not from a mathematical background, it would not be too difficult for him to understand the concepts.

4.4.1 Proof by Contradiction

Proof by contradiction requires starting from the opposite of what is to be proved. By proving the statement that you started with as incorrect and hence stating that since what we thought was incorrect, the intended statement was correct. In order to understand the concept, let us consider the following situation. Hari was fond of Indian classical music. He had a good ear for the complexities of distinct compositions and the depth of lyrics. A person called 'Funny Ting' came into picture and changed the rules of the game. He started singing songs from a book called 'nursery rhymes by those who never went to school, that too on music lifted from the Internet albums. Hari's annoyance was reflected from his blogs, and his Facebook posts. Almost everyone in the country seemed to love Mr Ting, probably, due to his previous legendary works (before coming to the mainstream music industry). One day, Mr Ting is found dead in his house. Investigators found that someone, mercilessly, forced him to listen to his own songs repeatedly. The investigators assumed that it could be Hari, for obvious reasons. The case goes to

court. In the court, Hari produces a CCTV footage showing Hari playing with his dogs. The court discharges Hari owing to the contradiction found in the investigators' theory. The court stated that since a person cannot be present at two places at the same time, Hari could not have committed the crime. The episode can be summarized in Fig. 4.2.

To prove	● Hari is innocent

Investigator	● He is guilty

Contradiction	● CCTV footage showing him playing with his dogs

Court	● He is innocent ● Investigators' claims are incorrect

Figure 4.2   Hari not involved in Funny Ting's murder

Now, let us consider a more concrete example. Here, we intend to prove that $\sqrt{3}$ is irrational. First, assume that it is rational. Now, since $\sqrt{3}$ is now rational (as per our assumption), $\sqrt{3} = \dfrac{p}{q}$, where p and q do not have a common factor.

This implies that $3 = \dfrac{p^2}{q^2}$, therefore, 3 is a common factor of p and q. This is contradictory to the statement from which we started. Therefore, $\sqrt{3}$ is an irrational number. Illustration 4.4 explores another example of the technique.

Illustration 4.4 Prove that $2 + \sqrt{3}$ is an irrational number.

Solution Let us assume that $2 + \sqrt{3}$ is a rational number.

$$2 + \sqrt{3} = \frac{p}{q}$$

This implies

$$\sqrt{3} = \left(\frac{p}{q}\right) - 2$$

That is,

$$\sqrt{3} = \left(\frac{p - 2q}{q}\right)$$

Now p and q are integers, therefore, $(p - 2q)/q$ is a rational number, but the left-hand side is an irrational number (proved in the previous illustration). Therefore, our supposition was incorrect and hence $2 + \sqrt{3}$ is an irrational number.

4.4.2 Proof by Mathematical Induction

Proof by mathematical induction requires three steps. In the first step, the given equation is verified by substituting the initial values of n. Often, the initial values are 1 and 2. Once this is done, we move on to the second step.

In the second step, we assume that the statement is true for $n = k$. Using this assumption, if we are able to prove that the given statement is also true for $n = (k + 1)$, then the statement is assumed to be true for every value of n. In order to understand the concept, let us explore some illustrations.

Illustration 4.5 Prove that $1^2 + 2^2 + \cdots + n^2 = \dfrac{n \times (n+1) \times (2 \times n+1)}{6}$ using mathematical induction.

Solution

Step 1 **Verification:** Put $n = 1$ on both sides of the given expression.

$$\text{LHS} = 1^2$$

$$\text{RHS} = \frac{1 \times (2) \times (3)}{6} = 1$$

On substituting $n = 2$ on both sides, we get

$$\text{LHS} = 1^2 + 2^2 = 5$$

$$\text{RHS} = \frac{2 \times (3) \times (5)}{6} = 5$$

Therefore, the given statement is true for $n = 1$ and 2. Let us now move to the second step.

Step 2 Let the statement be true for $n = k$.

That is, let
$$1^2 + 2^2 + \cdots + k^2 = \frac{k \times (k+1) \times (2 \times k+1)}{6}$$

Step 3 Substituting $n = k + 1$ on the LHS of the given equation, we get

$$1^2 + 2^2 + \cdots + k^2 + (k+1)^2$$

i.e.,
$$\frac{k \times (k+1) \times (2 \times k+1)^2}{6} + (k+1)^2$$

or
$$\frac{(k+1) \times (k+2) \times (2 \times k+3)}{6}$$

Substituting $n = (k + 1)$ on the RHS, we get $\dfrac{(k+1) \times (k+2) \times (2 \times k+3)}{6}$

Since on substituting $n = (k + 1)$ on both sides, we get the same thing; therefore, the statement is true for $n = (k + 1)$ when it is true for $n = k$. Hence, the statement is universally true.

Illustration 4.6 Prove that $1^3 + 2^3 + \cdots + n^3 = \dfrac{n^2 \times (n+1)^2}{4}$ using mathematical induction.

Solution

Step 1 **Verification:** Put $n = 1$ on both sides of the given expression.

$$\text{LHS} = 1^3$$

$$\text{RHS} = \frac{1 \times (4)}{4} = 1$$

On substituting $n = 2$ on both sides, we get

$$\text{LHS} = 1^3 + 2^3 = 9$$

$$\text{RHS} = \frac{4 \times (9)}{4} = 9$$

Therefore, the given statement is true for $n = 1$ and 2. Let us now move to the second step.

Step 2 Let the statement be true for $n = k$.

That is, let $$1^3 + 2^3 + \cdots + k^3 = \frac{k^2 \times (k+1)^2}{4}$$

Step 3 Substituting $n = k + 1$ on the LHS of the given equation, we get

$$1^3 + 2^3 + \cdots + k^3 + (k+1)^3$$

i.e.,

$$\frac{k^2 \times (k+1)^2}{4} + (k+1)^3$$

or,

$$\frac{(k+1)^2 \times (k+2)^2}{4}$$

Substituting $n = (k + 1)$ on the RHS, we get $\dfrac{(k+1)^2 \times (k+2)^2}{4}$.

Since on substituting $n = (k + 1)$ on both sides, we get the same thing; therefore, the statement is true for $n = (k + 1)$ when it is true for $n = k$. Hence, the statement is universally true.

Illustration 4.7 Prove that $5^{2n} - 1$ is divisible by 24, using mathematical induction.

Solution

Step 1 **Verification:** Put $n = 1$ on both sides of the given expression,

$$\text{LHS} = 5^2 - 1 = 24, \text{ which is divisible by 24}$$

On substituting $n = 2$ on both sides, we get

$$\text{LHS} = 5^4 - 1 = 624$$

This is also a multiple of 24.

Therefore, the given statement is true for $n = 1$ and 2. Let us now move to the second step.

Step 2 Let the statement be true for $n = k$.
That is, let $5^{2k} - 1$ is divisible by 24.

Step 3 Substituting $n = k + 1$ on the LHS of the given equation, we get

$$5^{2k + 1} - 1$$

i.e., $$25 \times 5^{2k} - 1$$

$$24 \times 5^{2k} + (5^{2k} - 1)$$

Now, the second part is a multiple of 24 (proved in the last step) and the first part is also divisible by 24. Therefore, $24 \times 5^{2k} + (5^{2k} - 1)$ is divisible by 24.

Therefore, the statement is true for $n = (k + 1)$ when it is true for $n = k$. Hence, the statement is universally true.

4.5 AMORTIZED ANALYSIS

The method described in this section helps us to analyse the performance of a data structure. It helps to establish the worst-case bounds of an algorithm. There are three ways discussed in this section, in which this analysis can be carried out. However, the choice depends on the situation. According to Cormen, Amortized Analysis may be defined as follows.

The concept of Amortized Analysis can be related to the concept of hash tables. Hash tables are effective tools for searching. The readers who are not familiar with the hash table can refer to Appendix A1 for a brief overview. However, those who have studied data structures must have read about it, and the problems associated with them.

The most contentious point in designing a hashing scheme is the size of the hash table. As observed by many researchers and authors, the size of the hash table should be neither too large nor too small. No matter how good the design is, it can always lead to 'Overflow.' In such cases, dynamic tables come to our rescue. In dynamic hash tables, as soon as overflow occurs, a new table is created with double the number of spaces as the old one, and the elements from the previous table are copied to the new one. Complexity

Definition An amortized analysis is any strategy for analysing a sequence of operations to show that the average cost per operation is small, even though a single operation within the sequence might be expensive [1].

analysis: on the face of it, the operation has a complexity of $O(n^2)$, in the worst case. Since a single insertion would have a complexity of $O(n)$ and there are n such operations, the worst-case complexity becomes $O(n^2)$. However, a more in-depth analysis of the insertions would yield an eccentric result.

Tarjen's Method

Consider, for example, the sequence of insertions depicted in Fig. 4.3. Here, if the value of $j - 1$ is an exact power of two, then the operation would have a cost of $O(j)$, and in the other case 1.
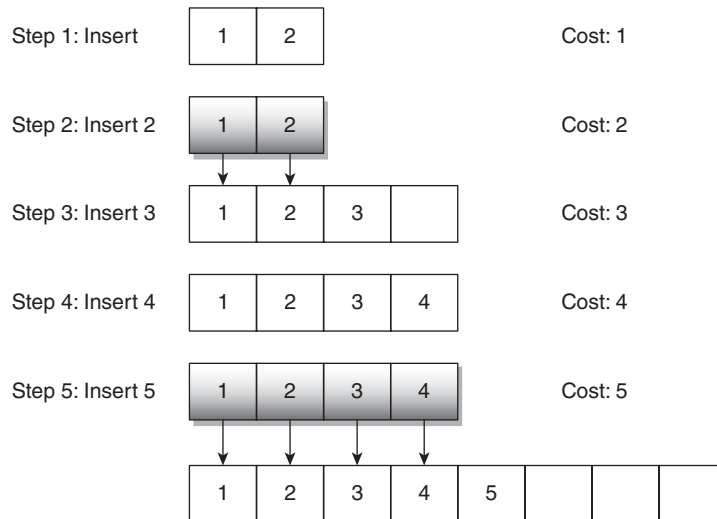
Figure 4.3   Amortized analysis

The aggregate cost comes out to be $(1 + 2 + 3 + 1 + 5) = 12$. Had the complexity been $O(n^2)$ the cost would have been proportional to 25. However, it comes out to be 12. Moreover, for the next three insertions, the cost would be 1, thus making the aggregate cost of 8 iterations as $1 + 2 + 3 + 1 + 5 + 1 + 1 + 1 = 15$ and not 64. It can be easily proved that the cost, in any case, would be $\leq 3 \times n$, thus making it $O(n)$. The above method was introduced by Tarjen.

Accounting Method

The second more intricate method of amortized analysis is the accounting method. In this method, a fictitious cost is assigned to each operation, or a fee is required to perform an operation (henceforth be denoted by C). The amount, which is not used, acts as a saving in a bank. The premise of the analysis is that the bank balance should never become negative. The following equation summarizes the concept:

$$\sum_{i=1}^{n} Ci \leq \sum_{i=1}^{n} C_i$$

Potential Method

The third method, referred to as potential method, is based on the concept of dynamic programming; in this, the difference in the costs of the previous two operations is proportional to the cost of the current operation. The cost is determined by a function called the *potential function*.

It has been observed by Rebecca Fiebrink, Princeton University, that both the potential method and the accounting method give the same result (or almost same result). However, as stated earlier, the choice depends on the situation. It may also be stated here that the above section intends to introduce the concept of Amortized Analysis. Appendix A1 deals with the potential method and the accounting method in detail. The readers who wish to comprehensively cover the topic may refer to Appendix A8.

4.6 PROBABILISTIC ANALYSIS

The analysis of an algorithm requires an in-depth knowledge of the mathematical concepts like probability. The concept of probability has been dealt within the appendix of this book. It is strongly recommended that those who are not familiar with the idea of probability should go through the appendix before starting this section. However, a basic knowledge of probability would help us to understand the probability analysis, for example, viva problem helps us to explain the probabilistic analysis. The problems discussed are just representative examples. It is expected that one would be able to apply problem reduction approach to apply the analysis in other problems as well.

In Chapter 1, it was mentioned that one of the ways of dealing with the algorithms is to analyse our resources and then decide which algorithm would suit our needs. The following discussion would help us to achieve that goal.

4.6.1 Viva Problem

Hari joins a university as an assistant professor. He is asked by one of his seniors to conduct an internal viva for computer graphics. He has a book of graphics, having 15 chapters and 25 questions at the end of each chapter. So, the total number of questions that he can ask is $25 \times 15 = 375$. In the class, there are 60 students. The problem is to find out the probability that two students would get the same question. This is important as students are asked questions in Hari's cabin. Outside his cabin, the rest of the students are waiting and as soon as a person comes out after giving viva, the students waiting for their turn ask him the questions that were asked, in the hope of getting the same question.

Now, since there are 375 questions, the probability of a question being asked is $\dfrac{1}{375}$. If the events (asking question to a student) are deemed independent, two students

would be asked the same question, with a probability of $\dfrac{1}{375} \times \dfrac{1}{375} = \dfrac{1}{140625}$, which is pretty low.

Had there been n questions in the book, the probability of asking same questions to two students would have been $\dfrac{1}{n} \times \dfrac{1}{n} = \dfrac{1}{n^2}$.

The probability that at least two out of k people will get the same question can be calculated as follows. Since it is easy to find out the probability of two questions being different, one minus this probability would give us the probability of two questions being same. If there are k students in a class, then the conditional probability that out of n questions $(n -(k - 1))$ have not been asked is $\dfrac{n-k+1}{n}$. For $k = 0, 1, \ldots, (k - 1)$, this becomes

$$1 \times \frac{n-k+1}{n} \times \frac{n-(k-1)+1}{n} \times \cdots \times \frac{n+1}{n}$$

Since, $1+k \leq e^k$, the above expression $\leq e^{-\frac{1}{n}} \times e^{-\frac{2}{n}} \times \cdots = e^{-k \times \frac{k-1}{2n}} \leq \dfrac{1}{2}$.

$k^2 - k \leq 2n \ln 2$, if n is 375, then the value of k comes out to be 23. Therefore, there must be at least 23 people in the class in order to make the probability of the same question being asked greater than 1/2.

4.6.2 Marriage Problem

The person in the previous problem conducts the exam successfully and is therefore offered an extension of a year by the university. He, therefore, decides to marry. He consults his 'Guru' who advises him not to marry a person having birthday on the same month as him. After reaching home, he gets himself registered on a matrimony site and the next day finds some suggestions. On the basis of the above discussion, he finds the minimum number of girls, whose details Hari must see in order to make the probability of her birthday being in a different month as that of Hari. In an unrelated development that 'Guru' is arrested in some fraud case and Hari is left to fend for himself.

4.6.3 Applications to Algorithms

The probabilistic analysis helps us to analyse a given algorithm 'X', with a set of inputs 'I'. The analysis helps us to find the complexity of the algorithm or to compare the algorithm with other algorithms or it helps us to find ways to improve the algorithm. Figure 4.4 depicts the concept of the analysis.
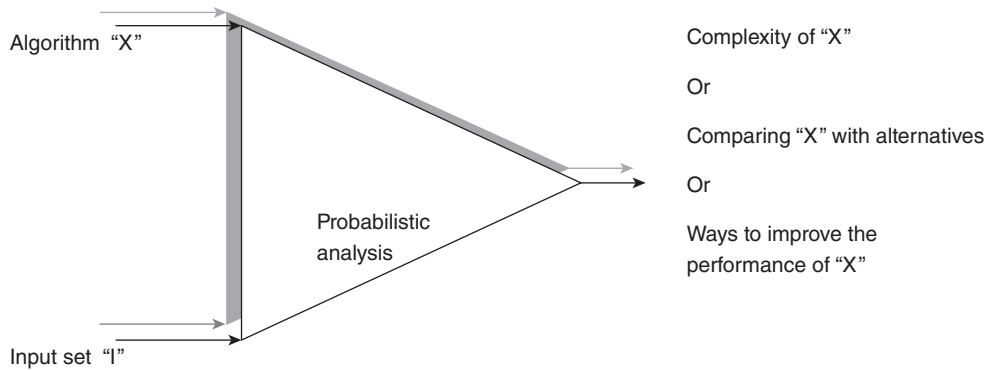
Figure 4.4 Probabilistic analysis of algorithms

4.7 TAIL RECURSION

Tail recursion is a special case of recursion. The tail call is, generally, the concluding call of a particular procedure. It can be implemented with no additional stack frame being added to the call stack. The crafting of such call is referred to as tail call elimination.

Those versed with functional programming should have an idea of the fact that the tail call elimination is assured by the language standards. As an example, let us consider the following code, which calculates the factorial of a number.

Listing 3

```
int fact(int number)
        {
int f=1;
                return (fact(number, f));
        }
int fact(int number, int iteration)
        {
        if(number==0)
                return(1)
        else
                {
                iteration++;
                return(fac((number –1), iteration)*number);
                }
        }
```

4.8 CONCLUSION

This chapter examined the application of core mathematical concepts. This chapter is a continuation of Chapter 2. The above theories would help us to find the complexities of recursive algorithms such as those in Chapters 2 and 3. The proving techniques would

be useful in proving various theorems in chapters related to NP-complete and NP-hard problems. If one wants to explore the concept, then he should follow any book on discrete mathematics. The importance of the concepts described in this chapter extends to many more topics.

Points to Remember

- The complexity of factorial of a number through recursion is $O(n)$.
- The complexity of finding the nth Fibonacci term through recursion is ϕ^n, where ϕ is the gold number.
- The complexity of binary search is $O(\log n)$.
- An argument can be proved through many methods; some of them are 'Proof by Contradiction' and 'Mathematical Induction'.
- Mathematical Induction has three steps: verification, assumption, and induction.
- In proof by contradiction, we assume the negation of the given argument to be true and then prove that our supposition was incorrect.
- The Amortized Analysis helps to establish the worst-case bounds of an algorithm.

KEY TERM

Amortized analysis It is any strategy for analysing a sequence of operations to show that the average cost per operation is small, even though a single operation within the sequence might be expensive.

EXERCISES

I. Multiple Choice Questions

1. Which of the following is a method to solve recursive equation?
   (a) Tree method
   (b) Substitution method
   (c) Master theorem
   (d) All of the above
2. A generating function is used in context of recursive functions to
   (a) Derive explicit formula from recursive formula
   (b) Derive recursive formula from explicit formula
   (c) Both
   (d) None of the above
3. Which of the following precisely represent the complexity of Tower of Hanoi?
   (a) $O(2^n)$ (c) $O(n)$
   (b) $O(2n)$ (d) None of the above

4. Which of the following precisely the complexity of Fibonacci series?
   (a) $O((1+\sqrt{(5)}/2)^n)+O((1-\sqrt{(5)}/2)^n)$
   (b) $O((1+\sqrt{(5)}/2)^n)-O((1-\sqrt{(5)}/2)^n)$
   (c) ϕ^n, where ϕ is gold number
   (d) None of the above

5. Which of the following is the complexity of the algorithm having equation $T(n) = T(n-1)+n$?
   (a) $O(n)$ (c) $O(n^3)$
   (b) $O(n^2)$ (d) None of the above

6. Which of the complexity of the algorithm having recursive equation $T(n)=T(n-1)+1$?
   (a) $O(n)$ (c) $O(n^3)$
   (b) $O(n^2)$ (d) None of the above

7. Which of the following can be used to solve the recursive equation of the form $T(n) = aT\left(\dfrac{n}{b}\right)+f(n)$?
   (a) Master theorem
   (b) Tree method
   (c) Both of the above
   (d) None of the above

8. Which of the following is not a proving technique?
   (a) Pumping lemma
   (b) Contradiction
   (c) Induction
   (d) Inception

9. Which of the following requires assuming the kth instance of the equation to be correct and proving the $(k+1)$th instance to be correct?
   (a) Induction
   (b) Conduction
   (c) Convection
   (d) Radiation

10. Which of the following is generally used to prove the statements of the type '$8+\sqrt{7}$' is an irrational number?
    (a) Induction   (c) Conduction
    (b) Contradiction   (d) Radiation

II. Review Questions

1. Discuss the concept of Amortized Analysis.
2. What is Amortized Analysis, how is it helpful in analysing the dynamic hash tables?
3. What are the different methods of Amortized Analysis?

4. Prove that in the aggregation method of Amortized Analysis, the cost is $\leq 3 \times n$, where n is the number of iterations.
5. Explain the concept of probabilistic analysis using appropriate examples.

III. Application-Based Questions

1. A set of numbers called L numbers are such that the first number of the sequence is 1, the second number 1, and the rest of the numbers are the sum of previous two numbers. Write an algorithm to generate the nth L number.
2. Write a recursive algorithm to find the sum of two numbers.
3. Write a recursive algorithm to calculate 'a' to the power of 'b', 'a' and 'b' entered by the user.
4. Write a recursive algorithm to reverse a string entered.
5. Prove that $3 + \sqrt{5}$ is an irrational number.

IV. Numerical Problems

1. Find the complexity of the algorithms having time complexity given as follows.
   (a) $T(n) = T(n-1) + T(n-2) + T(n-3)$
   (b) $T(n) = T(n-1) + 2 \times T(n-2)$
   (c) $T(n) = 3 \times T(n-1) + 1$
   (d) $T(n) = T(n-1) + 2$
   (e) $T(n) = 5T(n-1) + 6T(n-2)$
   (f) $T(n) = 5T(n-1) + 6$
   (g) $T(n) = T(n-2) + 3$
   (h) $T(n) = n \times T(n-2), T(1) = 1, T(2) = 1$
   (i) $T(n) = 3T\left(\dfrac{n}{2}\right) + 1$
   (j) $T(n) = T(n-1) + T(n-2) + T(n-3), T(1)$
   $= 1, T(2) = 1, T(3) = 1$

Problems Based on Probabilistic Analysis

2. In the viva problem given in Section 4.6.1, find the minimum number of students required to make the probability of two students getting the same question equal to 1/4?
3. In the viva problem given in Section 4.6.1, find the minimum number of students required to make the probability of two students getting the same question equal to 1/2, if the number of questions is increased to 575?
4. In the viva problem given in Section 4.6.1, find the minimum number of students required to make the probability of two students getting the same question equal to 1/2, if the number of questions is changed to 100?

5. In the marriage problem, find the number of bio-data that Hari must see so that the probability of the girl's birthday being different from him is greater than 1/2.

6. In the marriage problem, find the number of bio-data that Hari must see so that the probability of the girl's birthday being different from him is greater than 1/4.

7. How many persons must be there in a room so that the probability of two persons having birthday on the same day is greater than 1/3?

Answers to MCQs				
1. (d)	3. (a)	5. (b)	7. (a)	9. (a)
2. (c)	4. (c)	6. (a)	8. (d)	10. (b)

Annexure
The Tower of Hanoi

The Tower of Hanoi is a mathematical puzzle, which consists of three rods. Initially, there are n disks in the first rod such that the disk i is above the disk j, the radius of j is greater than that of i.

The objective is to transfer the disks into the last rod, such that at no point in time a larger disk is above a smaller disk. Moreover, only one disk can move at a time. So, if there are n disks 1, 2, 3,… (2 above 1 and 3 above 2) having radii r_1, r_2, …, r_n in the first rod, such that $r_1 > r_2 > r_3$ and so on, then at the end there should be n disks 1, 2, 3,… (2 above 1 and 3 above 2) having radii r_1, r_2, …, r_n, in the last rod, such that $r_1 > r_2 > r_3$ and so on.

The minimum number of moves to move n disks from the first peg to the last one is $2^n - 1$. The corresponding recurrence relation and its solution have been explained in Illustration 4.3.

Interestingly, the pegs have been used to guess the age of the world in the legends of many cultures including Hinduism and Buddhism. According to a legend, the priests of Kashi Vishwanath temple in India were asked by Brahma, the creator, to move 64 disks from first peg, of the three pegs, to the third peg. Even if it takes 1 s to move a disk to another peg, then it would take $2^{64} - 1$ s, or more than 580 billion years, to complete the task.

How to Solve the Problem?

There can be many solutions to the above problem. The simplest of which is the alternate move between the smallest disk and its previous disk.

The recursive solution would require placing of all, but the last, disks from the first to the second peg. This should be followed by placing the last disk in the third peg and then moving rest of them to the third one, in a way that the constraints are met. Figure depicts the steps involved in the transfer of three disks, from the first to the third rod.

Why is this Problem Important?

- The problem is an excellent example of recursion.
- The game is used by neuropsychologists to check if there is any defect in the frontal lobe of the brain.
- It is difficult for human beings to solve the problem but amazingly Argentine ant, *Linepithemahumile* can solve it using their pheromones.
- A version of the problem has also a connection with the shortest path problem, discussed in Chapter 10.
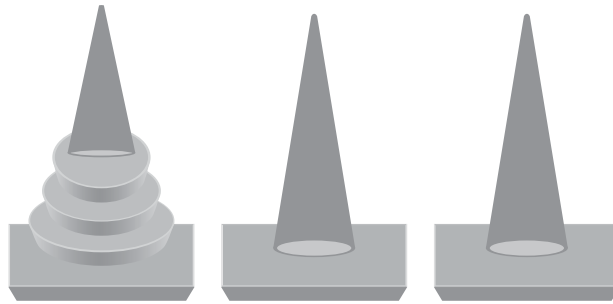
(a) Initially there are three disks in the first peg
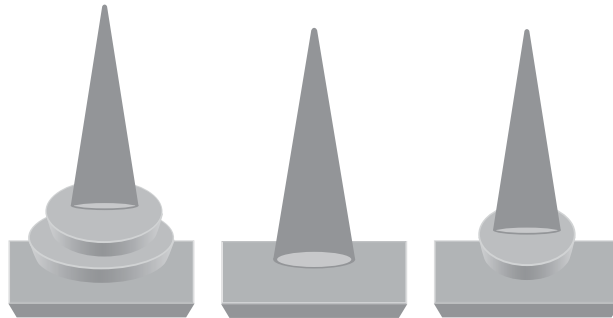
(b) The smallest disk is then moved to the third peg
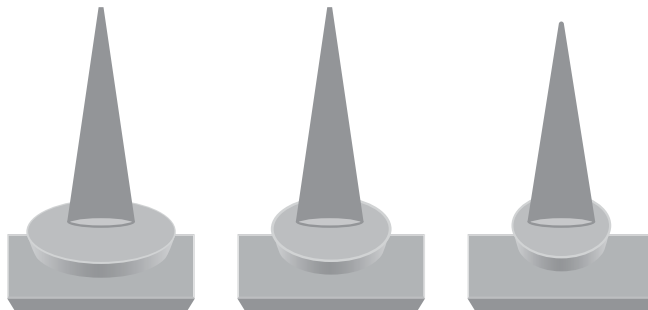
(c) The second disk is then moved to the second peg
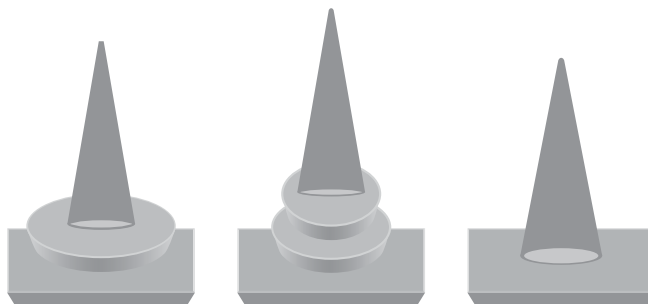
(d) The smallest disk is then stacked on the middle peg

Figure Solution of Tower of Hanoi, with three disks (*contd*)
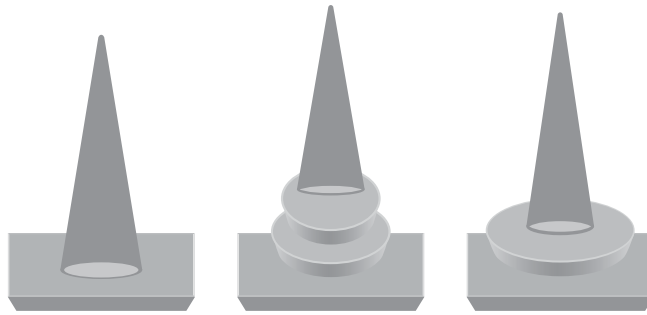
(e) The largest disk is moved to the last peg
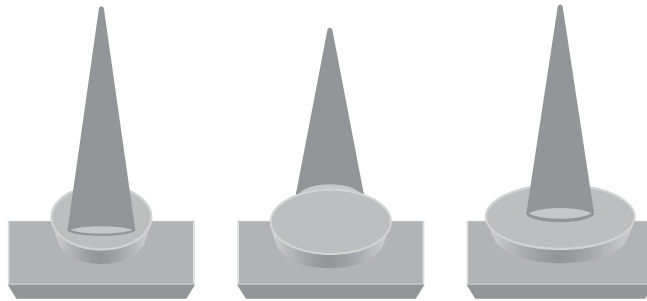
(f) The smallest disk is then stacked on the first peg
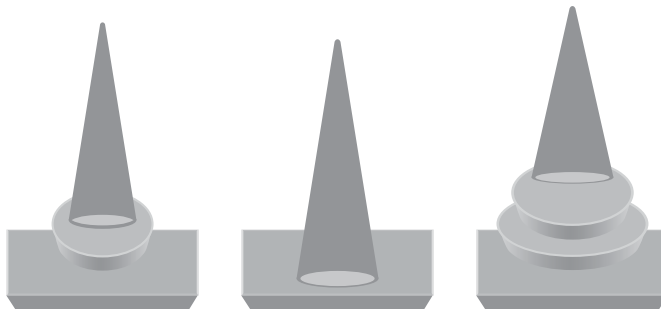
(g) The middle disk is then stacked on to the largest disk on the last peg
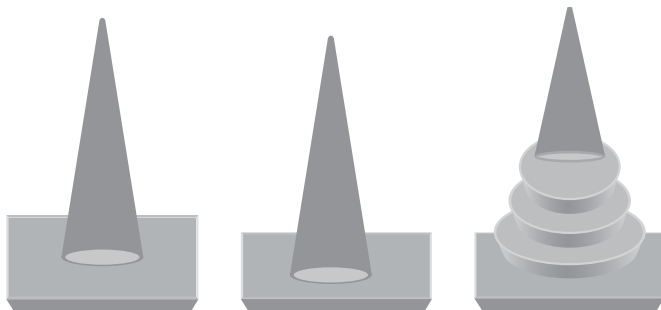
(h) The smallest disk is then moved to the last peg

Figure (Contd) Solution of Tower of Hanoi, with three disks

SECTION II

DATA STRUCTURES

Bad programmers worry about the code.
Good programmers worry about data structures and
their relationships.

— *Linus Torvalds*

Basic Data Structures

OBJECTIVES

After studying this chapter, the reader will be able to

- Understand the concept of data structures
- Appreciate the importance of abstract data structures
- Understand and use arrays and implement various operations such as search and traverse
- Understand the linked list and various operations on a singly linked list
- Differentiate between a doubly linked list and a circular linked list
- Understand the concept of stacks
- Implement stacks using arrays and linked list
- Understand the concept of queues
- Implement queues using arrays and linked list
- Understand the importance of data structures vis-à-vis algorithms

5.1 INTRODUCTION

For most of the readers, the primary aim of reading this book is to become an accomplished programmer or an accomplished algorithm designer. It is important to understand the concept of data structures in order to do that. It is essential to be able to develop algorithms, analyse them, and make them as efficient as possible. This efficiency can be attained in two ways, either by the techniques explained in the following sections or by using efficient data structures. This chapter concentrates on the latter. In order to make things happen efficiently, it is essential to organize the data so that its retrieval becomes easy. This branch of computer science deals with not just the organization of data and its retrieval, but also the processing alternatives. Finally, in order to become a good programmer, one must understand both data structures and algorithms.

Even the basic data types, provided by a language, such as int, float, and char, are considered as data structures. They are called *primal data structures*. The more intricate ones such as arrays, stacks, and linked lists described in the following sections are called

non-primal data structures. The latter can be linear as in the case of a stack or a queue or can be non-linear like a graph or a tree. Trees and graphs have been discussed in Chapters 6 and 7, respectively, of this book.

Each data structure comes with standard operations such as create, delete, update, and traverse. Let us now explore the tremendous world of data structures. In the journey, one must try to empower oneself with the immense powers of the data structures so that the war of algorithms can be fought and won.

5.2 ABSTRACT DATA TYPES

Abstract data types (ADTs) refer to the abstraction of certain class of types that have similar behaviour. Such data structures are defined by the operations that can be performed on the above said class. At times, the restrictions on such operation(s) are also stated along with the ADT. One of the classical examples of an ADT is stack. Stack is a linear data structure that follows the principle of last in first out. The behaviour of a stack may be defined by push, pop, underflow, and overflow operations. The push operation inserts an element into the stack at the position indicated by the value of TOP. The pop operation takes out an element from the stack and decrements the value of TOP. The overflow operation checks if the value of TOP increments MAX −1, where MAX is the maximum number of elements that can be stored in the stack. The underflow operation, on the other hand, checks if the pop operation is running when the value of TOP is −1, that is, there is no element in the array and still pop is being invoked. The stack ADT has been implemented using both arrays and linked list in the following sections. In the above discussion, the various operations that can be performed on a stack and the constraints on the operations have been discussed. The above discussion also brings forth the point that an ADT can be implemented using various data types.

Though there is no regular convention for defining them, the ADTs find their application even in the fascinating field of Artificial Intelligence wherein groups, rings, etc. are described using ADT. ADTs are also an important ingredient of Object Oriented Programming (OOP), wherein they are widely used in design by contract.

The above approach not only provides flexibility and facilitates change as needed but also gives a way to understand the concept of abstraction in OOP.

5.3 ARRAYS

An array is a linear data structure, wherein homogeneous elements are located at consecutive locations. An array can be of any standard data type; for example, an integer array cannot store a character and a string array would not store a user-defined structure. Hence, all arrays are said to be homogeneous. Moreover, if an integer type array starts from a location, say 2048, then the first element stored at 2048 would have its neighbour

stored at 2050 (assuming that an integer takes 2 bytes of memory). The ith element, in this case, would be stored at the location, which can be found by the following formula:

$$\text{Address of } i\text{th element} = \text{Base Address} + (i - 1) \times \text{size of (int)}$$

The following discussion explains some of the basic operations on arrays.

5.3.1 Linear Search

An element stored in an array (ITEM), having n elements, can be easily found by a simple traversal. This procedure would be henceforth referred to as linear search. Although the algorithm has also been discussed earlier, the following explanation will revisit the concept and will help the user to implement the procedure. The variable FLAG, in Algorithm 5.1, is initially 0, indicating that the value has not been found. If the element is found, then the value of FLAG becomes 1. At the end of the procedure, if FLAG remains 0, it means that the element has not been found. In case the element to be found occurs in the array more than once, the value of FLAG remains 1.

Algorithm 5.1   Algorithm for linear search (Array [] a, int n, int ITEM)

```
{
    FLAG = 0;
    int i = 0;
    while (i<n)
        {
        if (a[i] == ITEM)
            {
            FLAG = 1;
            print: "FOUND";
            }
            i++;
        }
    if (FLAG == 0)
        {
        Print: 'Not Found';
        }
}
```

Complexity: If the element to be found is present at the first position, then the complexity would be O(1). In the other extreme case, the element may not be present in the given array, or even be present at the last position. In such case, the complexity would be $O(n)$. What matters the most is the average complexity which, in the case of linear search, is $O(n)$.

There is another, more efficient, search technique referred to as binary search, which requires the use of Divide and Conquer. The algorithm has been discussed in Chapter 9.

5.3.2 Reversing the Order of Elements of a Given Array

In order to reverse the order of elements of an array, the following procedure is employed. A loop takes the ith element from the beginning and the ith element from the end (or the $(n - i - 1)$th element from the beginning) and swaps them.

Algorithm 5.2   Reorder (int[] a)

```
//temp is a temporary variable
//i is the counter
for(i=0; i<n; i++)
    {
    temp = a[i];
    a [i] = a [n-i-1];
    a [n-i-1] = temp;
    }
```

Complexity: Every statement inside the block runs $(n/2)$ times, so the complexity of the above algorithm becomes $O(n)$.

5.3.3 Sorting

The sorting of a given array $[a_1, a_2, a_3, \ldots,]$ produces an array $[b_1, b_2, b_3, \ldots,]$, such that $b_i > b_j$, if $i > j$ (or for that matter $b_i < b_j$, if $i > j$). The concept has been dealt with in Chapter 8 and has been carried forward in Chapter 9.

5.3.4 2D Array

A 2D array contains rows and columns. The base index of rows is 0 and so is that of the first column. There are two ways of storing the elements of a 2D array, the row major or the column major. In the row major technique, the first row elements are stored, followed by those in the second row and so on. A 3×2, 2D array has been depicted as follows:

$$a_{00} \quad a_{01} \quad a_{02}$$
$$a_{10} \quad a_{11} \quad a_{12}$$

A 2D array has many applications. One of the most important is matrix operations. A matrix is a 2D array. These operations are used in graphics and animations along with many other things. The basic operations like that of addition and subtraction are $O(n^2)$ algorithms. For example, a matrix A of order $m \times n$, when added to another matrix B of the same order, would require the execution of $O(n^2)$ instructions, if $m = n$, otherwise $O(mn)$ (refer to the code that follows). This is because there is a loop within a loop.

```
for (i=0; i<n; i++)
    {
    for(j=0; j<n; j++)
        {
        c[i][j]=0;
        for(k=0; k<n; k++)
            {
            c[i][j]+=a[i][k]*b[k][j];
            }
        }
    }
```

Complexity: The multiplication of two matrices requires three levels of nesting, thus making the complexity as $O(n^3)$. However, the complexity can be greatly reduced by a method explained in Chapter 9 of this book.

5.3.5 Sparse Matrix

A sparse matrix is a special type of matrix in which most of the elements are zero. In such cases, there is hardly any need to store each and every element of the matrix. Only the non-zero elements of the matrix can be stored along with their corresponding row and column numbers. For example, consider the following 5×5 sparse matrix. There are only four elements. The rest of the elements are zero and hence are shown in the following array:

$$\begin{pmatrix} 3 & \cdots & 2 \\ \vdots & \ddots & \vdots \\ 21 & \cdots & 11 \end{pmatrix}$$

The above elements can therefore be stored as follows. Here, the first column depicts the element, the second column depicts the row number, and the third depicts the column number of the element. This method requires only 12 memory locations, as against 25 required to store the whole array.

3	0	0
2	0	4
21	4	0
11	4	4

5.4 LINKED LIST

A linked list is a data structure whose basic unit is a node. Each node has two parts namely: data and link. The data part contains the value, whereas the link part has the address of the next node. This helps to connect various nodes of a list. The last node of the list has NULL in the link part, thus indicating that nothing is attached after the last node. In the discussion that follows the first node would be denoted by FIRST. In C, a linked list can be created using structures, the code of which is written as follows.

```
struct node
    {
    int data;
    struct node * LINK;
    };
```

In the above code, `node` is a structure having data part, which is of any standard data type and the link, which is a pointer to the node itself. Figure 5.1 depicts the node.
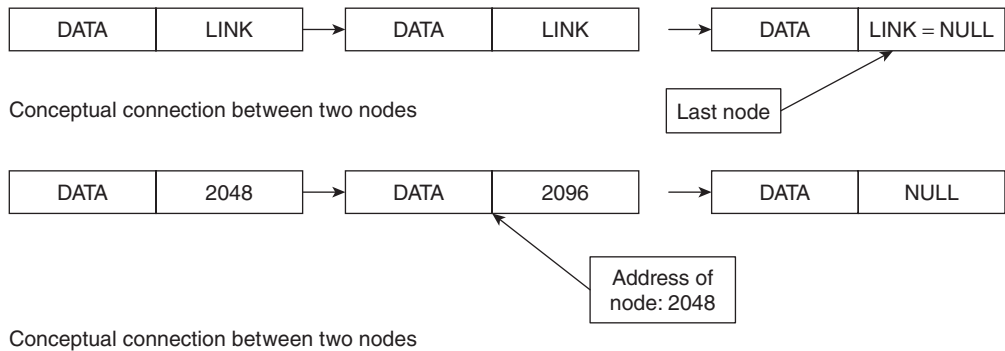
Figure 5.1   Nodes in a linked list

5.4.1 Advantages of a Linked List

Insertion and deletion in a linked list are easier as compared to an array. Moreover, a linked list does not suffer from the problem of limited placeholders as in the case with arrays. Linked lists are flexible, efficient, and provide more functionality as compared to an array. However, the linked list makes use of pointers, which make the implementation of a linked list a bit difficult. Moreover, the linked list algorithms are more involved as compared to that of arrays. However, the advantages of using a linked list are far more than its disadvantages.

5.4.2 Creation of a Linked List

In order to create a linked list, allocate memory to the node (in the case of C, it can be done via malloc()). This is followed by setting the value in the DATA part of the node. If one wants to insert a new node, then the address of the next node is set in the LINK field. If there is just a single node in the LIST, then the LINK of the first node becomes NULL (Fig. 5.2).
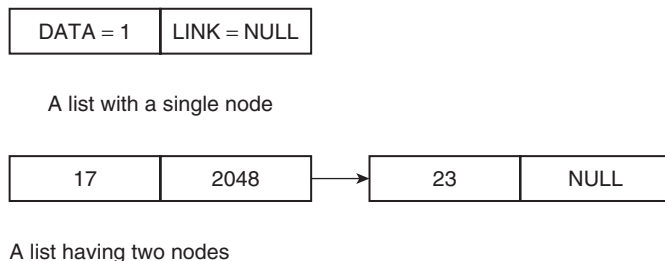
Figure 5.2   A list with a single node and with two nodes

5.4.3 Insertion at the Beginning

In order to insert a node at the beginning of a linked list, the following steps are required:
    INPUT: VALUE and LIST

Algorithm 5.3   Insert_beg()

```
{
//Create a node called temp.
     struct node * temp;
Allocate memory to temp.
//Now put the given value (VALUE) in the data part of temp.
    temp->DATA = VALUE;
//Set the LINK part of temp to FIRST.
     temp->LINK = FIRST
Rename temp to FIRST
}
```

Complexity: $O(1)$.

The process is depicted in Fig. 5.3. In this figure, a linked list having elements such as 11, 43, and 28 is taken. An element is to be inserted at the beginning having a value = 23.
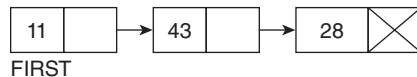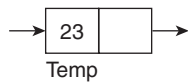
(a) Create a new node called temp, and set the value of DATA to the given value (23 in this case)
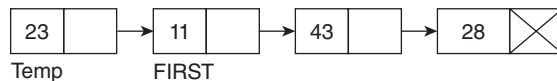
(b) Point the LINK of temp to FIRST
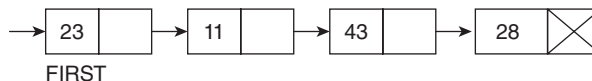
(c) Rename temp as FIRST

Figure 5.3   Insertion at the beginning of a linked list

5.4.4 Insertion at End

In order to insert an element at the end, first of all, a pointer, PTR, is set at the beginning. The list is traversed till the LINK of the present node becomes NULL.

A new node called TEMP is created, the DATA of which is set to the given value. The LINK of PTR is then set to TEMP. The LINK of temp is then set to NULL, indicating that it has become the last node. The process is depicted in Fig. 5.4.

(a) Create a new node called temp, and set the value DATA to the given value (23 in this case), set PTR to FIRST

(b) Traverse till PTR reaches the last node

(c) Set LINK of PTR to temp and LINK of temp to NULL
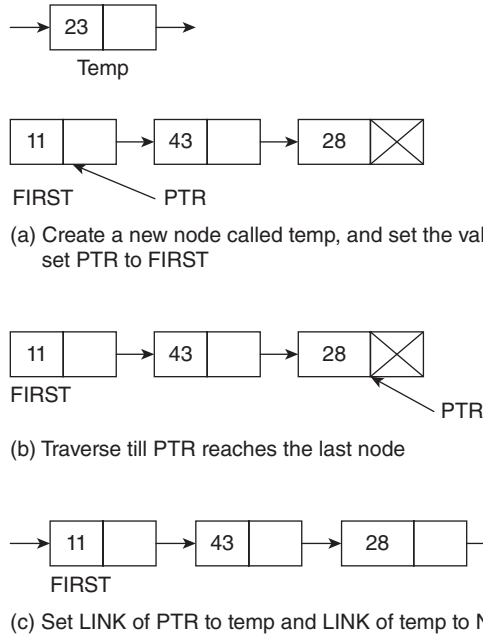
Figure 5.4 Adding node at the end

Algorithm 5.4 Insert_end (VALUE)

```
{
PTR = FIRST;
while (PTR -> LINK! = NULL)
    {
    PTR = PTR-> LINK;
    }
Create a new node called TEMP;
SET TEMP->DATA = VALUE;
SET PTR ->LINK = TEMP;
TEMP->LINK = NULL;
}
```

5.4.5 Inserting an Element in the Middle

In order to insert an element after the node having DATA = 'x', follow the below steps.

Set PTR to FIRST. Traverse till the DATA of PTR becomes 'x'. Create a new node called TEMP and set the DATA of Temp to the given value. Now set the LINK of Temp to the next of PTR, and the LINK of PTR to TEMP. The algorithm is given as follows. Figure 5.5 depicts the process.
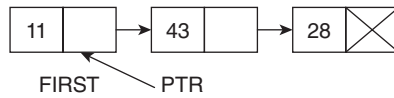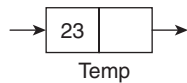
(a) Create a new node called Temp, and set the value of DATA to the given value (23 in this case), set PTR to FIRST. The value of 'x' is 43
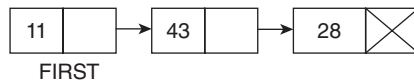
(b) When PTR reaches the node having value 43, stop
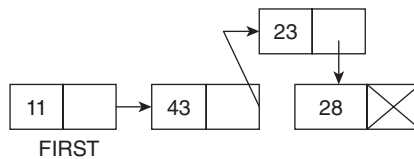
(c) Set LINK of PTR to Temp and LINK of Temp to PTR

Figure 5.5   Inserting node after a given value

Algorithm 5.5   Insert_middle (VALUE)

```
{
PTR = FIRST;
while (PTR -> DATA! = VALUE)
    {
    PTR = PTR-> LINK;
    }
Create a new node called TEMP;
SET TEMP -> DATA = VALUE;
SET TEMP -> LINK = PTR -> LINK;
SET PTR ->LINK = TEMP;
}
```

Complexity: Since this is a linear algorithm, so the complexity is $O(n)$.

5.4.6 Deleting a Node from the Beginning

In order to delete a node from the beginning, the DATA of FIRST is first stored in a backup. This is followed by renaming the FIRST → LINK (node to which the pointer of FIRST points) to FIRST.

Algorithm 5.6   Delete_beg()

```
{
int backup = FIRST->DATA
Rename (FIRST->LINK) as FIRST;
}
```

5.4.7 Deleting a Node from the End

In order to delete a node from the end, first of all we traverse till the last but one node. This can be done with the help of a pointer, PTR.

```
PTR = FIRST;
while ((PTR -> LINK)->LINK! = NULL)
    {
    PTR = PTR->LINK;
    }
```

After the above code has been executed, PTR reaches the last but one node.
Now save the value of PTR->LINK in the backup
`int backup = (PTR->LINK)->DATA;`
In the final step, the pointer of PTR is set to NULL.
`PTR->LINK = NULL;`
The complete algorithm is as follows.

Algorithm 5.7   Delete_end()

```
{
PTR = FIRST;
while ((PTR-> LINK)->LINK != NULL)
    {
    PTR = PTR->LINK;
    }
int backup = (PTR->LINK)->DATA;
PTR->LINK = NULL;
}
```

5.4.8 Deletion from a Particular Point

In order to accomplish the task, the pointer PTR is first set to FIRST. This is followed by traversal till the requisite value 'x' is found. Now, the LINK of PTR is set to the (PTR -> LINK -> LINK).

The algorithm is left as an exercise for the reader.

The web resources of this book contain the programs of all the aforementioned operations.

> **Tip:** A stack can be implemented using a linked list by combining two algorithms: insert_end() and delete_end() of a singly linked list. In this case, the element can be inserted only at the end and can be taken out from the end only.

> **Tip:** A queue can be implemented using a linked list by combining two algorithms: insert_end() and end_beg() of a singly linked list. In this case, the element can be inserted only at the end and can be deleted only from the beginning.

5.4.9 Doubly Linked List

The node of a linked list may also have two links namely: PREV and NEXT. Such a linked list is called a doubly linked list.

The insertion and deletion in such a linked list are a bit involved. However, the extra effort pays as this can be used to represent a number of useful data structures like a binary tree. The node of a doubly linked list is shown in Fig. 5.6.

PREV	DATA	NEXT

(a) A node of a doubly linked list

PREV	22	NEXT		PREV	31	NEXT

PREV	15	NEXT = NULL

(b) An example of a doubly linked list
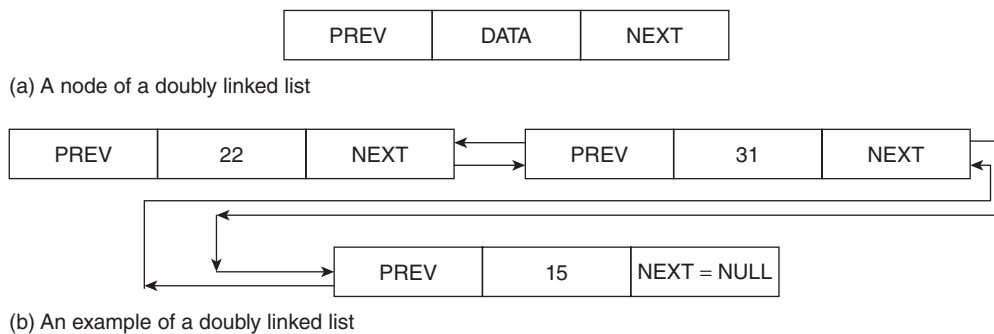
Figure 5.6   An example of a doubly linked list

The premise of insertion of a node at the beginning, middle, and end in a doubly linked list is fundamentally the same as that of a singly linked list. However, in the case of a doubly linked list, both the pointers, PREV and NEXT are to be taken care of. For example, in inserting a node at the beginning, a node called TEMP should be created. The value of the DATA part of TEMP should be set to VALUE (the given value). After this, the LINK of TEMP should be set to the FIRST of the given list. The PREV of the FIRST should be set to TEMP. Since TEMP has to become the FIRST of the list, the PREV of TEMP should be set to NULL. Finally, TEMP should be renamed as FIRST. The algorithm of the above process is given as follows.

Algorithm 5.8   Insert_beg_doubly

```
{
    Set PTR = FIRST;
    Create a new node called TEMP;
```

```
      Set TEMP->DATA = VALUE;
      Set TEMP->LINK = PTR;
      Set PTR->PREV = TEMP;
      TEMP-> PREV = NULL;
      FIRST = TEMP;
}
```

In inserting a node at the end, PTR has to traverse till the last node. A new node called TEMP is to be created whose DATA is set to VALUE.

Now, the LINK of PTR is set to TEMP, the PREV of TEMP to PTR, and the LINK of TEMP to NULL. The process is summarized in the following algorithm.

Algorithm 5.9   Insert_end_doubly

```
{
SET PTR = FIRST;
While(PTR -> LINK ! = NULL)
   {
   PTR = PTR->LINK;
   }
Create a new node called TEMP. TEMP-> DATA = VALUE;
Set PTR-> LINK = TEMP;
Set TEMP-> PREV = PTR;
Set TEMP-> LINK = NULL;
}
```

In inserting a node in the middle, PTR is made to traverse till 'x' is found. The LINK TEMP is then set LINK PTR. The PREV of the (PTR->LINK) becomes TEMP. The LINK of PTR is then set to TEMP. This is followed by setting the PREV of TEMP to PTR.

While deleting a node from the beginning, the FIRST is simply set to the (FIRST-> LINK) and the second node is renamed as FIRST.

Deletion from the end requires PTR to traverse till the last but one node and then make it LINK as NULL.

Deletion from the middle also follows the same approach as was done in a singly linked list, keeping in mind that PREV pointers are to be catered with.

5.4.10   Circular Linked List

A circular linked list is one in which the LINK of the last node points to the FIRST node. Here, the circular list can be implemented both by a singly linked list and by a doubly linked list. The algorithms pertaining to the circular linked list are left as an exercise for the reader. However, the web resources of this book contain programs pertaining to a doubly linked list.

5.5 STACK

When we pile up our books on our study table, we pick up the book which we had kept at the end, that is, the book which was at the last index is popped first, the initial index being that of the book which was kept initially. The same thing happens when we open the 'open file' dialog in MS Word and *via* the 'open file' dialog we open the 'browse' dialog. Till the browse dialog is not closed, we cannot close the 'open file' dialog, and until the 'open file' dialog is closed, we cannot close the Word application.

Such data structures that follow the principle of Last In First Out are referred to as a stack. The data structure can be implemented in a static fashion with the help of arrays or in a dynamic fashion with the help of a linked list.

> **Definition** Stack is a linear data structure, which follows the principle of last in first out.

The variable TOP keeps track of the last element of the stack. Initially, the value of TOP in the static implementation of stack is −1. As we add the elements, the value of the variable increases. However, this increase is possible only if the value of TOP is less than a maximum threshold, henceforth be referred as MAX. If an element is added onto a stack, wherein the value of TOP is (MAX −1), then a condition known as 'Overflow' occurs.

The deletion of the element, from a stack, is possible if the value of TOP is not −1. If the value of TOP is −1 and the pop operation is invoked, then a condition known as 'Underflow' occurs.

Terminology:

MAX:	Maximum number of elements in a stack
TOP:	Initially: −1
	Maximum value: MAX −1
push():	A method which increments the value of TOP and inserts value 'item' in the stack.
pop():	A method which pops value from the stack and decrements the value of TOP.
Underflow:	Condition wherein pop is invoked and the value of TOP is −1
Overflow:	Condition wherein push is invoked and the value of TOP is (MAX −1)
Starting index:	−1

5.5.1 Static Implementation of Stack

The static implementation of a stack is done with the help of an array. The initial value of the variable TOP is set to −1. The push (int item) function increments the

value of TOP and inserts the value onto the stack. Algorithm 5.10 depicts the push function.

push()

Algorithm 5.10   push(int item)

Input: A variable of the type int, array a[ ].
Output: none
Strategy: Increment the value of TOP and insert the item at the index denoted by the new value of TOP.

```
{
if(TOP == MAX-1)
    {
        Print "Overflow";
    }
else
    {
    TOP++;
    a[TOP]=item;
    }
}
```

---

pop()

The pop algorithm pops or takes out an element from the top of the stack. The algorithm basically decreases the value of the variable TOP. However, this is possible only if the value of TOP ! = −1.

Algorithm 5.11   pop( )

Input: none
Output: none
Strategy: If the value of TOP is not −1, then pop an item from the top of the stack and decrement the value of TOP by −1.

```
POP ()
if (TOP !=-1)
    {
        TOP--;
    }
else
    {
        print 'Underflow';
        }
}
```

5.5.2 Dynamic Implementation of Stack

The dynamic implementation of a stack requires a linked list. The push operation traverses to the end of the list and joins the new node at the end of the list. In order to do so, a pointer PTR, initially at the FIRST of the linked list, traverses till the NEXT of the node is NULL. At the end of this step, PTR would be at the last node of the list.

A new node TEMP is created. The DATA part of the new node would contain the value to be inserted and the NEXT of PTR would be set to NULL. The NEXT of PTR would then point to the new node TEMP. Algorithm 5.12 depicts the dynamic implementation of the push operation.

Algorithm 5.12   push(int item)

```
// A node has two parts DATA and NEXT.
//PTR is a pointer to a node.
//FIRST is the first node of the list
//TEMP is a temporary node.
{
PTR=FIRST;
while( PTR-> NEXT !=NULL)
    {
    PTR = PTR -> NEXT;
    }
//Create a new node called TEMP.
TEMP-> DATA = item;
TEMP->NEXT = NULL;
PTR->NEXT = TEMP;
}
```

The pop operation removes an element from the top of the stack in question. In order to carry out the operation, a pointer to node, PTR, is set to FIRST. The PTR traverses till the NEXT of PTR is NULL. That is, go to the second last node in order to remove the last node. It is like cutting a branch of a tree. In order to cut a branch, we will not sit on the branch, which we intend to cut but on the branch just before that. In order to remove an element from the linked list, we need not to physically separate it, just make the NEXT pointer of the previous node NULL. The following Algorithm 5.13 depicts the implementation of the pop operation.

Algorithm 5.13   pop ( )

Input: none
Output: the element at the TOP of the stack.

```
// A node has two parts DATA and NEXT.
//PTR is a pointer to a node.
//FIRST is the first node of the list
//TEMP is a temporary node.

{
    PTR=FIRST;
    while((PTR-> NEXT)->NEXT !=NULL)
    {
    PTR = PTR -> NEXT;
    }
PTR->NEXT = NULL;
}
```

5.5.3   Applications of Stack

One of the most important applications of stack is conversion of one type of expression into another. An expression can be written in three forms: infix, prefix, and postfix. In the infix form, the operator is written in between the two operands; in the prefix form, the operator is written before two operands, whereas in the postfix form, the operator is written after the two operands. For example, if two numbers stored in variables 'a' and 'b' are to be added and the result is to be stored in a variable called 'c', then the infix expression would be $c = a + b$, the postfix would be $c = ab +$, and the prefix would be $c = +ab$.

A more complex expression would make the things more clear. The second example is as follows:

$$c = a + b - c \times d + e$$

Figure 5.7 shows the evaluation of the expression.

Stacks help us to evaluate postfix expressions as explained in the next section.

Infix	$a + b - c \times d + e$
Evaluation of infix	$a + b - (c \times d) + e$
Evaluation of infix	$(a + b) - (c \times d) + e$
Evaluation of infix	$((a + b) - (c \times d)) + e$
Postfix	$((a \ b +) (c \ d \times) -) e +$ $\quad a \ b + c \ d \times - e +$
Prefix	$+ (- (+ a \ b) (\times c \ d)) e$ $\quad + - + a \ b \times c \ d \ e$
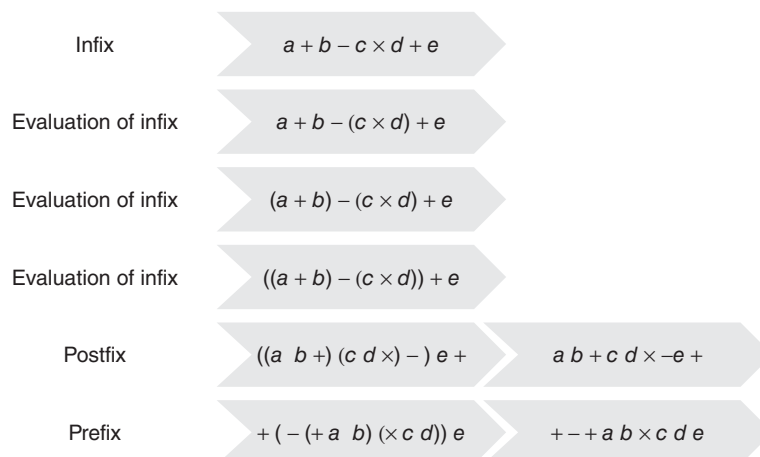
Figure 5.7   Evaluation of an expression

5.5.4 Evaluation of a Postfix Expression

Evaluation of a postfix expression refers to finding its value. Stacks can be used to evaluate a postfix expression. The procedure to accomplish the above task is pretty simple. The given expression is scanned from left to right. The symbols scanned are processed as follows. The operands are put into the stack. (The string representing the infix expression is initially set to NULL). When an operator 'op' is encountered, then two symbols 'x' and 'y' are popped from the stack. The expression 'y', 'op', and 'x' is evaluated and put into stack. At the end, whatever is left in the stack is the final answer. In order to understand the above procedure, let us consider the postfix expression in Fig. 5.8.
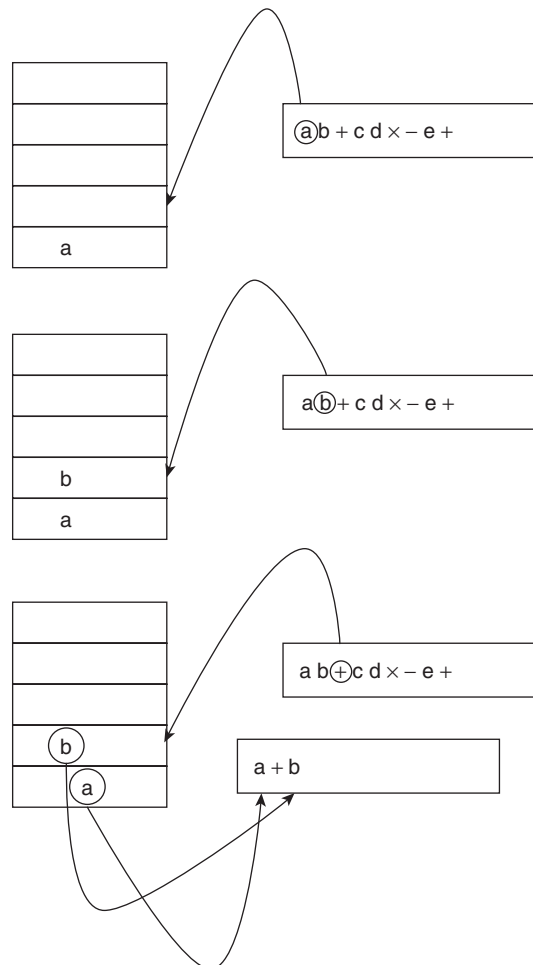
Figure 5.8 Evaluation of a postfix expression (*Contd*)
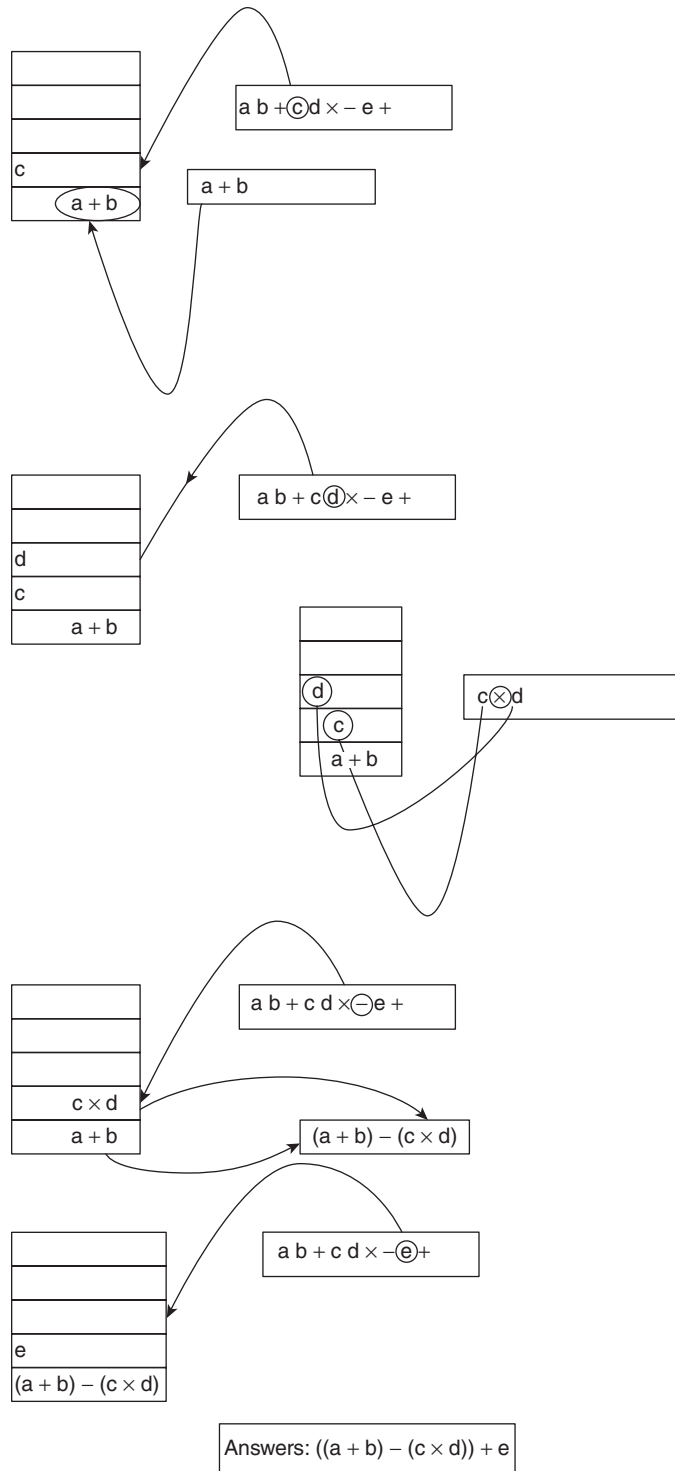
Figure 5.8 (Contd) Evaluation of a postfix expression

5.5.5 Infix to Postfix

Another application of stack is the conversion of an infix expression into postfix. The procedure for converting the expression is as follows.

Algorithm 5.14 InfixToPostfix (E) returns P

Input: Expression E, containing operators, operands, opening parentheses, and closing parentheses
Output: The expression P, which is the postfix of E.

```
{
P = φ;
Add ')' at the end of E and '('at the top of the stack;
for each  x ∈ P  do
    {
    if (x is an operand)
        {
        add it to P;
        }
    else if (x is '(')
        {
        push it to the stack;
        }
    else if (x is ')')
        {
        y=pop(S);
        while (y!='(')
            {
            add y to P;
            y=pop(S);
            }
    else if (x is an operator)
            {
        add it to the stack, S, if the operator on the top of the stack has a
        lower priority;
        otherwise pop the existing operator from stack and add the incoming
        operator at the top of the stack
            }
    }// end of for
return P;
}
```

Complexity: Each symbol is checked and processed. The complexity of the procedure discussed earlier is therefore $O(n)$.

Illustration 5.1 Convert the following expression to postfix:

$$((a + b) - c * d)$$

Solution

$$E: ((a + b) - c * d)$$
$$P = \varphi$$

Add ')' at the end of E and '(' at the top of the stack;
The processing of the given expression has been shown in the following table.

Symbol Scanned	Stack	P
(	((	NULL
(	(((	NULL
a	(((	a
+	(((+	a
b	(((+	ab
)	((	ab+
−	((−	ab+
c	((−	ab+c
*	((−*	ab+c
d	((−*	ab+cd
)	(	ab+cd*−
)	NULL	ab+cd*−

5.5.6 Infix to Prefix

Another application of stack is the conversion of an infix expression into the prefix. The procedure is the same as that of converting an infix expression to the postfix except for the fact that the given expression is first reversed and then the procedure is applied. Moreover, the expression obtained after applying the procedure is also reversed to get the final answer. The procedure for converting the expression is as follows.

Algorithm 5.15   InfixToPrefix (E) returns P

Input: Expression E containing operators, operands, opening parentheses, and closing parentheses.
Output: The expression P, which is the prefix of E.

```
{
P = φ;
E = Expression obtained by reversing the order of symbols in E;
Add ')' at the end of E and '(' at the top of the stack;
for each  x ∈ P  do
    {
    if (x is an operand)
        {
        add it to P;
        }
    else if (x is '(')
        {
        push it to the stack;
        }
    else if (x is ')')
        {
        y=pop(S);
        while (y!='(')
            {
            add y to P;
            y=pop(S);
            }
    else if (x is an operator)
        {
        add it to the stack, S, if the operator on the top of the stack has a
        lower priority;
        otherwise pop the existing operator from stack and add the incoming
        operator at the top of the stack
        }
    }
    }// end of for
P = expression obtained by reversing the order of symbols of P;
return P;
}
```

Complexity: Each symbol is checked and processed. The complexity of the above procedure is therefore $O(n)$.

Illustration 5.2 Convert the following expression to prefix:

$$(a+b)-c*d$$

Solution

E: $(a + b) - c * d$

E obtained by reversing the order of symbols: $d * c -)b + a($

$P = φ$;

Add ')' at the end of E and '(' at the top of the stack;

The processing of the given expression has been shown in the following table:

Symbol Scanned	Stack	P
None	(	NULL
d	(*	d
c	(*	dc
–	(–	dc*
)		dc*–
b		dc*–b
+	+	dc*–b
a	+	dc*–ba
(	+(	dc*–ba
)	+	dc*–ba
		dc*–ba+

P: *dc*–ba+*

P obtained by reversing the order of symbols: +ab–*cd

Answer: +ab–*cd

5.6 QUEUE

Queue is a linear data structure that follows the principle of First In First Out. A queue can be implemented using an array or a linked list. A queue implemented using an array is referred to as *a static* queue whereas those implemented using a linked list is called a *dynamic queue*. Figure 5.9 summarizes the classification.
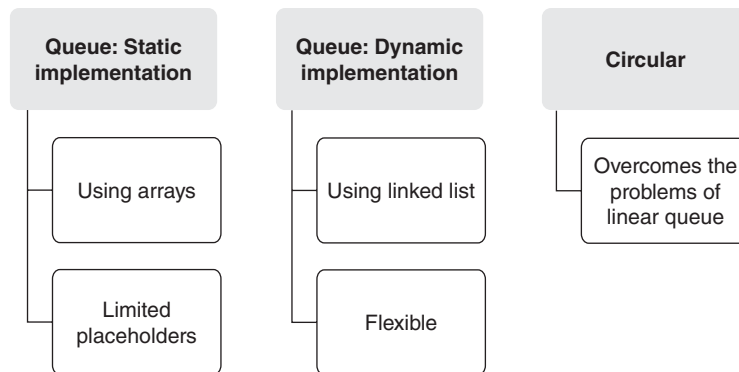
Figure 5.9 Queues and their types

5.6.1 Static Implementation

A queue implemented using an array has two indicators FRONT and REAR. In an empty queue, both the FRONT and REAR are −1. When the first element is inserted in the queue,

both FRONT and REAR become 0. On further insertion, the value of REAR increments and the new values are added to the new position. However, this is not possible if the value of REAR is MAX −1, where MAX is the maximum number of elements, an array can have. If one tries to enter an element in a queue whose REAR is MAX −1, then a condition referred to as Overflow is raised, indicating that no more elements can be added to the queue.

On deleting an element from a queue, the deletion must be from FRONT, hence the value of FRONT increments by 1. However, this is not possible if the queue is empty, that is FRONT=REAR= −1 (Underflow Condition).

The algorithms for the insertion and deletion from a queue are as follows.

Algorithm 5.16   Insert_Queue(int VALUE)

```
{
if (REAR = MAX-1)
    {
    print 'Overflow';
    }
else if(FRONT = REAR = -1)
    {
    FRONT = REAR = 0;
    Queue[REAR] = VALUE;
    }
else
{
REAR ++;
Queue[REAR ] = VALUE;
}
}
```

Algorithm 5.17   Delete_Queue()

```
{
if(FRONT = REAR = −1)
    {
    print: 'Underflow';
    }
else if (FRONT = REAR)
    {
    FRONT = REAR = −1;
    }
else
    {
    FRONT ++;
    }
}
```

Tip: The dynamic implementation of a queue can be done by insert_end() and delete_begin() algorithms of linked lists (refer to Section 5.4).

5.6.2 Problems with the Above Implementation

The problem with the above implementation is that the Overflow condition may be raised even when the queue has some empty cells. For example, consider the situation depicted in Fig. 5.10.
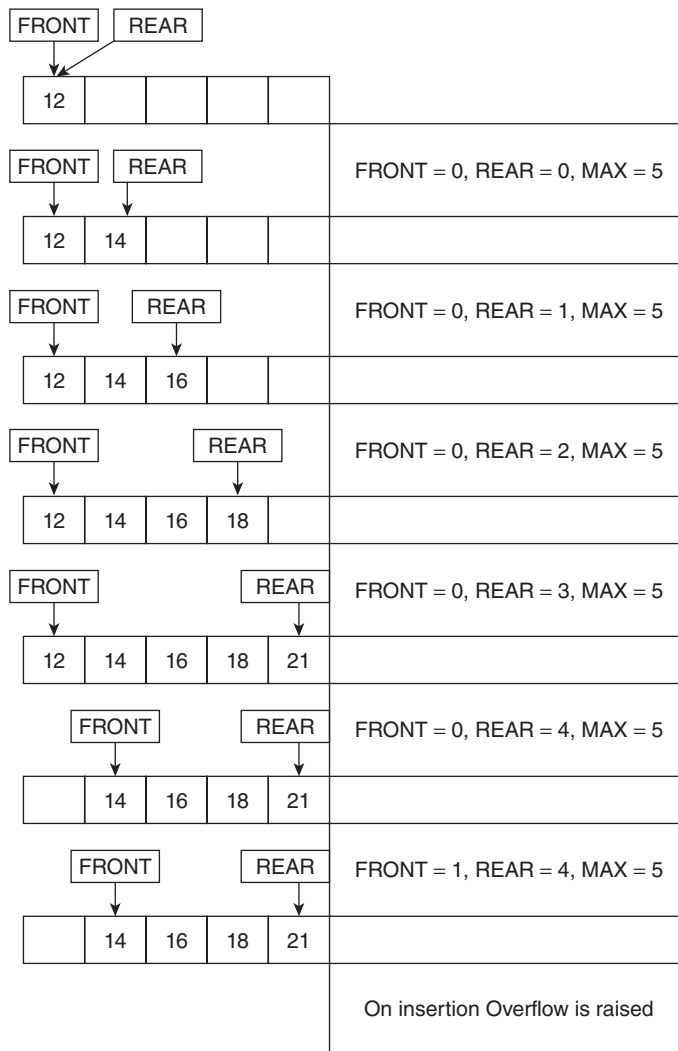
Figure 5.10 Example of insertion and deletion in a queue

In the figure, 12, 14, 16, 18, and 21 are added to a queue having MAX = 5. After which, 12 is deleted. The queue has empty place. However, on insertion, the Overflow condition is raised as the value of REAR is MAX − 1.

5.6.3 Circular Queue

The above problem can be handled by what is called a circular queue. In a circular queue, the REAR is connected to the 0th index of the array. This makes way for the element entering the array if the value of REAR is MAX − 1 and FRONT is not 0. However, the following points must be observed in the reference to a circular queue as against a linear queue.

In a circular queue, if there is just one element then FRONT = REAR. This condition is the same as that of a linear queue. In a circular queue, the algorithm for insertion increments the value of REAR but also takes its Mod with (MAX), so that if the value of REAR is MAX − 1, it becomes 0. The algorithm for insertion of an element in a circular queue is as follows.

Algorithm 5.18   Insert_Circular_Queue ()

```
{
  if (REAR = FRONT =-1)
        {
        REAR = FRONT = 0;
        Circular_Queue[REAR] = VALUE;
        }
  else if (FRONT = (REAR +1) % MAX)
        {
        Print 'Overflow';
        }
  else
        {
        REAR = (REAR + 1 )% MAX;
        Circular_Queue[REAR] = VALUE;
        }
}
```

The algorithm for deletion is given as follows.

Algorithm 5.19   Delete_Circular_Queue()

```
{
if (REAR = FRONT =-1)
```

```
    {
    Print 'Underflow';
    }
else if (REAR = FRONT)
    {
    REAR = FRONT = -1;
    }
else
    {
    FRONT = (FRONT + 1)% MAX;
    }
}
```

5.6.4 Applications of a Queue

The queue data structure finds applications in many fields. A few of them are listed here.

1. In a printer, the print commands are queued and hence the command, which was given first, is printed first followed by the next one. This is also referred to as *spooling*.
2. The operating system assigns the CPU to different processes. There can be many ways of doing this. One of the most common ways is a technique called Round Robin technique. In this technique, the CPU is allotted to different processes for a fixed time slot. For example, the slice (time) is two units and the four processes take six units, four units, and eight units, respectively. Then, the CPU executes the first process for the first two units, then it goes to the second process for two units, and then to the third process. The second cycle also proceeds like this. In the third cycle, the processor goes to the first, second, and then the third process. In the next cycle, only third process remains. The process is depicted in Fig. 5.11.
3. Queues are also used in various customer services.
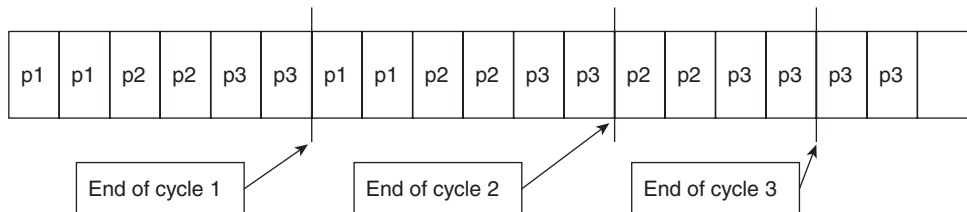
p1 is the first process, p2 is the second, and p3 is the third process. A block represents a unit of time.

Figure 5.11 Round Robin scheduling

5.7 CONCLUSION

The chapter introduced the concept of data structures, which is the soul of algorithm design. Without the knowledge of data structures, the study of algorithms is like an attempt to attain *Nirvana* without understanding the pains of living beings. The chapter also gave a short description of ADTs.

The chapter explored the types of stacks, their static and dynamic implementations, and their applications. It also introduced the concept of queues and provided an insight into their implementation. Finally, all the algorithms given in this chapter have been implemented in C. We can find their implementation in the web resources of this book. It is advised that before having a look at those implementations, the reader must at least try to implement them. The above concepts as well as the chapters that follow extensively use the concepts explained earlier.

Points to Remember

- Abstract data types (ADTs) refer to the abstraction of certain class of types that have similar behaviour.
- An array is a linear data structure, wherein homogeneous elements are located at consecutive locations.
- The complexity of linear search is $O(n)$.
- The complexity of addition, subtraction of two matrices of order $n \times n$ is $O(n^2)$.
- The complexity of conventional matrix multiplication is $O(n^3)$.
- Most of the elements in a sparse matrix are 0.
- The insertion and deletion in a linked list are easy as compared to an array.
- A stack is used in conversion of expressions (like from infix to postfix), in recursion, etc.
- Round Robin algorithm of CPU scheduling uses a queue.
- The static implementation of a stack is done using an array and the dynamic implementation is done using a linked list.
- Problem of having empty spaces and still showing 'overflow' in a linear queue can be solved by using a circular queue.
- If a stack has n elements then the space complexity is $O(n)$ and the time complexity of each operation is $O(1)$.
- The complexity of insertion or deletion at the beginning in the linked list is $O(1)$.
- The complexity of insertion at the end in a linked list is $O(n)$.
- The complexity of deletion from the end in a linked list is $O(n)$.
- The complexity of insertion or deletion in the middle of a linked list is $O(n)$.

KEY TERMS

Circular linked list A circular linked list is one in which the NEXT of the last node points to the FIRST node.

Circular queue In a circular queue, the REAR is connected to the 0th index of the array.

Doubly linked list Each node in a doubly linked list has two links namely: PREV and NEXT.

Linked list A linked list is a data structure whose basic unit is a node. Each node has two parts namely: DATA and LINK. The DATA part contains the value whereas the LINK part has the address of the next node.

Queue A queue is a linear data structure that follows the principle of First In First Out.

Stack It is a linear data structure that follows the principle of Last In First Out.

EXERCISES

I. Multiple Choice Questions

1. Which of the following follows the principle of First In First Out (FIFO)?
   - (a) Stack
   - (b) Queue
   - (c) Tree
   - (d) Graph

2. Which of the following follows the principle of Last In First Out (LIFO)?
   - (a) Stack
   - (b) Queue
   - (c) Tree
   - (d) Graph

3. Which of the following is a linear data structure?
   - (a) Stack
   - (b) Graph
   - (c) Tree
   - (d) None of the above

4. Which of the following is a non-linear data structure?
   - (a) Stack
   - (b) Queue
   - (c) Tree
   - (d) None of the above

5. Which of the following are facilitated by a data structure?
   - (a) Access of data
   - (b) Organization of data
   - (c) Manipulation of data
   - (d) All of the above

6. Which of the following must be defined by an abstract data structure?
   - (a) Operations on the data
   - (b) Constraints on operations
   - (c) Both
   - (d) None

7. Which of the following uses a queue?
   - (a) Round Robin
   - (b) Round Martin
   - (c) Martin Robin
   - (d) Robin Robin

8. Which of the following does not use a stack?
   - (a) Round Robin
   - (b) Conversion to postfix
   - (c) Evaluation of postfix
   - (d) Conversion to prefix

9. Which of the following is used in the dynamic implementation of a queue?
   (a) Stack
   (b) Linked List
   (c) Array
   (d) None of the above
10. Which of the following is the best in terms of flexibility?
   (a) Linked List
   (b) Both
   (c) Array
   (d) All of the above

II. Review Questions

1. What is an abstract data type?
2. What is meant by data structures? Classify them.
3. What are the applications of a queue?

III. Application-based Questions

1. Write an algorithm to insert an element in a queue. In addition, write the steps to delete an element.
2. Implement a queue using a linked list.
3. Write an algorithm to implement a stack using array.
4. Write an algorithm to implement a stack using a linked list.
5. Write an algorithm for
   (a) Inserting a node at the beginning of a singly linked list
   (b) Inserting a node at the end of a singly linked list
   (c) Inserting a node after a specific position in a singly linked list
   (d) Deleting a node at the beginning of a singly linked list
   (e) Deleting a node from the end of a singly linked list
   (f) Deleting a specific node from a singly linked list
6. Write an algorithm for
   (a) Inserting a node at the beginning of a doubly linked list
   (b) Inserting a node at the end of a doubly linked list
   (c) Inserting a node after a specific position in a doubly linked list
   (d) Deleting a node at the beginning of a doubly linked list
   (e) Deleting a node from the end of a doubly linked list
   (f) Deleting a specific node from a doubly linked list
7. Write an algorithm for inserting a node in a circular linked list.
8. Write an algorithm for deleting a node from a circular linked list.
9. Write an algorithm to reverse a linked list.
10. Write an algorithm to find the maximum element from a linked list.
11. Write an algorithm to find the minimum element from a linked list.
12. Write an algorithm to sort a linked list.
13. How do you find whether a given list has a cycle or a NULL terminated node?

14. Solve the above problem in $O(n)$ time. (HINT: Explore Floyd's Cycle finding algorithm).
15. Write an algorithm to find the nth node from the end in a linked list.

Answers to MCQs

1. (b)	3. (a)	5. (d)	7. (a)	9. (b)
2. (a)	4. (c)	6. (c)	8. (a)	10. (a)

CHAPTER 6

Trees

OBJECTIVES

After studying this chapter, the reader will be able to

- Understand the concept and applications of trees
- Understand binary trees and its types
- Store tree using an array and linked list
- Draw a tree when pre-order, post-order, and in-order traversals of trees are given
- Craft a tree when two traversals are given
- Understand binary search tree
- Explain B-tree
- Define and craft a heap
- Understand heapsort
- Define and use binomial and balanced trees
- Learn LL, LR, RR, and RL rotations

6.1 INTRODUCTION

Trees are perhaps one of the most useful data structures. They are used in parsing, in compilation, in game algorithms, and even in artificial intelligence algorithms. This chapter explores the fascinating world of trees and the basic operations therein. Utmost care has been taken to present the concepts in a simple way. The chapter begins with the definition of a tree, the representation of different types of trees, and then explains their traversals. Special types of trees like binary search trees and height balanced trees have also been dealt in the chapter. Finally, the chapter introduces the concept of heaps, insertion of an element in a heap, and heapsort.

Definition A tree is a non-linear data structure that has two components, namely nodes and edges. Nodes are joined by edges. This data structure has no cycle and no isolated vertex/edge.

6.2 BINARY TREES

Figures 6.1(a) and (b) are some examples of trees. Figure 6.1(c) depicts an example of a graph which is not a tree.
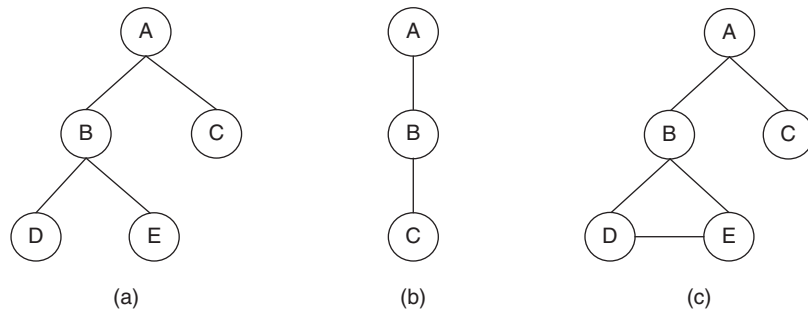
Figure 6.1 (a) and (b) Examples of graphs which are trees; (c) is not a tree

The most common tree that would be discussed in the chapter is a binary tree. A *binary tree* is one that has maximum of two children. There are many variants of a binary tree. A *strictly binary tree* is the one in which each node has either two children or no child. In a *complete binary tree*, each node has two children except the last level wherein the nodes do not have any child. Figure 6.2 shows examples of a complete binary and a strictly binary tree.

Before proceeding any further, it would be better to understand the terms used in the discussion that follows. Table 6.1 presents the basic terminology of trees.
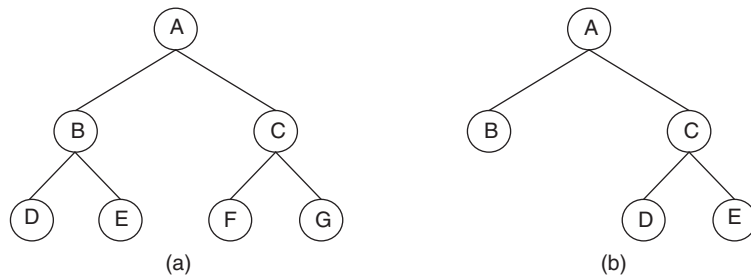
Figure 6.2 (a) Complete binary and (b) strictly binary trees

Table 6.1 Basic terminology

Component	Definition
Node	Basic unit of tree that contains data and may have children
Edge	The line that connects two nodes
Parent	Parent of a node is a node from which that node has been derived
Leaf node	A node that does not have any children
Root	The node of the tree of which all other nodes are children
Degree of a node	The number of children of a node is referred to as its degree
Degree of a tree	The maximum degree of any node in a tree
Level of a tree	The root is at level 0, its children at level 1, and so on
Depth	Maximum level of any node is referred to as depth of a tree
Siblings	Nodes having same parents

For example, in Fig. 6.2(a), A is the root of the tree. D, E, F, G are the leaves. Since D and E are the children of B, that is, they have same parent, therefore, D and E are siblings. The degree of A is 2, that of B is 2, C is 2, and of D, E, F, and G is 0. The maximum degree of any node in this tree is 2, therefore, the degree of the tree is 2. The root A is at level 0, B and C are at level 1, and the leaves are at level 2. The depth of the tree is 2.

6.3 REPRESENTATION OF TREES

A tree can be represented as an array or a linked list. The array representation of trees requires the root node to be stored at the 0th index of the array. The right child of the root is stored at the 2nd index and the left child is stored at the 1st index. In general, if a node is stored at the nth index of an array, then its right child would be stored at the $2 \times n + 2$nd index and the left child would be stored at the $2 \times n + 1$st index.

It may be noted, though, that such a representation can be used only in the case of a binary tree. Some examples of the above representation are depicted in Fig. 6.3. Moreover, such representation becomes inefficient when an unbalanced tree is stored as an array. Figure 6.3(c) is one such example.
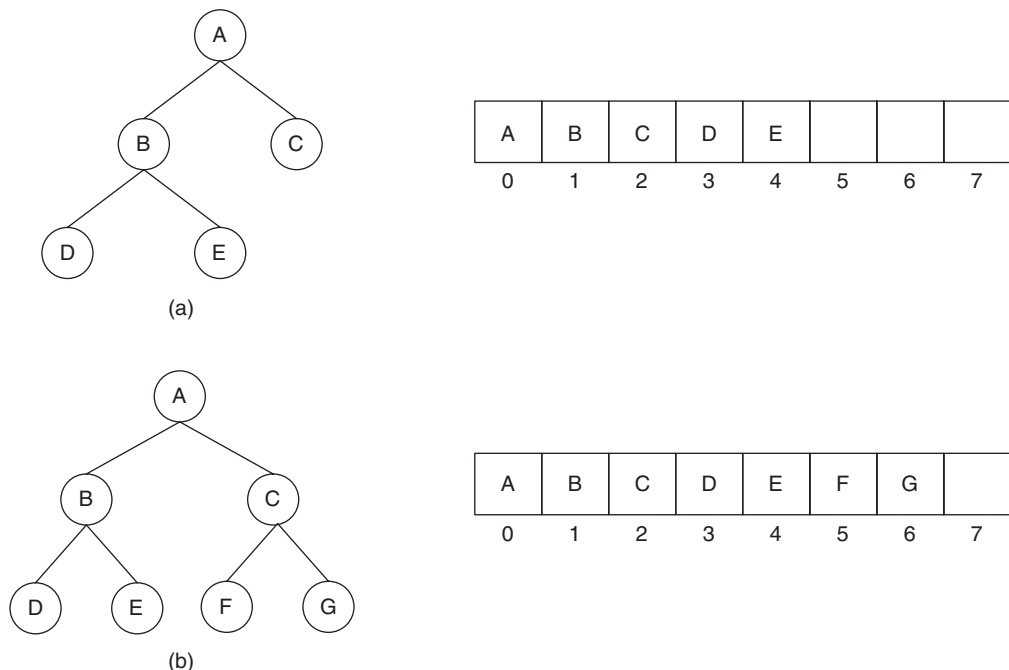
Figure 6.3 Representation of trees using arrays (*Contd*)
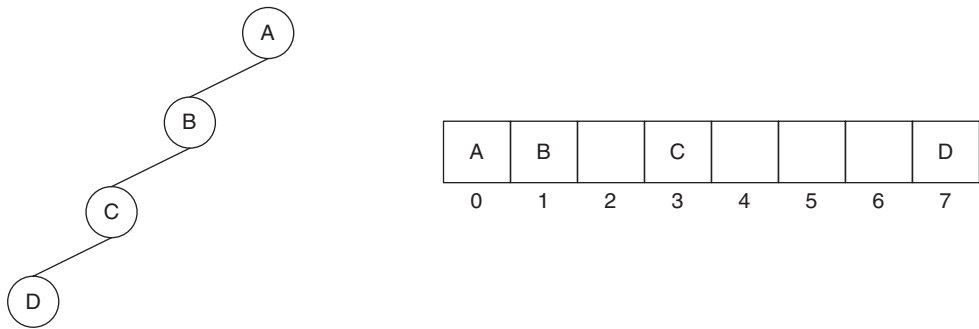
(c)

Figure 6.3 (Contd) Representation of trees using arrays

A more efficient way of storing a tree is using a linked list. A doubly linked list has two pointers, PREV and NEXT. If PREV is pointed to the left child and NEXT is pointed to the right child, then the trees shown in Fig. 6.4 would be represented as depicted in the figure. The linked list representation, though efficient, leads to the problem of dangling pointers. This problem is handled via a representation called *threaded trees*, wherein a NEXT or PREV, if not pointed to any node, points to the node determined by the pre-order traversal. The concept of pre-order traversal is explained in Section 6.5. The in-order representation of the tree shown in Fig. 6.4(b) is DBEAFCG. In the threaded tree, D and G point to the root, as there is nothing to the left of D and to the right of G. However, the NEXT of D points to B, as is evident from the in-order traversal and so

(a)

(b)
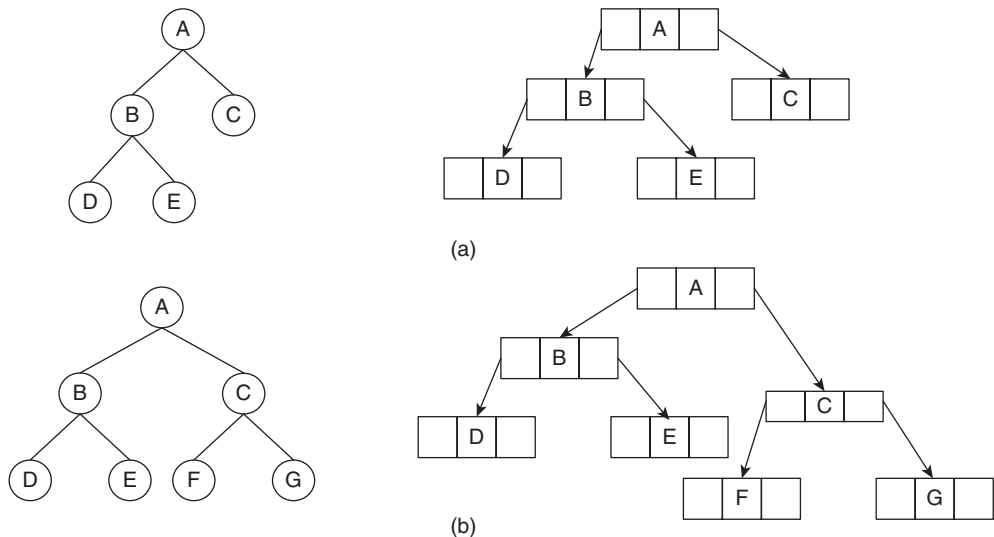
Figure 6.4 (a) and (b) Linked list representation of trees; (c) threaded representation of the tree shown in (b) (*Contd*)

(c)
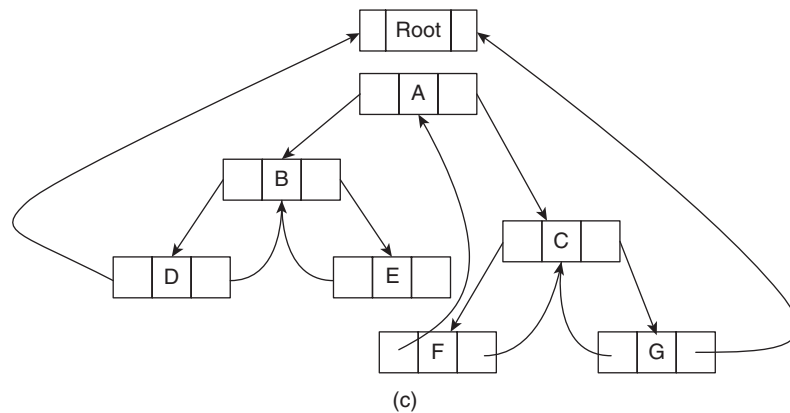
Figure 6.4 (Contd) (a) and (b) Linked list representation of trees; (c) threaded representation of the tree shown in (b)

does the PREV of E. The rest of the pointers have also been arranged as per this procedure. This representation reduces the number of un-pointed links and hence solves the problem of dangling pointers.

6.4 APPLICATIONS OF TREES

This section explores the various applications of trees. Trees are used in many sub-disciplines of Computer Science, from sorting data in a tape to organizing data in a database. The understanding of these applications would give you a reason to study trees and algorithms related to them. The various applications of trees are as follows.
- Organization of Data
  - o Sorting
  - o Searching
- To represent hierarchical relationship
- Compression of data using Huffman codes (refer to Illustration 6.2)
- To represent the parse structure of a sentence. Parsing, here, refers to a phase of a compiler in which it takes tokens as an input and gives parse trees as the output.
- To parse a sentence in natural language processing. Natural language processing is a part of artificial intelligence wherein text in natural language is converted into that which can be understood by a computer.
- They are also used for sorting.
- Trees are often used to specify hierarchical relationships. In such a tree if 'b' dominates 'a' then 'b' generally comes above 'a'.

Figure 6.5 represents the hierarchical structure of an organization where Academics is governed by VP (academic) and Administration by VP (administration) and both work under the President.
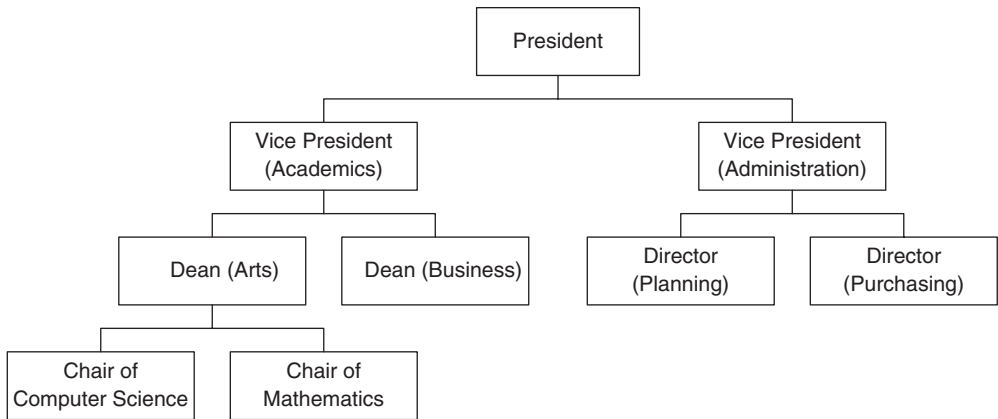
Figure 6.5 Representation of hierarchical structures using trees

If in a single elimination tournament Sachin plays with Tachin (Group 1). In the second group, Nitin plays with Nikhil; whoever wins in Group 1 plays with whoever wins in Group 2. The final round throws a winner who wins the tournament. The tree for the above situation is shown in Fig. 6.6.
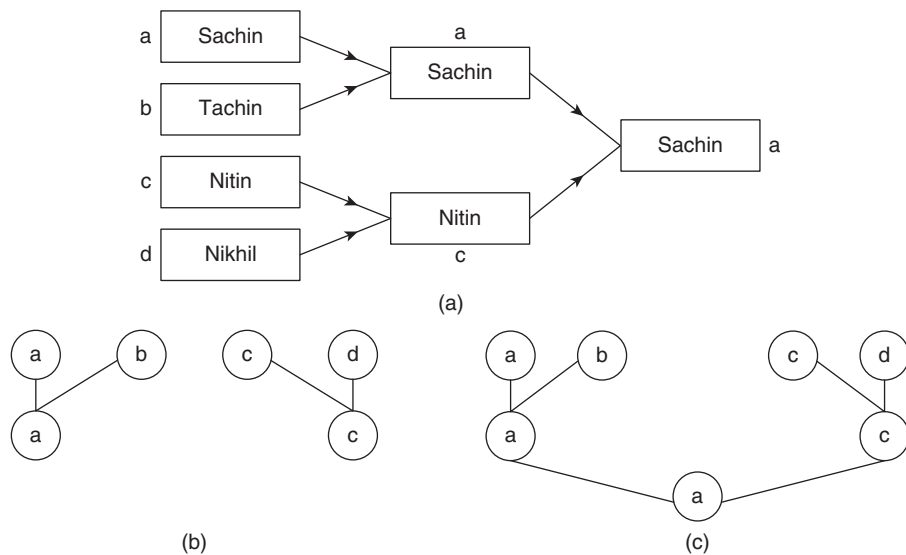
Figure 6.6 (a) Tournament selection level 1; (b) Tournament selection (c) Tournament selection

Illustration 6.2 Another important application of the concept of trees can be Huffman code. The code has input data represented as characters of variable length bit string. It is an efficient alternative to the ASCII code used generally.

Example: The Huffman tree is shown in Fig. 6.7.
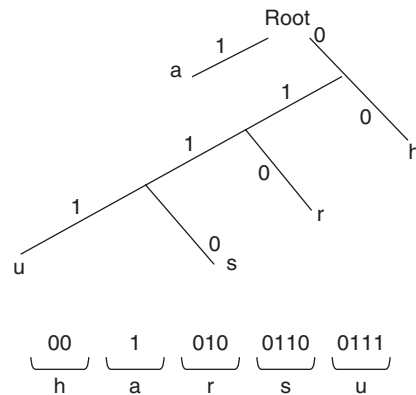
Figure 6.7 Huffman tree

In the above tree, a is represented by 1

    h by 00

    r by 010

    s by 0110

    u by 0111

Huffman gave an algorithm to create code that sees the frequency of occurrence of characters to find the code which occupies minimum space. This code is used in digital communication.

Observation

Let T be a graph with n vertices. Then, the following are equivalent:

(a) T is a tree

(b) T is connected and acyclic

(c) T is connected and has $(n - 1)$ edges

(d) T is acyclic and has $(n - 1)$ edges

Acyclic graph: A graph with no cycle is called an acyclic graph.

Illustration 6.3 The 4-Queens problem: It is required to place 4-Queens on a 4×4 chessboard, so that no 2-Queens are in the same row/same column or the same diagonal. The problem can be solved by backtracking. However, a brief overview of the concept has been provided here.

Solution Figure 6.8, 1 represents a queen. We start by placing the queen at (1,1). The next queen is to be placed in the next row. However, it should not be placed in the same column as the first row or in the same diagonal. We keep on placing the queens in this fashion. However, if we are stuck in a situation wherein we cannot place any more queens, then we backtrack to the last feasible option and try other options. The problem has been informally dealt within this section. However, it has been formally dealt within Chapter 12.

Figure 6.8 (tree of 4×4 matrices):

Root:
```
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
```

Left child:
```
1 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
```

Right child:
```
0 0 0 0
1 0 0 0
0 0 0 0
0 0 0 0
```

Left-left:
```
1 0 0 0
0 0 0 0
0 1 0 0
0 0 0 0
```

Left-right:
```
1 0 0 0
0 0 0 0
0 0 0 0
0 1 0 0
```

Under right child:
```
0 0 0 0
1 0 0 0
0 0 0 0
0 1 0 0
```

Under Left-right:
```
1 0 0 0
0 0 1 0
0 0 0 0
0 1 0 0
```

Next (right branch):
```
0 0 1 0
1 0 0 0
0 0 0 0
0 1 0 0
```

Bottom (right branch):
```
0 0 1 0
1 0 0 0
0 0 0 1
0 1 0 0
```
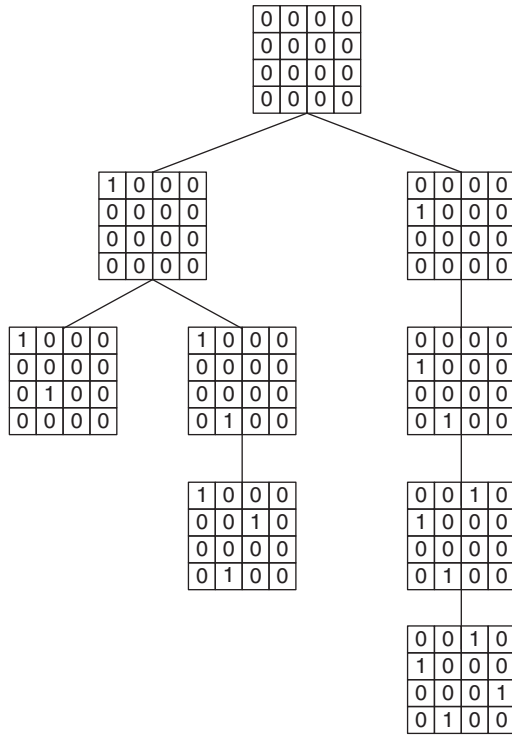
Figure 6.8 A possible solution of the 4-Queens problem

Illustration 6.4 Find the number of nodes in a complete binary tree.

Solution In a complete binary tree, there is a single node at the first level, two nodes at the second level, four at the third, and so on. The situation is depicted in Fig. 6.9.

First level = 2^0 nodes
Second level = 2^1 nodes
Third level = 2^2 nodes

$(n-1)$ levels ⟶ (2^{n-1}) nodes
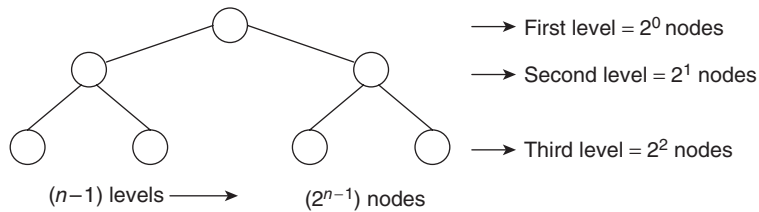
Figure 6.9 Number of nodes in a binary tree

Therefore, the total number of nodes $= 2^0 + 2^1 + 2^2 + \cdots + 2^{n-1} = 1 + 2 + 2^2 + \cdots + 2^{n-1}$. Since, the sum of n terms of a GP is

$$S = a(r^n - 1)/(r - 1)$$

where a is the first term and r is the common ratio.

$$S = (2^n - 1)/(2 - 1) = (2^n - 1)$$

Illustration 6.5 In a complete binary tree, the depth is d and if number of nodes is N, find depth in terms of N (Fig. 6.10).
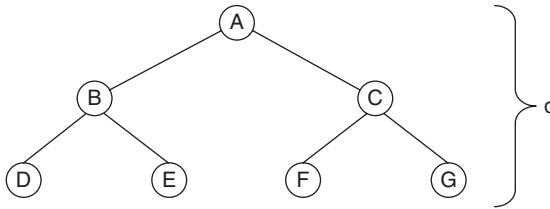
Figure 6.10

Solution If depth is d, then number of nodes

$$N = (2^n - 1)$$

where N is number of nodes.

$$N + 1 = 2^d$$
$$\log (N + 1) = d \log 2$$
$$d = \log_2(N + 1)$$
$$d = \log(N + 1)$$

6.5 TREE TRAVERSAL

The detailed definition of traversal has been given in Chapter 7. Informally, it is the order in which the vertices of a tree are processed. A binary tree can be traversed in three ways namely pre-order, in-order, and post-order.

6.5.1 Pre-order Traversal

In a *pre-order traversal*, first the root is processed followed by the left sub-tree and then the right sub-tree, which in turn are processed in the same order. The traversal procedure can be summarized as

- Process the root
- Process the left sub-tree in pre-order
- Process the right sub-tree in pre-order

6.5.2 In-order Traversal

In an in-order traversal, first the left sub-tree is processed in in-order then the root is processed followed by the right sub-tree in in-order. The traversal procedure can be summarized as

- Process the left sub-tree in in-order
- Process the root
- Process the right sub-tree in in-order

6.5.3 Post-order Traversal

In a post-order traversal, the left sub-tree is processed in post-order, then the right sub-tree in post-order, followed by the root. The traversal procedure can be summarized as

- Process the left sub-tree in post-order
- Process the right sub-tree in post-order
- Process the root

Figure 6.11 shows the in-order, pre-order, and post-order traversals of a tree.
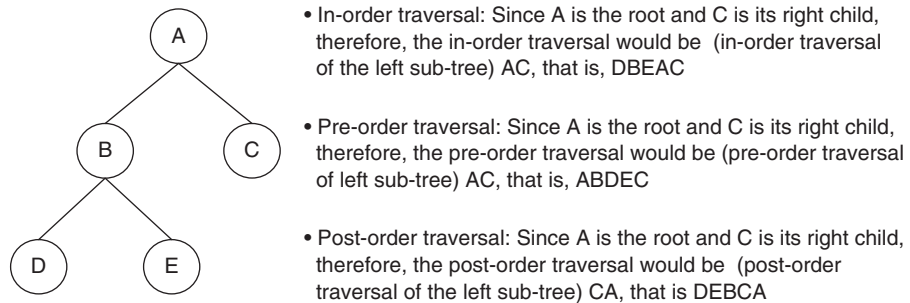
- In-order traversal: Since A is the root and C is its right child, therefore, the in-order traversal would be (in-order traversal of the left sub-tree) AC, that is, DBEAC

- Pre-order traversal: Since A is the root and C is its right child, therefore, the pre-order traversal would be (pre-order traversal of left sub-tree) AC, that is, ABDEC

- Post-order traversal: Since A is the root and C is its right child, therefore, the post-order traversal would be (post-order traversal of the left sub-tree) CA, that is DEBCA

Figure 6.11   Traversals of a binary tree

6.6 TO DRAW A TREE WHEN PRE-ORDER AND IN-ORDER TRAVERSALS ARE GIVEN

Illustration 6.6   The in-order traversal of a tree is DBEACF and its pre-order traversal is ABCDEF. Draw the tree.

Solution The pre-order traversal has root at the beginning:

<p style="text-align:center">ABCDEF</p>

Therefore, root of the required tree is A.

Step 1 First of all, look for the root in the in-order traversal

<p style="text-align:center">DBE A CF</p>

<p style="text-align:center">left   right</p>

So the in-order traversal of left sub-tree of the required tree is DBE.

Step 2 Now, look for DBE in pre-order

<p style="text-align:center">A BDE CF</p>

Therefore, B must be the root of left sub-tree.

Step 3 Then, look for B in the in-order traversal of left sub-tree:

<p style="text-align:center">D B E</p>

<p style="text-align:center">left   right</p>

Therefore, left sub-tree is as shown in Fig. 6.12.
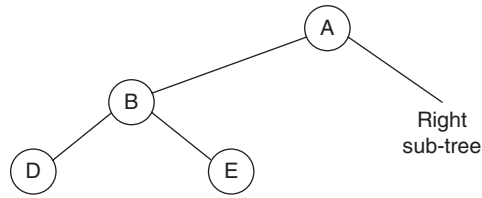
Figure 6.12

Now to create the right sub-tree, follow the same procedure.

$$CF \quad \text{in-order}$$
$$CF \quad \text{pre-order}$$

Therefore, C is the root and F is right child of C. The tree is as shown in Fig. 6.13.
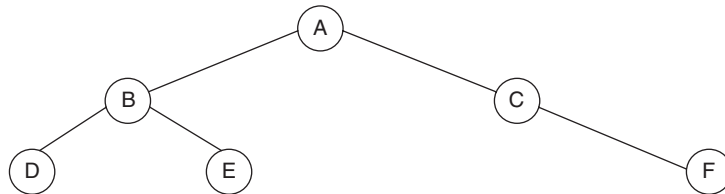
Figure 6.13   Tree for Illustration 6.6

Illustration 6.7   Write the pre-order and in-order traversals of the tree shown in Fig. 6.14.
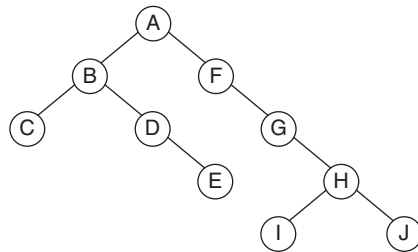
Figure 6.14   Tree for Illustration 6.7

Solution Pre-order traversal of the given tree can be found out with the help of the following algorithm:
- Process the root
- Process the left sub tree in pre-order
- Process the right sub-tree in pre-order

  In the given tree, the root is A

The pre-order traversal of the left sub-tree can be found via the procedure given in Fig. 6.15.
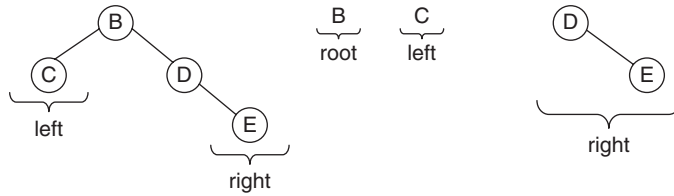
Figure 6.15 Pre-order traversal for right sub-tree

In the same way, the pre-order traversal of the right sub-tree can be found via the procedure depicted in Figs 6.16–19. Therefore, pre-order of the complete tree is

A BC DE FGHIJ
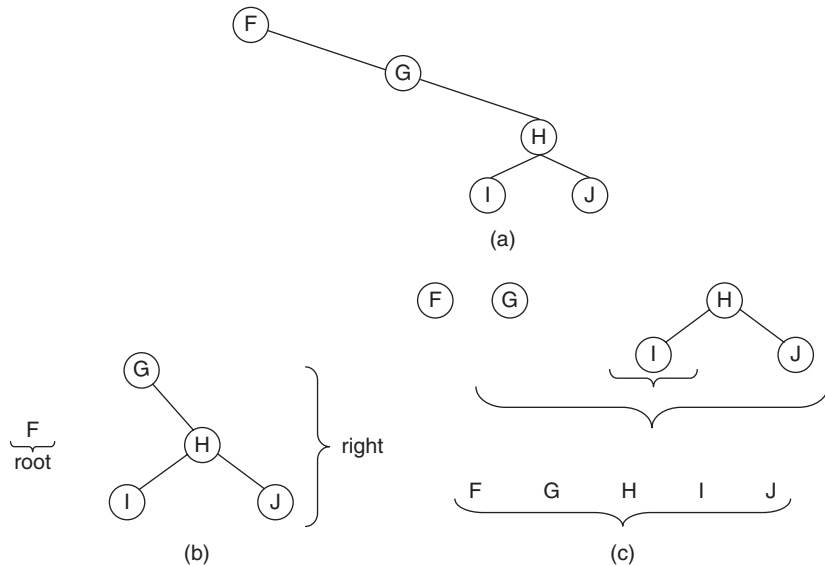
Figure 6.16 Pre-order traversal for right sub-tree
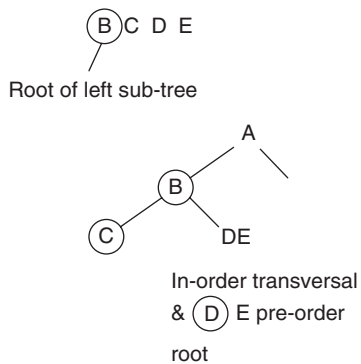
Figure 6.17
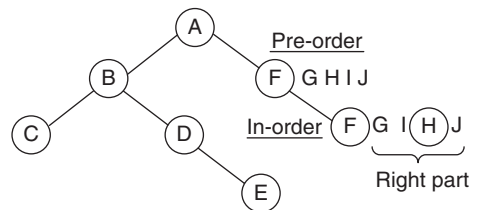
Figure 6.18
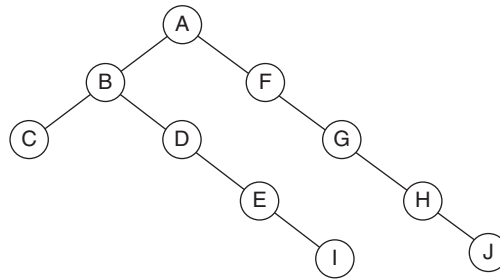
Figure 6.19

Illustration 6.8 Use the pre-order traversal ABCDEFGHIJ and the in-order traversal CBDEAFGHIJ of the tree to find the structure of the tree.

Solution First of all, find the root of tree (by using the pre-order)

$$\text{ABC DE FGHIJ}$$

After that, look for the root in the in-order traversal of the tree

$$\text{CBDE A FGIHJ}$$

Then, look for CBDE in the pre-order traversal of the given tree.

Illustration 6.9 The post-order traversal of a binary tree is

$$\text{AB} + \text{C} * \text{DE}/-$$

and its in-order traversal is

$$\text{A} + \text{B} * \text{C} - \text{D/E}$$

Find the tree.

Solution In the post-order traversal, we have the root at the end.

Therefore, root of the tree is '−'.

Now, look for '−' in the in-order

$$\text{A} + \text{B} * \text{C} - \text{D/E}$$

The left sub-tree is A + B * C.

When we look at its post-order

$$\text{AB} + \text{C}$$

We see that * is the root of the left sub-tree (Fig. 6.20).
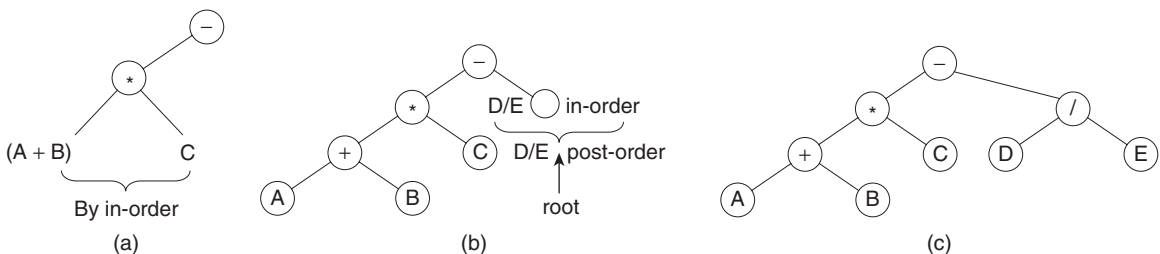
Figure 6.20

Illustration 6.10 The following figure shows the shortcut to find pre-order and post-order traversals of a given tree (Fig. 6.21).
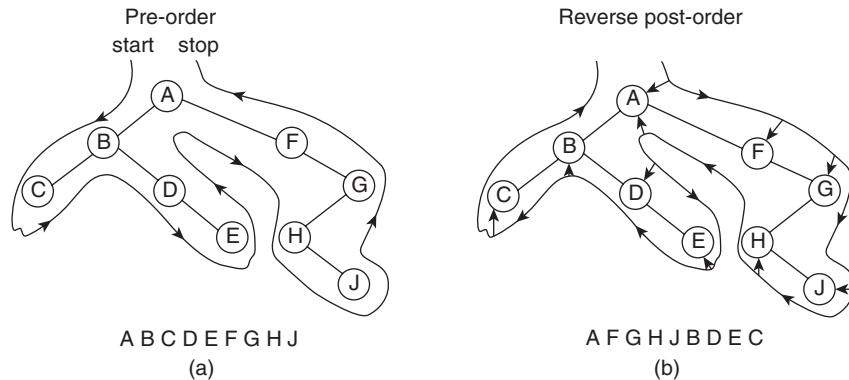
Figure 6.21

Solution Reverse post-order,

$$A F \quad G H J \quad B D E C$$

Taking the reverse

$$C E D B J H G F A$$

Which is the post-order traversal (Fig. 6.21(b)).
The pre-order can be found by Fig. 6.21(a).

6.7 BINARY SEARCH TREE

Binary search tree is a binary tree (where each node has at most two children) in which a new node is added at the left of the tree if it is smaller and to the right if it is bigger in value.

Example: Create a binary tree (binary search tree) from the following values:

$$23, 10, 12, 5, 4, 91, 18, 2, 28$$

First value = 23
Let us make it the root
Second value = 10 which is less than 23
Therefore, it will be positioned to the left of the root (23).
Third value = 12. This is less than 23 and hence will be moved to its left. But 12 is greater than 10, and will be moved to its right.
Now next value is 5; 5 is less than 23, that is, root → value. Therefore, we move to left, that is, ptr = ptr → left
Now ptr → value = 10. But 5 is less than ptr → value. Therefore, we create a new node on the left of 10.

Next value is 4 moving in the same way as we started
Now we encounter 91 which is
> root → value
Therefore, we create a new node in the right
Now next is 18
Now next is 2
and finally we have 28.
Figure 6.22 shows the complete binary search tree.
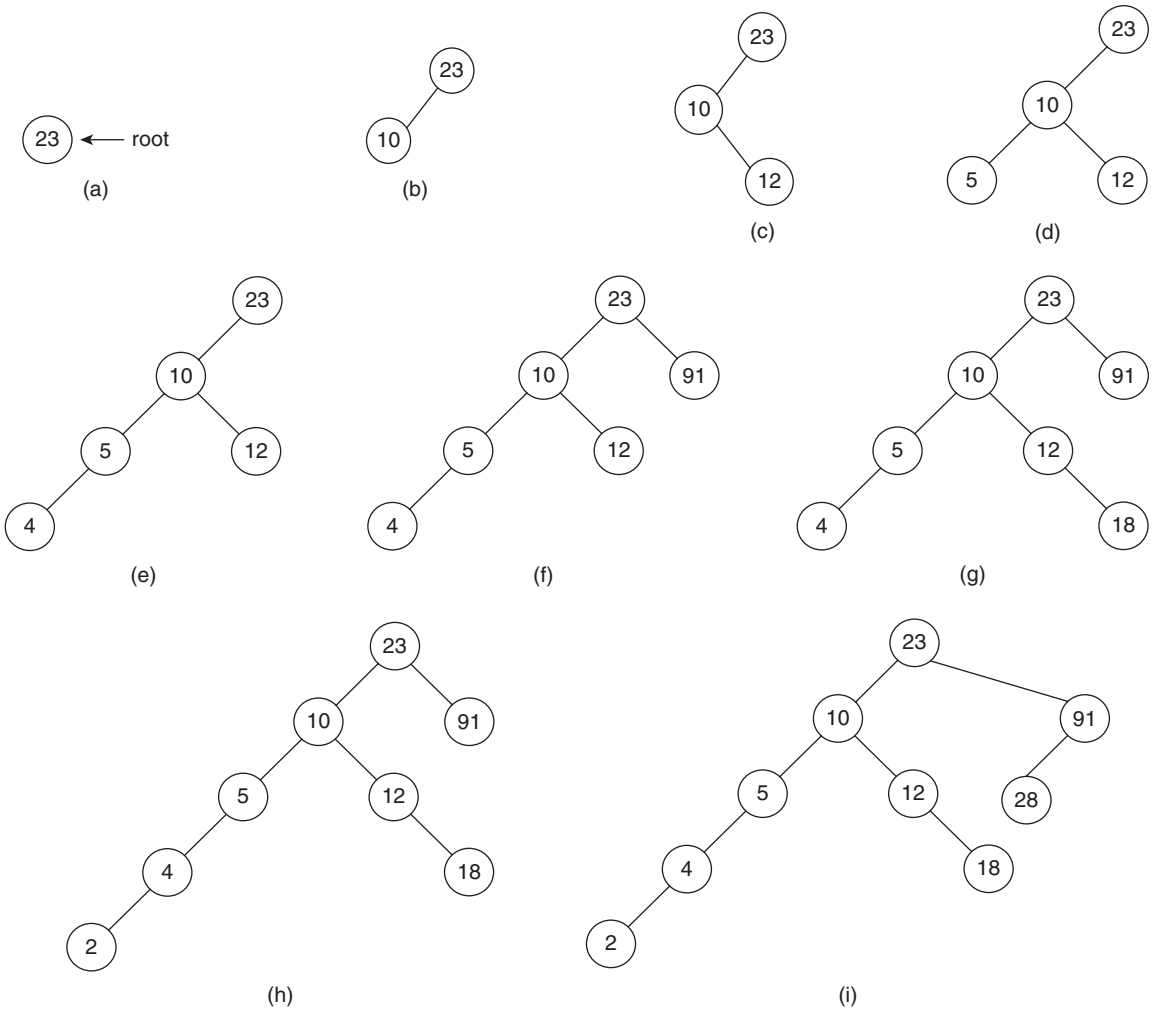
Figure 6.22   Creation of a binary search tree

Let us summarize with the help of creation of binary search tree.

Algorithm 6.1 Creation of binary search tree

Input: Binary search tree and the item which needs to be inserted
Output: The tree with an additional element.

```
Add node (x,root)

{
ptr=root;
if (ptr = = NULL)
    {
     Create a new node;
      node→value = x;
      node→left = node→right = NULL;
}
else
 {
 If (node→value < x)
    {
    Ptr = node→left
    if (ptr = NULL) create
    new node and new node→value = x
    else add node (x, ptr)
    }
else
    {
    ptr = node→right;
    if (ptr = NULL) create new node & new node→value = x
    else
    add node (x, ptr);
    }
}
}  end of algorithm.
```

Illustration 6.11 Search for node having value 12 in the tree shown in Illustration 6.10.

Solution The basic idea behind searching is summarized in the following pseudocode:

```
Put the pointer ptr = root.
    ptr = root

    if ptr→value = 12 (the value to be searched)
    Print success & exit.

    else
```

```
if (ptr→value < 12)
{ ptr = ptr→right;
}
else
ptr = ptr→left;(refer to Figure 6.23)
```
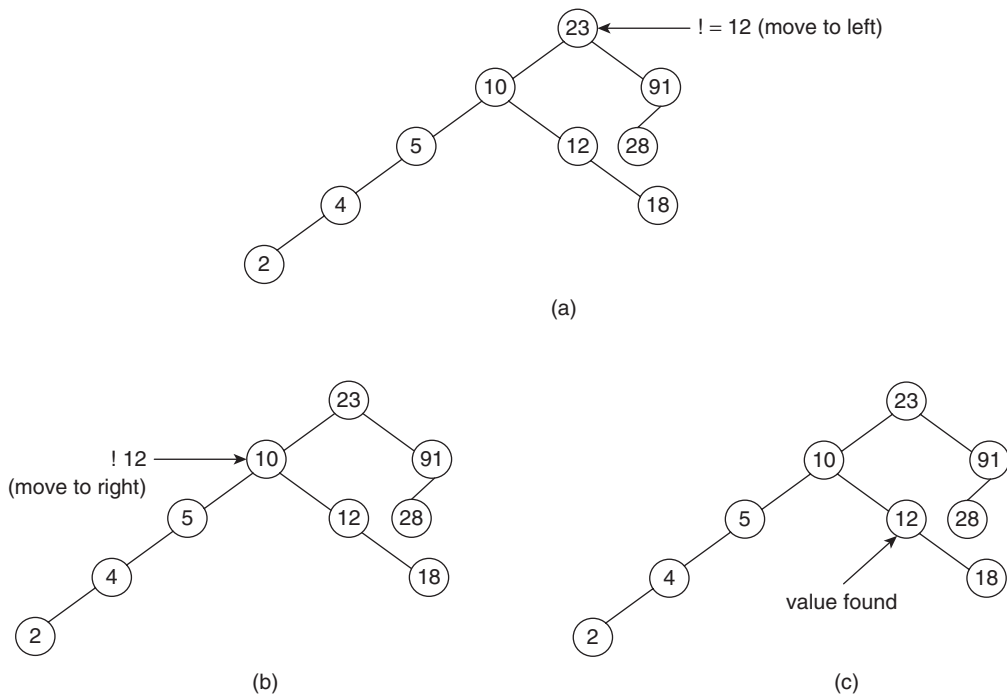

Figure 6.23 Searching in a binary search tree

Illustration 6.12 How many comparisons are needed to search a value in a binary search tree, of depth *d*?

Solution In the worst case, it will be *d* and in the best case it will be 1; if the element to be searched is present at root itself.

Illustration 6.13 Describe the two major searching techniques in a linear array.

Solution

(a) *Linear search*: Here each and every element is seen and we proceed further one by one. The pseudocode of linear search is as follows.

```
Linear search (a[], x, n)
```

```
{||*a[] is the array where element is to be searched, x is the element to be
searched and n is the number of elements in the array.
```

```
for (i = 0; < n; i++)
{
  if (a[i] = = x)
    {
    print: "found";
    exit ();
    }
} || if we come out of loop then || element not found.
    Print: "element not found"
```

(b) *Binary search*: Here we have a sorted array a[ ], we check the value to be searched at low (first index of array).

```
if (a[low] = = x)
    {
    Print: "found";
    exit ();
    }
or at the last value.
else if (a[high] = = x)
    {
    print: "found";
    exit ();
    }
Or we found out mid which is (low + high)/2
If (a[mid] = = x)
    {
    print: "found";
    exit ();
    }
if (element is not found and,
    x < a[mid])
    {
then we take left subarray
that is, low is same
    high is mid - 1
    }
else (if x > a[mid]) {
then we take right subarray
that is, low = mid + 1
    high is same
}
}
```

The above algorithm has been formally discussed in Illustration 9.4, which is very similar to binary search tree.

6.8 B-TREE

A B-tree is a balanced tree in which all the leaf nodes are at the same level. The order of a B-tree is denoted by M. In order to keep the height of the tree minimum, a node has a minimum of two and a maximum of ($M/2$) children. Each node has maximum of ($M - 1$) keys. When a new value is inserted, then it is inserted at the requisite place. If the number of values at a level exceeds ($M - 1$), then the tree is split into left and the right sub-trees. In order to understand the concept, let us take an example of a B-tree with $M = 5$. It may be noted that when the number of values exceed the maximum value ($M - 1$), then the elements to the left of the median move to the left sub-tree and those to the right move to the right sub-tree. Figure 6.24 explains the example.
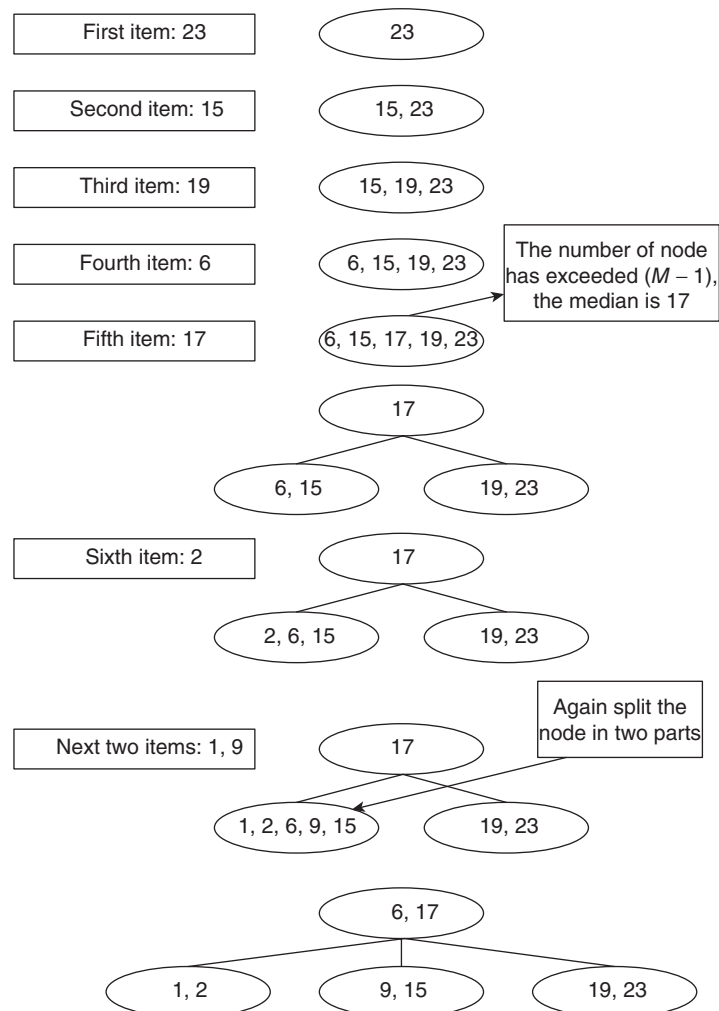
Figure 6.24   Insertion in a B-tree

6.9 HEAP

A binary heap is a binary tree in which the value of root is always greater (or less) than either of its children. The elements in a complete binary tree are added in the order depicted by Fig. 6.25. Heap can be of two types, max-heap or min-heap. In a max-heap, the parent is greater than either of its child, whereas in a min-heap, the parent is always lesser than either of its child.
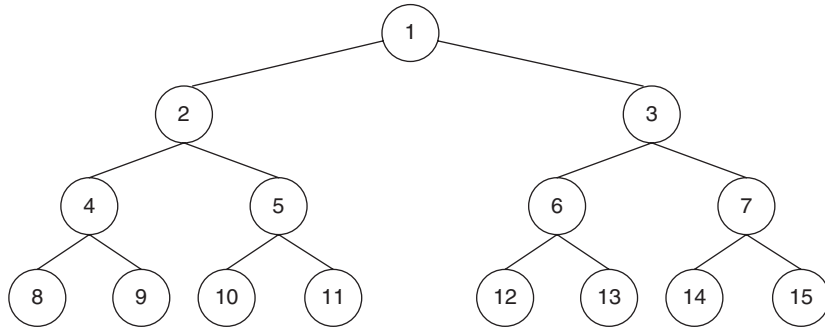
Figure 6.25   An example of a min-heap

6.9.1 Creation of a Heap

In order to create a heap, a simple algorithm is followed. The first element becomes the root. The next element goes to the position indicated by Fig. 6.26(b). However, if the
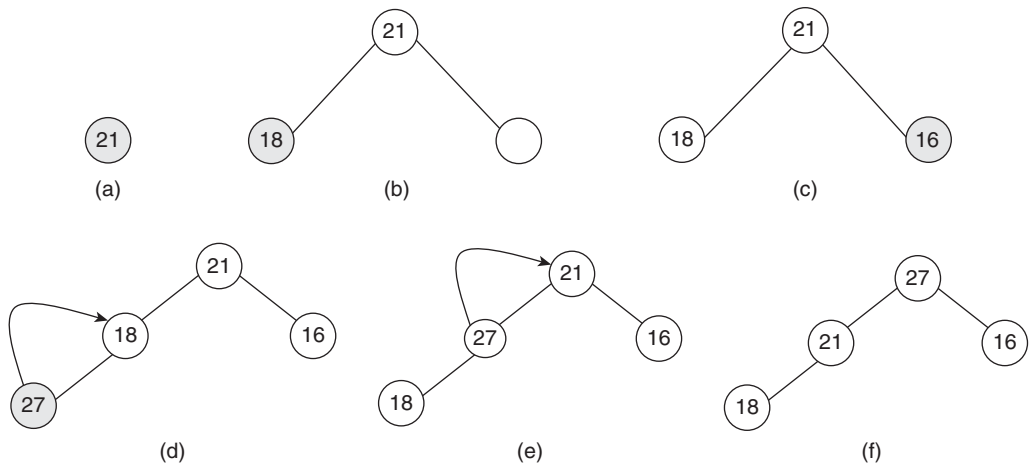
Figure 6.26   (a) The first element becomes the root of the heap; (b) the next element becomes the left child of the root, if it is lesser than the root; (c) the next element becomes the right child of the root, if it is lesser than the root; (d) since the child is greater than the parent it is swapped; (e) 27 is greater than its new parent, so it is swapped with the root of the heap resulting in (f)

element is greater than its parent, then it is swapped with its parent. The process continues till each parent is greater than either of its child. It may be noted here that a heap is fundamentally different from a binary search tree. In a binary search tree, the right child is greater than the parent, whereas the left child is smaller. In the case of max heap, the parent is greater than either of its children. In order to understand the concept, let us consider the following illustration.

Illustration 6.14 Create a heap out of the following elements:

$$21, 18, 16, 27, 23, 17, 18$$

Solution

Step 1 The first number is 21. A new node, root node, is created and its value is set to 21 (Fig. 6.26(a)).

Step 2 18 goes to the 2nd position in the binary tree depicted in Figs 6.26(a) and (b).

Step 3 16 becomes the right child of the root (Fig. 6.26(c)).

Step 4 27 becomes the left child of 18. However, it is greater than 18, it is swapped with 18, it is greater than its new parent. 'So it is swapped with the root of the heap'(Figs 6.26(d) and (e)).

Step 5 23 becomes the right child of 21. However it is greater than 21, therefore, it is swapped with 21 (Figs 6.27(a) and (b)).
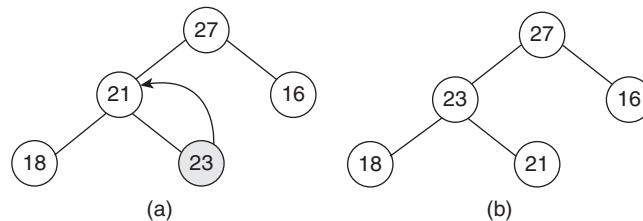
Figure 6.27   (a) 23 swapped with its parent; (b) Adjusted tree

Step 6 17 then becomes the left child of 16 (Fig. 6.28(a) and the adjusted tree is Fig. 6.28(b)).
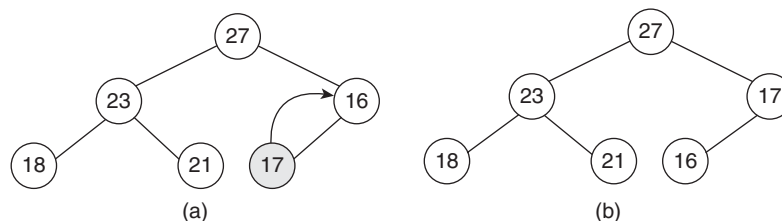
Figure 6.28   (a) 17 becomes the left child of 16 and then swapped
with 16; (b) adjusted tree

Algorithm 6.2   Insertion in a heap

Input: Heap h, int x
Output: A heap in which x is placed at appropriate position
Strategy: Discussed above

```
insert (Heap h, int x)
      {
      //Heap is stored in an array
      if (heap is empty)
            {
            h[0]=x;
            }
      else
            {
            // the current element is to be placed at position n
            if(x<h[n/2])
                  {
                  h[n]=x;
                  }
            else
                  {
                  h[n]=x;
                  while (h[n]<h[n/2])
                        {
                        swap (h[n], h[n/2]);
                        }
                  }
            }
         }
```

Complexity: Swaps are done at most $\log n$ times. So the worst-case complexity is $O(\log(n))$, whereas the best-case complexity is $O(1)$.

6.9.2 Deletion from a Heap

Deletion of the leaf from a heap is easy as the leaf is simply removed as shown in Fig. 6.29(a). However in the case of internal node, the deletion may require re-heapification, for example deletion of a node 23 would result in a heap shown in Fig. 6.29(c).

6.9.3 Heapsort

In heapsort, the given elements are first arranged in a heap. Then the elements are removed one by one. After every deletion, the elements are re-heapified.

If the heap in question is a max-heap, then we get the elements in the descending order. In the case of a min-heap, the output is an ascending sequence.
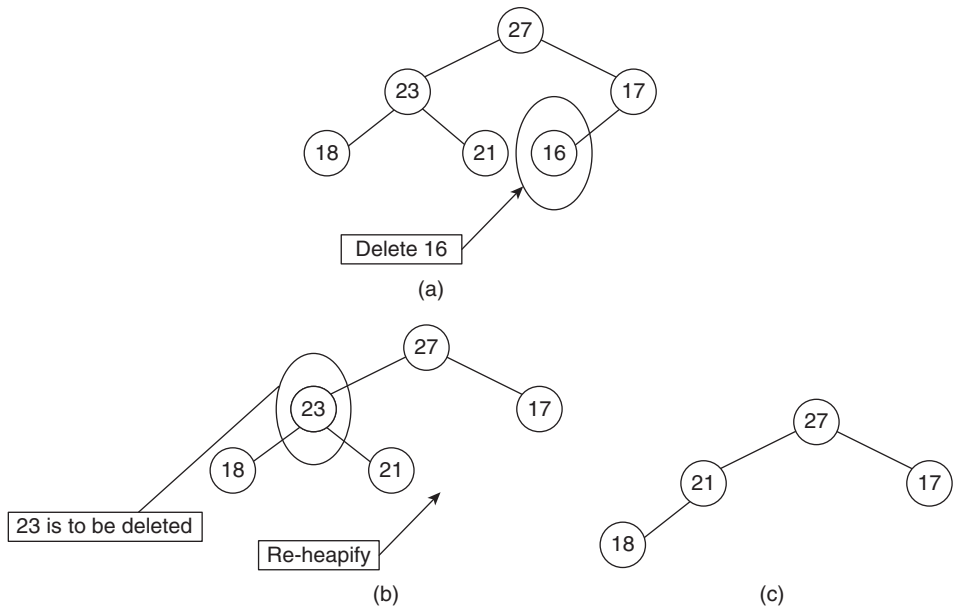
Figure 6.29 (a) Step 1: deletion of 16; (b) step 2: deletion of 23; (c) step 3: repositioning of 21

The following algorithm presents the formal procedure of heapsort.

Algorithm 6.3   Heapsort

Input: A list of elements
Output: Sorted elements
Strategy: Discussed above
Heapsort (List elements) returns sorted_list

```
    {
    heap h=heapify (elements);
//the elements are heapified and inserted into a heap namely h
    i=0;
    while(i!=n)
        {
        x=delete(
//the delete function removes the element at the root of the heap and inserts
it into x
        insert(sorted_list, x);
//the element x is inserted into sorted_list
        }
    }
```

Complexity: As explained earlier, the algorithm requires heapify. The worst-case complexity of heapify is $(\log(n))$; therefore, the complexity of the algorithm is $O(n \log n)$.

6.10 BINOMIAL AND FIBONACCI HEAP

Binomial heap is a data structure similar to a binary heap with some additional constraints to facilitate the union operation. A binomial heap satisfies the property of a minimum heap. That is, the element at the root is always less than that of either of its child. The structure of a binomial tree follows the concept of recursion. A binomial tree of order 0 has just one node. Figure 6.30(a) depicts the tree. A binomial heap of order 1 has two nodes, the second being the child of the first. Figure 6.30(b) shows the tree. A binomial heap of order 2 can be crafted via the above two trees. Figure 6.30(c) shows a binomial heap of order 2, Fig. 6.30(d) shows a binomial heap of order 3, and finally, Fig. 6.30(e) shows a binomial heap of order 4. In general, a binomial heap of order k has a binomial tree of order 0 as its first child, that of order 1 as its second child, and so on. The leftmost child of the root, in this case, would be a binomial tree of order $(k - 1)$.
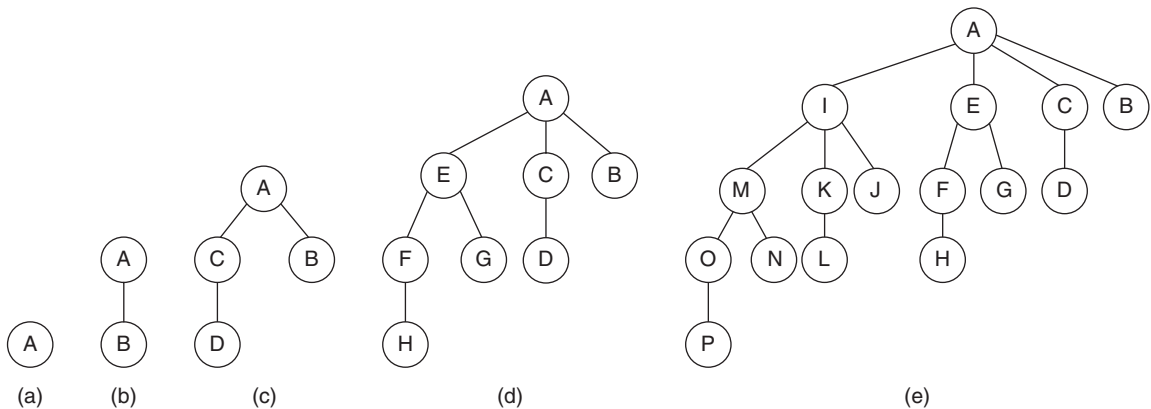
Figure 6.30 Binomial tree; (a) binomial heap of order 0, (b) order 1, (c) order 2, (d) order 3; (e) order 4

The above organization makes the simple merger of two trees.

There is another tree called Fibonacci trees in which, in the extreme case, each node can even be a separate tree. However, it follows the minimum heap property. The operations can be lingered on to a later point. Two major operations in Fibonacci tree are merge and decrease. Merge merges the two Fibonacci trees, the implementation of which is as simple as concatenating two lists. The decrease operation splits the tree into different trees.

6.11 BALANCED TREES

A binary search tree can, at times, be skewed. For example, consider the following example. A binary search tree has to be created from 12, 9, 5, 3, 2, 1. The first element 12 becomes the root of the tree. Since 9 is lesser than 12, it becomes the left node of the root. Now, 5 is lesser than 9, it becomes the left node of 9. In the same

way, 3 becomes the left node of 5, 2 becomes the left node of 3 and 1 becomes the left node of 2. The tree is depicted in Fig. 6.31.

In the static representation of this tree 12, being the root would be stored at the 0th index, 9 at 1st, 5 at 3rd, 3 at 7th, 2 at 15th, and finally 1 at 31st position.

| 12 | 9 | | 5 | | | 3 | | 2 | | 1 |

0   1   2   3   4   5   6   7   … 15   … 31

The array has 31 position but only 6 are filled. Thus, the efficiency is less than 20%.
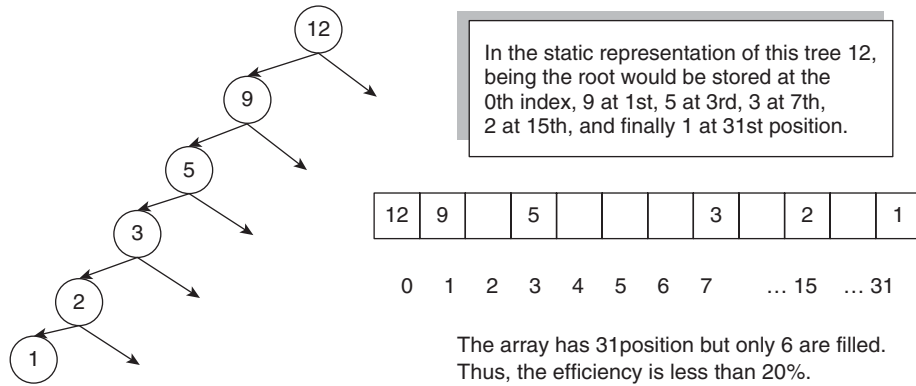
Figure 6.31   A skewed binary tree

The above problem can be solved by what we call height balanced trees. In order to understand the concept, let us try to understand the concept of balance factor. Balance factor of a binary tree is defined as the difference of height of left sub-tree and right subtree. A binary tree is called a balanced tree if the balanced factor of each node is 0, 1, or −1. The figure depicts some examples of balanced and unbalanced trees. The tree of Fig. 6.32(a) is not balanced since the BF of node A and B are 2. Similarly, the tree of Fig. 6.32(b) is also not balanced. However, trees of Figs 6.32(c) and (d) are balanced, since the balanced factor of each node is 0, 1, or, −1.
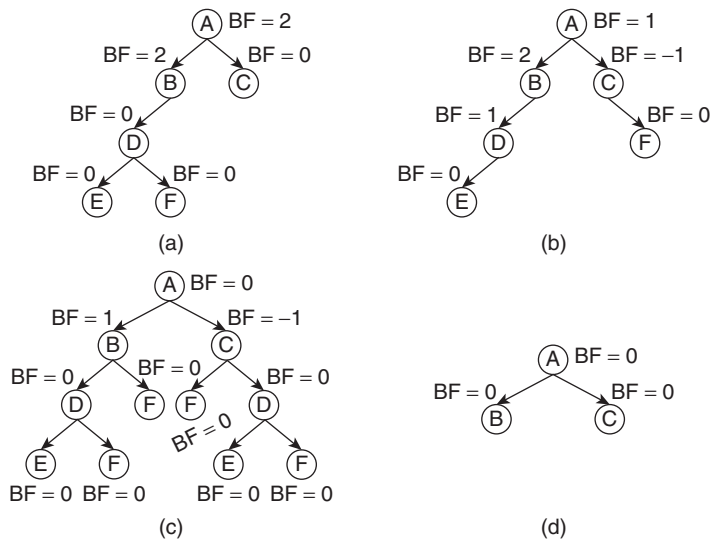
Figure 6.32   Examples of trees that are not balanced (a and b) and those that are balanced (c and d)

One of the most innovative methods of dealing with the above problem, that is converting an unbalanced tree to a balanced one, was shown in Fig. 6.33. The method requires a minor change in the configuration of the given tree as and when a new node is added to the tree; which makes the balance factor of the tree anything except for 0, 1, or −1. In such cases, LL, RR, LR, or RR rotations are used to make the tree balanced.

For example, a tree in Fig. 6.33(a) is a balanced tree. But on adding a new node at the position depicted by Fig. 6.33(b), the tree becomes unbalanced. As per the method suggested by the figure, the tree should be reconfigured to that depicted in Fig. 6.33(b) (second tree), to make it balanced again. The idea is simple. A new node D is added to the left of left, as in left of BL, which is to the left of A. The addition of this node would make the
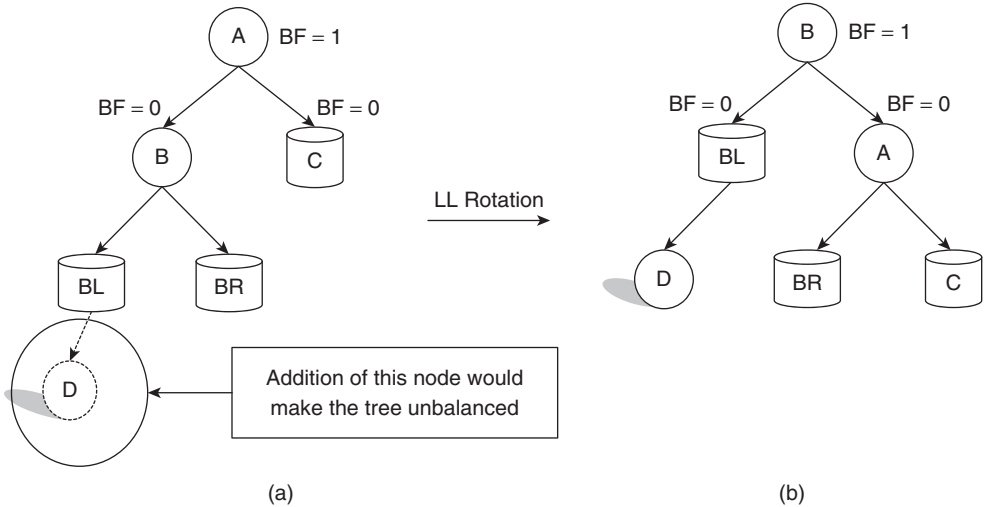
Figure 6.33 LL rotation

tree unbalanced. So as to make the tree balanced again, the tree is reconfigured so that B becomes the new root, A becomes the right node of the new root and C becomes the right node of A. Since BR (the right sub-tree of B) as to the left of A earlier, indicating that the value of each of its node is less than A; in the rearranged tree it goes to the left of A.

In order to understand the RR rotation, let us consider a tree in Fig. 6.34(a), which is a balanced tree. But on adding a new node at the position depicted by Fig. 6.34(b), the tree becomes unbalanced. As per the method suggested by the figure, the tree should be reconfigured to that depicted in the figure, to make it balanced again. The idea is simple, a new node D is added to the right of the right, as in right of CR, which is to the right of A. The addition of this node would make the tree unbalanced. So as to make the tree balanced again, the tree is reconfigured so that C becomes the new root, A becomes the left node of the new root and CR becomes the right node of C. Since CL (the left sub-tree of C) was to the right of A earlier, indicating that the value of each of its node is greater than A; in the rearranged tree, it goes to the right of A.
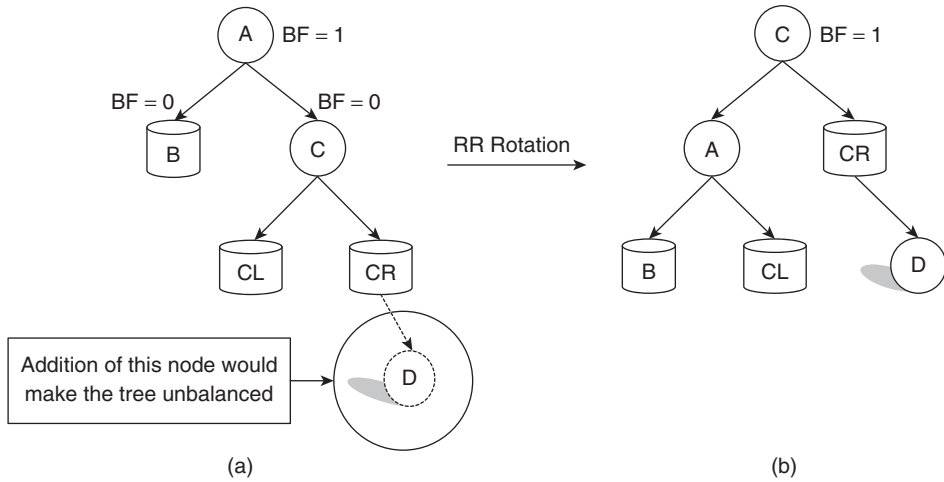
Figure 6.34 RR rotation

Figures 6.35 and 6.36 depict the concept of RL and LR rotations.
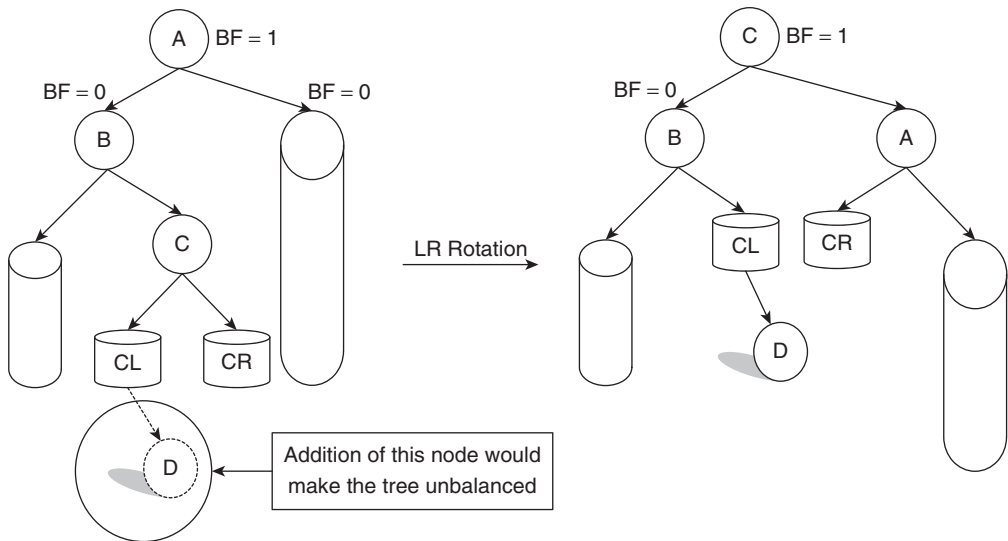
Figure 6.35 LR rotation

The above discussion explores the addition of a new node and the effect of that addition on the balance factor of a tree. The deletion of a node from a balanced AVL tree also requires rearranging the tree in order to make it balanced. Such rearrangements are referred to as R0, R1, and R − 1 rotations.
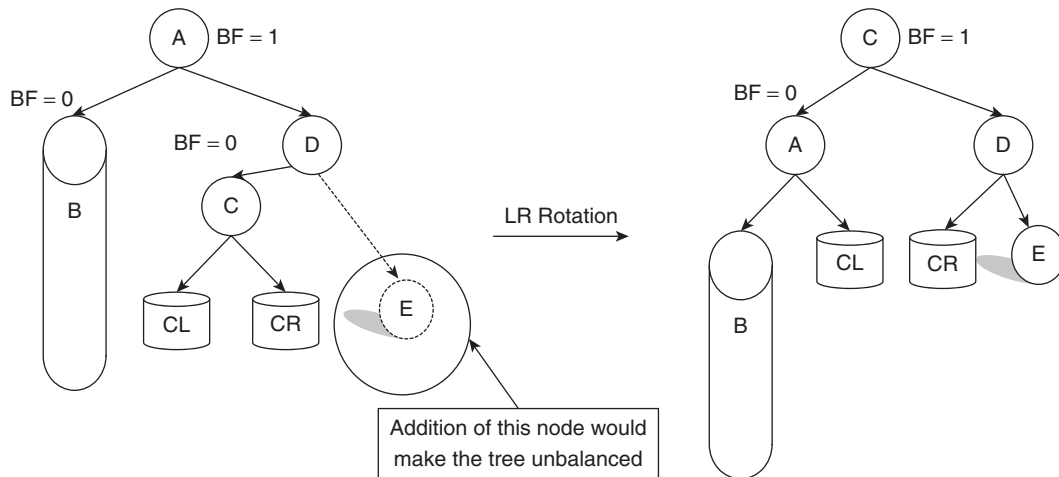
Figure 6.36 RL rotation

6.12 CONCLUSION

The chapter introduced a very important concept called trees. The definition of trees has been discussed first in the chapter. From the definition, it can be inferred that every graph is a tree but every tree is not a graph. The trees majorly discussed in the chapter are binary trees that have a maximum of two children. There are many classifications of binary trees one of which is a binary search tree. This data structure helps in the easy addition and retrieval of values. However, such trees tend to become imbalanced as more values are added. In order to handle this problem, balanced trees discussed in Section 6.11 are used. The chapter also discussed the concept of heaps, which form the basis of one of the most efficient sorting called heapsort. The merging of two heaps, at times, becomes inefficient. This problem is handled by Binomial and Fibonacci heaps discussed in the chapter.

Points to Remember

- Each tree is a graph.
- A binary tree has at maximum two children.
- The number of children, of each node, in a strictly binary tree is either 2 or 0.
- The number of children, of each node, in a complete binary tree is two except for the last level. The degree of nodes of the last level is 0.
- In a balanced tree, the balanced factor is 0, 1, or −1.
- A binomial heap is a minimal heap.
- Heapsort is more efficient than selection or bubble sort.
- The tree traversals can be recursive or non-recursive.

KEY TERMS

Balanced factor The balanced factor of a node is the difference of the height of the left sub-tree and the height of the right sub-tree of the node.

Balanced tree A balanced tree is one in which each node has balanced factor 0, 1, or −1.

Binary tree It is one which has maximum of two children.

Binary search tree It is a binary tree (where each node has at most two children) in which the new node is added at the left of the tree if it is smaller and to the right if it is bigger in value.

B-tree A B-tree is a balanced tree in which all the leaf nodes are at the same level. In order to keep the height of the tree minimum, a node has a minimum of two and a maximum of (M/2) children. Each node has maximum of (M − 1) keys.

Binary heap A binary heap is a binary tree in which the value of root is always greater (or less) than either of its children.

Strictly binary tree It is the one in which each node has either two children or no child.

In-order traversal In an in-order traversal, the left sub-tree is processed in in-order then the root is processed followed by the right sub-tree in in-order.

Pre-order traversal In a pre-order traversal, a root is processed followed by the left sub-tree and then the right sub-tree, which in turn are processed in the same order.

Post-order traversal In a post-order traversal, the left sub-tree is processed in post-order then the right sub-tree in post-order followed by the root.

Strictly binary tree In a strictly binary tree, each node has two nodes except the last level wherein the nodes do not have any child.

Tree A tree is a non-linear data structure that has two components namely nodes and edges. Nodes are joined by edges. This data structure has no cycle and no isolated vertex.

EXERCISES

I. Multiple Choice Questions

1. Which of the following does not have a cycle and does not have an isolated vertex?
   - (a) Tree
   - (b) Plex
   - (c) Graph
   - (d) All of the above

2. Which of the following is more flexible Binomial or Fibbonici heap?
   - (a) Binomial
   - (b) Fibonacci
   - (c) Both
   - (d) Insufficient data

3. Which property is followed by a Binomial heap?
   - (a) Each node has a maximum of two children
   - (b) Each node can have either two or zero children
   - (c) The value of each node is less than its children
   - (d) None of the above

4. Which property is followed by a complete binary tree?
   (a) Each node has a maximum of two children
   (b) Each node can have either two or zero children
   (c) The value of each node is less than its children
   (d) None of the above

5. Which property is followed by a strictly binary tree?
   (a) Each node has a maximum of two children
   (b) Each node can have either two or zero children
   (c) Each node has exactly two children except for the last level in which a node does not have any children
   (d) None of the above

6. Which of the following is the most efficient sorting technique?
   (a) Heap sort
   (b) Selection sort
   (c) Bubble sort
   (d) All the above are equally efficient

7. Which of the following is not a type of tree?
   (a) Binomial   (c) Binary search
   (b) Fibonacci   (d) All of the above are trees

8. In a binomial tree what is the order of second sub-tree of the root (from the right)?
   (a) 2   (c) 0
   (b) 1   (d) None of the above

9. Which of the following is the best data structure in terms of height?
   (a) B-tree   (c) AVL tree
   (b) Binary search tree   (d) None of the above

10. Which of the following is not true for a B-tree?
    (a) A B-tree is a balanced tree in which all the leaf nodes are at the same level
    (b) A node has a minimum of two and a maximum of (M/2) children
    (c) Each node has maximum of ($M - 1$) keys
    (d) All the above points are true for a B-tree

II. Review Questions

1. Define a tree and give examples of graphs which are not trees.
2. Define the following:
   (a) Binary tree   (f) Level
   (b) Strictly binary tree   (g) Degree of a tree
   (c) Complete binary tree   (h) Siblings
   (d) B-tree   (i) Root
   (e) Heap

3. What is a binary search tree? Write an algorithm to insert and delete values from a binary search tree.

4. What is a heap? Write an algorithm to insert and delete values from a heap.

5. What is a B-tree? Write an algorithm to insert and delete values from a B-tree.

6. Write a non-recursive algorithm for in-order, pre-order, and post-order traversals.

7. How will you find out maximum element in a binary search tree?

8. Write an algorithm to delete a tree.

9. What are the advantages of an AVL trees?

10. Write an algorithm to find the deepest element in a tree.

11. Write an algorithm to find the average of the values of nodes of a binary search trees.

12. Write an algorithm to find the total sum of values of nodes of a tree.

13. Using Illustration 6.11, write an algorithm for binary search ().

III. Numerical Problems

1. In the tree given below.
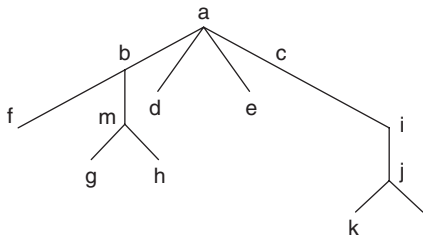
(a) Find parents of g and l.

(b) Find ancestors of h and f.

(c) Find descendants of c, b.

(d) Find terminal vertices.

(e) Find external vertices.

(f) Find sub-tree rooted at j

(g) Find sub-tree rooted at 6.

(h) Is the tree obtained in (g) a binary tree?

(i) Find siblings of f and i.

(j) What can you say about two vertices having same parent?

2. Draw the following graphs or explain why they cannot be drawn:

(a) Six edges, eight vertices

(b) Acyclic four edges, six vertices

(c) Tree, 5 vertices degree of each maximum 2

(d) Tree, 4 terminal vertices, 6 internal vertices

(e) Tree, 6 internal vertices having degree
    2, 1, 1, 1, 3, 3

3. Explain why a forest is a union of trees.

4. Show that graph G with *n* vertices and fewer than *n* − 1 edges is not connected.

5. How many trees can be formed from the following graph:
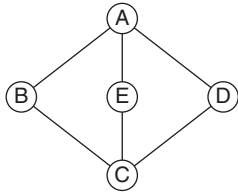
6. Draw tree structure of your organization (or college).
7. Eight-Queens problem:
   Eight-Queens are to be placed on the 8×8 chessboard so that no two are in the same row or same column or same diagonal.
8. Draw any three spanning trees out of the following graphs:
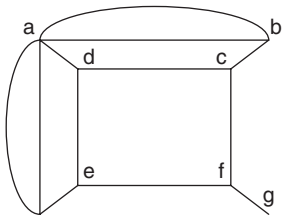
9. Which of the following graphs are a tree? Explain.

(a)

(d)

(b)

(e)

(c)
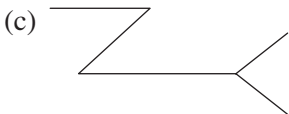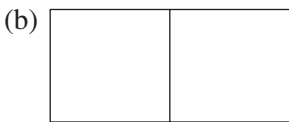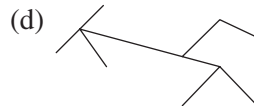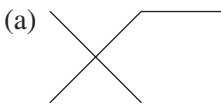
10. For what value of n is the complete graph a tree?
11. For what n is the n cube a tree?
    (For definition of n cube refer to graphs)
12. Decode each of bit strings using Huffman code given.
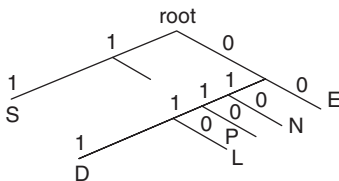
(a) 011000010

(c) 111001110100111

(b) 01110100110

13. With the help of tree of Q12, encode the following strings:
    (a) SEAL   (c) NEAD
    (b) LAP
14. What techniques are used except for Huffman code to store text?
15. Show that the tree is the planar graph.
16. If we draw a tree of an expression and then take its pre-order then it is called prefix form.
    Now find prefix form of the following:
    (a) (A + B) + (C/D)   (d) (A * B − C/D + E)/(A/B/C − D +
    (b) ((A − C) * D) + (A + (B − D))   D)/(A + B +C)
    (c) (A + B) * (C − D)   (e) (A + B + C) * D
17. If we make a binary tree of an expression and find its post-order transversal then it is called postfix form. Find the postfix form of Q16.
18. Draw binary tree for A + (B + C − D + E) * F/G, and write all three transversals.
19. Draw tree for each of two traversals given below:
    (a) Pre-order: + AB   (e) Pre-order: sin x
        In-order: A + B   Post-order: x sin
    (b) Post-order: AB + CD + *   (f) Pre-order: || > AB > CD
        In-order: A + B * C + D   In-order: A > B || C < D
    (c) Pre-order: * + AB + CD   (g) Pre-order: &&&& < AB < BC <
        In-order: A + B * C + D   CD
    (d) Post-order: AB/CD/+   In-order: (A < B) && (B < C) &&
        In-order: A/B + C/D   (C < D)
20. Draw binary search tree from the following data:
    (a) 5, 4, 3, 2, 6, 7, 8, 9, 1, 11.
    (b) 100, 90, 80, 70, 60, 50, 40, 65, 55, 45.
    (c) 1, 4, 7, 11, 2, 5, 8, 12, 3, 6.
    (d) 2, 5, 8, 9, 10, 12, 14, 18, 21, 81.
21. In the following binary search tree, write every step if we search 21:
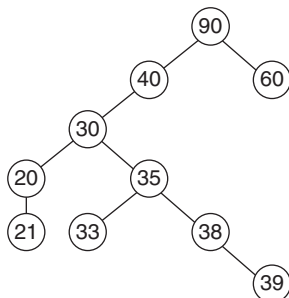
    In the above tree, what happens if we delete 30?

22. Odd node 28 to the above tree.
23. What happens if in tree of Q22 we try to add 40 again?
24. In the following array, apply linear search and binary search to search 28 and hence compare the number of comparisons
    [5, 14, 28, 29, 30, 35, 40, 45, 50]
25. Find time complexity of linear search and binary search.

Answers to MCQs

1. (a)	3. (c)	5. (c)	7. (d)	9. (a)
2. (b)	4. (b)	6. (a)	8. (b)	10. (d)

CHAPTER 7

Graphs

OBJECTIVES

After studying this chapter, the reader will be able to

- Understand the concept of graphs
- Define a graph and understand its representation
- Enlist the different types of graphs
- Understand and implement graph traversal algorithms
- Explain the concept of connected components
- Understand topological sorting

7.1 INTRODUCTION

This chapter introduces the concept of graphs which is the soul of algorithm analysis and design. It is the foundation of the building that would be constructed in our journey through the book. It is important to understand the representation and algorithm of graphs before starting off with the design techniques (Section III of the book).

7.2 CONCEPT OF GRAPH

A graph is a non-linear data structure consisting of two components (V, E), where V is the finite, non-empty set of vertices and E is the set of edges. The set E has elements in the form $(x, y): x, y \in V$. Figure 7.1 shows a graph having vertices depicted by the set $V = \{A, B, C, D\}$ and set of edges $E = \{(A, B), (A, C), (B, C)(B, D)(C, D)\}$. This graph is an *undirected graph*. So, there is a path from A to B and also from B to A.
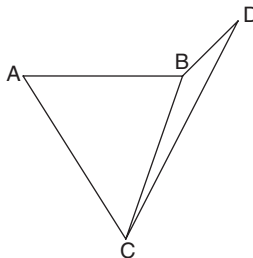
Figure 7.1 An example of an undirected graph

However, a graph, like that in Fig. 7.2, that has edges directing from one vertex to another is called *directed graph*. In this case, though, there is a path from A to B but there is no path from B to A.
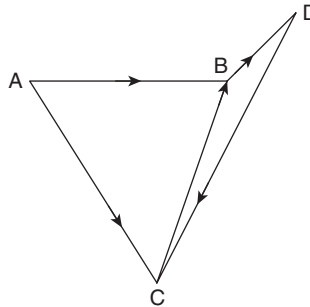
Figure 7.2 An example of
a directed graph

7.3 REPRESENTATION OF GRAPH

A graph can be represented in many ways. Some of them are via matrix and via linked list. The *matrix representation* of a graph generally has rows and columns representing the vertices of the graph. The value at a particular position can be 1 or 0 depending on whether there is an edge between the vertices represented by the corresponding row and column. The matrix representation of a graph depicted in Fig. 7.1 is as follows:

$$
\begin{array}{c@{\quad}cccc}
 & A & B & C & D \\
A & 0 & 1 & 1 & 0 \\
B & 1 & 0 & 1 & 1 \\
C & 1 & 1 & 0 & 1 \\
D & 0 & 1 & 1 & 0
\end{array}
$$

The first element of the first row is 0 as there is no edge from A to A. However, there is an edge between A and B, therefore, the matrix has 1 at the element at row 1 and column 2. In the same way, the element at row 1 and column 4 is 0. The numbers here can be more than 1 if there are more than one edge between two vertices.

The second way of representing the same graph is via *linked lists*. The linked list of each vertex would contain the vertices connected to that vertex. For example, the linked list of A would have B and C. The linked list of this graph is depicted in Fig. 7.3.
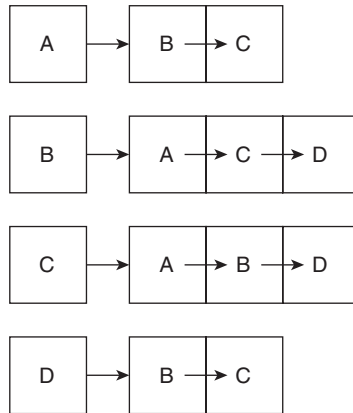
Figure 7.3 Linked list of
graph 1

7.4 CYCLIC GRAPHS: HAMILTONIAN AND EULERIAN CYCLES

A graph can also be classified as a cyclic or a non-cyclic graph. A cyclic graph is one that has at least one path of the form $v_i, v_{i+1}, ..., v_i$. A graph that does not have a cycle is referred to as a non-cyclic graph. A non-cyclic graph that does not have an isolated vertex is also called a tree. The concept of trees has been discussed in detail in Chapter 6. However, even cyclic graphs find their application in many disciplines—from networking to biotechnology.

A special type of cycle called Hamiltonian cycle is also one of the greatly researched topics in graph theory. A Hamiltonian cycle is one which covers all the vertices but no vertex is repeated except for the first. It is, however, not always the case that a graph has a Hamiltonian cycle. Figure 7.4 shows an example of a graph that contains Hamiltonian cycle. The corresponding Hamiltonian cycle is shown in Fig. 7.5. There can be more than one Hamiltonian cycle in a graph.
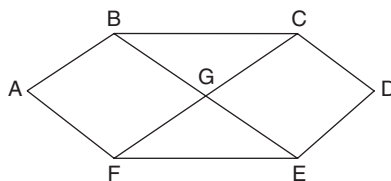
Figure 7.4   A graph with a
Hamiltonian cycle
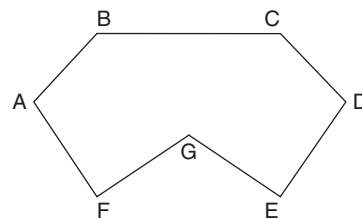
Figure 7.5   One of the
Hamiltonian cycles of the
graph shown in Fig. 7.4

There is another type of cycle called Eulerian cycle. This cycle covers all the edges but no edge is repeated twice. Figure 7.6 is one of the examples of graphs that have an Eulerian cycle. Figure 7.7 is an example of a graph that has an Eulerian cycle but not a Hamiltonian cycle. Figure 7.8 shows a graph that has a Hamiltonian cycle but not an Eulerian cycle. The Hamiltonian cycle is shown in Fig. 7.9. Figure 7.10 shows a graph that has neither of the cycles.
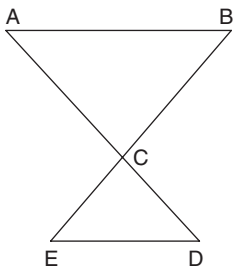
Figure 7.6 Graph with an Eulerian cycle
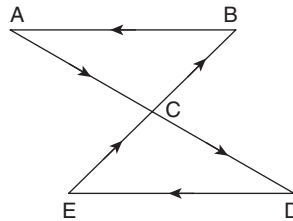
Figure 7.7 The Eulerian path of the graph shown in Fig. 7.6: ACDECBA
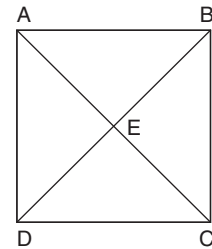
Figure 7.8 An example of a graph that has Hamiltonian cycle but not an Eulerian cycle
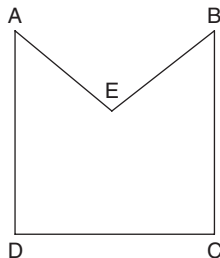
Figure 7.9 The Hamiltonian cycle of graph shown in Fig. 7.8
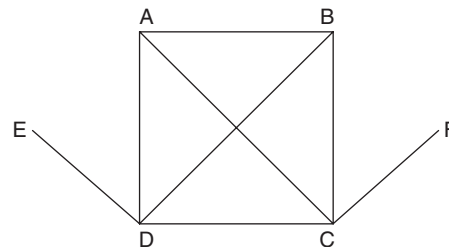
Figure 7.10 Example of graph having neither Eulerian cycle nor Hamiltonian cycle

7.5 ISOMORPHIC AND PLANAR GRAPHS

Having understood the basics of graphs, let us move forward to the concept of isomerism. In order to understand the concept let us take an example. Two persons were asked to solve the following problem:

'Draw and label five vertices a, b, c, d, and e, connect: a & b, b & c, c & d, d & e, a & e'. They answered the same question in two different ways. The first person drew a graph as depicted in Fig 7.11, whereas the second person drew a graph as depicted in Fig. 7.12. Here, both graphs 1 and 2 have

(a) Same number of vertices.
(b) Same number of edges.
(c) One-to-one correspondence.
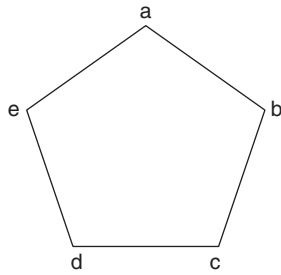
Figure 7.11 Graph drawn
by the first person
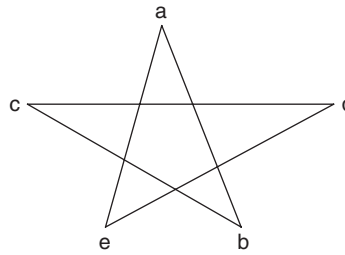
Figure 7.12 Graph drawn by
the second person

Any two graphs that follow the above three properties are referred to as *isomorphic graphs*. The graphs G1 and G2 are isomorphic if for the same ordering of their vertices and their adjacency matrices, discussed earlier, are equal convertible. For example, the adjacency matrix of the graphs shown in Figs 7.11 and 7.12 are as follows.

Adjacency Matrix for Graph 1

$$
\begin{array}{c}
\begin{array}{ccccc} a & b & c & d & e \end{array} \\
\begin{array}{c} a \\ b \\ c \\ d \\ e \end{array}
\left(\begin{array}{ccccc}
0 & 1 & 0 & 0 & 1 \\
1 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 1 \\
1 & 0 & 0 & 1 & 0
\end{array}\right)
\end{array}
$$

Adjacency Matrix for Graph 2

$$
\begin{pmatrix}
0 & 1 & 0 & 0 & 1 \\
1 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 1 \\
1 & 0 & 0 & 1 & 0
\end{pmatrix}
$$

The matrices are same (in this case, even if the matrices are reducible to each other, they are isomorphic), therefore, they are isomorphic.

If a graph can be drawn on a plane, then it is referred to as a *planar graph*. If it cannot be drawn on a plane or it has intersecting edges, then it is called a *non-planar graph*. In order to understand the concept, let us consider the following example.

Illustration 7.1 Three cities C_1, C_2, and C_3 are to be connected via the expressway to each of three destination cities C_4, C_5, C_6. Can a system of roads be designed so that roads do not cross each other?

Solution The graph shown in Fig. 7.13 depicts the situation. It can be seen that if the first city is connected to all the three destination cities, the second can also be connected. However, it is not possible to connect the third city to all the three destination cities without having to intersect any edge.
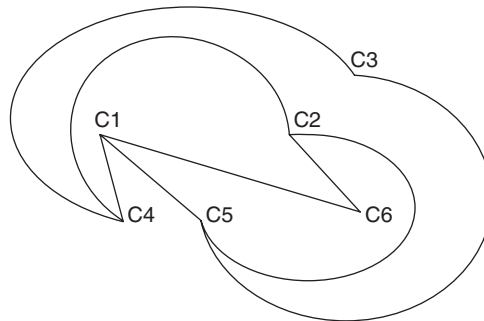
Figure 7.13 Example of a non-planar graph

That is, there is no way to connect C_3 and C_6. The answer to the above problem is that it is not possible to design a system of roads that will not intersect each other. Therefore, a 3×3 graph on a K_5 graph is deemed to be non-planar. It may be stated here that K_5 is a fully connected graph with 5 vertices.

In fact a planar graph always satisfies a relation between the number of edges (e), number of vertices (V), and the number of faces (f),

$$f = e - V + 2$$

If a graph is drawn in a plane divided into regions called faces, then it will be a planar graph.

In the graph depicted in Fig. 7.14, the number of edges, $e = 9$ and the number of vertices, $V = 6$. Therefore, number of faces $= 9 - 6 + 2 = 5$.
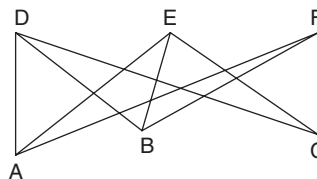
Figure 7.14 A K_{33} graph is always non-planar

So, the relation is not satisfied, hence K_{33} is not a planar graph. In a planar graph $2e > 4f$

Since, in the above graph if
$$2e > 4f$$
then
$$2e > 4(e - V + 2)$$
also
$$e = 9, V = 6$$

therefore, $18 \geq 20$, which is a contradiction.

The above theorem is referred to as Kuratowski's theorem, an example of which is stated as follows.

Kuratowski's Theorem

A graph is planar if it does not contain a sub-graph homomorphic to K_5 or K_{33}. A graph can be reduced to another homomorphic graph by reducing an edge or a vertex.

For example, let us consider a graph G, depicted in Fig. 7.15.

The graph of Fig. 7.17 is K_{33}. So, graph G can be reduced to K_{33} and therefore G″ is not planar. Another theorem that helps us to find out whether the given graph is a planar is called Euler's theorem. The theorem relates e, v, and f, as stated earlier. The theorem and its proof have been explained as follows:
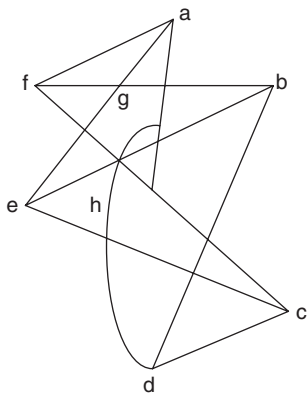
Figure 7.15 Graph G
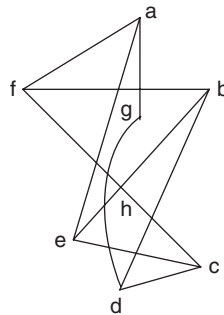
Figure 7.16 Graph G′, homomorphic to G, obtained by the reduction of G
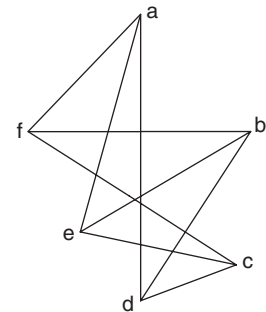
Figure 7.17 Graph G″, homomorphic to G′, obtained by the reduction of G′

Euler's Formula for Graphs

In a planar graph with v vertices, e edges, and f faces

$$f = e - v + 2$$

We prove the theorem using mathematical induction. Mathematical induction has three steps:

We start with verification of the theorem by taking a graph with a single edge (Fig. 7.18(a)).

That is, suppose, $e = 1$

i.e., one edge

Then, $v = 2$ and $f = 1$

$$f = 1 - 2 + 2 = 1$$

So, in this case, the theorem holds.

In the second step, we suppose the relation to be true for $e = k$.

Let the theorem be true for $n = k$, that is, for k vertices,

$$f = e - v + 2$$

If we increase the number of vertices by 1 and edge by 1, then the number of faces will remain same (Fig. 7.18(b)).

If we increase number of vertices by 1 and edges by 2, then there will be an extra face (Fig. 7.18(c)).
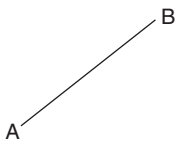
Figure 7.18(a) Graph G′
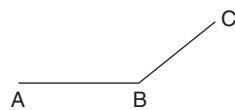
Figure 7.18(b) Adding an extra edge to the graph
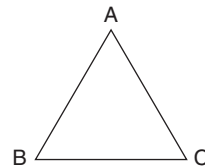
Figure 7.18(c) A planar graph with $v = 3$

If we increase V by 1 and e by 3, there would be 2 more faces (Figs. 7.19 – 7.21). Even in this case, the above formula holds.

The theorem is, therefore, true for $n = k + 1$, if it is true for $n = k$. By the principle of mathematical induction, the theorem is always true.
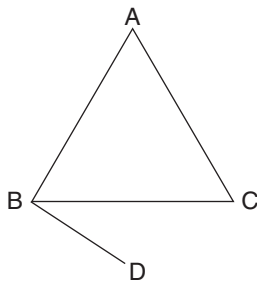
Figure 7.19 Adding an extra edge to graph of Fig. 7.18(c)
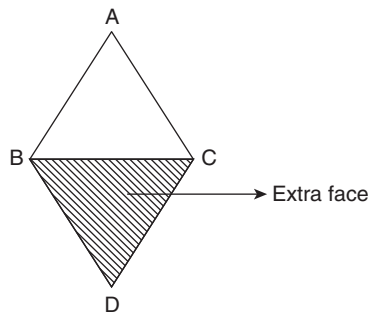
Figure 7.20 Adding two extra edges to the graph
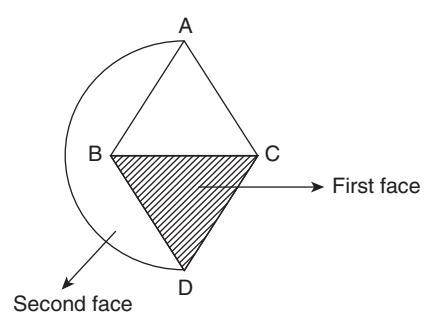
Figure 7.21 Adding three extra edges and one vertex to the graph

Definition The traversal of a graph is the order in which the vertices must be visited.

7.6 GRAPH TRAVERSALS

Having seen the basics of graph theory, let us now move to some intricate problems. One of the most used concepts in the chapters that follow is graph traversals.

This section explores the topic, presents its algorithm, and exemplifies the procedure. Graph traversal can be either breadth first search (BFS) or depth first search (DFS). BFS explores the first level of the graph followed by the next levels, whereas DFS explores the given graph depth-wise. In order to implement DFS, we need a stack. However, BFS can be implemented via a queue. Both the algorithms have their advantages and disadvantages. Figure 7.22 shows the two types of traversals.
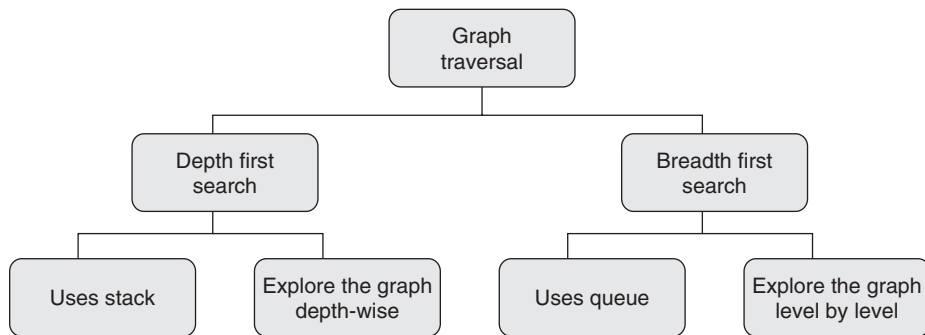
Figure 7.22 Types of graph traversals

7.6.1 Breadth First Search

The first algorithm discussed in this section is BFS. BFS uses a queue. An array already_visited[ ] keeps track of the vertices that have already been visited. The array is a global array. The algorithm starts with a loop that initializes each element of already_visited[ ] as 0. Now when a node is processed, then its adjacent vertices are put into a queue. The process is repeated till the queue is empty. Algorithm 7.1 depicts the above process.

Algorithm 7.1 Breadth first search

Input: Graph G
Output: Sequence of vertices
Strategy: Discussed above
Problem: The problem in this algorithm can be understood by the following example: Suppose there is a vertex at the fifth level which is to be found. However, each level has 128 vertices. The algorithm would be able to find out the answer, but after $(128 \times 3 + 1)$ iterations, DFS would be a better choice in this case.

```
Algorithm BFS (v)
{
// BFS of G at the beginning we have vertex v for node i visited [i] = 1 if i has
```

```
//already been visited. The graph G and array visited [] are global
for(i = 1 to n)
visited [i] = 0
// initialize the array elements to 0.
u = v
visited [v] = 1;
repeat
  {
  for all vertices ω adjacent from u do
{ if (visited [ω] = = 0)
    {
add ω to q;
    visited [ω] = 1;
    }
}
if q is empty then return;
Delete u from q;
} until (false);
}
```

Complexity: If there are l level and *b* children at each level, then the number of children at the first level would be *b*, at the second level b^2, and so on. So the total complexity would be proportional to (b^{n+1}).

Illustration 7.2 Find the BFS of the graph depicted in Fig. 7.23.
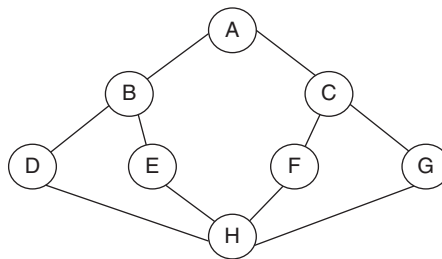
Figure 7.23 Graph for Illustration 7.2

Solution Since it is easier to see the adjacent nodes of a given node via linked list representation, an adjacency list of the given graph is drawn first (Fig. 7.24). This is followed by the implementation of the above algorithm. The stages of BFS have been shown as follows:

Stages in BFS
First of all find the adjacency nodes of the root node.
    B is adjacent to A (Fig. 7.25), so B is processed.
    C is also adjacent to A (Fig. 7.26), therefore C is processed.

Load nodes
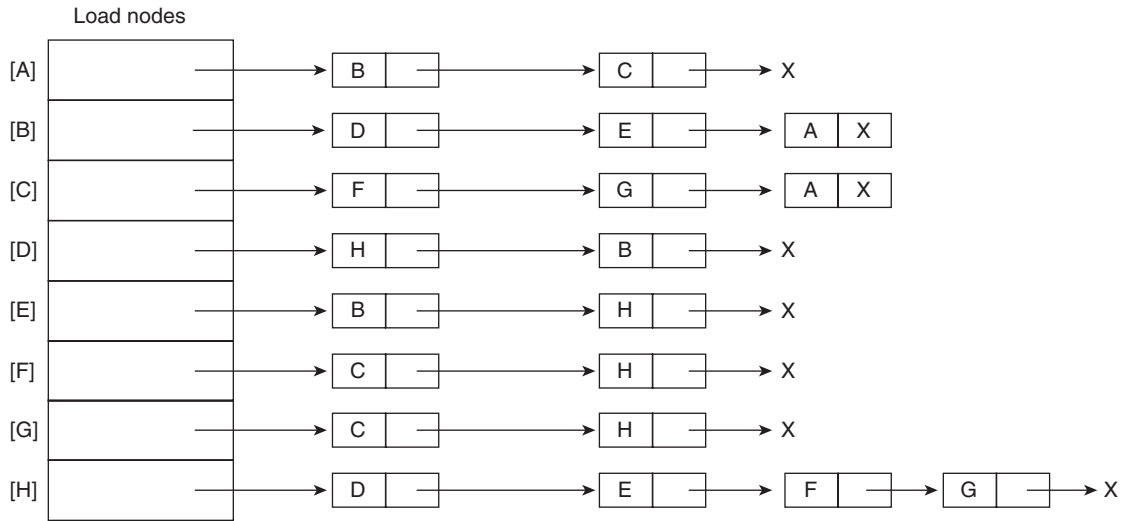
Figure 7.24   Adjacency list of the graph of Illustration 7.2
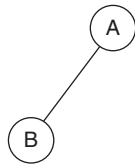
Figure 7.25   Illustration 7.2,
Step 1: Process A and then B
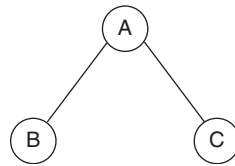
Figure 7.26   Illustration 7.2,
Step 2: Process C

Since A does not have any more adjacent nodes, we now find the nodes adjacent to B (Fig. 7.27).

D and E are adjacent to B, so E and D are processed.

Since B does not have any more adjacent nodes, so we find the nodes adjacent to C.
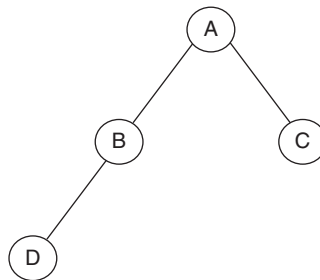
Figure 7.27   Illustration 7.2, Step 3: Move to the
next level and Process D

F, G and A are adjacent to C, however A has already been processed so F and G are processed.

Similarly, next node to be processed is H (Figs 7.28 and 7.29).
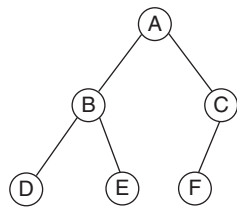
Figure 7.28 Illustration 7.2,
Step 4: Process E and then F
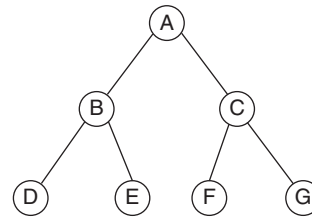
Figure 7.29 Illustration 7.2, Step 5:
Process G and then move to the next level

Since all the nodes of this level have been processed, we now move on to the next level (Fig. 7.30).
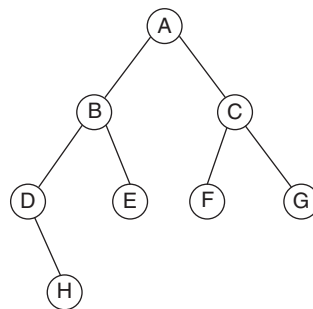
Figure 7.30 Finally, Process H

Finally, H is processed. Therefore, the order in which nodes have been processed is A B C D E F G H.

Complexity: The above example also brings forth the point that if adjacency matrix is used, then the complexity would be O(n). However, if an array is used, the process takes O(n^2) time.

7.6.2 Depth First Search

The next algorithm discussed in this section is DFS. DFS uses a stack. An array already_visited[ ] keeps track of the vertices that have already been visited. The array is a global array. The algorithm starts with a loop that initializes each element of already_visited[ ] as 0. The algorithm is based on the concept of recursion. The algorithm is called by passing the adjacent nodes of the root node to the function again. Recursion uses stack. Therefore, the root node is processed first followed by the node adjacent to it in the next level. The control then considers the next node as the present node. The process is repeated till the stack is empty. Algorithm 7.2 depicts the above process.

Algorithm 7.2 Depth first search

Input: Graph G
Output: Sequence of vertices
Strategy: Discussed above

Problem: The problem in this algorithm can be understood by the following example: Suppose that the node which we are looking for is the second node of the second level and there are 128 levels in the graph. The algorithm would go to the desired node after processing 129 nodes. In such cases, BFS works in a better way.

```
(DFS)
Graph G(V, E) with n vertices and array visited [] initially set to 0.
{
visited [v] : = 1;
for each vertex ω adjacent from V do
  {
if (visited [ω] = 0) then DFS (ω);
  }
}
```

Complexity: $O(V + E)$, if we make use of adjacency list. If, on the other hand, an adjacency matrix is used, the complexity becomes $O(V^2)$.

Illustration 7.3 Find the DFS of the graph depicted in Fig. 7.23.

Solution Since it is easier to see the adjacent nodes of a given node via linked list representation, an adjacency list of the given graph is drawn first (Fig. 7.24).

This is followed by the implementation of the above algorithm. The stages of DFS have been shown as follows.

Stages in DFS
First of all find the adjacency nodes of the root node.
   B is adjacent to A
   So process B (Fig. 7.31).
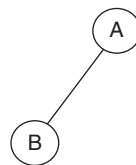
Figure 7.31   Step 1: Process A and then B

D is the first node adjacent to B, so D is processed in the next step (Fig. 7.32)
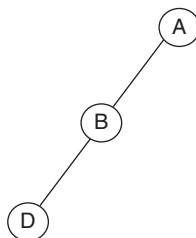
Figure 7.32   Step 2: Process D, being the first adjacent node of B

H is the first node adjacent to D, so H is processed in the next step (Fig. 7.33)
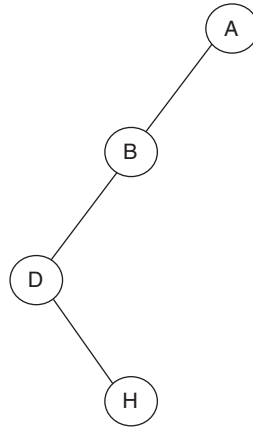
Figure 7.33 Step 3: Process H, being the
unprocessed adjacent node of D

E is the unprocessed adjacent node of H, so E is processed in the next step (Fig. 7.34)
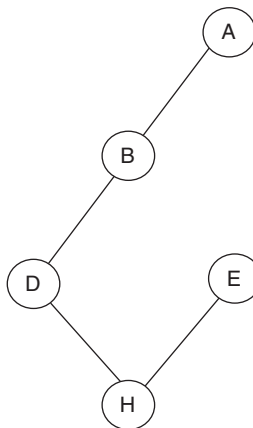
Figure 7.34 Step 4: Process E, being the only
unprocessed adjacent node of H

This is followed by what is referred to as backtracking. The control now goes to H and process the next unprocessed adjacent node of H, since there is no node, adjacent to E, which is still unprocessed (Fig. 7.35).

The node adjacent to F is then processed. Figure 7.36 shows the next step.

Finally, G is processed and the process ends (Fig. 7.37).
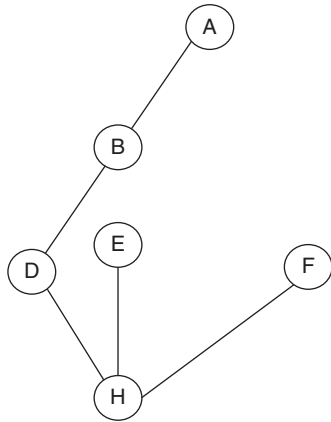
Figure 7.35   Step 5: Process F, which is an
unprocessed adjacent node of H
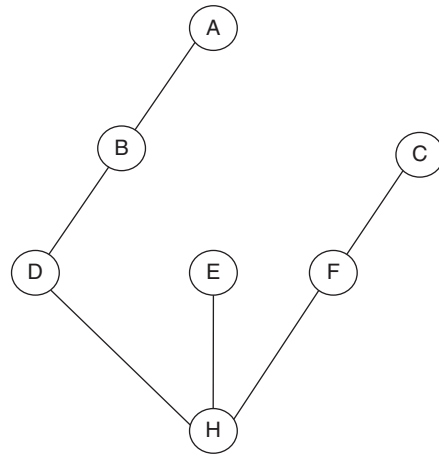
Figure 7.36   Step 6: Process C, which is an
unprocessed adjacent node of F
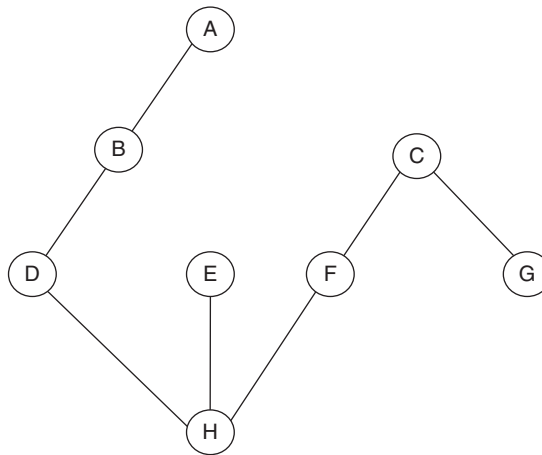
Figure 7.37   Step 7: Process G, which is an
unprocessed adjacent node of C

Therefore, the order is
AB D HE FCG,
which is the DFS of the graph of Illustration 7.3.

7.7 CONNECTED COMPONENTS

The connected components of a graph, with respect to a vertex, are those that can be reached
from a given node via any node. Those components that cannot be reached from the given
vertex are referred to as *disconnected components*. For example, in Fig. 7.38, the components
A, B, C, D, and E are connected with respect to A as they can be reached from A. However,
F and G cannot be reached via A, therefore, they are disconnected with respect to A.
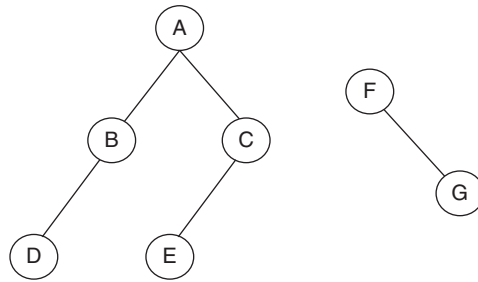
Figure 7.38 A, B, C, D, E are connected
components with respect to A

The question that arises is how do we find out the connected components with respect to a given vertex? The answer is simple. The traversal techniques, described earlier, help us to find the connected components. One can start from the given vertex and then apply BFS or DFS. The nodes that appear in either of the traversals depict the connected components with respect to a given node. The rest of the components are, however, disconnected.

If the 'with respect to' concept is not considered, then there are two connected components in Fig. 7.38, namely C1 and C2. The components have been shown in Fig. 7.39.
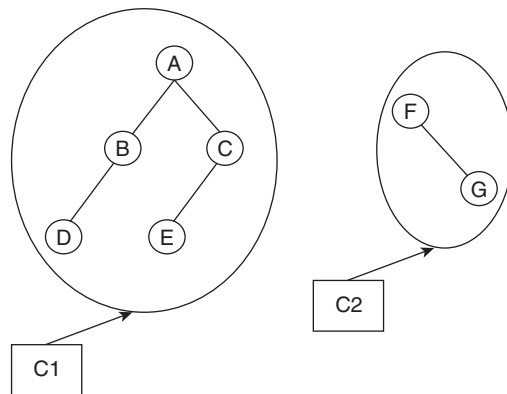
Figure 7.39 C1 and C2 are disconnected

7.8 TOPOLOGICAL SORTING

Topological sorting is a method of sorting in which the vertices of a given graph are traversed in the order in which they appear in the graph. The terminology makes more sense if we are talking about a directed graph. In a directed graph, any order that has only forward pointing vertices is a valid topological sorting. For example, based on the graph shown in Fig. 7.40, the order depicted in Fig. 7.41 is one of the correct topological sorts,
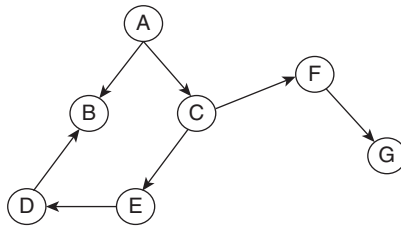
Figure 7.40 Graph G
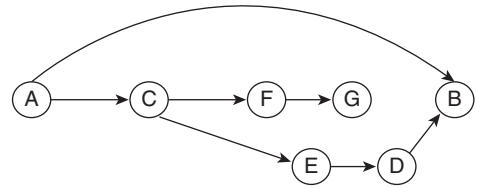
Figure 7.41 A correct topological sort

whereas the one shown in Fig. 7.42 is not the correct topological sorts. For small graphs, topological sorts can be found by not following any particular algorithms. However, for larger graphs, they are found by following Algorithm 7.3.
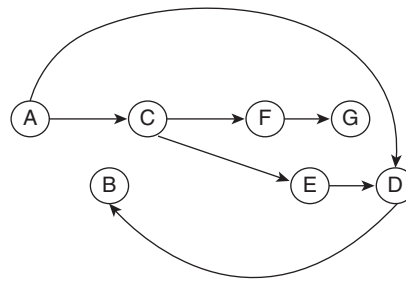
Figure 7.42 An incorrect topological sort

Algorithm 7.3 Algorithm to find topological sort of a given graph

Input: A directed graph
Output: A sequence depicting the topological sort of the graph.
Constraints: The algorithm does not work for a graph having cycles.
Procedure: Topological sort (directed graph G) returns sequence S

```
{
    While (there is :some V_j ∈ V in G:V has no incoming edge.)
    ∀V_i ∈ V in G:V having no incoming edge.
        {
        S = S ∪ {V_i};
        Remove all the outgoing edges of V_i;
        }
Return S;
}
```

Complexity: Since the algorithm performs computations on N vertices, the complexity of the topological sort is $O(|V| + |E|)$, where $|V|$ is the number of vertices and $|E|$ is the number of edges.

Example: Refer to Illustration 7.4.

Illustration 7.4 Find the topological sort of the graph depicted in Fig. 7.43.

Solution
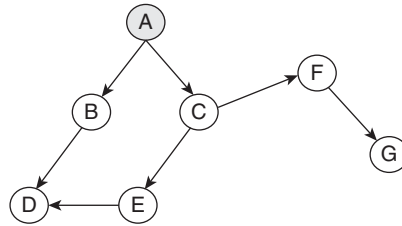
Figure 7.43 Select A, as it does
not have any incoming edge

Step 1 First of all, find all the vertices having no incoming edge. In this case, the vertex A is the one having no incoming edge. So A is the vertex that would go to the output set S (Fig. 7.43). That is, $S = S \cup \{A\}$.

Step 2 Now, remove all the outgoing vertices of A (Fig. 7.44).
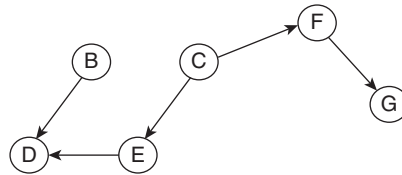
Figure 7.44 Remove all the
outgoing edges of A

Step 3 In this step, either of B or C can be selected. Here, we select B (Fig. 7.45). That is,

$$S = S \cup \{B\}$$
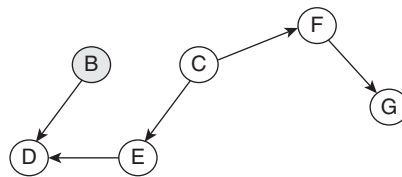
Figure 7.45 Select B, as it does not
have any incoming edge

Step 4 Now, we remove all the outgoing vertices of B (Fig. 7.46).
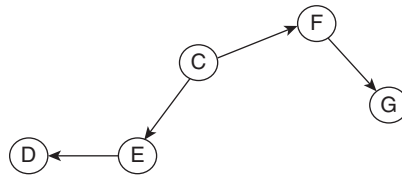
Figure 7.46 Remove all the outgoing edges of B

Step 5 In this step, C should be selected, as it does not have any incoming edge (Fig. 7.47).

$$S = S \cup \{C\}$$
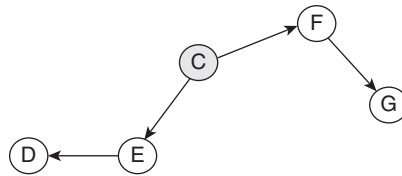
Figure 7.47 Select C, as it does not have any incoming edge

Step 6 Now, we remove all the outgoing vertices of C (Fig. 7.48).
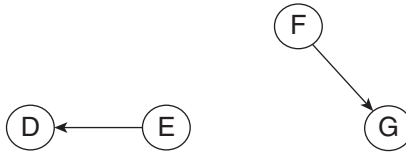
Figure 7.48 Remove all the outgoing edges of C

Step 7 Now E or F can be selected. Here, we select E (Fig. 7.49).
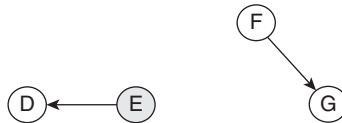
Figure 7.49 Select E, as it does not have any incoming edge

Step 8 Now, we remove all the outgoing vertices of E (Fig. 7.50).
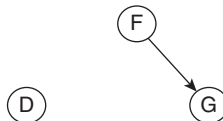
Figure 7.50 Remove all the outgoing edges of E

Step 9 In this step, D can be selected, followed by F and then G for the reasons explained above.

Therefore, the topological sort of the graph is {A, B, C, E, D, F, G}.

7.8.1 Applications of Topological Sorting

Topological sorting has many applications. Some of the applications include ordering of courses in a university. This is because each course has some prerequisites. A student should not (in some cases cannot) opt for a course until he has completed the prerequisites.

The concept is also used in detecting deadlocks. One of the reasons for deadlock is two processes are competing for resources currently held by others. In such cases, topological sorting can determine whether the resources required by a process are available. The readers are advised to explore the mechanism of formula evaluation in Excel. Topological sorting is used in this evaluation also.

7.9 SPANNING TREE

For a given graph $G = (V, E)$, the spanning tree is a connected sub-graph with no cycle, which covers all the vertices of the tree.

For example, one of the spanning trees of the graph shown in Fig. 7.51(a) are depicted in Fig. 7.51(b).
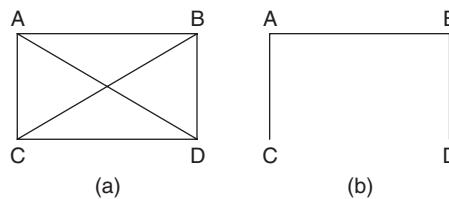
Figure 7.51 (a) Graph G; (b) spanning tree of graph G

The concept has been dealt with in Section 11.2 of Chapter 11. In fact, the concept finds applications in almost all the design techniques discussed in Section III of the book. The concept is used even in shortest path problems, discussed in Section III of the book. The applications, however, have been enlisted as follows:

• Finding shortest path
• Routing packets in a network
• Finding paths in testing and many more.

7.10 CONCLUSION

This chapter explores the concept of graph including the definition of graph, its representation, types, and applications. The linked list representation discussed in the chapter is used in traversal and sorting algorithms. The chapter also discussed the types of graphs

that are important if we want to apply the concepts in solving problems related to routing and cryptography. The traversals of graphs have been discussed and exemplified in the chapter. Connected components are generally found by using either DFS or BFS; therefore, the section has been included after discussing the former topics. The third section of the book discusses the spanning tree and shortest path algorithms (like Kruskal, Prim's, etc.) in detail. This chapter surely forms the basis of the subject and hence should be understood before proceeding further.

Points to Remember

- $G_{3,3}$ or a G_s is never planar.
- A graph can be represented using a matrix or a linked list.
- An adjacency matrix requires V^2 space, whereas adjacency list requires $(V + E)$ space.
- There can be more than one topological sorts of a graph.
- A spanning tree covers all the vertices of a graph.
- A graph that satisfies $f = e - v + 2$ (where f is the number of faces, e is the number of edges, v is the number of vertices) is a planar graph.
- DFS uses memory in a more efficient way as compared to BFS.
- In shortest path algorithms, BFS is preferred as compared to DFS.

KEY TERMS

Cycle A cyclic graph is one that has at least one path of the form $v_i, v_{i+1}, \ldots, v_i$. A graph that does not have a cycle is referred to as a non-cyclic graph.

Euler cycle A cycle that covers all the edges of a graph is called an Euler cycle.

Graph It is a non-linear data structure consisting of two components (V, E), where V is the finite, non-empty set of vertices and E is the set of edges. The set E has elements in the form $(x,y): x,y \in v$.

Hamiltonian cycle A cycle that covers all the vertices of a graph without having to cover any vertex twice, is called a Hamiltonian cycle.

Topological sorting The vertices are traversed in the order in which they appear in the graph.

EXERCISES

I. Multiple Choice Questions

1. Which of the following is true?
   - (a) Every graph is a tree
   - (b) Every tree is a graph
   - (c) Both of the above
   - (d) None of the above

2. A graph $G = (V, E)$ is a fully connected graph with $|V| = n$. How many edges does G have?

(a) $n \times \dfrac{n+2}{2}$
(c) n

(b) $n \times \dfrac{n-2}{2}$
(d) None of the above

3. A graph can be represented by which of the following?
   (a) Matrices
   (c) Both
   (b) Linked list
   (d) None of the above

4. A cycle that covers all the vertices of the given graph, without having to cover any vertex twice is called
   (a) Hamiltonian cycle
   (c) Hamming cycle
   (b) Euler cycle
   (d) None of the above

5. A cycle that covers all the edges of the given graph, without having to cover any edge twice is called
   (a) Hamiltonian cycle
   (b) Euler cycle
   (c) Hamming cycle
   (d) None of the above

6. Which of the following is not true?
   (a) Every graph has a Hamiltonian cycle
   (b) A graph can have both Euler and Hamiltonian cycle
   (c) A graph that has Euler cycle may not have a Hamiltonian cycle
   (d) None of the above

7. Which of the following is never planar?
   (a) 3×3
   (c) 1×1
   (b) 2×2
   (d) None of the above

8. Which of the following is never non-planar?
   (a) 3×3
   (c) 4×4
   (b) 2×2
   (d) None of the above

9. If a graph satisfies the equation $f = e - v + 2$, then it is
   (a) Planar
   (c) Hamiltonian
   (b) Euler
   (d) None of the above

10. Breadth first search requires which of the following?
    (a) Stack
    (c) Both
    (b) Queue
    (d) None of the above

11. Depth first search can be implemented by using
    (a) Stack
    (c) Both
    (b) Queue
    (d) None of the above

12. Which of the following is topological sorting?
    (a) Vertices of a graph are traversed in the order in which they appear in the graph
    (b) Vertices of a graph are traversed in the opposite order in which they appear in the graph
    (c) None of the above
    (d) Both of the above
13. A tree that covers all the vertices of a given graph is called a
    (a) Search tree
    (b) Parse tree
    (c) Spanning tree
    (d) None of the above
14. Which algorithm is used for graph colouring?
    (a) Welsh–Powell
    (b) Fulk Flockerson
    (c) Bellman Ford
    (d) None of the above
15. What is chromatic number?
    (a) Minimum number of colours required to colour a graph such that no two adjacent vertices have same colour
    (b) Maximum number of colours required to colour a graph such that no two adjacent vertices have same colour
    (c) Minimum number of colours required to colour a graph such that two adjacent vertices have same colour
    (d) Maximum number of colours required to colour a graph such that two adjacent vertices have same colour

II. Review Questions

1. What is a graph? How are graphs represented in computer?
2. Write the algorithm for depth first search and discuss its complexity.
3. Write the algorithm for breadth first search and discuss its complexity.
4. What is a planar graph?
5. State and prove Euler's formula for planar graph.
6. What is a face? Can a graph have infinite faces?
7. Show that a planar graph can be coloured using just five colours.
8. Show that K_s is not planar.
9. Show that DFS into all nodes in G is reachable from root node.
10. Explain the concept of connected components. How do you find out the connected components with respect to a given vertex?
11. Write an algorithm to find the minimum number of colours that can be assigned to each vertex so that no two adjacent vertices will have same colour.

Hint to the above problem is:

Welsh–Powell Algorithm

If input is graph G

   (a) Order vertices in decreasing order degree.

   (b) Assign first colour say C_1 to the first vertex then in sequential order assign next colour to each vertex.

We can assign C_1 to that vertex which is not adjacent to V_1.

III. Numerical Problems

  1. Determine whether the following graphs G1 and G2 are isomorphic.

(a)

G1   G2

(b)

G1   G2

(c)

G1   G2

(d)

G1   G2

(e)

G1

G2
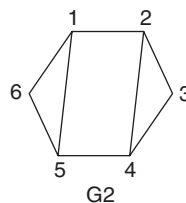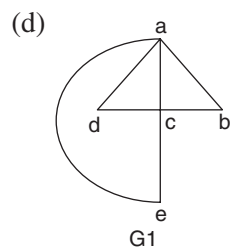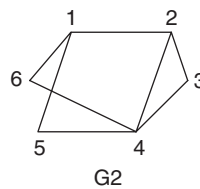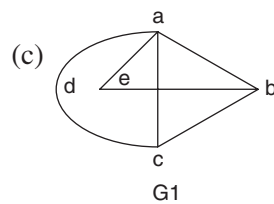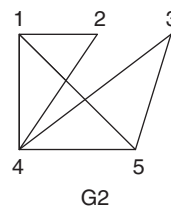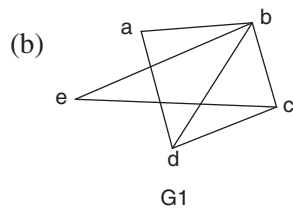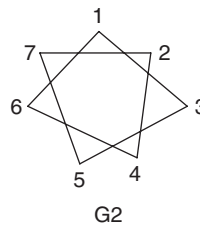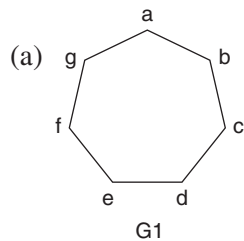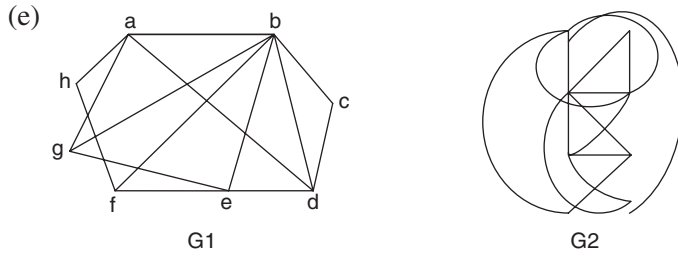
2. Redraw the following so that edges do not cross.

(a)

(b)

(c)
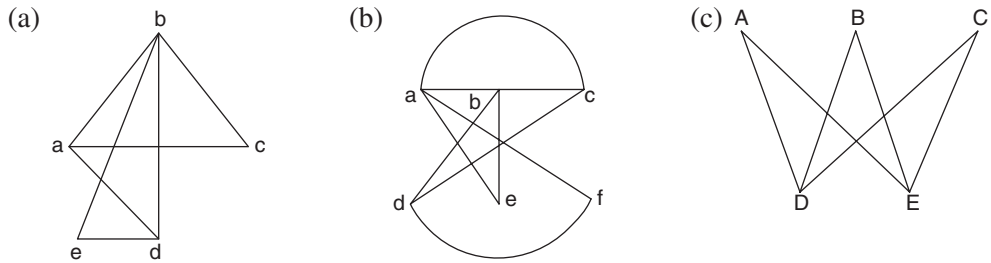
3. Show that the following graphs are not planar by reducing them to K_5 or K_{33}.

(a)

(b)
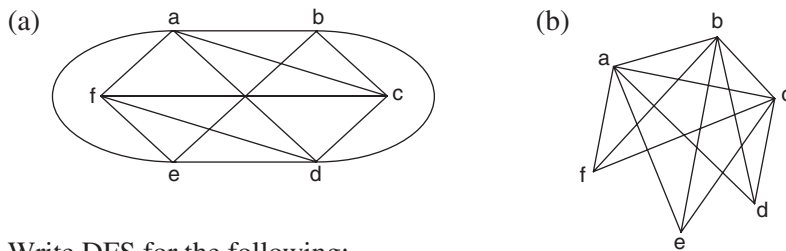
4. Write DFS for the following:
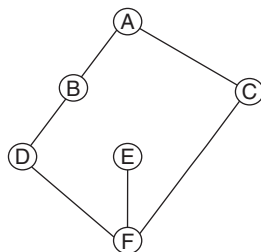
5. For the above graph (Question 4) write the BFS.

6. Write BFS for the following graph:
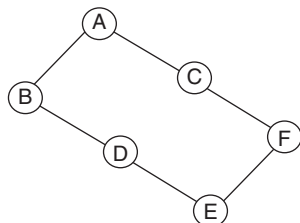
7. Write BFS and DFS for the following graph and explain each step.
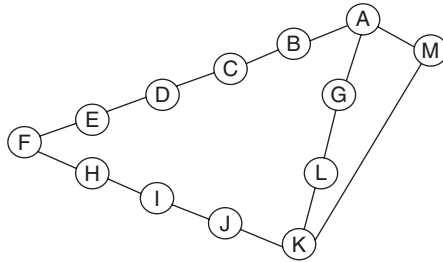
8. Use Welsh–Powell algorithm to determine upper bound to the chromatic number of the following graph.
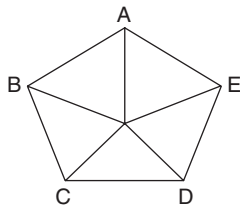
9. Find a topological sort of the following graph:
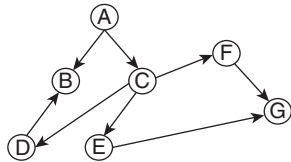
10. Find a topological sort of the following graph:
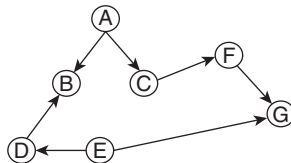
Answers to MCQs

1. (b)	4. (a)	7. (c)	10. (b)	13. (c)
2. (d)	5. (b)	8. (b)	11. (c)	14. (a)
3. (c)	6. (a)	9. (a)	12. (a)	15. (a)

CHAPTER   8

Sorting in Linear and Quadratic Time

OBJECTIVES

After studying this chapter, the reader will be able to

- Understand the concept of sorting
- Classify sorting algorithms
- Understand the concept of stability with respect to sorting
- Explain the importance of counting the number of swaps and comparisons in sorting algorithms
- Understand the procedure, complexity, and problems of the following algorithms:
  - o Selection
  - o Bubble
  - o Modified bubble
  - o Insertion
  - o Shell
  - o Counting
  - o Radix
  - o Bucket
- Compare various algorithms

8.1 INTRODUCTION

Nowadays landlines are not that important in India as they were in the 1980s and 1990s. In those days, there was something called 'Directory'. The Directory had the names and phone numbers of the residents of a city. When I was young, I used to wonder what would happen had the names in that directory not been in the sorted order. That is exactly what the importance of sorting is. Though sorting has been one of the most researched topic, it is not exactly known what would have happened had sorting not been developed by human beings. The following discussion explores this topic and presents the algorithms of the involved techniques.

Section 8.3 discusses various classifications of sorting. Sections 8.4–8.10 discuss the various sorting techniques. The techniques discussed in Sections 8.4–8.7 are quadratic as

the time complexity of these algorithms is $O(n^2)$ (at least in the average case). However, the techniques presented in Sections 8.8–8.10 are linear.

8.2 SORTING

Definition Given a list of elements $\{a_1, a_2, \ldots, a_n\}$, sorting is a procedure that rearranges the elements of the array such that for any two elements in the sorted list, a_i and a_j, $a_i < a_j$.

It may be noted that an array with a single element is deemed to be sorted.

The process is carried out to facilitate searching. A sorted array is easy to search and maintain. Binary search, for instance, can be applied only to a sorted file. Searching a file, via binary search, takes $O(\log n)$ time, whereas searching via linear search takes $O(n)$, which is considerably larger than the former (for larger values of n). For instance, if the value of n is 1024, the ratio of time taken by linear search is approximately 10 times of that taken in binary search.

8.3 CLASSIFICATION

Sorting can be classified in many ways. It can be internal or external; in place or not in place; linear or quadratic; and classification on the basis of memory requirements. In the following discussion, complexity analysis is done on the basis of number of swaps and the number of comparisons (see Fig. 8.1).
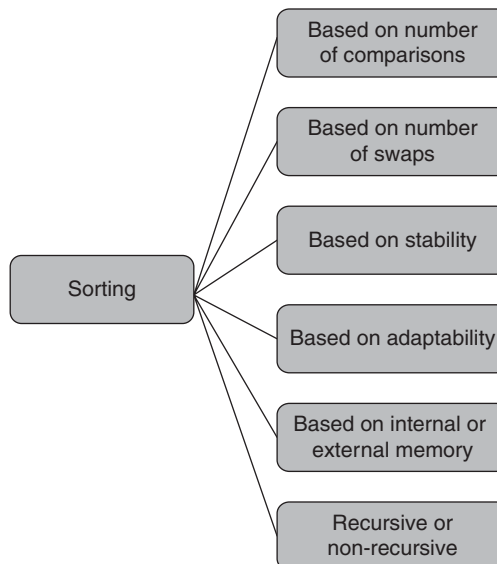
Figure 8.1 Classification of sorting

8.3.1 Classification Based on the Number of Comparisons

In this classification, the number of comparisons is the basis. For example, in the selection sort, there are $O(n^2)$ comparisons (same is the case with bubble sort).

8.3.2 Classification Based on the Number of Swaps

This can be considered as one of the criteria for deciding which algorithm is the best amongst the given algorithms that are $O(n^2)$. For instance, though both bubble sort and selection sort are $O(n^2)$ algorithms, bubble sort requires $O(n^2)$ swaps, whereas selection sort requires $O(n)$ swaps.

8.3.3 Classification Based on Memory

In this classification, the basis is the amount of memory required by the algorithm. As discussed later, an algorithm may or may not require external memory. Moreover, the extra memory required by an algorithm may depend on the number of elements present in the list. For example, merge sort requires memory proportional to the number of elements in the array. The type of memory used by a sorting algorithm is also helpful in classification. The algorithms that use main memory are referred to as *internal algorithms*, whereas those that use the secondary memory are called *external algorithms*.

8.3.4 Use of Recursion

Some algorithms, such as quick sort, use recursion. Some of them, such as selection sort, can be implemented via a non-recursive algorithm. Some, however, can be implemented using both. For example, merge sort can be implemented either by recursion or by a non-recursive algorithm (see Fig. 8.2).
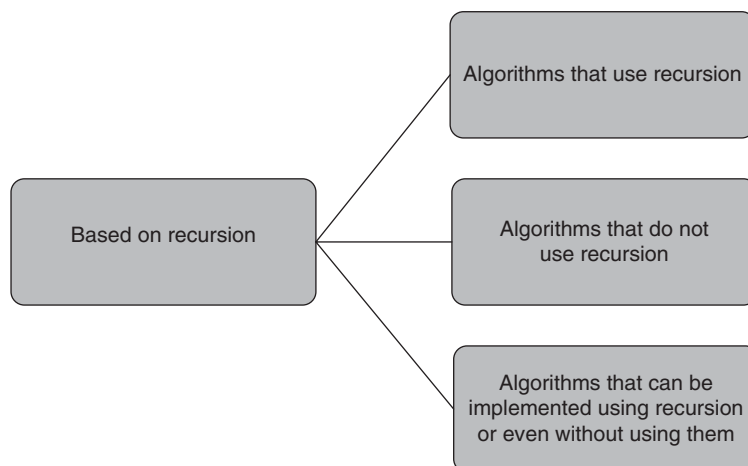
Figure 8.2 Classification based on the use of recursion

> **Tip:** Algorithms such as merge sort can be implemented both by recursion and without recursion.

8.3.5 Adaptability

Some of the algorithms perform better in specific situations. For example, the conventional bubble sort is an $O(n^2)$ algorithm irrespective of the input. However, the modified selection sort presented in the following section performs better if the input is sorted or partially sorted. So, the algorithms which consider the sortedness of the input are referred to as *adaptive algorithms*.

8.3.6 Stable Sort

At times, a given list has more than one contender for a particular position in the sorted list. In such cases, the sorted list can be different depending upon the relative positions of those contenders. If the sorted list has same position of the contenders as in the given list, it is referred to as stable; otherwise the sorting is unstable.

For example, two teams having three members each. If the relative ordering in the sorted list is preserved, like in Fig. 8.3, then it is a *stable sort*.
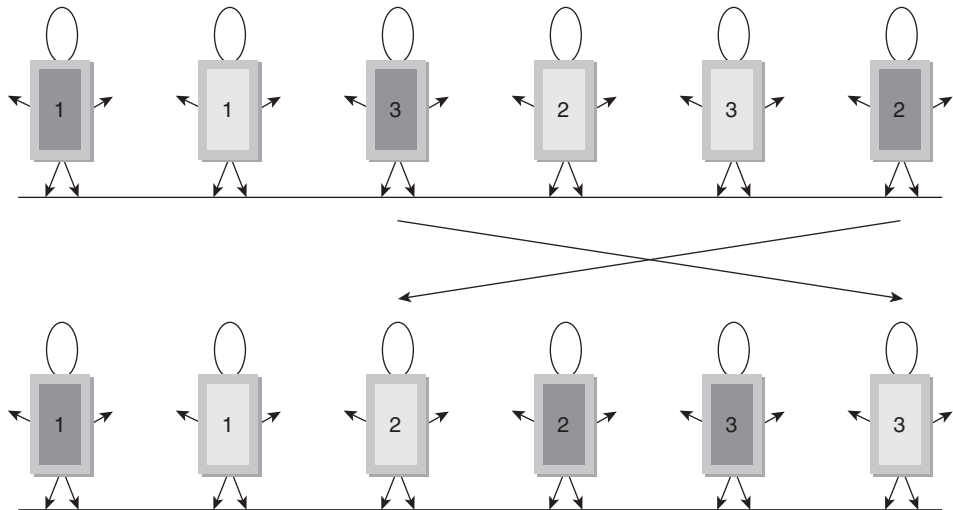
Figure 8.3 Stable sort

In the other case, wherein the relative ordering is not preserved, it is called *unstable sort*. For example, the sorting depicted in Fig. 8.4 is an unstable sort.
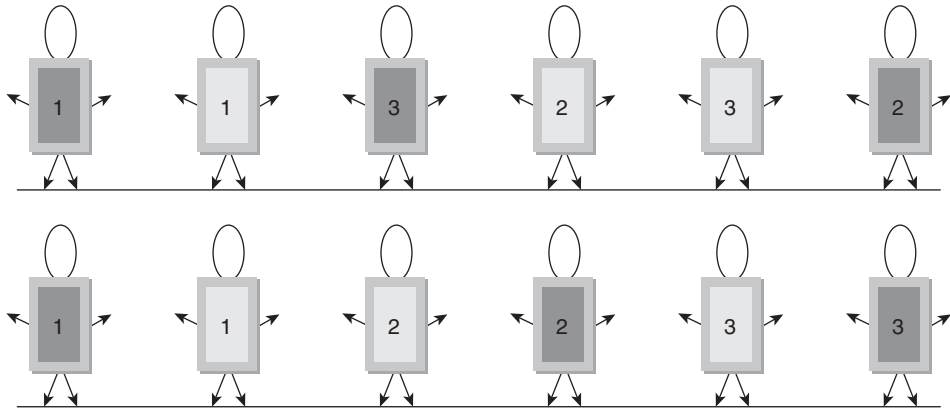
Figure 8.4 Unstable sort

The unstable sorting algorithms can be made stable by comparing the key also, this would, however, increase the time and memory complexity of the procedure. One of the questions that come in mind is regarding the utility of such algorithms. Such algorithms are good when a two-level sorting is to be implemented. The following section discusses the various sorting techniques.

8.4 SELECTION SORT

As discussed earlier, sorting means the arrangement of a set of numbers in an order. In this section, a sorting technique called selection sort has been discussed.

In selection sort, the element at the first position is compared with all other elements. The number being compared and the number, to which it is compared to, are swapped, if the number to be compared is smaller. The same procedure is repeated for the element at all the other positions. In order to understand the procedure, consider an array {7, 1, 5, 4, 9, 2, 3, 10, 8, 6}. The first element, namely 7, is compared with all other elements starting from 1. Since 1 is smaller than 7, it is swapped with 7. The same process is repeated until the element at the first position is compared with all other elements. Similarly, the elements at the second, third, and fourth positions are compared with all other elements above them. The process has been depicted in Figs 8.5–8.9.

In a similar manner, iteration 6 would place 6 at the sixth position; iteration 7 would place 7 at the seventh position; iteration 9 would place 9 at the ninth position, and 10 at the tenth position. Tenth iteration would not be needed as the only element remaining would be 10. One element, as stated earlier, cannot be unsorted.

> **Tip:** The process shown in Fig. 8.7 is not the only way of implementing selection sort. The minimum element in the remaining list can be found by divide and conquer, in each iteration. This would reduce the complexity to $O(n \log n)$.
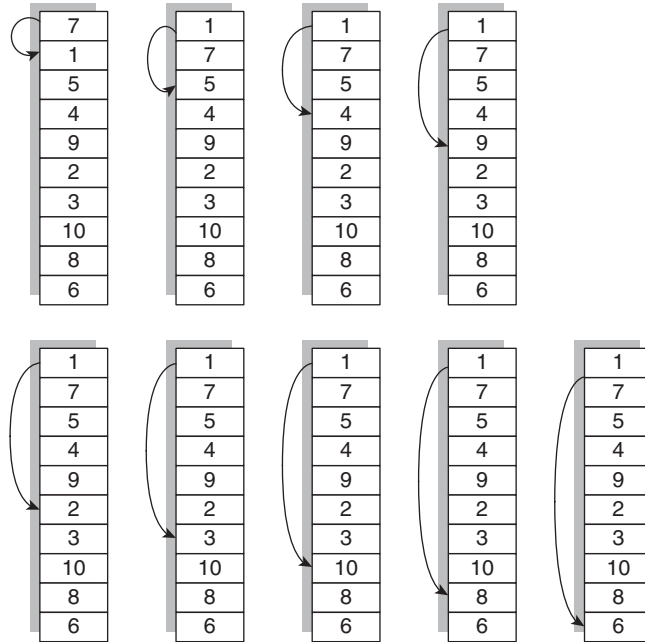
Figure 8.5 Selection sort, after iteration 1, the smallest
element comes at the first position
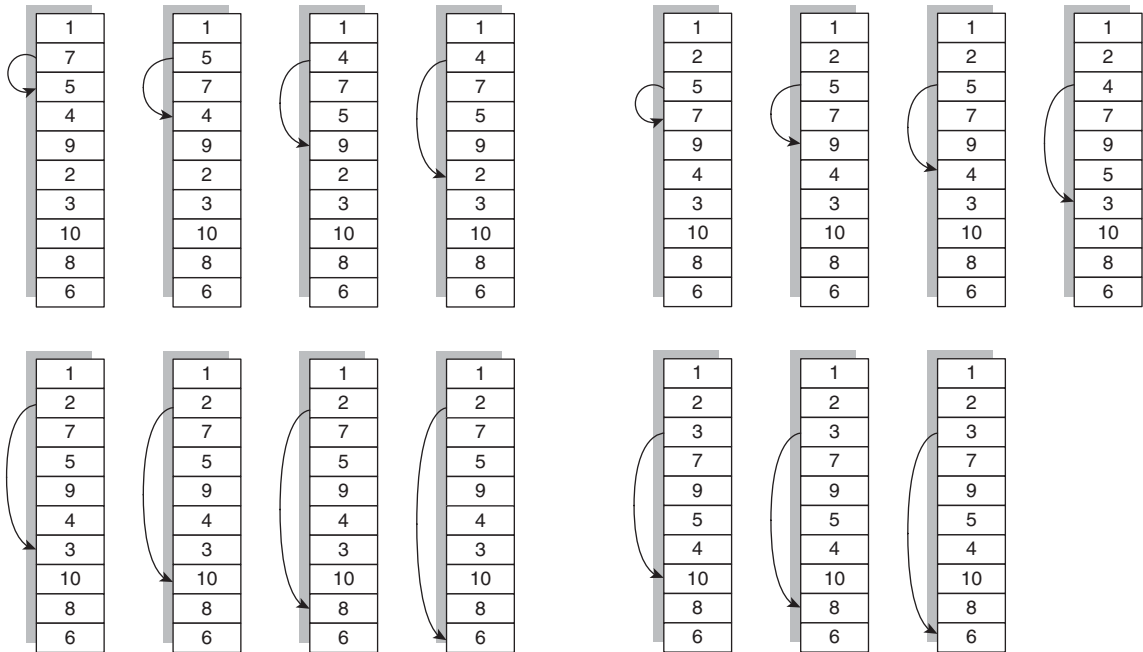
Figure 8.6 Selection sort, after iteration 2,
the second smallest element comes at the
second position

Figure 8.7 Selection sort, after iteration 3,
the third smallest element comes at the third
position
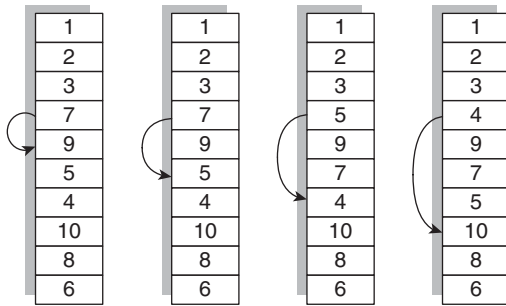
Figure 8.8 Selection sort, after iteration 4, the fourth smallest element comes at the fourth position
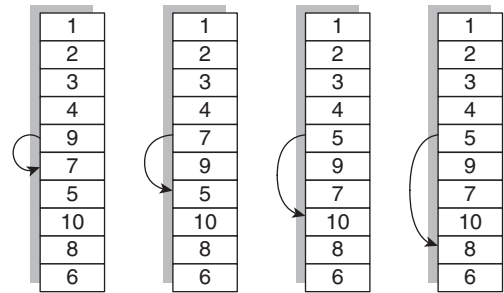
Figure 8.9 Selection sort, after iteration 5, the fifth smallest element comes at the fifth position
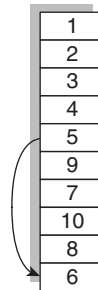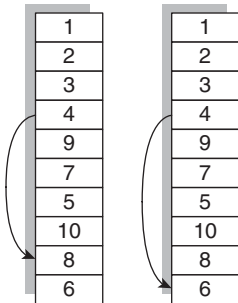
Therefore, in each iteration, the smallest element from amongst the yet unsorted elements is found and placed at its proper position. The algorithm selects the smallest element in each iteration; therefore, it is called selection sort (see Algorithm 8.1).

Algorithm 8.1 Selection sort

Input: An array 'a' containing n elements.
Output: A sorted array.
Constraints: No constraints

```
SELECTION SORT (a, n) returns a
  {
  i=0;
  // (n-1) iterations
  while(i<(n-1))
    {
    j=i+1;
    while(j<n)
      {
```

```
    if(a[j]<a[i])
      {
      temp=a[j];
      a[j]=a[i];
      a[i]=temp;
      }
    j++;
    }
  i++;
  return a;
  }
}
```

Complexity: Since there is a loop inside another loop, the first loop runs $(n − 1)$ times and for each $(n − 1)$ iteration the inner loop runs $(n − i − 1)$ times, where i is the iteration number of iterations. The complexity of the algorithm, therefore, becomes $O(n^2)$.

Number of comparisons: $2((n-1)+(n-2)+(n-3)+...1)=\dfrac{n(n-1)}{2}=\dfrac{(n^2-n)}{2}=O(n^2)$

Problem: The algorithm has a high complexity, as discussed above.

8.5 BUBBLE SORT

The technique discussed in Section 8.4 does not scale well. It has a high complexity and lesser scope for improvement. In this section, a sorting technique called bubble sort has been discussed. This section also introduces an improved version of *bubble sort* referred to as modified bubble sort.

In bubble sort, the element at the first position is taken and compared with the item at the second position. The number being compared and the number to which it is compared to are swapped, if the number to be compared is smaller. The same procedure is repeated for the element at the second and the third positions. In order to understand the procedure, consider the same array that we considered in the previous section {7, 1, 5, 4, 9, 2, 3, 10, 8, 6}. The first element, namely 7, is compared with the second element that is 1. Since 1 is smaller than 7, it is swapped with 7. The same process is repeated for the rest of the elements. The process has been depicted in Figs 8.10–8.12.

In a similar manner, iteration 4 would place fourth largest element at the fourth position from the end; iteration 7 would place fifth largest element at the fifth position from the end and so on.
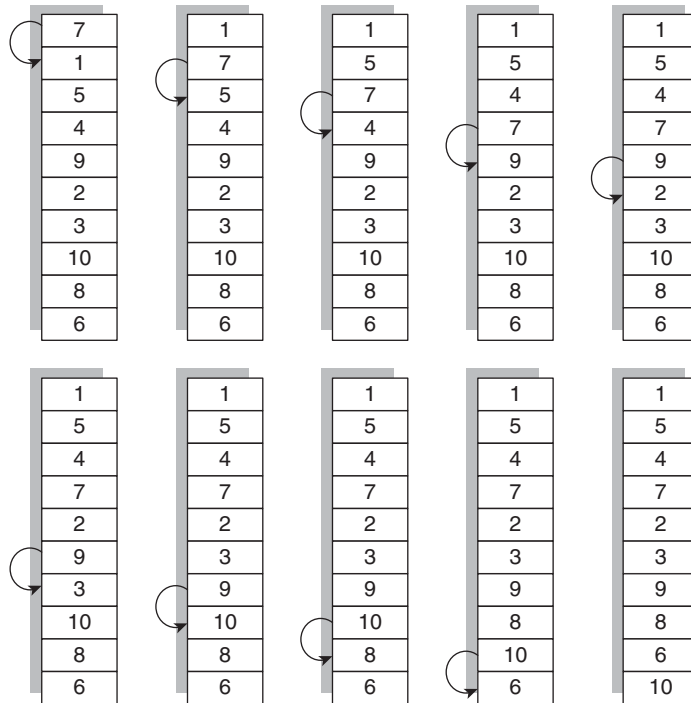
Figure 8.10 Bubble sort, after iteration 1, the largest element comes at the last position
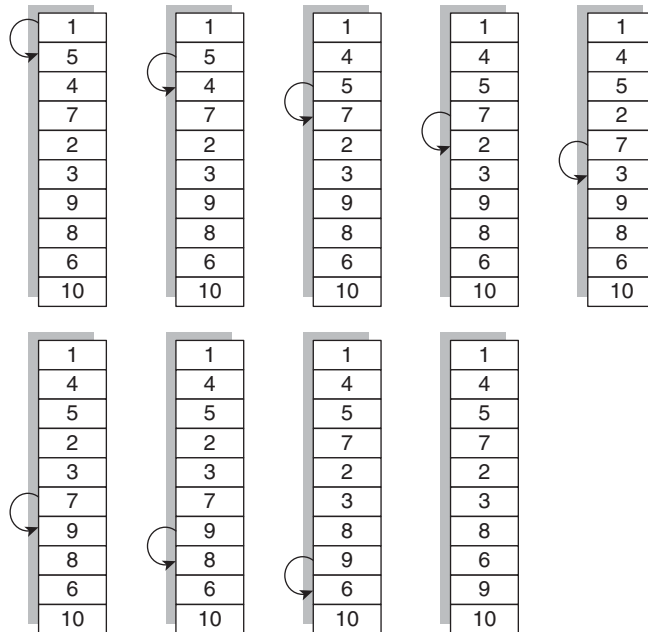
Figure 8.11 Bubble sort, after iteration 2, the second largest element comes at the second last position
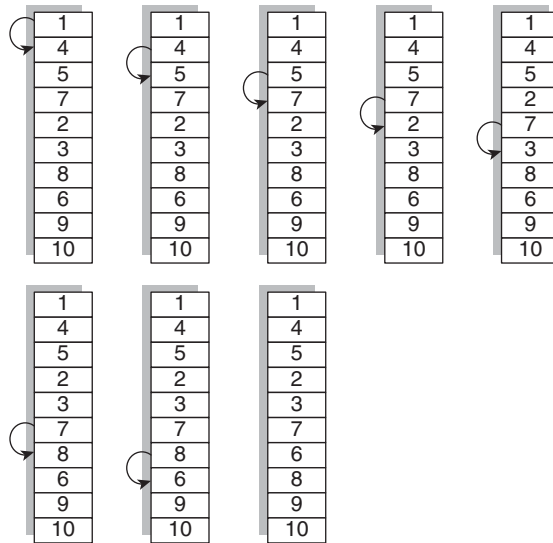
Figure 8.12 Bubble sort, after iteration 3, the third largest element comes at the third last position

In each iteration, the largest element from amongst the yet unsorted elements is found and placed at its proper position (see Algorithm 8.2).

Algorithm 8.2 Bubble sort

Input: An array 'a' containing n elements.
Output: A sorted array.
Constraints: No constraints

```
BUBBLE SORT (a, n) returns a
  {
  i=0;
  // (n-1) iterations
  while(i<(n-1))
    {
    j=0;
    while(j<n-1-i)
      {
      if(a[j+1]<a[j])
        {
        temp=a[j];
        a[j]=a[i];
        a[i]=temp;
        }
      j++;
      }
    i++;
    }
    return a;
}
```

Complexity: Since there is a loop inside another loop, the first loop runs $(n - 1)$ times and for each $(n - 1)$ iteration, the inner loop runs $(n - i - 1)$ times, where i is the iteration number iterations. The complexity of the algorithm, therefore, becomes $O(n^2)$.

Number of comparisons: $\dfrac{n^2}{2}$

Problem: The algorithm has a high complexity. However, the modified version presented in Algorithm 8.3 has a lower complexity.

$$= (n-1) + (n-2) + \cdots + 1 = \frac{n(n-1)}{2} = O(n^2)$$

> **Tip:** There are two ways of judging a sorting algorithm. This can be done either by the number of comparisons or by the number of swaps.

In the modified version of the algorithm, a flag is made which is initialized to a 0. However, inside the inner loop, if the swaps are no more there, then the sorting may be deemed as completed (Algorithm 8.3).

Algorithm 8.3   Modified bubble sort

Input: An array 'a' containing n elements.
Output: A sorted array.
Constraints: No constraints

```
MODIFIED BUBBLE SORT (a, n) returns a
  {
  i=0;
  flag=1;
  while(i<(n-1) && flag==1)
    {
    j=0;
    flag=0;
    while(j<(n-1-i)
      {
      if(a[j+1]<a[j])
        {
        temp=a[j];
        a[j]=a[j+1];
        a[j+1]=temp;
        flag=1;
        }
      j++;
      }
    i++;
    }
  return a;
    }
```

Complexity: Since there is a loop inside another loop, the first loop runs $(n-1)$ times and for each $(n-1)$ iteration, the inner loop runs $(n-i-1)$ times, where i is the iteration number iterations. The complexity of the algorithm, therefore, becomes $O(n^2)$. However, the above algorithm works well for the arrays which have already been sorted.

Number of comparisons: $\dfrac{n^2}{2}$.

8.6 INSERTION SORT

Insertion sort is a sorting technique, which inserts an item at its correct position in a partially sorted array. The technique is a well-known one as it is used in sorting playing cards. In order to understand the technique, let us consider the following example. The number cards {7, 1, 5, 4, 9, 2, 3, 10, 8, 6} are given and are required to arrange them in order.

The first card, numbered 7, comes first and is placed in the list containing sorted cards (Fig. 8.13(a)). The next card has 1 on it; 1, being lesser than 7, is placed before 7 (Fig. 8.13(b)). The third card, numbered 5, is placed between 1 and 7 (Fig. 8.13(c)). The steps of sorting have been depicted in Fig. 8.13 (see also Algorithm 8.4).

Algorithm 8.4   Insertion sort

Input: An array 'a' containing n elements.
Output: A sorted array.
Constraints: No constraints

```
INSERTION SORT (a, n) returns a
  {
  i=1;
  while(i<=(n-1))
    {
    temp=a[i];
    j=i;
    while((temp<a[j-1])&&(j>=0))
      {
      a[j]=a[j-1];
      j--;
      }
    a[j]=temp;
    i++;
    }
    return a;
  }
```

Complexity: Since there is a loop inside another loop, the first loop runs $(n-1)$ times and for each $(n-1)$ iteration, the inner loop runs for the requisite number of times. The complexity of the algorithm, therefore, becomes $O(n^2)$. As a matter of fact, the worst-case, best-case, and the average-case complexities of this algorithm is $O(n^2)$.
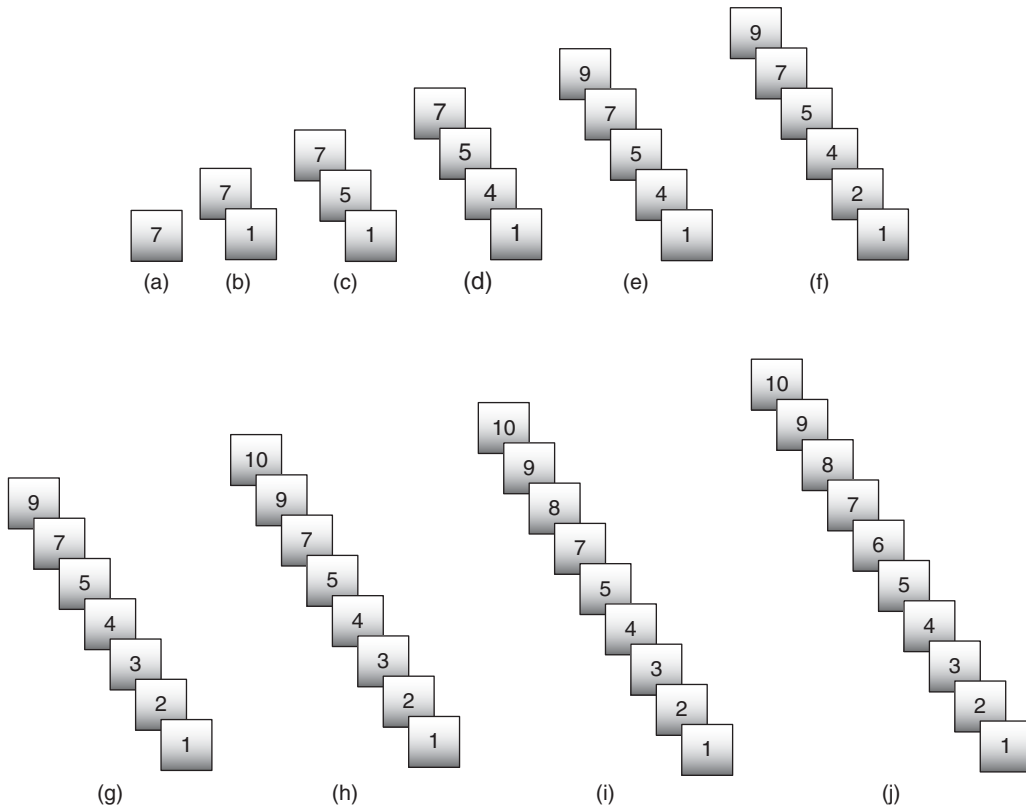
Figure 8.13 Figure depicting the steps of insertion sort

Number of comparisons: $\dfrac{n^2}{4}$ in the average case.

Problem: The algorithm has a high complexity.

8.7 DIMINISHING INCREMENTAL SORT

This sorting technique is also called *Shell sort* as it was invented by Donald Shell. The technique discussed in the previous section works well for an array, which is already sorted. This technique is a generalization of the previous technique. As a matter of fact the algorithm is same as insertion sort in the last step. However, till that time most of the inversions have already been done (see Algorithm 8.5).

Algorithm 8.5 Shell sort

```
function shell_sort(arr,n){
gap = floor(n/3);
while(gap > 0)
```

```
{
for(i = gap; i < n; i++)
  {
    temp = arr[i];
   j = i;
    while(j >= gap && arr[j - gap] > temp)
 {
 arr[j] = arr[j - gap];
    j -= gap;
}
arr[j] = temp;
}
gap = floor(gap/2);
}
  return arr;
  }
```

Complexity: The worst-case complexity of the above algorithm is $O(n^2)$. However, it is best of all the $O(n^2)$ algorithms. As a matter of fact, the number of comparisons goes on decreasing as and when we proceed.

Problem: The algorithm has a high complexity.

8.8 COUNTING SORT

The elements of the given array are up to a given number k. A temporary array, temp_array[], is taken. This keeps track of the number of elements before a[i]. This helps to place a[i] at its requisite position. The elements of the array, temp_array[], are initially set to 0. This is followed by keeping a[i] at the position depicted by its value, in the temp_array. The number of elements before a[i] is then calculated. In order to understand the algorithm, let us consider an example. The input array, for instance, is {2, 8, 3, 6, 10}. The length of the temporary array, in this case, would be 11, as the maximum element in the array is 10 (see Fig. 8.14, Algorithm 8.6).

Algorithm 8.6 Counting sort

Input: a[], int n (number of elements)
Output: b[], the sorted array
Temporary memory: temp_array[], having k elements where k is the maximum element of the array.
```
Counting Sort (a[], n) returns sorted_array[]
  {
  for(i=0;i<k;i++)
    {
```

```
        temp_array[i]=0;
        }
    for(i=0;i<n;i++)
        {
        temp_array[a[i]]=temp_array[a[i]]+1;
        }
    for(i=1;i<k;i++)
        {
        temp_aray[i]=temp_array[i]+temp_array[i-1];
        }
    for(i=n-1;i>=0;i--)
        {
        temp_array[a[i]]=temp_array[a[i]]-1;
        sorted_array[temp_array[a[i]]]=10
        }
    }
```

Complexity: $O(n)$.
Problem: Extra space needed
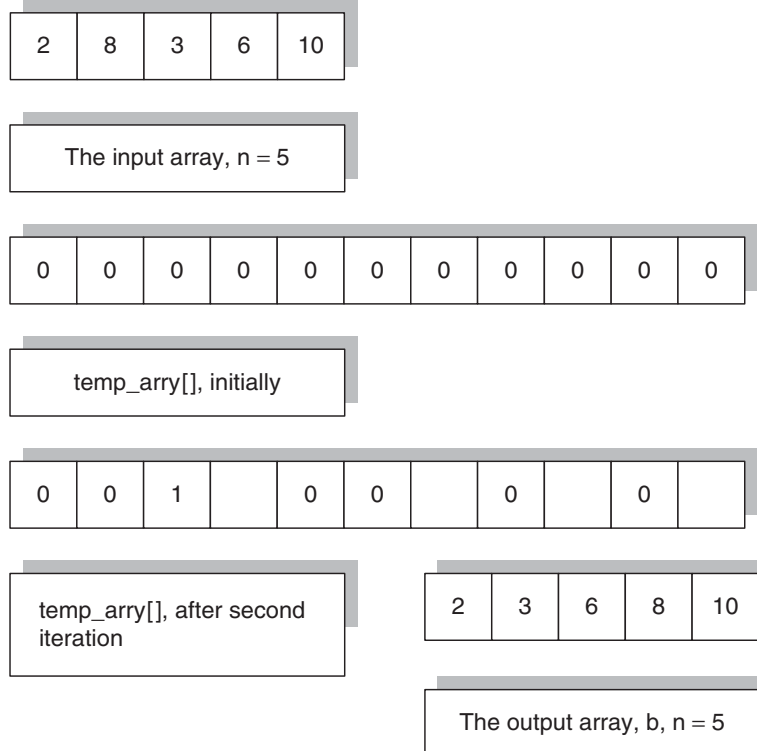
Figure 8.14 Counting sort

8.9 RADIX SORT

Radix sort algorithm sorts the elements of a given set by considering the digits at various places. The process can be understood by taking an example. If the given set is {23, 34, 67, 89, 123, 63, 39, 212, 90}, then the following steps must be followed in order to generate the sorted set.

Step 1 Place elements according to units place in holders having indices from 0 to 9. Figure 8.15 depicts the first step.
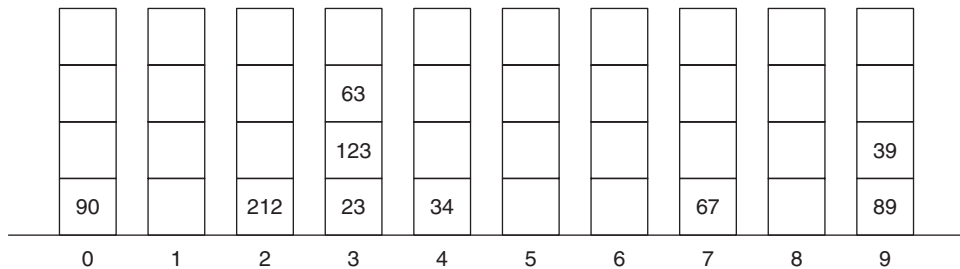
Figure 8.15 Radix sort, elements placed in order of their unit's place

Next, we read the list from left to right. The output of this step would be {90, 212, 23, 123, 63, 34, 67, 89, 39}.

In the next step, the above array would be arranged in order of their tens place. The situation is depicted in Fig. 8.16.
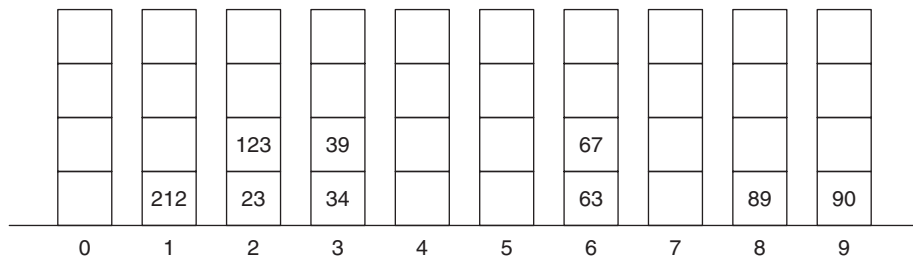
Figure 8.16 Radix sort, elements placed in order of their tens' place

Next, we read the list from left to right. The output of this step would be {212, 23, 123, 34, 39, 63, 67, 89, 90}.

In the next step, the above array would be arranged in order of their hundreds place. The situation is depicted in Fig. 8.17.
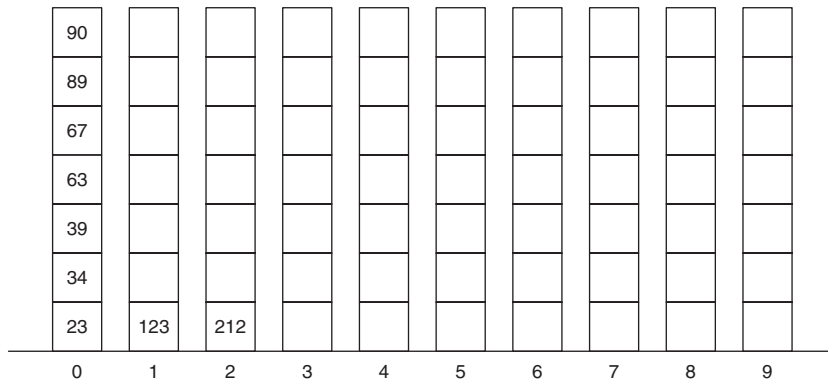
Figure 8.17 Radix sort, elements placed in order of their hundreds place

Next, we read the list from left to right. The output of this step would be {23, 34, 39, 63, 67, 89, 90, 123, 212} (see Algorithm 8.7).

Algorithm 8.7 Radix sort(a[], n) returns a sorted array

```
k=number of digits in the maximum element of the given array
for(i=1;i<=k;i++)
  {
  Sort the list in accordance with digit di.
  }
```

Complexity: There are k iterations, the complexity of the above algorithm is, therefore, $O(N)$.

Problem: The algorithm needs 9 arrays. The number of elements in the array should be at least the maximum number of j digit numbers in the list, for all j.

8.10 BUCKET SORT

Bucket sort is a sorting technique that takes $O(n)$ time. As per the literature review, it has also been considered as a version of counting sort. The algorithm assumes that the elements of the given array are in the range $[0, k - 1]$ (see Algorithm 8.8). The process depicted in the algorithm requires ith element to be put into its corresponding bucket and then reading the data structure depicting the buckets.

Algorithm 8.8 Bucket sort(a[], n) returns a sorted array

```
{
k= the maximum number in the given array.
int Bucks[k];
//Bucks is the array depicting the buckets
//initialize the elements of Bucks to -1
```

```
for(i=0;i<k;i++)
  {
  Bucks[i]=-1;
  }
for(i=0;i<n;i++)
  {
  Bucks[a[i]]=a[i];
  }
j=0;
for(i=0;i<k;i++)
  {
  if(Bucks[i]!=-1)
    {
    b[j]=Bucks[i];
    j++;
    }
  }
  return b[];
}
```

Complexity: The complexity of the above algorithm is $O(n)$, as there are loops which run one after another, however, there are no nested loops.

Problem: The requirement of extra memory makes the above algorithm a bit repulsive.

8.11 CONCLUSION

The discussion in this chapter focuses on sorting algorithms having linear or quadratic time complexity. However, there are some more sorting algorithms such as merge sort and quick sort, which have been discussed in Chapter 9. The technique called heapsort has been discussed in Section 6.12 of Chapter 6.

Table 8.1 presents the time complexity and important attributes of various algorithms presented in the chapter.

Table 8.1   Summary of sorting algorithms

Algorithm	Worst-case time complexity	Linear/ quadratic	Stable/not stable
Selection	$O(n^2)$	Quadratic	No
Bubble	$O(n^2)$	Quadratic	Yes
Insertion	$O(n^2)$	Quadratic	Yes
Shell	$O(n^2)$	Quadratic	No
Counting	$O(n)$	Linear	
Radix	$O(n)$	Linear	
Bucket	$O(n)$	Linear	

The chapter forms the basis of data structures and algorithms and is amongst the most important topics of the discipline. The questions on this topic are amongst the favourites in interviews. The web resources of this book contain the implementations of all the algorithms discussed in the chapter in C language.

Points to Remember

- In the classification based on number of comparisons, the number of comparisons is the basis.
- In the classification based on the number of swaps, the efficiency of an algorithm depends on the number of swaps.
- In classification based on memory, the basis is the amount of memory required by the algorithm.
- The sorting algorithms can also be classified on the basis of recursion. Some algorithms, such as quick sort use recursion. Some of them such as selection sort can be implemented via a non-recursive algorithm.
- Some of the algorithms perform better in specific situations.
- In some situations, a given list has more than one contender for a particular position in the sorted list. In such cases, the sorted list can be different depending upon the relative positions of those contenders.

KEY TERMS

Adaptive sorting algorithms The algorithms which consider the 'sortedness' of the input are referred to as adaptive.

Insertion sort It is a sorting technique, which inserts an item at its correct position in a partially sorted array.

Selection sort The process of selection of least (or greatest) element from the remaining list and replacing it with the current element is called selection sort.

Sorting Given a list of elements $\{a_1, a_2, \ldots, a_n\}$, sorting is a procedure which rearranges the elements of the array such that for any two elements, in the sorted list, a_i and a_j, $a_i < a_j$ if $i < j$.

Stable/unstable sort At times, the sorted list can be different depending upon the relative positions of those contenders. If the sorted list has same position of the contenders as the given list, it is referred to as stable otherwise the sorting is unstable.

EXERCISES

I. Multiple Choice Questions

1. In which of the following sorting would be helpful?
   - (a) Searching
   - (b) Merging
   - (c) Nursing
   - (d) None of the above

2. Which of the following can be a criteria (or criterion) for judging the efficiency of a sorting algorithm?
   (a) Number of swaps
   (b) Number of comparisons
   (c) Stability
   (d) All of the above

3. Which of the following does not require auxiliary memory?
   (a) Merge sort
   (b) Quick sort
   (c) None of the above
   (d) Both of the above

4. Which of the following uses main memory?
   (a) Internal
   (b) External
   (c) Both of the above
   (d) None of the above

5. Which of the following uses external memory?
   (a) Internal
   (b) External
   (c) Both of the above
   (d) None of the above

6. What is the best-case complexity of modified bucket sort?
   (a) $O(n)$
   (b) $O(n^2)$
   (c) $O(n \log n)$
   (d) None of the above

7. Which of the following find the minimum value in the given array and swaps it with the current position?
   (a) Selection
   (b) Bubble
   (c) Insertion
   (d) None of the above

8. Which of the following requires $\frac{n^2}{2}$ swaps both for the average and the worst case?
   (a) Bubble
   (b) Selection
   (c) Insertion
   (d) All of the above

9. Which of the following requires n swaps?
   (a) Selection
   (b) Bubble
   (c) Insertion
   (d) All of the above

10. Which one of the following is also called diminishing incremental sort?
    (a) Insertion
    (b) Shell
    (c) Bubble
    (d) Selection

11. What is the worst-case complexity of best known Shell sort?
    (a) $O(n)$
    (b) $O(n \log n)$
    (c) $O(n(\log n)^2)$
    (d) None of the above

12. Which one of the following is stable?
    (a) Selection
    (b) Shell
    (c) Heap
    (d) Bubble

13. Which of the following is not stable?
    (a) Selection
    (b) Bubble
    (c) Insertion
    (d) Merge

14. Which of the following is not an $O(n)$ algorithm?
    (a) Counting sort
    (b) Bucket sort
    (c) Radix sort
    (d) Selection sort

15. 128 MB data is to be sorted using a 16 MB RAM. Which of the following solution would help us to accomplish the task?
    (a) A combination of quick sort and merge sort
    (b) A combination of selection and bubble
    (c) The task cannot be accomplished without more memory
    (d) Counting sort
16. A picture has 128 colours. The number of pixels in the picture is 10,000. Which sorting algorithm can be used to sort the pixels most efficiently?
    (a) Insertion sort   (c) Selection sort
    (b) Bubble sort   (d) Counting sort

II. Review Questions

1. Explain the procedure of bubble sort. What modification is needed to improve the efficiency of conventional bubble sort?
2. Explain the procedure of selection sort. Why is selection sort considered better than bubble sort?
3. Explain the procedure of insertion sort. Calculate the number of swaps and comparisons in insertion sort.
4. Explain the procedure of shell sort. Why is Shell sort better than bubble sort?
5. Explain the procedure of counting sort. Why is it a linear sorting algorithm?
6. Explain the procedure of radix sort. How does it depend on the number of digits of the given list?
7. Differentiate between internal and external sorting.

III. Numerical Problems

1. Sort the following lists and calculate the number of swaps and comparisons required in each case:
    (a) 1, 100, 1000, 10,000, 100,000, 1,000,000
    (b) 1, 3, 5, 7, 9, 11
    (c) 1, 2, 1, 5, 1, 7, 8, 1, 9, 10, 1, 23, 1
    (d) 1, 1, 1, 1, 1, 1, 1, … (2048 times)
    By
    (e) Bubble sort   (h) Counting sort
    (f) Insertion sort   (i) Radix sort
    (g) Selection sort   (j) Shell sort

Answers to MCQs

1. (a)	3. (b)	5. (b)	7. (a)	9. (a)	11. (c)	13. (a)	15. (d)
2. (d)	4. (a)	6. (a)	8. (a)	10. (b)	12. (d)	14. (d)	16. (d)

SECTION III

DESIGN TECHNIQUES

If you optimize everything, you will always be unhappy.

— Donald Knuth

Divide and Conquer

OBJECTIVES

After studying this chapter, the reader will be able to
- Understand the concept of divide and conquer (D&C)
- Explain Master theorem and apply it to solve recursive equations
- Find the maximum and minimum number from a list using D&C
- Infer quick sort and find its complexity in all cases
- Interpret the algorithm, analysis, and complexity of merge sort
- Recognize the procedure and complexity of the following problems:
  - o Finding the pair having closest distance
  - o Convex hull
  - o Selection

9.1 INTRODUCTION

The term 'divide and conquer' is generally used in the sociological context. It generally refers to the strategy of breaking the unity in a given population in order to achieve some intended goal. It is easy to psychologically handle and manipulate a smaller group rather than handling a larger one. The divide and conquer strategy used in algorithms works only if there is a way to club together the solutions of the sub-problems. In contrast, the strategy used in sociological context does not require a strategy to merge the segregated population.

9.2 CONCEPT OF DIVIDE AND CONQUER

In order to accomplish a task using the above technique, the following steps need to be carried out:

Step 1 Divide the domain into SMALL (atomic level), where SMALL is the basic unit whose solution is known.
Step 2 Solve individual sub-problems using the solution of SMALL.
Step 3 Combine the sub-solutions to get the final answer.

In order to understand the above process, let us consider an example shown in Algorithm 9.1. An array of *n* numbers is given and it is required to find the maximum element of the array using divide and conquer approach.

Algorithm 9.1   MAX

Input: An array a[], consisting of *n* elements.
Output: The maximum element of the array, MAX.
Strategy: Divide the array into two parts and continue dividing the sub-parts till one element remains in the array. For example, an array of eight elements would be divided into two arrays of four elements. The two arrays would then be divided into four arrays of two elements each and in the last step, the four arrays would be divided into eight arrays of one element each.

    Now the largest element from two arrays (containing a single element) would be the solution of the sub-problem. The procedure is repeated for the rest of the elements also. After this step, four elements would remain. Now, these four elements would be paired in groups of two, to give two elements which are greatest in their respective groups. This is followed by the selection of the largest element.

```
Find_Max( int[] a, low, high) returns max {
// a[] is an integer array having n elements, low is the first index and high is
the last index of the array. //max is the maximum value of the array.
if (low == high)
        {
        max = a [low];
        return max;
        //if the array has just one element return it
        }
else
        {
        mid =(low +high)/2;
        int x= FindMax(a[], low, mid);
        int y = FindMax (a[], mid+1, high);
        if(x> y)
                {
                return x;
                }
        else
                {
                return y;
                }
        }
//return the maximum of the left and the right sub-array.
}
```

Complexity: Initially, there are n elements, after the first iteration, there would be $n/2$ elements in each array. The process stops when a single element remains in the array. This would require $\log_2 n$ steps. After this a single element is selected in each step. So, the complexity is proportional to $\log_2 n$, i.e.,

$$T(n) = O(\log_2 n)$$

Figure 9.1 illustrates the implementation of the above algorithm.
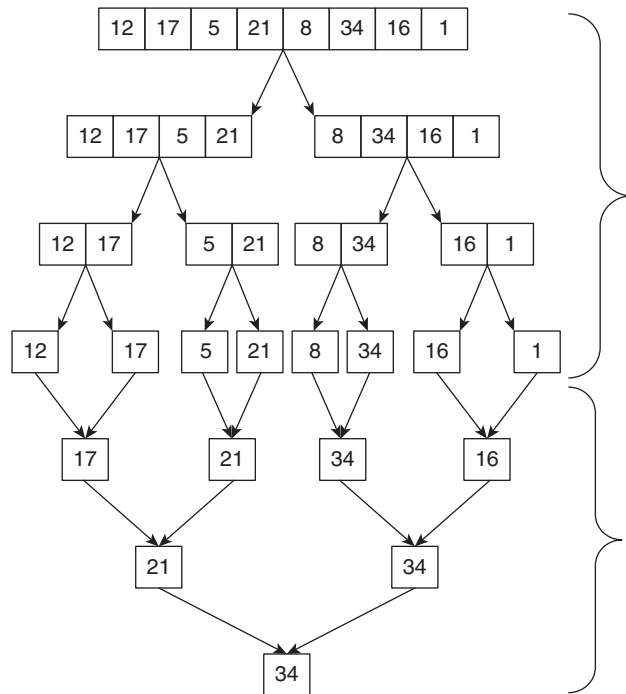
Figure 9.1 Finding the maximum element by divide and conquer

Applications: The above strategy can also be used to calculate the minimum from the array. In order to calculate the minimum value, instead of the maximum, choose the minimum at each step. Figure 9.2 illustrates the process of selection of minimum element using divide and conquer approach.

The rest of the chapter relies heavily on recursive equations and their solutions. Therefore, it is important to understand the procedures of solving them. Section 9.3 discusses Master theorem that helps us to find time complexity for such equations.
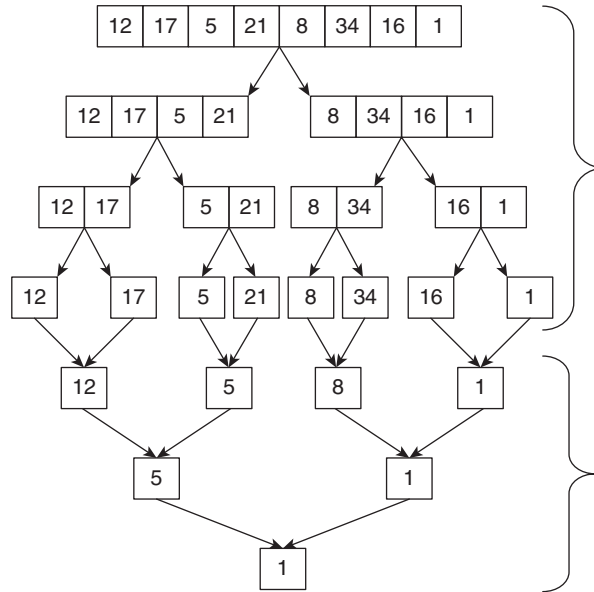
Figure 9.2   Finding minimum element by divide and conquer approach

9.3 MASTER THEOREM

The method of substitution and recursion trees, discussed in Chapter 4, helps us to solve the recursive equations but is tedious. The other method to solve the recursive equations is by Master theorem. Master theorem can be used to solve many recursive equations without having to draw the recursion tree or using substitution. In fact, the solution of the equations becomes easy using the theorem. Although the theorem can be applied only to a particular type of recursive equation and the theorem requires learning the three cases, it still makes the work easy.

The Master theorem can be employed to solve the recursive equations of the form:

$$T(n) = a \times T\left(\frac{n}{b}\right) + f(n) \tag{9.1}$$

where $f(n)$ is asymptotically positive, $a \geq 1$ and $b > 1$.

The theorem is used to find the asymptotic bounds of $T(n)$. According to the theorem,

Case 1 If $f(n) = \theta(n^{\log_b a})$, then $T(n) = \theta(\log n \times n^{\log_b a})$

Case 2 If $f(n) = O(n^{(\log_b a) - s} - \varepsilon)$, then $T(n) = \theta(n^{\log_b a})$, where ε is a positive number.

Case 3 If $f(n) = \Omega(n^{(\log_b a) + s} - \varepsilon)$, then $T(n) = \theta(f(n))$, where ε is a positive number.

The theorem is not technically correct as n/b may not always be an integer. However, writing n/b or using ceiling or floor will give the same answer as far as asymptotic notation is concerned. In order to understand the above theorem, let us examine a few illustrations. The three illustrations that follow depict each case. Illustration 9.1 depicts Case 1, Illustration 9.2 depicts Case 2, and Illustration 9.3 exemplifies the third case.

Illustration 9.1 The recursive equation of time complexity of an algorithm is given by

$$T(n) = 4 \times T\left(\frac{n}{2}\right) + n^2, \text{ find the asymptotic bounds of } T(n).$$

Solution In the given equation, the value of 'a' is 4, the value of 'b' is 2, and the value of $f(n) = n^2$. In this equation, $n^{\log_b a} = n^{\log_2 4} = n^2$.

Since, $f(n) = n^{\log_b^a}$, $T(n) = \theta\left(\log n \times n^{\log_b a}\right) = \theta\left(\log n \times n^{\log_2 4}\right) = \theta\left(\log n \times n^2\right)$

Illustration 9.2 The recursive equation of time complexity of an algorithm is given by

$$T(n) = 4 \times T\left(\frac{n}{2}\right) + n, \text{ find the asymptotic bounds of } T(n).$$

Solution In the given equation, the value of 'a' is 4, the value of 'b' is 2, and the value of $f(n) = n$. In this equation, $n^{\log_b a} = n^{\log_2 4} = n^2$.

Since, $\qquad f(n) = n^{\log_b a - 1}, T(n) = \theta\left(n^{\log_b a}\right) = \theta\left(n^{\log_2 4}\right) = \theta\left(n^2\right)$

Illustration 9.3 The recursive equation of time complexity of an algorithm is given by

$$T(n) = 4 \times T\left(\frac{n}{2}\right) + n^3, \text{ find the asymptotic bounds of } T(n).$$

Solution In the given equation, the value of 'a' is 4, the value of 'b' is 2, and the value of $f(n) = n^3$. In this equation $n^{\log_b a} = n^{\log_2 4} = n^2$.

Since, $\qquad f(n) = n^{\log_b a + 1}, T(n) = \theta(f(n)) = \theta(n^3)$

Having seen one instance of each of the above three cases, let us now understand the meaning of Eq. (9.1). The equation can be perceived as the time required to divide a problem of size n, into a sub-problems of size n/b and then combining the results of those sub-problems in time $f(n)$. The strategy is same as that followed in divide and conquer (D&C) approach, discussed earlier in the chapter.

As stated earlier, not every equation can be solved using the Master theorem. Examples of some equations that cannot be solved using Master theorem are as follows:

$$T(n) = n \times T\left(\frac{n}{2}\right) + n^2: \text{ The value of '} a \text{' is } n. \text{ So, '} a \text{' is not constant.}$$

- $T(n) = 3T\left(\dfrac{n}{2}\right) + \dfrac{n}{\log n}$: The difference between $f(n)$ and $n^{\log_b a}$ should be polynomial

- $T(n) = 2nT\left(\dfrac{n}{3}\right) + n^4$: 'a' is not constant

- $T(n) = 3T\left(\dfrac{n}{2}\right) - n$: $f(n)$ is not positive

The application of Master theorem in finding the time complexity of an algorithm can be understood by taking an example of binary search (Illustration 9.4) and merge sort (Illustration 9.5).

Illustration 9.4 Find the complexity of binary search using Master theorem.

Solution In binary search, an array consisting of n elements is divided into two arrays of ($n/2$) elements each. The time required to combine the results is $O(1)$. The value of 'a', therefore, becomes 1 and the value of 'b' becomes '2'. Hence, the value of $n^{\log_b a}$ is $n^0 = 1$. Since, the value of $f(n)$ is $O(1)$, the time complexity of binary search becomes $\theta(\log_2 n)$.

Illustration 9.5 In the case of merge sort, wherein the equation of $T(n)$ is $T(n) = 2 \times T\left(\dfrac{n}{2}\right) + n^1$, find the asymptotic bounds of $T(n)$.

Solution In the equation, the value of 'a' is 2, the value of 'b' is 2,
    The value of $n^{\log_b a}$ becomes n^1 and $f(n)$ is n^1.
    Since $f(n)$ and $n^{\log_b a}$ are same, the complexity becomes $\theta(n \times \log_2 n)$.

Illustration 9.6 Solve the following recursive equation by Master theorem:

$$T(n) = 3T\left(\dfrac{n}{2}\right) + n^2$$

Solution The value of 'a' = 3, 'b' is 2, and $f(n)$ is n^2.

The value of $\qquad\qquad\qquad\qquad \log_b a = \log_2 3$

Since $\qquad\qquad\qquad\qquad\qquad n^{\log_2 3} < n^2$

$$T(n) = \theta(n^2)$$

Illustration 9.7 Solve the following recursive equation by Master theorem:

$$T(n) = 2T\left(\dfrac{n}{3}\right) + n^2$$

Solution The value of 'a' = 2, 'b' = 3, and $f(n) = n^2$.

The value of $$\log_b a = \log_3 2$$

Since $$n^{\log_3 2} < n^2 \text{ (for large } n\text{'s)}$$

$$T(n) = \theta(n^2)$$

Illustration 9.8 Solve the following recursive equation by Master theorem:

$$T(n) = T\left(\frac{n}{2}\right) + 7n$$

Solution The value of 'a' = 1, 'b' = 2, and $f(n) = 7n$. The value of $\log_b a = \log_2 1$. Since $n^0 < n$,

$$T(n) = \theta(n)$$

Illustration 9.9 Solve the following recursive equation by Master theorem:

$$T(n) = 2^n T\left(\frac{n}{2}\right) + n^n$$

Solution Since 'a' is not constant, Master theorem cannot be applied.

Illustration 9.10 Solve the following recursive equation by Master theorem:

$$T(n) = 27T(n/3) + n$$

Solution The value of 'a' = 27, 'b' = 3, and $f(n) = n$

The value of $$\log_b a = \log_3 27 = 3$$

Since $$n^3 > n \text{ (for large } n\text{'s)}$$

$$T(n) = \theta(n^3)$$

Illustration 9.11 Solve the following recursive equation by Master theorem:

$$T(n) = 2T(n/2) + n$$

Solution The value of 'a' = 2, 'b' = 2, and $f(n) = n$.

The value of $$\log_b a = \log_2 2 = 1$$

Since

$$\therefore \qquad T(n) = \theta(n \log n)$$

Illustration 9.12 Solve the following recursive equation by Master theorem:

$$T(n) = 2T(n/2) + n \log n$$

Solution Since the difference between $f(n)$ and $n^{\log_b a}$ is not polynomial, Master theorem cannot be applied. The equation, however, can be solved using substitution (refer to Chapter 4).

Illustration 9.13 Solve the following recursive equation by Master theorem:

$$T(n) = 2T(n/4) + \sqrt{n}$$

Solution The value of 'a' = 2, 'b' = 4, and $f(n) = \sqrt{n}$.

The value of $$\log_b a = \log_2 4 = 2$$

Since $$n^2 > \sqrt{n} \ \text{(for large } n\text{'s)}$$

$$T(n) = \theta(n^2)$$

Illustration 9.14 Solve the following recursive equation by Master theorem:

$$T(n) = \left(\frac{1}{2}\right) T \left(\frac{n}{2}\right) + \frac{1}{n}$$

Solution Again, the difference between $f(n)$ and $n^{\log_b a}$ is not a polynomial, Master theorem cannot be applied.

Illustration 9.15 Solve the following recursive equation by Master theorem:

$$T(n) = 16T(n/4) + n!$$

Solution Here also, the difference between $f(n)$ and $n^{\log_b a}$ is not polynomial, and hence Master theorem cannot be applied.

Illustration 9.16 Solve the following recursive equation by Master theorem:

$$T(n) = \sqrt{2}\,T\left(\frac{n}{2}\right) + \log n\,(n/2) + \log n$$

Solution The value of 'a' = $\sqrt{2}$, 'b' = 2, and $f(n) = \log n$.

The value of $$\log_b a = \log_{\sqrt{2}} 2 = \frac{\log 2}{\log \sqrt{2}} = 2$$

Since $$\sqrt{n} > \log n$$

$$T(n) = \theta(n^2) = \left(\sqrt{\theta}\right)$$

9.3.1 Proof of Master Theorem

Step 1 In this step, the following premise would be proved:

$$T(n) = aT\left(\frac{n}{2}\right) + f(n), n > 1 \text{ and } T(n) = 1 \text{ if } n = 1, \text{ then}$$

$$T(n) = n^{\log_a b} + \sum_{j=0}^{\log_b n - 1} a^j f\left(\frac{n}{b^k}\right)$$

Case 1 Let us assume that n is an exact power of b

Let $$b^k = n$$

Initially, the size of the problem is n. In the next iteration, there would be 'a' sub-problems of size b^{k-1}. The cost of combining the solutions will be $f(b^{k-1})$. Each child has 'a' children and the cost of combining the solutions would be $f(b^{k-2})$. In the ith iteration, there would be a^i nodes at distance i from the root and the cost of combining the solutions would be $f(b^{k-j})$ or $f(n/b^k)$.

In order to calculate the number of levels, let us put $n/b^j = 1$

$$n = b^j$$
$$j \log b = \log n$$
$$j = (\log b/\log n)$$
$$j = \log_n b$$

So, at the last level, the number of children would be $a^j = a^{\log_b n}$.

Since there are a^j children at this level, the total cost would be $\sum_{j=0}^{\log_b n - 1} a^j f\left(\dfrac{n}{b^k}\right)$.

The number of nodes in the last level would be $a^{\log_b n} = n^{\log_a b}$. Therefore, $T(n)$ can be considered as

$$T(n) = n^{\log_a b} + \sum_{j=0}^{\log_b n - 1} a^j f\left(\frac{n}{b^k}\right) \text{ (Fig. 9.3)}$$
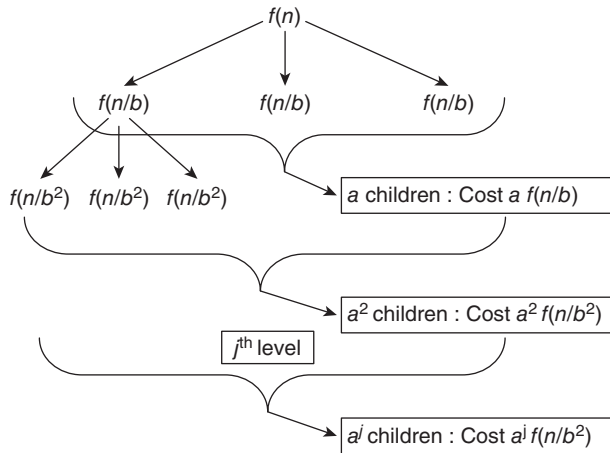
Figure 9.3 Master theorem: The root has 'a' children and the cost of combining the solutions of the sub-problems is $f(n)$

Case 2 When n/b is not an integer
In this case, we will either use floor or ceiling, that is,

$$T(n) = aT\left(\left\lceil \frac{n}{b} \right\rceil\right) + f(n)$$

or

$$T(n) = aT\left(\left\lfloor \frac{n}{b} \right\rfloor\right) + f(n)$$

The first case can be considered as the upper bound since $\left\lceil \dfrac{n}{b} \right\rceil \geq \dfrac{n}{b}$ and $\left\lfloor \dfrac{n}{b} \right\rfloor \leq \dfrac{n}{b}$.

Step 2 In this step, the following premise would be proved:

If $(n) = a \times T\left(\dfrac{n}{b}\right) + f(n)$, where $f(n)$ is asymptotically positive, $a \geq 1$ and $b > 1$.

Then

Case 1 If $f(n) = \theta(n^{\log_b a})$, then $T(n) = \theta(\log n \times n^{\log_b a})$

Case 2 If $f(n) = \Omega(n^{(\log_b a) - \varepsilon})$, then $T(n) = \theta(\log n^{\log_b a})$, where ε is a positive number.

Case 3 If $f(n) = \Omega(n^{(\log_b a) + \varepsilon})$, then $T(n) = \theta(f(n))$, where ε is a positive number.

Having understood the procedure of solving the recursive equations, let us now move to involved applications of divide and conquer. The following section discusses one such application that is 'Quick Sort'.

9.4 QUICK SORT

In quick sort, the pivot is compared with the first and the last element of the list. The pointer which moves from left to right, starting from the first element, would henceforth be called i. The pointer which moves from right to left, starting from the last element of the list would henceforth be called j.

The pivot is compared with $a[i]$, where $a[]$ denotes the array. If $a[i]$ is less than the pivot, i is incremented. The process continues till $a[i]$ remains less than pivot.

The pivot is then compared with $a[j]$, where $a[]$ denotes the array. If $a[j]$ is greater than the pivot, j is decremented. The process continues till $a[i]$ remains less than pivot.

When $a[i] >$ pivot and $a[j] <$ pivot, they are swapped. The process stops when i becomes greater than j. At this point, the pivot is placed at the position denoted by the index where i becomes greater than j. This step places pivot at its appropriate position. After this step, the elements less than pivot will be to the left of pivot and those greater than pivot will be to the right of the pivot. The left sub-array and the right sub-array will now undergo the above procedure. Finally, a sorted array is obtained. The following algorithm presents a formal approach to quick sort.

Algorithm 9.2   Partition (a, x, y)

Input: An array: a[ ], low: the first index of the array, high: the last index of the array.
Output: A sorted array

Strategy: Discussed above

```
// Within a[x], a[x+1],….,a[y-1] the elements are rearranged in such a manner
that if initially // t=a[x], then after the completion a[q]=t for some q between
x and high-1, a[k]<=t for m<=k<q, // and a[k]>=t for q<k<high. q is returned.
```

```
Algorithm: Partition
{
        v=a[m];
        i=m;
        j=p;
    repeat
    {
      repeat
          i=i+1;
      until (a[i]>=v);
      repeat
          j=j-1;
      until (a[j] <=v);
    if (i<j) then swap (a, i, j);
    } until (i>=j); a[i] >= a[j];
    a[m] = a[j]; a[j] =v; return j;
}
Algorithm: swap (a, i, j)
// Exchange a[i] with a[j].
{
        p= a[i];
        a[i] = a[j];
        a[j] = p;
}
Algorithm: QuickSort (low, high)
{
    // Sorts the elements a[low]… a[high] into ascending order;
    // a [n+1] is considered to be defined and must be >= all the elements in a
    [1: n].
    if (low<high) then // If there are more than one element
    {
      // divide array into two sub arrays.
          j= Partition (a, low, high);
              // j is the position of the partition element.
      Quicksort (low, j-1);
      QuickSort (j+1, high);
      }
}
```

Complexity: In the average case, the array would be divided into two arrays of equal size. Each sub-array will be divided into two arrays of number of elements half of the

parent. The process continues till just one element remains in the child array. The situation is depicted in the following diagram. The process stops when

$$\frac{n}{2^i} = 1$$

i.e., $n = 2^i$

or, $i = \log_2 n$

Since, there are $\log_2 n$ levels, the complexity of algorithm becomes $O(n \times \log_2 n)$.

Figure 9.4 depicts the tree corresponding to the situation.
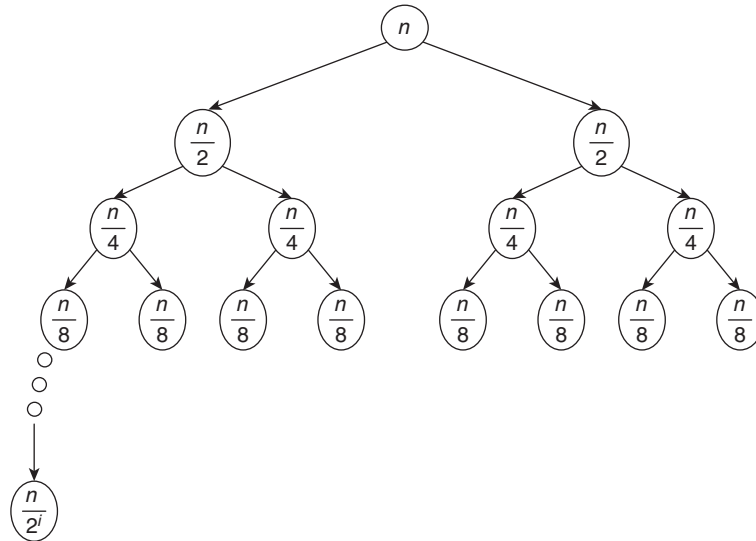
Figure 9.4 Average-case complexity of quick sort

9.4.1 Worst-case Complexity

If partition leads to segregation of the array into two parts, one having $(n-1)$ elements and the other having 1 element (in the next iteration also the pattern is repeated). This happens when the array is already sorted. In that case, the array would be divided into two parts and the first part will always have a single element. The situation is depicted in the following diagram. The number of comparisons in this case would be

$$T(n) = (n-1) + T(n-1),\ T(1) = 1$$

i.e., $\quad T(n) = (n-1) + (n-2) + \cdots + 1$

or, $\quad T(n) = n \times \dfrac{n-1}{2}$

or, $\quad T(n) = O(n^2)$

Figure 9.5 depicts the above division and Fig. 9.6 depicts the working of partition and quick sort.
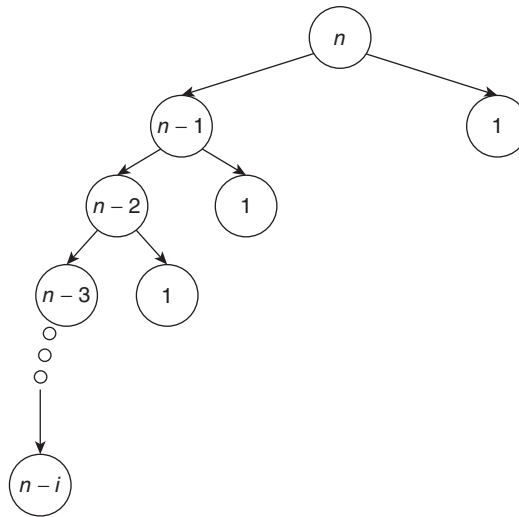
Figure 9.5   Worst-case complexity of quick sort
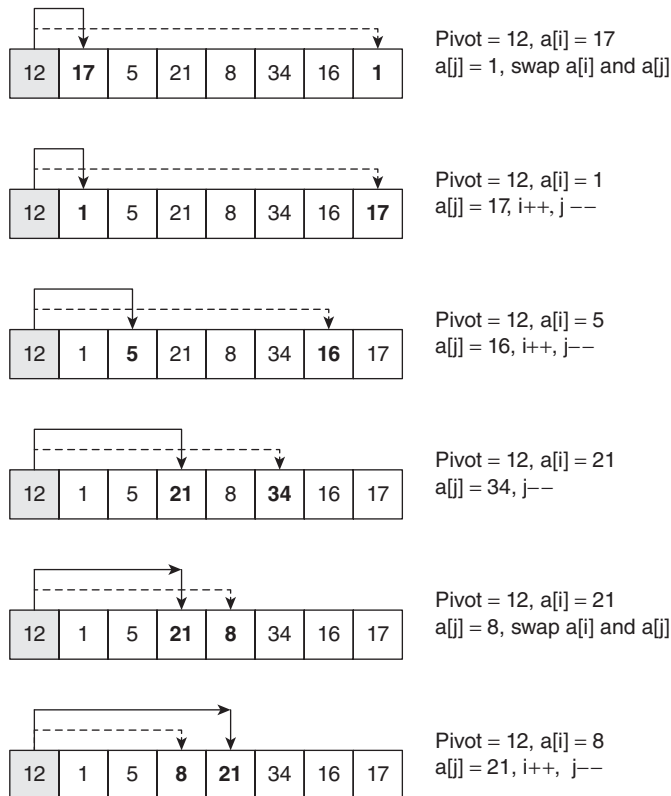
Figure 9.6   Quick sort and partition (*Contd*)

| 1 | 5 | 8 | 12 | 21 | 34 | 16 | 17 |

Pivot = 12, i > j, so place
pivot at the position.

| 1 | 5 | 8 |

The above procedure is to be applied to
the sub-array to the left of the previous pivot.

| 1 | 5 | 8 |

The result of the application of procedure
to the left sub-array.

| 21 | 34 | 16 | 17 |

The above procedure is to be applied to the
sub-array to the right of the previous pivot.

| 16 | 17 | 21 | 34 |

The result of the application of procedure
to the right sub-array.

| 1 | 5 | 8 | 12 | 16 | 17 | 21 | 34 |

Final result

Figure 9.6   (Contd) Quick sort and partition

9.5 MERGE SORT

Merge sort is a sorting algorithm that takes $O(n \log n)$ time to complete as against quick sort in which worst-case complexity is $O(n^2)$.

The algorithm is composed of two independent parts, Merge(array1[ ], array2[ ], n1, n2) and MergeSort(array[ ], low, high). The Merge() function merges two sorted arrays to give another sorted array. The MergeSort(…), on the other hand, uses Merge(…) to sort a list of elements. The algorithm can be implemented both by using recursion and without using recursion. However, the present section discusses the recursive version of merge sort. Table 9.1 depicts the terminology of merge sort.

Table 9.1   Merge sort terminology

Name	Meaning
array1[]	First array
$n1$	Number of elements in the first array
array2[]	Second array
$n2$	Number of elements in the second array
C[]	Array which stores the merged array
Low	The lower index of the array, initially, its value is generally 1
High	The higher index of the array, initially, its value is generally $(n-1)$, where n is the number of elements in the given array
Merge	Procedure, which merges two sorted arrays
MergeSort	Procedure, which sorts two arrays to give a sorted array

Algorithm 9.3 Merge

//The algorithm merges two arrays, array 1 and array 2, to form another array
c[]. The procedure for merging two sorted arrays is looking at the elements at the
indices indicated by i and j. The index k states the position at the result array.

```
Merge (array1 [], array2 [], n1, n2) returns c[]
{
      i=0;
      j=0;
      k=0;
      while(( i<n1)&&(j<n2))
                {
              if(array1[i]<array2[j])
                      {
                      c[k]=array1[i];
                      k++;
                      i++;
                      }
              else if(array2[j]<array1[i])
                      {
                      c[k]=array2[j];
                      k++;
                      j++;
                      }
              else
                      {
                      c[k]=array1[i];
                      k++;
                      i++;
                      j++;
                      }
                }
      if(i<n1)
              {
              while(i<n1)
                      {
                      c[k++]=array1[i++];
                      }
              }
      else if(j<n2)
              {
              while(j<n2)
                      {
                      c[k++]=array2[j++];
                      }
              }
      return c[];
}
```

Algorithm 9.4 Merge sort

```
MergeSort(array, low, high)
{
int mid= (low + high)/2;
int array1[], array2[];
array1[]=MergeSort(array, low, mid);
array2[]=MergeSort(array, mid+1, high);
n1=n/2;
n2=n/2;
array3[]= Merge(array1, array2, n1,n2);
return array 3[];
}
```

In order to understand the algorithm, let us have a look at one illustration. The problem requires you to sort an array using merge sort. The process is depicted in Fig. 9.7.
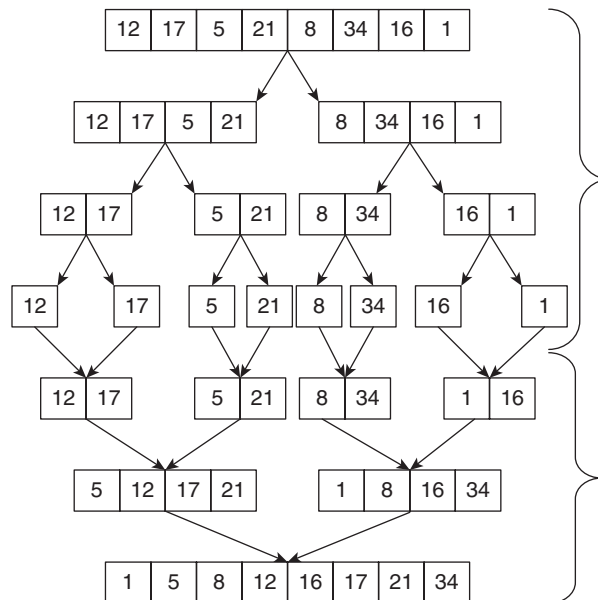
Figure 9.7 Steps of merge sort

The following discussion analyse merge sort with the help of tree method. The analysis is similar to the average case of quick sort discussed earlier. The first call will split the array, having *n* elements into two arrays having *n*/2 elements each. In the next step, there would be four arrays of (*n*/4) elements. At the end of the divide step, there would be just one element (Fig. 9.8),
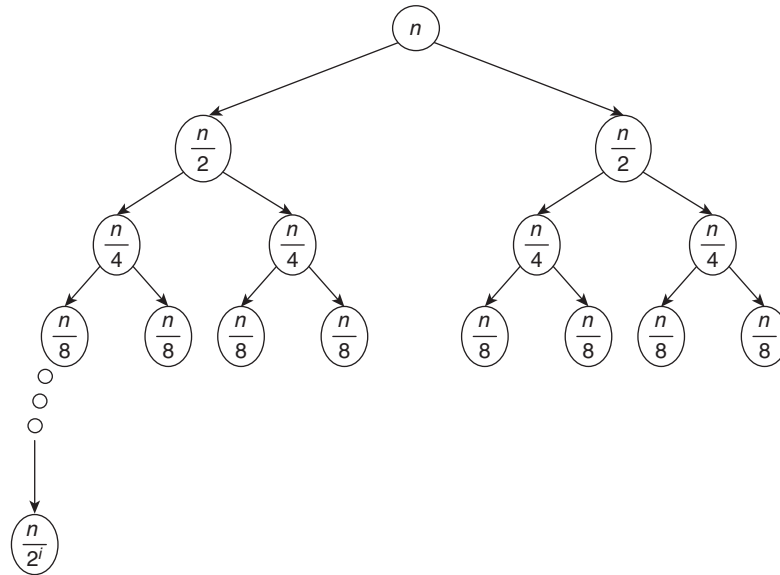
Figure 9.8 Complexity of merge sort

$$\frac{n}{2^i} = 1$$

i.e., $n = 2^i$

or, $i = \log_2 n$

Since, there are $\log_2 n$ levels, the complexity of algorithm becomes $O(n \times \log_2 n)$.

The complexity of merge sort is same as that of the average-case complexity of quick sort. However, the memory requirement of this algorithm is too large. The algorithm requires a colossal amount of auxiliary memory and is, therefore, perceived as less efficient, generally.

9.6 SELECTION

An array $a[]$, having n elements is given as an input. 'Selection' selects the kth largest element of a. The process makes use of the partition algorithm discussed earlier. The first iteration finds the correct position of the first element. If the position is equal to k, then the process stops, otherwise the next element is sent to the partition algorithm. The process continues till the result obtained is equal to the value of k.

Algorithm 9.5 Selection

Input: Array a[], and the first and the last index
Output: The requisite element

```
SELECTION (a[], int low, int high, int k) returns value
{
      i=0;
      FOUND=0;
      while(FOUND !=1)
            {
            int pos=Partition(a[], low, high, a[i]);
            if( pos==k)
                  {
                  FOUND=1;
                  return a[i];
                  }
            }
}
```

Note:

The above algorithm assumes that *a*[] is unsorted. The use of SELECT makes sense only if *a*[] is unsorted. Had *a*[] been sorted we would have gone to the *k*th position straight away.

Complexity: The above algorithm runs $O(n)$ in the worst case and $O(1)$ times in the best case.

The recursive algorithm of SELECT uses the same premise as binary search. If the correct element is not found in the first iteration, the left or the right part of that position is explored as per the case. The complexity, though reduces, but the need of the algorithm is not justified as it is already sorted.

Algorithm 9.6   Select recursive

```
SELECT RECURSIVE(a[], int low, int high, int k) returns value
{
      i=0;
      FOUND=0;
      while(FOUND !=1)
            {
            int pos=Partition(a[], low, high, a[i]);
            if( pos==k)
                  {
                  FOUND=1;
                  return a[i];
                  }
            else if(pos<k)
                  {
                  SELECT RECURSIVE (a[], low, pos, k)
```

```
                          }
            else
                  {
                  SELECT RECURSIVE (a[], pos, high, k)
                  }
            }
      }
```

Complexity: The worst-case complexity of the above algorithm is $O(n^2)$, since partition requires $O(n)$ time in the worst case. Partition is called $O(n)$ in the worst case. However, the average-case complexity of SELECT is $O(n)$. This can be proved as follows:

$$T(n) \le cn + \frac{1}{n}\left(\begin{array}{l}\text{Total time taken for processing } (n-1) \text{ elements}\\ + \text{Total time taken for processing } (i-1) \text{ elements}\end{array}\right)$$

$$T(n) \le cn + \frac{1}{n}(C_1 n + C_2 n)$$

$$T(n) \le O(n)$$

The algorithm can also be implemented without using recursion. The procedure has been left as an exercise for the readers.

9.7 CONVEX HULL

Convex hull is a set of points that makes a cover of n points such that no point lies outside the cover. All the line segments must be inside the polygon. In order to understand the point, let us consider the set of points, shown in Fig. 9.9. In this case, ABCDEA is the convex hull of the set $\{A, B, C, D, E, F, G\}$. The process of finding a convex hull of a given set of points has been explained in the following discussion.
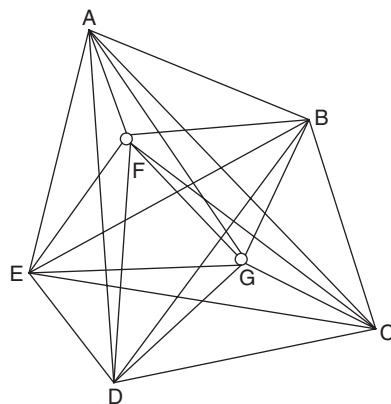
Figure 9.9 Convex hull

The first step requires the elicitation of all the triangles that can be formed. In the case of Fig. 9.9, these are as depicted in Table 9.2. As is evident from the table, there are 34 triangles.

> **Tip:** If there are n points, no three of which are collinear, then the number of triangles would be ^n_3C, that is, $\dfrac{n\times(n-1)\times(n-2)}{2\times3}$. The first step of the above procedure would be $O(n^3)$.

Table 9.2   Triangles that can be formed from the polygon of Fig. 9.9

ABC	AEF	BFG
ABD	AEG	CDE
ABE	AFG	CDF
ABF	BCD	CDG
ABG	BCE	CEF
ACD	BCF	CEG
ACE	BCG	CFG
ACF	BDE	DFE
ACG	BDF	DFG
ADE	BDG	EFG
ADF	BEF	
ADG	BEG	

The next step would be to check whether a given point lies inside or outside a given triangle. This requires $7 \times 35 = 245$ calculations.

> **Tip:** If there are n points, no three of which are collinear, then the number of calculations for finding out a convex hull would be $n\times {^n_3C}$, that is, $\dfrac{n^2 \times(n-1)\times(n-2)}{2\times3}$. The first step of the above procedure would be $O(n^4)$.

The above algorithm is summarized in Algorithm 9.7.

Algorithm 9.7   Convex hull

Input: n vertices
Output: Set 'a' which represents the set of vertices forming convex hull

```
CONVEX HULL (Set of n points) returns a set of points which forms the convex hull,
in set a
```

```
{
for any three points {n_i, n_j, n_k}
            {
            FLAG=0;
            a=∅;
            for each n_t ∈ Set of nodes
                        {
                        if(n_t does not lie in the triangle {n_i, n_j, n_k}
                                    {
                                    FLAG=1;
                                    }
                        }
            if(FLAG ==0)
                        {
                        a= a Union n_i
                        }
            }
return a;
}
```

Complexity: The complexity of the above algorithm is $O(n^4)$ as discussed above.

The divide and conquer approach of solving the above problem requires the hull to be divided into an upper hull and a lower hull. The upper hull can further be divided into two parts. The solutions of the two parts can be connected via a line called tangent. The portion to the right of the left hull and to the left of the right hull is then ignored. The complexity of such algorithm would be $O(n \log n)$.

The approach divides the set of n points into two sets, each having $n/2$ points. The initial n points, though, need to be sorted. Sorting requires $O(n \log n)$ time. The procedure requires $O(n)$ time to craft the final solution. The final equation of the above therefore becomes

$$T(n) = 2 \times T\left(\frac{n}{2}\right) + O(n)$$

By Master theorem, the final solution becomes

$$O(n \log n)$$

9.8 STRASSEN'S MATRIX MULTIPLICATION

Suppose you have two 2×2 matrices, A and B.

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$$

$$B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

The multiplication of the above two matrices can be performed by eight scalar multiplications, i.e.,

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \times \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

If

$$C_{11} = A_{11} \times B_{11} + A_{21} \times B_{21}$$
$$C_{12} = A_{11} \times B_{12} + A_{12} \times B_{22}$$
$$C_{21} = A_{21} \times B_{11} + A_{22} \times B_{21}$$
$$C_{22} = A_{21} \times B_{12} + A_{22} \times B_{22}$$

Therefore, the computational complexity is n^3. This can be proved by Master theorem. The recurrence relation of Strassen's matrix multiplication is $T(n) = 8T\left(\dfrac{n}{2}\right) + n^2$, which gives $T(n) = n^{\log_2 8} = n^3$.

> **Tip:** A 4×4 matrix can also be solved by the above equations. In that case $C_{11}, C_{12}, C_{21}, C_{22}$ would be 2×2 matrices. In general, an $n \times n$ matrix can be segregated into $\dfrac{n}{2} \times \dfrac{n}{2}$ matrices. This divide and conquer approach requires complexity of $O(n)$.

Strassen proposed a novel method of multiplication of two matrices, which had only seven scalar multiplications. The recurrence relation of the method, therefore, becomes

$$T(n) = 7T\left(\frac{n}{2}\right) + n^2, \text{ which gives } T(n) = n^{\log_2 7} = n^{2.81}, \text{ instead of } n$$

The equations are as follows:

$$P = (A_{11} + A_{22})(B_{11} + B_{22})$$
$$Q = (A_{21} + A_{22})(B_{11})$$
$$R = (A_{11})(B_{12} + B_{22})$$
$$S = (A_{22})(B_{21} + B_{11})$$
$$T = (A_{11} + A_{22})(B_{22})$$
$$U = (A_{21} - A_{11})(B_{11} + B_{12})$$
$$V = (A_{12} - A_{22})(B_{21} + B_{22})$$

$$C_{11} = P + S - T + V$$
$$C_{12} = R + T$$
$$C_{21} = Q + S$$
$$C_{22} = P + R - Q + U$$

The matrix is given by
$$C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

Tip: A 4×4 matrix can also be solved by the above equations. In that case $C_{11}, C_{12}, C_{21}, C_{22}$ would be 2×2 matrices. In general, an $n \times n$ matrix can be segregated into $\dfrac{n}{2} \times \dfrac{n}{2}$ matrices. The divide and conquer given by Strassen has complexity $O(n^{2.81})$.

The above method can be understood by considering the following illustration.

Illustration 9.17 Multiply the following matrices using divide and conquer.

$$A = \begin{pmatrix} 1 & 2 & 1 & 2 \\ 3 & 8 & 2 & 2 \\ 5 & 1 & 4 & 9 \\ 6 & 2 & 5 & 0 \end{pmatrix}, \quad B = \begin{pmatrix} 1 & 2 & 5 & 6 \\ 3 & 4 & 7 & 8 \\ 9 & 1 & 4 & 5 \\ 2 & 3 & 6 & 7 \end{pmatrix}$$

Solution

The matrix can be segregated into four 2×2 matrices, $A_{11}, A_{12}, A_{21},$ and A_{22}.

$$A_{11} = \begin{pmatrix} 1 & 2 \\ 3 & 8 \end{pmatrix}$$

$$A_{12} = \begin{pmatrix} 1 & 2 \\ 2 & 2 \end{pmatrix}$$

$$A_{21} = \begin{pmatrix} 5 & 1 \\ 6 & 2 \end{pmatrix}$$

$$A_{22} = \begin{pmatrix} 4 & 9 \\ 5 & 0 \end{pmatrix}$$

The matrix can be segregated into four 2×2 matrices, B_{11}, B_{12}, B_{21}, and B_{22}

$$B_{11} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

$$B_{12} = \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix}$$

$$B_{21} = \begin{pmatrix} 9 & 1 \\ 2 & 3 \end{pmatrix}$$

$$B_{22} = \begin{pmatrix} 4 & 5 \\ 6 & 7 \end{pmatrix}$$

Now $A \times B$ is given by the set of equations given by Strassen. Each of P, Q, R, S, T, U, V is evaluated by multiplying two 2×2 matrices, which can be done by applying Strassen's equation again. The process is summarized in Fig. 9.10.
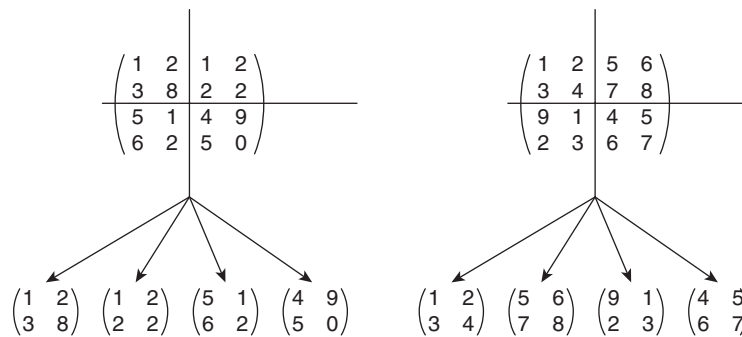
Figure 9.10   Dividing a 4×4 matrix into four 2×2 matrices

9.9 MINIMUM DISTANCE BETWEEN n POINTS

Given a set of n points, where a point is a pair (x, y), having x and y as coordinates. It is required to find two points that are nearest to each other. In order to accomplish the task, the points are first arranged in ascending order (as per the values of x coordinates). This is followed by finding out the mid-point. The strategy of divide and conquer then comes to our rescue. The premise behind the division is that the distance between two points has to be minimum, since there is just one value. While combining the sub-solutions, the following strategy is used. The distances $d1$ and $d2$ between the units to be combined is also compared with $d3$, the distance between the second point of the first pair and the first point of the second pair (Figs 9.11 and 9.12). The minimum amongst the three becomes the result of the function that combines the two units to form a larger unit. The minimum_of_three (int, int, int) function returns the minimum value amongst the three arguments (Algorithm 9.8).
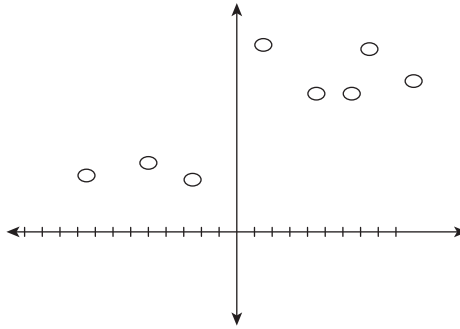
Figure 9.11 Arrange the given points as per their *x* coordinates
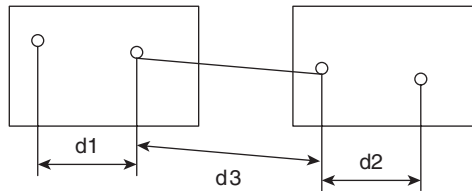
Figure 9.12 The strategy of combining the basic blocks

Algorithm 9.8 Minimum_of_three(x, y, z) returns minimum_distance

Input: Distances *d*1, *d*2, and *d*3.
Output: The minimum distance

```
      {
//x, y and z are three distances depicted by d1, d2 and d3 in Fig. 9.11. Minimum
distance is the minimum amongst the three
      minimum_distance = (d1<d2)? (d1<d3? d1: d3):(d2<d3? d2: d3);
      return minimum_distance;
      }
```

Complexity: $O(1)$;
The comparison operator, *a* op *b*? a:b means that if (*a* op *b*) is true *a* is returned; else *b* is returned.

Algorithm 9.9 Minimum_distance (*n* points of the form (x_i y_i)) returns minimum distance

Input: *n* points (x_1, y_1), (x_2, y_2),…, (x_n, y_n)
Output: Minimum distance

```
{
Arrange points in increasing order of x_i 's
```

```
X[]= heap_sort(x1, x2, …, xn);
mid = find_mid(x1, x2, …, xn);
//find_mid finds the middle point of the given set
d1 = Minimum_distance ((x₁, y₁), (x₂, y₂),…, (mid, y));
d2 = Minimum_distance ((mid +1, y), (x₂, y),…, (xₙ, yₙ));
d3 = |distance between mid and mid+1|;
d = minimum_of_three (d1, d2, d3);
return d;
}
```

Complexity: The complexity of heapify is $O(n \log n)$. That of `minimum_of_three()` is $O(1)$. However, it runs $O(n)$ times. Hence, the overall complexity is $O(n \log n)$.

9.10 MISCELLANEOUS PROBLEMS

The earlier discussion focuses on the application of divide and conquer. There are some more problems on which this technique can be applied. Some of them have been covered in the following discussion. The reader is advised to go through the latest research in this topic by exploring ACM Digital library and IEEE Explore. The aim of the following discussion is to discuss the applicability of divide and conquer in the stated problems. It is left to the reader to implement and analyse the following.

9.10.1 Multiplying Numbers Using Divide and Conquer

If an n-bit binary number is to be multiplied by another n-bit number, where $n = 2^m$. The numbers can be split into two halves. Let the numbers be n_1 and n_2. The number n_1 is split into two parts say n_{11} and n_{12}. The number n_2 is split into two parts say n_{21} and n_{22}. The numbers can be multiplied by using the following formula:

$$n_1 \times n_2 = (n_{12} \times 2^{n/2} + n_{11}) \times (n_{22} \times 2^{n/2} + n_{21})$$
$$= (n_{12} \times n_{22} \times 2^n + (n_{22}n_{11} + n_{12}n_{21}) \times 2^{n/2} + (n_{11}n_{21}))$$

The above technique would make the problem which multiplies four multiplications of $(n/2)$ bits and three additions. The complexity of the above turns out to be

$$T(n) = 4T\left(\frac{n}{2}\right) + \theta(n)$$

Applying Master theorem, we get $T(n) = \theta(n^2)$.

The algorithm for the above procedure is as follows.

Algorithm 9.10   Number multiplication

Input: Two n-bit numbers n_1, n_2
Output: A 2n-bit binary number

```
multiply (n1, n2) returns n
```

```
{
        n₁ = n₁₁ n₁₂.
        n₂ = n₂₁ n₂₂.
return multiply (multiply (multiply(n₁₂, n₁₁), 2ⁿ) + (multiply ((multiply(n₂₂n₁₁)
    + multiply(n₂₂n₂₁)), 2^(n/2)) + multiply(n₁₁, n₂₁))

}
```

Complexity: $\theta(n^2)$

9.10.2 Defective Chessboard Problem

A chessboard is generally a $2^n \times 2^n$ pane board. In the case of a normal chessboard, all the panes are identical. However, in the case of a defective chessboard, a pane would be different from the rest of the three. This different panes need to be found out. The strategy used to locate the 'different pane' would be the divide and conquer strategy, as discussed in this sub-section.

As stated earlier, the divide and conquer algorithms work if the problem can be divided into various sub-problems and those sub-problems can be independently solved. These small solutions would be merged together to form a large correct solution.

The strategy is simple, the original chessboard is divided into four chessboard with size $2^{n-1} \times 2^{n-1}$. The defective location would be in one of these four sub-boards. The rest three would be OK.

The bigger chessboard can be divided into the smaller ones until we are able to get a 1×1 chessboard. The correctness of a single block is an elementary problem. At the last step, the problem would become a yes/no problem. The solution of this would tell us which of the four calling functions contain this defective cell.

The solution then propagates in a bottom-up fashion, which finally leads us to the correct solution to the problem. As far the complexity of the problem is concerned, the following equation depicts the recursive relation:

$$T(n) = T(n-1) + O(1)$$

which gives $T(n) = O(n)$, on applying Master theorem.

9.11 CONCLUSION

Divide and conquer algorithms are helpful when a problem in hand can be divided into sub-problems. The core idea of solving the problem and the sub-problem should remain the same. If same strategy can be applied to all the sub-problems, then one should opt for divide and conquer approach. One disadvantage of this method is that usually it is implemented via recursion, which requires a lot of memory and a different way of dealing with the problem. Though divide and conquer successfully reduces the time complexity

of many problems, it is not the solution to all the problems. This chapter introduced the concept of divide and conquer and discussed the ways to find the time complexity of a recursive algorithm, which forms the backbone of the paradigm. Figure 9.13 summarizes the approach.
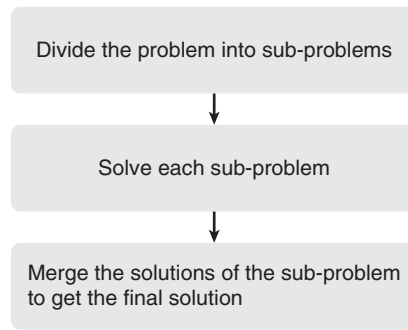
Figure 9.13 Divide and conquer

Points to Remember

- Master theorem helps in finding the complexity of an algorithm that uses the divide and conquer approach.
- Quick sort uses the partition algorithm.
- Merge sort uses two algorithms—merge and merge sort.
- Partition algorithm is also used in selection.
- Quick sort is better than merge sort as it requires lesser space.
- Divide and conquer is used to solve problems like multiplication of binary numbers and convex hull.
- The complexity of multiplying two n-bit numbers via divide and conquer is $O(n^2)$.

KEY TERMS

Convex hull It is a set of points which make a cover of n points such that no point lies outside the cover.

Merge sort A sorting technique based on the concept of divide and conquer, which uses merge to craft a sorted array from two sorted arrays. It requires auxiliary memory of order $O(n)$.

Merge algorithm The algorithm takes two sorted arrays and returns one sorted array.

Partition The algorithm takes two arguments, an integer and an array, and finds the correct position of the given integer in the array.

Quick sort A sorting technique based on the concept of divide and conquer, which uses partition to find the correct position of a given element in the given array. The algorithm is better than merge sort, which though has same complexity, requires auxiliary memory.

EXERCISES

I. Multiple Choice Questions

1. Which of the following cannot be solved via divide and conquer?
   (a) Matrix chain multiplication
   (b) Merge sort
   (c) Quick sort
   (d) None

2. Which of the following has best 'worst-case complexity'?
   (a) Merge sort
   (b) Quick sort
   (c) Bubble sort
   (d) None of the following

3. Which of the following is best taking into consideration both time and memory?
   (a) Bubble sort
   (b) Selection sort
   (c) Quick sort
   (d) Merge sort

4. Which of the following depict the correct complexity of quick sort when the input array is sorted?
   (a) $O(n)$
   (b) $O(n^2)$
   (c) $O(n^3)$
   (d) $O(n \log n)$

5. Which of the following depict the correct complexity of merge sort when the input array is sorted?
   (a) $O(n)$
   (b) $O(n^2)$
   (c) $O(n^3)$
   (d) $O(n \log n)$

6. A researcher develops a technique to multiply two 2×2 matrices. The technique requires six multiplications. The complexity of the module that combines the sub-solutions is $O(n^2)$. Which of the following correctly represent the recursive equation depicting the complexity of the algorithm?

   (a) $T(n) = 6 \times T\left(\dfrac{n}{2}\right) + O(n^2)$

   (b) $T(n) = T\left(\dfrac{n}{2}\right) + 6 \times O(n^2)$

   (c) $T(n) = 6 \times T(n^2) + O\left(\dfrac{n}{2}\right)$

   (d) None of the above

7. Which of the following depicts the answer of the above question?
   (a) $O(n^{2.58})$
   (b) $O(n^{2.8})$
   (c) $O(n^2)$
   (d) None of the above

8. The researcher makes an arbitrary change in the module that merges the solutions. On analysing the module again, he finds out that the complexity of the module has now become $O(n^3)$. What would be the complexity of the overall algorithm now?
   (a) $O(n^{2.58})$
   (b) $O(n^{2.8})$
   (c) $O(n^3)$
   (d) None of the above

9. What can be said about the algorithm after changes have been made as per question 8?
   (a) It is as good as Strassen's matrix multiplication.
   (b) Its complexity is greater than that of Strassen's matrix multiplication.
   (c) Nothing can be said on the basis of the above data.
   (d) Matrix multiplication cannot be judged on the basis of complexity.

10. The researcher then develops a testing technique based on the concept of divide and conquer which divides the program into three parts (each part takes one-third of the paths), find paths in each part and then generates a combined solution using an algorithm that takes $O(n^4)$ time. Which equation correctly depicts the solution?

(a) $T(n) = 3 \times T\left(\dfrac{n}{3}\right) + O(n^4)$

(c) $T(n) = 3 \times T(n^4) + O\left(\dfrac{n}{3}\right)$

(b) $T(n) = T\left(\dfrac{n}{3}\right) + 3 \times O(n^4)$

(d) None of the above

11. What is the complexity of the above algorithm?
   (a) $O(n^{2.5})$
   (b) $O(n^4)$
   (c) $O(n^1)$
   (d) None of the above

12. Which of the following can be used to solve recursive equations?
   (a) Substitution
   (b) Master theorem
   (c) Tree method
   (d) All of the above

13. Which of the following techniques use recursion?
   (a) Divide and conquer
   (b) Backtracking
   (c) Both
   (d) None of the above

14. Which of the following statements is true?
   (a) Binary search is based on divide and conquer
   (b) Divide and conquer cannot be applied to ternary search
   (c) The time complexity of binary search is $O(n)$.
   (d) None of the above

15. If the sub-problems are such that each solution can be used at a later point (the sub-problems need not to be homogeneous), which strategy can be used?
   (a) Dynamic
   (b) Divide and conquer
   (c) Backtracking
   (d) None of the above

II. Review Questions

1. Explain the concept of divide and conquer with the help of an example.
2. Explain quick sort. Write the algorithm and derive its complexity in best, average, and worst case.
3. Explain merge sort. Write the algorithm and derive its complexity in best, average, and worst case.
4. Explain convex hull. Write the algorithm and derive its complexity.
5. Explain quick sort. Write the algorithm and derive its complexity in best, average, and worst case.
6. Explain the procedure of finding out the minimum from a given list using divide and conquer.
7. Explain the procedure of finding out the nearest pair from a given set of n points using divide and conquer.

8. What are the disadvantages of using divide and conquer?
9. Explain Master theorem. Give an example of each of the tree cases.
10. Prove Master theorem.

III. Numerical Problems

1. Solve the following using Master theorem:

   (a) $T(n) = 4 \times T\left(\dfrac{n}{4}\right) + n^2$

   (b) $T(n) = 4 \times T\left(\dfrac{n}{3}\right) + n^3$

   (c) $T(n) = 4 \times T\left(\dfrac{n}{2}\right) + n \log n$

   (d) $T(n) = 4 \times T\left(\dfrac{n}{5}\right) + n^{2.87}$

   (e) $T(n) = 36 \times T\left(\dfrac{n}{36}\right) + \log n$

2. If the elements of an array are sorted, what would be the complexity of sorting when using quick sort?

3. If the elements of an array are sorted in the reverse order (say the elements are arranged in increasing order, and they are arranged in decreasing order), what would be the complexity?

4. Devise an algorithm that takes an array as an input and gives a sorted array as output. However, as against quick sort, it has two pivot elements. The intermediate output should be as follows. The left side of the first pivot should contain elements less than the first pivot, the elements between the two pivots should have elements between the two pivots and the elements to the right of the second pivot should be greater than the second pivot.

5. Re-craft the above algorithm so that the number of inputs are k, where $k \le n$.

6. Suggest another way of finding out the correct position of pivot, except partition given in question 5.

			Answers to MCQs				
1. (a)	3. (c)	5. (d)	7. (a)	9. (b)	11. (b)	13. (c)	15. (a)
2. (a)	4. (b)	6. (a)	8. (c)	10. (a)	12. (d)	14. (a)	

CHAPTER 10

Greedy Algorithms

OBJECTIVES

After studying this chapter, the reader will be able to

- Understand the concept of greedy algorithms
- Define spanning tree
- Find spanning tree using greedy algorithms
- Solve knapsack problem using greedy algorithms
- Solve job sequencing problem using greedy algorithms
- Explain Kruskal's and Prim's algorithms
- Understand and solve single-source shortest path
- Explain coin changing problem
- Understand the importance of Huffman codes
- Solve optimal storage, subset cover, and container-loading problem using the greedy approach

10.1 INTRODUCTION

A study by the researchers in the University of Oxford published in the *Nature Communication* brings out an interesting fact. The study proves that greed is good. According to the study, in the hierarchically structured communities, the class at the top takes care of the lowest in the hierarchy, while competing with the classes at the top of other groups. The behaviour is similar to the chimps and monkeys. A strong chimp in the group protects some of its stooges and in the process tries to establish its superiority. So, this greed is good for those that are superior in their group and for those that at the lowest level in that group. It is easy to find examples of such type of behaviour in our vicinity also. Greed is good in the case of algorithms also. The problem-solving approach that incorporates a pinch of greed helps to attain the goals in a better way, perhaps in a lesser time.

10.2 CONCEPT OF GREEDY APPROACH

Greedy approach of solving a problem calls for the selection of the most promising intermediate solution at that instance. The intermediate solution which seems promising

at a point might not be that good in the long run. In order to solve a problem via greedy approach, we select an input. If the solution satisfies the greedy goal, then it is taken in the solution set; otherwise it is left. The process is depicted as follows in Algorithm 10.1.

Algorithm 10.1 Greedy (X, n)

```
{
SET Solution Set to φ
SELECT y: y∈ X.
If y is a feasible solution, then include it in the solution set, else
proceed.
REPEAT the above two steps till all the elements of the array X have
been processed.
Return solution set.
}
```

The greedy algorithms work most of the times; however, they are not always optimal. The economies run on the basis of greedy approach. Companies are able to survive because of greedy approach. Even those who rule us follow the greedy approach. The modus operandi may not be good for the people, the country, and the humanity, but at least gives transient gains to those who make the policies.

This section explains the greedy approach by taking an example of minimum cost spanning tree.

Definition A spanning tree of a graph is a tree that covers all the vertices and does not contain any cycle.

The spanning tree of a graph can be obtained by many methods. For a given graph $G = (V, E)$, the spanning tree is a connected sub-graph with no cycle, which covers all the vertices of the tree. The spanning trees of the graph shown in Fig. 10.1(a) are depicted in Fig. 10.1(b).

The present section examines the process of finding out the spanning tree of a graph by greedy approach. As stated earlier, a spanning tree covers all the vertices; however, in most of the practical applications, there is another goal which needs to be achieved. The goal is to minimize the total weight of the edges selected (the graph is a weighted graph). So, we can use greedy algorithms in order to accomplish the task. Since the main aim of the problem is to minimize the cost, it is an optimization problem. Most of the optimization problems follow the core principle of economics: to minimize the losses or to maximize the gains.

The input of a minimum cost spanning tree is a weighted graph. The greedy approach to find the minimum cost spanning tree calls for the decision (to select the requisite edge) to be taken at every step.
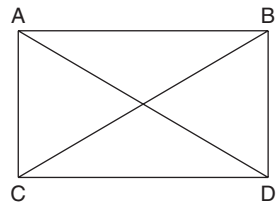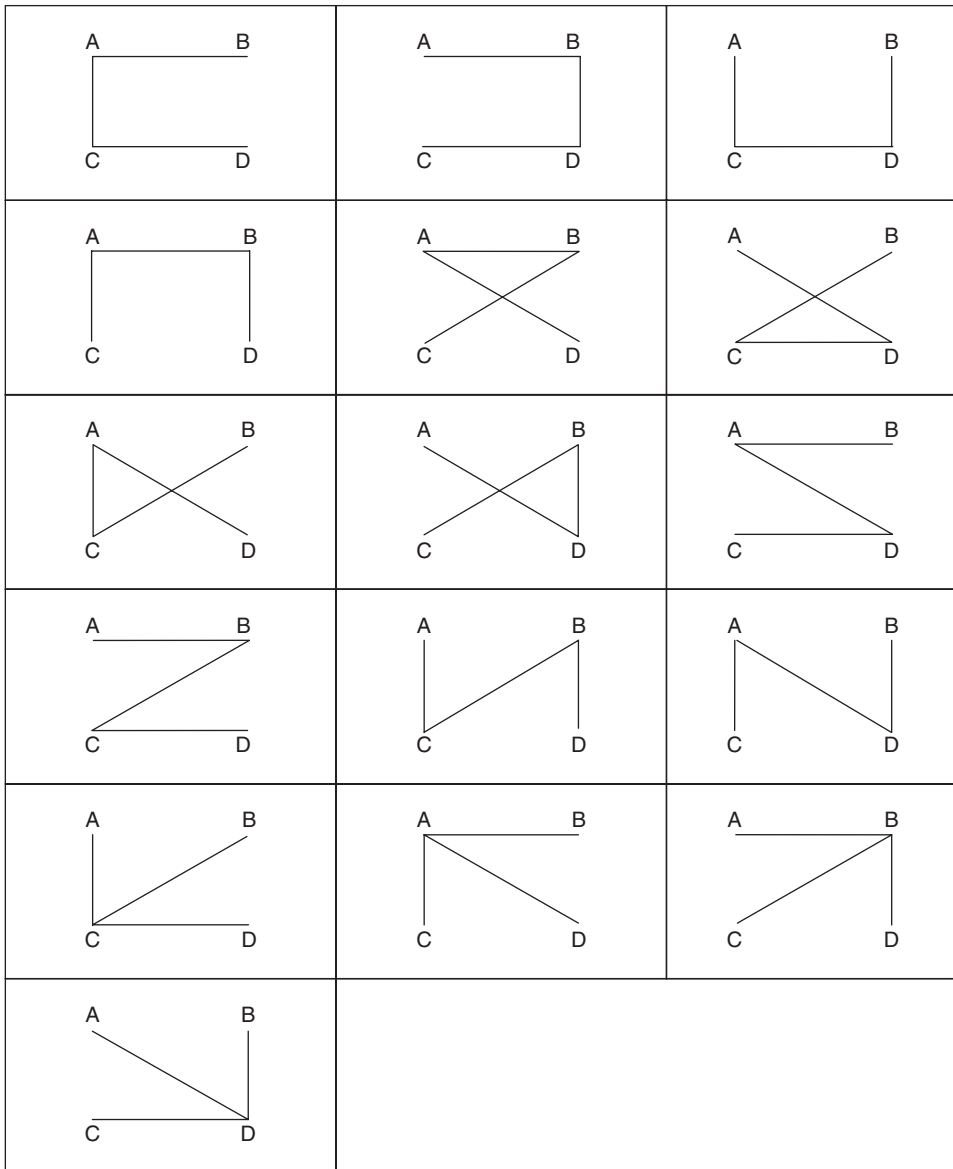
Figure 10.1 (a) Graph G; (b) Spanning trees of graph given in (a)

The concept can be understood with the help of the following example. A minimum cost spanning tree is to be formed from the graph depicting cities given in Fig. 10.2. Table 10.1 gives the distance between the various cities, for example, the cell at row 2 and column 3 of the table gives the distance between city 2 (read B) and city 3 (read C).
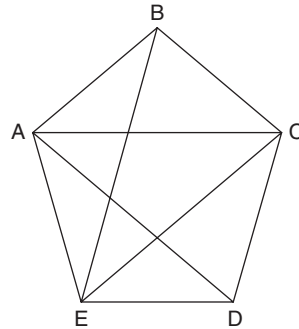
Figure 10.2 Graph whose spanning tree is to be found

Table 10.1 Cost matrix of Fig. 10.2

	A	B	C	D	E
A	0	13	42	21	14
B	13	0	26	∞	53
C	42	26	0	32	10
D	21	∞	32	0	11
E	14	53	10	11	0

In order to form a spanning tree, taking A as the source node, go from A to E, first of all, the minimum cost path from A is selected. As per the values given in Table 10.1, the minimum cost edge is from A to B (from the vertex A). The edge AB is therefore selected in the first iteration (Fig. 10.3). The vertices selected as yet would be termed as active henceforth.
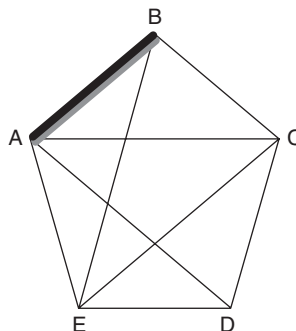
Figure 10.3 Finding spanning tree Step 1

This is followed by selecting the minimum cost edge from B or A (which are now active). Since the minimum cost edge from A or B is AE, we must head from A to E (Fig. 10.4). The edges selected as of now are AB and AE and the active vertices are A, B, and E.
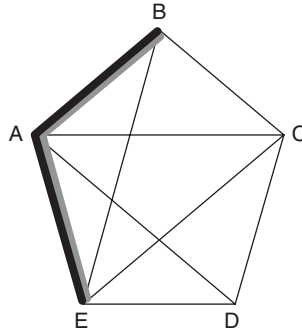
Figure 10.4　Finding spanning tree of graph in Fig. 10.2: Step 2

The edges adjacent to the active vertices are BC, AC, AD, ED, EC, and EB. The minimum cost edge from amongst these edges is EC. The edge EC is now selected and the final graph is shown in Fig. 10.5.
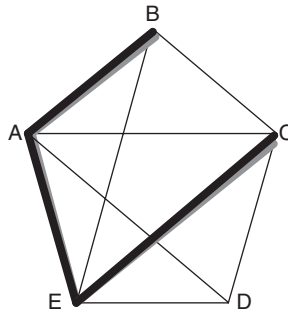
Figure 10.5　Finding spanning tree of graph in Fig. 10.2: Step 3

Now, the edges adjacent to the active vertices are BC, AC, AD, BE, AD, ED, and DC. The minimum cost edge from amongst these edges is ED. The edge ED is now selected and the final graph is shown in Fig. 10.6.
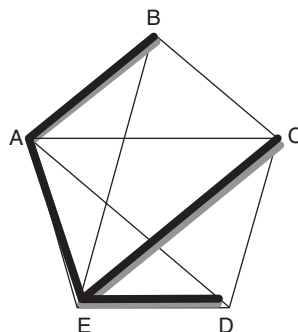
Figure 10.6　Finding spanning tree of graph in Fig. 10.2: Step 4

Since all the vertices have been covered, the task is now accomplished. The number of edges in a spanning tree of graph having *n* vertices should be $(n - 1)$. The resultant tree also has four edges. The above approach qualifies to be called greedy approach, as the decision is taken by considering the path which is the most promising, at that point of time. The sections that follow examine other methodologies to find the spanning tree of a graph.

10.3 0/1 KNAPSACK PROBLEM

In the knapsack problem, a subset of items is to be selected from the given set of items. The subset should completely (or almost completely) fill the bag and the profit earned by the selected elements should be maximum. The capacity of the bag, weights of the items, and the profit earned by selecting the items are given as an input of the problem.

Input
- The set of items $x : \{x_1, x_2, x_3, ..., x_n\}$.
- The weights of the above items $W : \{w_1, w_2, w_3, ..., w_n\}$ and
- The profits earned by picking the items $P : \{p_1, p_2, p_3, ..., p_n\}$.

Output
- $x_n = 1$ denotes that the item has been picked and $x_n = 0$ means that the item has not been picked.

Constraint
- The total weight of the selected items is less than or equal to the weight of the bag, i.e.,

$$x_1 \times w_1 + x_1 \times w_2 + x_3 \times w_1 ... \leq m \qquad (10.1)$$

where *m* is the weight of the bag.
- The profit earned is to be maximized, i.e.,

$$x_1 \times p_1 + x_1 \times p_2 + x_3 \times p_1$$

is to be maximized.

> **Definition** The selection of items from a given set in such a way that the total weight is less than or equal to the given weight, and the profit earned by picking up the elements is maximum is referred to as knapsack problem.

Solution strategy
1. Find out profit per unit weight of the items. This is because while selecting an edge, it is important to maximize the profit at the same time it is desired to fill the bag as much as possible.
2. Arrange the array obtained in the previous step in decreasing order.
3. Pick the items from the sorted array one by one till there is a space in the bag.

Algorithm 10.2   Knapsack (X, W, P, m) returns profit earned

```
{
//X is the array which has items, W is the array containing the weights of the
//items, P is the array containing profits of the items and m is the capacity of
//the Knapsack, t is the remaining weight, the variable p depicts profit earned
//Arrange the items in the non-increasing order of their p[i]/w[i];
t=m;
while (t>W[i])
        {
        Pick the item X[i];
        p=p+P[i];
        t=t-W[i];
        }
return p;
}
```

Figure 10.7 depicts the above procedure.
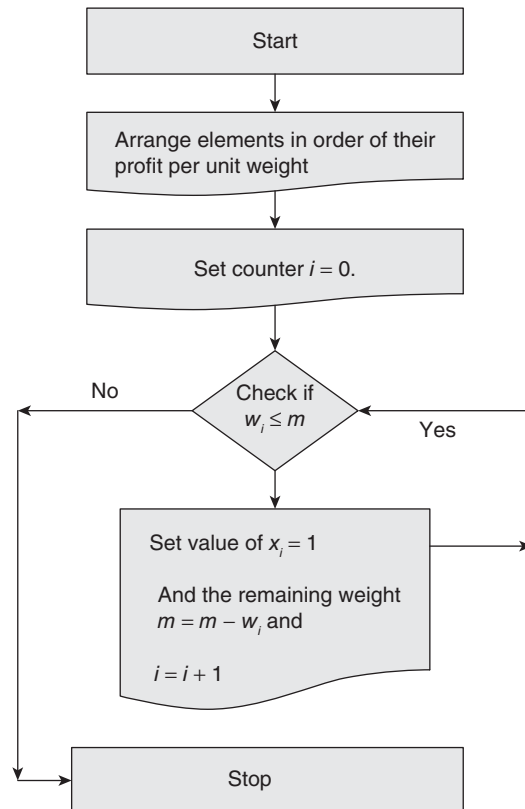
Figure 10.7   Procedure for solving knapsack problem

Applications

The problem, though, can be solved by many approaches; the approach discussed in this section is one of the easiest. Moreover, the greedy approach is sure to optimize a smaller instance of the problem. Knapsack problem is used in the following domains:

- data mining
- by the crawlers
- networking
- regression testing
- test data generation, etc.

However, the above approach will not work if the number of items is too high. In order to understand this, consider the following complexity analysis:

Complexity Analysis

Suppose there are n items in a set. The complexity of finding out profit per unit weight of each item would be $O(n)$. Moreover, the array needs to be sorted. In order to carry out the task, the complexity would be $O(n^2)$ or $O(n \times \log_2 n)$, depending upon the type of algorithm selected. This is followed by picking of an item and checking whether there is any more space in the bag. The total complexity will therefore be $O(n \log n)$ or $O(n^2)$, depending upon the algorithm. Obviously, if n is too large, then the time complexity would be too high.

However, Chapter 23 deals with Genetic Algorithms that helps to tackle the instance of the problem wherein the number of items is too large.

10.4 JOB SEQUENCING WITH DEADLINES

Job sequencing, as the name suggests, is the ordering of jobs with given deadlines and profits, in order to maximize the profit earned. Job sequencing, being an optimization problem, can be easily solved by the greedy approach. In the problem, there are n tasks: $\{x_1, x_2, x_3, ..., x_n\}$ having deadlines $\{d_1, d_2, d_3, ..., d_n\}$ and profits earned on accomplishing those jobs are $\{p_1, p_2, p_3, ..., p_n\}$. We are required to select the jobs in such a way that the selected jobs can be finished well within the deadlines and the profit earned by accomplishing the jobs is maximum. The ordering of profit helps to solve the problem via greedy approach.

Approach

The jobs can be arranged in the order of their profits. This is followed by the picking of the job with the highest profit and placing it at the index depicted by its deadline.

The rest of the jobs are to be picked such that they can be done on or before the deadline associated with them.

In order to do so, try placing the job at the position depicted by the deadline. If the position is full, then traverse backward one position at a time. If, however, none of the positions before (or on) the deadlines are empty, then that job cannot be done.

Algorithm 10.3   Job Sequencing(J[], P[], D[], n) returns X[]

Input
- A set *X* depicting jobs: $\{x_1, x_2, x_3, ..., x_n\}$
- A set *D* depicting deadlines: $\{d_1, d_2, d_3, ..., d_n\}$
- A set *P* depicting the profits earned on the accomplishment of the *i*th job: $\{p_1, p_2, p_3, ..., p_n\}$

Output
- The sequence in which jobs are to be done, in order to maximize the profit.

```
//P is the array containing profits earned when a particular job is carried out
//D[] is the array containing the corresponding deadlines
//n is the number of jobs
//X is the array that would contain the sequence in which jobs are to be
//executed in order to maximize the profit
{
netProfit = 0;
Arrange the jobs in the non-decreasing order of profits; rearrange the arrays of
Deadlines and Jobs correspondingly.
i=0;
while(i<n)
        {
        flag = 0;
        if(X[i] is not occupied)
                {
                flag=1;
                X[i]=J[i];
                }
        else
                {
                for(m=j[i];m>0;m--)
                        {
                        if(X[m] is not occupied)
                                {
                                flag=1;
                                X[m]=J[i];
                                }
                        else
                                {
                                 print: "Job cannot be completed";
                                }
                        }
                }
        }
}
```

Illustration 10.1 Solve the following job sequencing problem using greedy algorithm.

Job Number	1	2	3	4	5	6
Profit	300	250	130	212	100	424
Deadline	4	2	3	3	3	3

Solution In order to solve the problem we proceed as follows.

Step 1 First of all arrange the jobs in order of their profit.

Job Number: J	6	1	2	4	3	5
Profit: P	424	300	250	212	130	100
Deadline: D	3	4	2	3	3	3

Step 2 Now pick the job that gives the highest profit, and place it at the position depicted by its deadline.

	Job Number 6	

Step 3 For the job with second highest profit, place it at the number depicted by its deadline.

	Job Number 6	Job Number 1

Step 4 For the next highest profit job, place it at the number depicted by its deadline. Since, the second position is already filled, therefore, any position before the second that is empty is chosen.

Job Number 2	Job Number 6	Job Number 1

Step 5 As per the above logic, position number 3 is filled by job number 6. Therefore, proceed backward till an empty slot is detected.

Job Number 4	Job Number 2	Job Number 6	Job Number 1

Step 6 The rest of the jobs have deadlines whose positions have already been filled. Therefore, the remaining jobs cannot be done.

Hence, the profit incurred in doing the above jobs is
424 + 300 + 250 + 212 = 1186

Therefore, the total profit earned by accomplishing the selected jobs is 1186.

Complexity analysis: As per the complexity of the above algorithm is concerned, the sorting part will take a minimum of $n \log n$ time, depending upon the technique employed for sorting the array. The second part takes $O(n^2)$. Therefore, the overall complexity of the algorithm is $O(n^2)$.

10.5 KRUSKAL'S ALGORITHM

The Kruskal's algorithm finds out the minimum cost spanning tree by including the minimum cost edge in each step of the solution in the output tree; provided that a cycle is not formed by including that set in the solution.

In order to understand the algorithm, let us consider the following example. Phineas Flynn and Ferb Fletcher are two brothers who intend to make a roller coaster that connects their home (A) with Isabella's Firesite office (B), Baljeets home (C), Buford house (D), and Jeremy's home (E). It turns out that the cost of connecting Phineas' home to Isabella's office is $700. The cost of connecting their home with Jeremy's home is $800. Their home can be connected to Buford's home in $750. However, their home cannot be connected to any other house directly. Baljeet's home can be connected to the Firesite office in $820. The cost of connecting Baljeet's house to Buford's house is $810. Baljeet's home can be connected to Jeremy's house in $600. Jeremy's home can be connected to Firesite office in $220 and to Buford's house in $340. It may be assumed that the path for which the cost is not given may be considered infeasible. Figure 10.8 shows the diagram and Table 10.2 the corresponding costs.
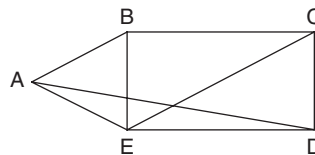
Figure 10.8 Graph

Table 10.2 Cost matrix

Cost	A	B	C	D	E
A	0	700	∞	750	800
B	700	0	820	∞	220
C	∞	820	0	810	600
D	750	∞	810	0	340
E	800	220	600	340	0

The brothers realize that there is no point in creating redundant paths. The task can be accomplished by making a spanning tree. Now, Phineas decides to make the roller coaster that connects all the points, still costs the least. In order to do so he, first of all, decides that a path BE should be constructed. This should be followed by the construction of the path which costs minimum from amongst the paths left. In the above question, this path is from B to E. Next, the path from E to C is selected. The edge BA will then be selected, thus making a spanning tree. It may be stated here that the spanning tree of a graph having n vertices has $(n - 1)$ edges. Since the given graph has 5 vertices, the number of edges in the spanning tree should have been 4. Figures 10.9–10.12 depict the steps followed.

Figure 10.9 Select BE as it is the minimum cost edge

Figure 10.10 Select ED as it is the minimum cost edge from amongst the edges left

Figure 10.11 Select EC as it is the minimum cost edge from amongst the edges left

Figure 10.12 Finally, select AB as it is the minimum cost edge from amongst the edges left

The above algorithm is referred to as Kruskal's algorithm. The algorithm selects an edge e in the kth step if

• It is the edge of minimum cost from amongst the edges left.
• It does not form cycle with the edges that have already been selected.
   Another example of the algorithm is as follows.

Illustration 10.2   Trace the steps of Kruskal's algorithm for the graph given in Fig. 10.13.
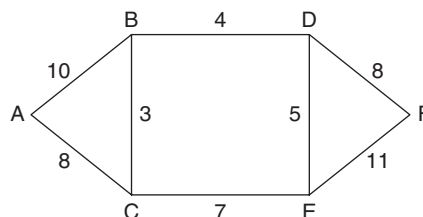
Figure 10.13 Graph

Solution
Given Graph:

Step 1 Since BC is the least cost edge, BC will be selected in the first iteration (Fig. 10.14).
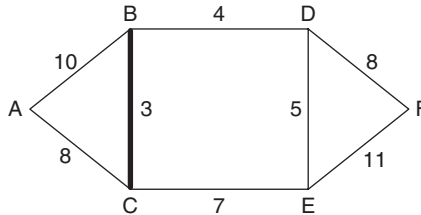
Figure 10.14 Step 1 of Kruskal's algorithm applied to the graph of Fig. 10.13

Step 2 Now from amongst the remaining edges BD has the least cost. Therefore, BD will be selected (Fig. 10.15).
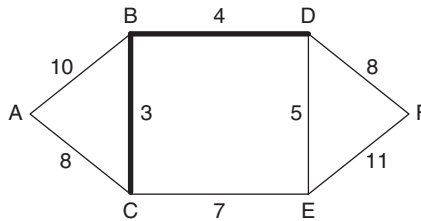
Figure 10.15 Step 2 of Kruskal's algorithm applied to the graph of Fig. 10.13

Step 3 Now from amongst the leftover edges, DE has the least cost and selecting DE will not lead to formation of a cycle. Therefore, DE will be selected (Fig. 10.16).
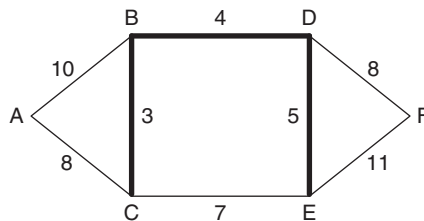
Figure 10.16 Step 3 of Kruskal's algorithm applied to the graph of Fig. 10.13

Step 4 Now since we cannot select CE, as it will lead to a cycle, AC and DF will be selected in the next two steps, one by one.
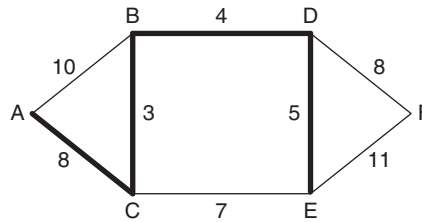
Figure 10.17 Step 4 of Kruskal's algorithm applied to the graph of Fig. 10.13

Step 5 The above tree is the required spanning tree of the given graph. The algorithm uses a heapify algorithm, which sorts the edges in order.
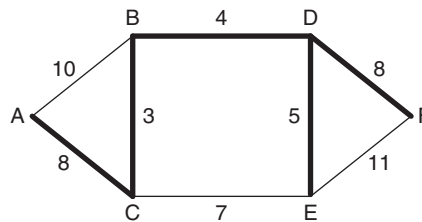
Figure 10.18 Step 5 of Kruskal's algorithm applied to the graph of Fig. 10.13

The formal algorithm is as follows. Algorithm 10.4 has two input parameters E, which is the set of edges and n, which is the number of vertices. The first step sets the parent of all the vertices as -1. The vertices will therefore be considered as isolated (not connected to any other vertex). The minimum cost is then set to 0. As stated earlier, the number of edges in a spanning tree is $(n-1)$, so the algorithm proceeds till $(n-1)$ vertices have been selected or till the heap becomes empty. An edge is then selected from the heap and taken in the solution, if it does not form a cycle. If, however, due to any of the two reasons stated above, we are not able to form a tree with $(n-1)$ vertices, then it is not possible to form a spanning tree.

Algorithm 10.4   Kruskal's algorithm

Input: A 2D matrix E, which comprises the cost of respective edges and n, the number of edges.
Output: The minimum cost spanning tree t.

Strategy: Discussed above

Kruskal (E, n) returns mincost

```
{
// E: set of edges and n: no. of vertices, t is the data structure which stores
the tree
The edges are arranged in ascending order of their costs using heapify algorithm.

for (i = 1 to n)
          {parent[i] = -1;
          }
i=0;
minimum cost = 0;

for (i=0; (i<n-1); i++)
     {
     if(heap not empty)
     {
     delete a minimum cost edge (u, v) from heap and heapify
     Find the vertices adjacent to u and v, call them j and k
     if (j and k are not same)
               {
               i=i+1;
               t (i, 1): = u;
               t (i,2): = v;
               mincost=mincost+cost [u,v];
               take the vertices j and k in the result set
               } // end of 'if'
     } // end of for
If (i≠ n-1)
     {
     Print ("no spanning tree");
     }
else

     {
     return mincost;
     }
}
```

Complexity: The algorithm arranges the vertices in order via heapify, this takes $O(|E| \log |E|)$ time. Moreover, there are no nested loops in the above algorithm. The complexity of a single loop is $O(|E|)$. The overall complexity of the algorithm therefore comes out to be $O(|E| \log |E|)$.

Theorem 10.1 Prove that Kruskal's algorithm is true.

Proof In order to prove the above theorem, mathematical induction can be applied. Now, the first step considers a graph with no edge. Since there is no edge, the theorem has to be true. Even if there is a single edge, there would be no sorting and hence the only edge would be selected in the spanning tree (Fig. 10.19).

Graph with no edge

Graph with a single edge

Figure 10.19 Graph with no edge and that with a single edge are spanning trees in themselves

Let the algorithm be true for $n = k$, which is true if k edges are selected. Now if one more edge is added to the graph from the heap having sorted edges, then the edge would keep the optimality of the graph intact. Moreover, according to the algorithm, the edge can be selected only if it does not form a cycle. In any case, the edge is selected if it has the least cost from amongst the left edges or it cannot be selected (Fig. 10.20).

This edge would be selected if its cost is least from amongst the remaining edges not selected as yet

This edge cannot be selected
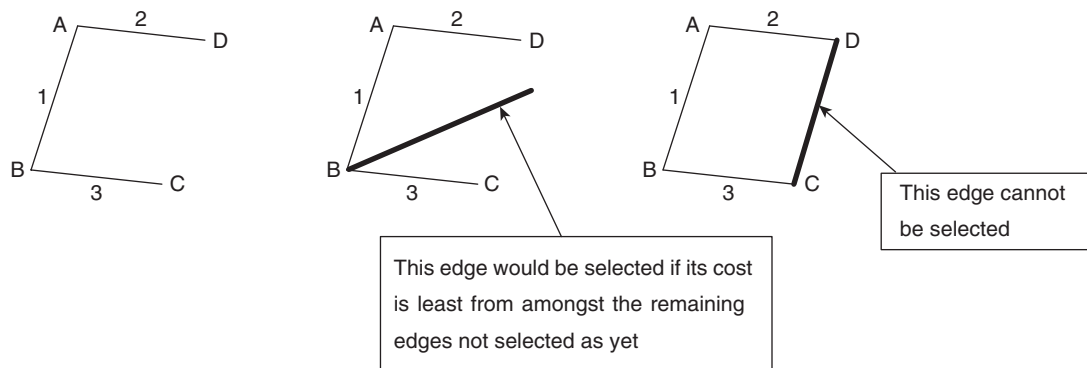
Figure 10.20 Step 2 of Kruskal's algorithm

10.6 PRIM'S ALGORITHM

The first section dealt with Kruskal's algorithm that finds out the minimum cost spanning tree. Greed is good in the case of algorithms also. The problem-solving approach that incorporates a pinch of greed helps to attain the goals in a better way, perhaps in a lesser time, also accomplishes the task of finding the minimum cost spanning tree. The algorithm is called *Prim's algorithm*. The first step of the algorithm finds out the minimum cost edge from the source node. This is followed by the addition of minimum cost edges, in a way that no cycle is formed. As stated in the first section, a spanning tree has $n - 1$ edges for a graph of n vertices. Therefore, the process continues till $(n - 1)$ edges have

been selected. This may not always be the case. There are graphs wherein the above task cannot be accomplished. In that case, an error message is printed. Algorithm 10.5 presents the Prim's method. The illustration following Algorithm 10.5 exemplifies the procedure.

Algorithm 10.5 Prim's algorithm

Input: A 2D matrix E, which comprises the cost of respective edges and *n*, the number of edges.
Output: The minimum cost spanning tree t.
Strategy: Discussed above

```
Prim(E,V) {
(k,l) = the edge of minimum cost;
mincost = cost(k,l).
// include the above edge in the tree
t(1,1)=k;
    t(1,2)=l;
     for(i=1 to n)

        {
        if (cost[i,l] < cost[i,k])
             {
near[i]=l;
             }
            else
             {
near[i]=k;
             }
             near[k]= near[l]=0;
        }
     for(i= 2 to n-1)
        {
             Find j so that if ( near [j] is not zero and (cost(j, near[j])
             is minimum)
             {
    t[i,1]=j;
    t[i,2]= near[j]
             Add the cost of (j, near[j]) to the minimum cost and make near[j]=0;
             }
for(k=1 to n)
             {
             if(near[k] is not zero and cost(k,near[k]))> cost[k,j])
                 {
                   near[k]=j;
                 }
             }
         return mincost;
}
```

Complexity: As is evident from the above algorithm, there is a loop within a loop. The complexity of the algorithm, due to this nested loop, becomes $O(n^2)$. The complexity can be reduced by using efficient data structures. The concept of red–black trees, given in Appendix A2, can greatly reduce the complexity of the algorithm.

Illustration 10.3 Trace Prim's algorithm for the graph given in Fig. 10.21.
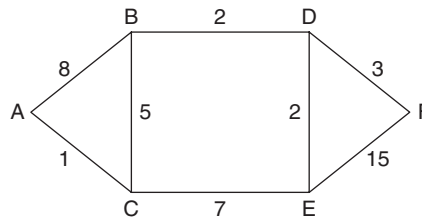
Figure 10.21 Graph

Solution
Given Graph:

Step 1 Start from A. Since from amongst all the edges from A, AC has the least cost, AC will be selected in the first iteration (Fig. 10.22).
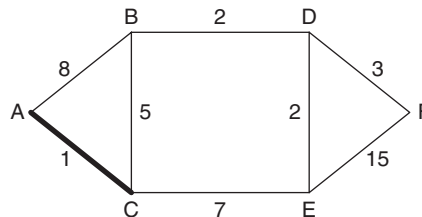
Figure 10.22 Step 1 of Prim's algorithm
applied to the graph of Fig. 10.21

Step 2 Now from amongst the active edges (AB, CE, and CB), BC has the least cost. Therefore, BC will be selected (Fig. 10.23).
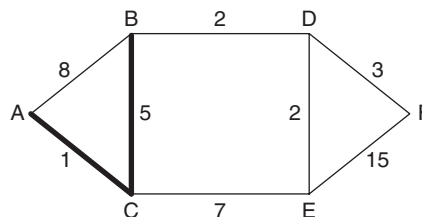
Figure 10.23 Step 2 of Prim's algorithm
applied to the graph of Fig. 10.21

Step 3 Now from amongst the active edges (AB, CE, and BD), BD has the least cost and selecting BD will not lead to formation of a cycle. Therefore, BD will be selected. In the above case, AB could not be selected even if its cost was lesser than BD, as selecting AB would have formed a cycle (Fig. 10.24).
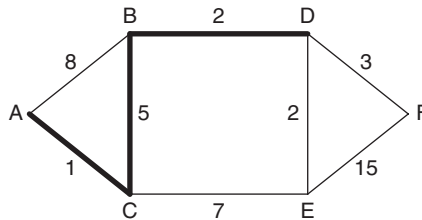
Figure 10.24   Step 3 of Prim's algorithm
applied to the graph of Fig. 10.21

Step 4 Now DE will be selected (Fig. 10.25).
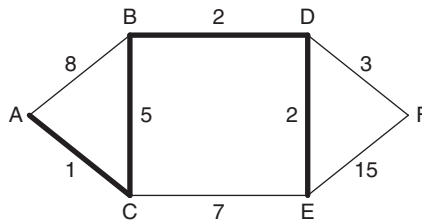
Figure 10.25   Step 4 of Prim's algorithm
applied to the graph of Fig. 10.21

Step 5 Finally, from amongst the remaining edges DF will be selected (Fig. 10.26).
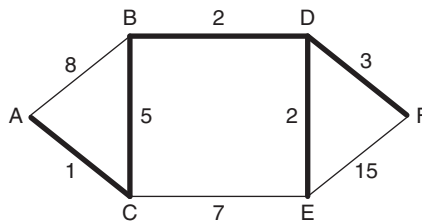
Figure 10.26   Step 5 of Prim's algorithm
applied to the graph of Fig. 10.21

The above tree is the required spanning tree of the given graph.

10.7 COIN CHANGING

Consider a set of currency notes of denomination $\{d_1, d_2, d_3, ..., d_n\}$. The set is arranged in descending order. The amount 'sum' is to be made out of the above notes in such

a way that the number of currency notes used is minimum. The greedy approach for accomplishing the above task would require selecting the maximum number of the highest denomination notes, followed by the second highest denomination, and so on. The formal algorithm for the above problem is given as follows in Algorithm 10.6.

Algorithm 10.6 Coin changing (D, sum) returns Y

```
/*D is the set containing denominations in decreasing order. The "sum" is the
amount to be generated by using minimum number of currency notes. Y is the array
that contains the number of notes of a particular denomination.*/
{
Set solution set Y = φ;
// Y is the set whose iᵗʰ element denotes the number of currency notes of
//denomination di.
seti=0;
remaining = sum; //the remaining variable denotes the amount left after the iᵗʰ
//iteration
while((remaining==0)||(remaining <dm))
        {
        if(remaining >di)
        {
        yi= sum%di;
        remaining= remaining - (di*yi);
        }
        else
        {
        yi=0;
        }
        i++;
        }
return y;
}
```

Complexity: The number of iterations in the above algorithm would be maximum n. So, the complexity of the above algorithm is $O(n)$.

Illustration 10.4 Trace the steps of coin changing problem, if the set of denominations is {50, 10, 5, 2, 1} and the amount to be given is 573.

Solution In order to solve the problem, we would choose as many notes of denomination 50 as possible. This would be followed by choosing notes of denomination 10. It is not feasible to choose notes of every denomination for a particular problem. In this case, we would not pick any note of denomination 5. The solution of the problem is depicted as follows:

Iteration 1
```
i=0;
di=50;
yi=573/50=11;
remaining= 573 - 11*50=23;
```

Iteration 2
```
i=1;
di= 10;
yi=23/10=2;
remaining= 23- 2*10=3;
```

Iteration 3
```
i=2;
di=5;
since di>remaining
yi=0;
```

Iteration 4
```
i=3;
di=2;
yi= 3/2 = 1;
remaining = 3 - 2*1=1;
```

Iteration 5
```
i=4;
di=1;
yi=1/1 = 1;
remaining = 1 - 1*1 =0;
Therefore, the solution set is {11, 2, 0, 1, 1}.
```

Failure of the Greedy Coin Changing Algorithm

The above algorithm does not work for every set of denominations. For instance, if the set of denominations is {1, 6, 10}, the above algorithm is not optimal.

For example, consider the amount is 47. The algorithm selects 4 notes of denomination 10, 1 note of denomination 6, and 1 note of denomination 1. However, if the amount is 12, then the algorithm selects one note of denomination 10 and 2 notes of denomination 1, thus selecting 3 notes. It can be observed that had we selected 2 notes of denomination 6, then the number of notes would have been less. So, the algorithm fails to ensure optimization for all the sets of denominations.

10.8 HUFFMAN CODES

A text needs to be encoded in order to minimize the number of bytes required to store it in the memory. If a piece of text is stored in the conventional way, which requires, say 7 bits, for each alphabet, then quite a lot of memory would be utilized for the task which could have been done in a much more efficient way. This section introduces Huffman encoding which would accomplish the task of minimizing the number of bytes to store a piece of text, using greedy approach.

The greedy approach would require assigning the code having minimum number of bits to an alphabet which is repeated maximum number of times. For example, if in a document, the alphabet 'a' is used 1024 times, whereas the rest of the alphabets are not repeated so often, then it would be better to assign a 1-bit code to 'a', thus saving $1024 \times 7 - 1024 \times 1 = 6144$ bits, assuming that each alphabet requires 7 bits in the conventional method.

The greedy approach selects a lesser bit code for a more frequently used bit and a code requiring more bits for a less frequently used alphabet. However, there is a catch. It may happen that the assignment may lead to a situation wherein there exists more than one decode for a single piece of encoded text. This situation, as we understand, should never arise and hence a mechanism needs to be developed which accomplishes the task of minimizing the number of bits in order to encode a text and avoid the potential ambiguity while decoding. Huffman code is one such technique.

This can be achieved by creating a tree like that given in Fig. 10.27. Note that, the leaves represent the code for a particular alphabet. The code can be assigned in accordance with the frequency of the alphabet.
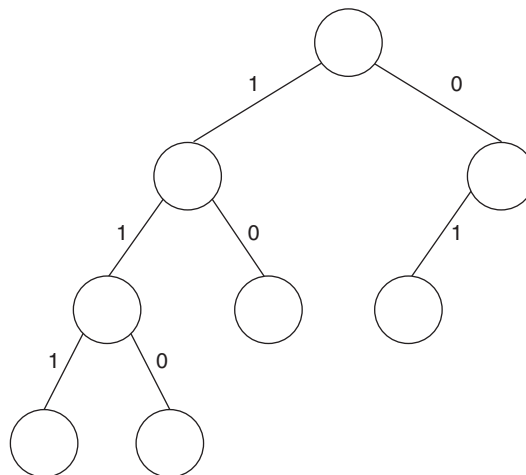
Figure 10.27 Example for Huffman codes

Now consider a piece of text that contains the alphabets shown in Table 10.3.

Table 10.3 Frequency of alphabets in a given text

Alphabet	Frequency
Space	20
A	15
B	7
C	4
G	3

Now, as it is evident from the above discussion that the symbol which has the highest frequency is allotted a code which has a least length. Since 'space' is encountered the maximum number of times, it is allotted code '0'. '01' is allotted to the character 'A', '10' to B, '110' to C, and '111' to G. Table 10.4 shows the codes for various characters.

Table 10.4 Codes for symbols

Symbol	Code
Space	0
A	01
B	10
C	110
G	111

The total number of bits required to store the given text is therefore 85. If a standard 8-bit code was used, then the number of bits would have been 343. Therefore, approximately 75% of space is saved by using the above code.

10.9 SINGLE-SOURCE SHORTEST PATH

Have you ever thought how the message we send from our computer finds its path to some other computer? The message travels via a complicated mesh of routers and tries to go via a path which has least cost. There are two issues involved in the above problem. The first issue that needs to be addressed is whether there is a path between source and the receiver and if there are more than one path, then which is the shortest? The above problem is a subclass of the problem which would be discussed in this section.

The section introduces single-source shortest path to find the shortest path from a node designated as 'source' to all other nodes.

Algorithm 10.7 Single-source shortest path

Input
- A graph G = (V, E)
- The cost corresponding to each edge

Output
- Shortest paths from the source node to all other nodes

Strategy
- Start from the source node and select the path which has minimum cost.
- The paths from the node selected and the source node are then explored. In order to go to a node, say X, the direct path from a source node and path via any of the selected nodes are considered, whichever is smaller is selected.

The array selected_vertex[] keeps track of the vertex selected at any instant. The initial value of each element is 0, it becomes 1 if that vertex is selected. Another array distance[] stores the minimum distance of a node from the source vertex. In the algorithm, i is a counter and n is the number of vertices in the graph.

```
Single Source shortest path returns distance[]
{
while ( i<n)
        {
        selected_vertex[i]=0;
        }
i=0;
selected_vertex[i]= 1;
while(i<n)
{
From amongst (n-1) edges (maximum) originating from I vertex, select the one
which has minimum cost.
Let the selected vertex be k.
selected_vertex[k]=1;
for each vertex m adjacent to i such that selected_vertex[m]=0
        {
        if(distance[k]>distance[m]+cost[m, k])
                {
                distance[k] = distance[m]+cost[m,k];
                }
        }
}
return distance[]
}
```

Complexity: The first loop runs n times, the second loop runs n times, and in each iteration, the inner loop runs, thus, making the complexity as $O(n^2)$. However, if all the shortest paths are to be determined using the above algorithm, then the complexity would be $O(n^3)$.

Illustration 10.5   Apply single-source shortest path to find the minimum distance from A of graph G1 (Fig. 10.28) to each vertex.
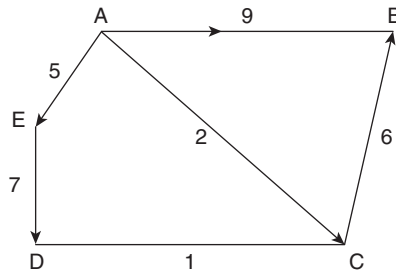
Figure 10.28 Graph G1

Solution From A, there are three outgoing edges having costs 5, 9, and 2. The edge having minimum cost is selected, as per the greedy approach (Fig. 10.29).
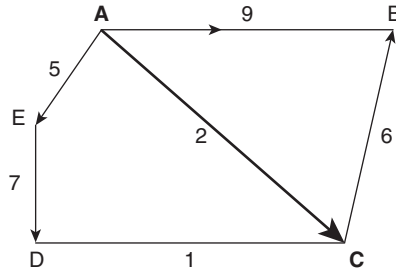
Figure 10.29 Graph G1, edge AC
selected

The minimum cost edge from amongst the remaining edges is CD. So in order to go from A to D, the optimal path is A→C→D (Fig. 10.30).
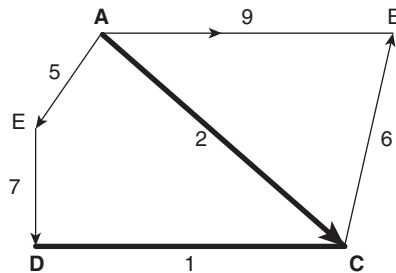
Figure 10.30 Graph G1, edge CD
selected

A packet can traverse from A to B in two ways, either directly or through C. However, the path from C costs 2 + 6 = 8, whereas the direct cost is 9 (Fig. 10.31).
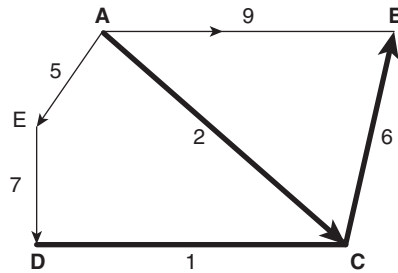
Figure 10.31 Graph G1, edge CB
selected

A packet can traverse from A to E directly. The path costs 5 (Fig. 10.32).
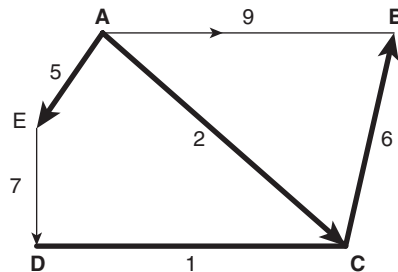
Figure 10.32 Graph G1, edge AE
selected

Though we select the minimum cost edge at a particular point, however, the cost is also being compared with other costs (i.e., via k) the algorithm cannot be called just greedy, it has a dynamic approach as well (Fig. 10.33).
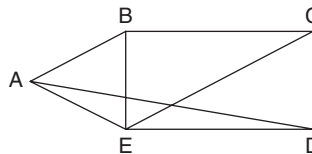
Figure 10.33 Graph

Illustration 10.6 Phineas Flynn intends to create a Metropolitan Area Network which connects the Firesite office to all other stations. Refer to Section 10.5 and solve the single-source shortest path problem using the data.

Solution The problem was solved using Kruskal's algorithm. Now, Phineas realizes that there is no point in creating redundant paths (refer Section 10.5). The task can be accomplished by using the single-source shortest path. In order to do so, he first of all, decides that a path AB should be constructed since A is the source. This should be followed by the construction of the path which costs minimum from amongst the paths left. In the above

question, this path is from B to E. Next, the path from E to C is selected. The spanning tree of a graph having *n* vertices has (*n* − 1) edges. Since the given graph has 5 vertices, the number of edges in the spanning tree should have been 4. Figures 10.34–10.37 depicts the steps followed.
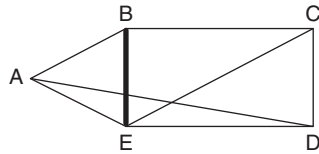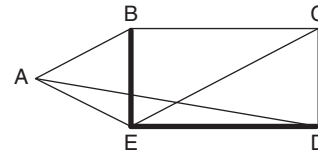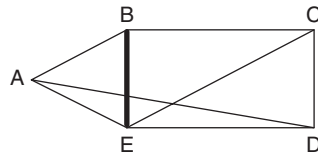

Figure 10.34 Select BE as it is the minimum cost edge

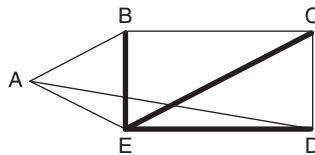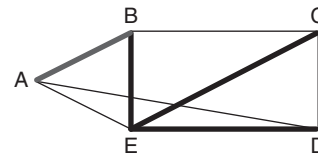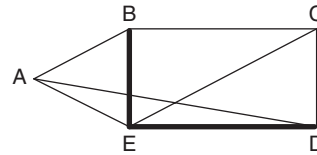**Figure 10.35** Select ED as it is the minimum cost edge from amongst the edges left as per vertices B and E are concerned
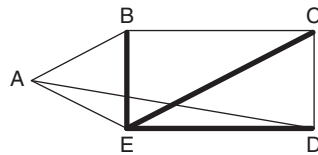
Figure 10.36 In order to go to C, the possible paths are B→E→C or B→C. However, going from E gives no added advantage. The direct edge is therefore selected
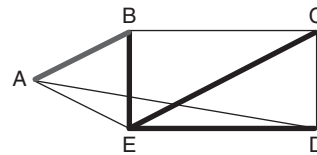
Figure 10.37 Finally, select AB as it is the least cost path between A and B

As an exercise we are required to apply this algorithm for all the nodes and create their matrices. We will observe that the space complexity of each iteration would be $O(n^2)$ and that of finding all the matrices would be $O(n^3)$.

10.10 MISCELLANEOUS PROBLEMS

This section discusses some other problems which can be handled via greedy approach. The problems are important but their algorithms are not that intricate. The section deals with the container-loading problem, the subset cover problem, and the optimal storage problem.

10.10.1 Container Loading Problem

There are *n* containers, which are to be loaded on a ship. The capacity of the ship is *c*, so not all containers can be selected. There is an x_i and a w_i associated with each item.

Let $x_i = 1$ denote that the ith container has been selected and $x_i = 0$ denote that the container has not been selected. The weight of the ith container is w_i. The aim is to select items in such a way that it maximizes the sum of weights of the selected items.

Algorithm 10.8   Container loading problem

Input
Items $1 \leq i \leq n$
Weights: $w = \{w_1, w_2, ..., w_n\}$ where w_i denotes the weight of ith element. Capacity of the ship: c.

Output
$x = \{x_1, x_2, ..., x_n\}$, where $x_i = 1$ denotes that the item has been selected and $x_i = 0$ denotes that the ith element has not been selected.

Strategy
First of all, the elements are arranged in order of their weights. Let m denote the capacity left in the ship. Initially, the value of m would be c. The elements are then selected one by one till no more elements can be selected.

```
Container Loading (W [], c) returns X
{
m=c;
i=0;
Sort the set of elements, in the decreasing order of the weights;
while (m < wᵢ)
        {
        xᵢ = 1;
        m = m - wᵢ;
        i++;
        }
return x;
}
```

Complexity: The complexity depends on the type of sorting algorithm chosen. It would be advisable to sort the elements using heapify as the complexity of heapify is just $n \times \log n$. This step is followed by a loop which traverses a maximum of n times. So, the overall complexity of the above algorithm is $(n \times \log n)$.

Importance: It has been observed by many researchers that underperformance of the above algorithm results in unsatisfied customers and unnecessary costs. The problem has, therefore, been dealt with by many researchers using heuristics methods. The paper 'Container Loading Problem: A State of the Art Review' by Andreas Bortfeldt gives a broader perspective of the problem. The paper examines the various approaches that have been used to solve the problem and their advantages and disadvantages. However, the present section has a limited scope of giving the solution of the problem via greedy algorithms. In the paper, the author has observed that the

solutions proposed so far are very approximate and the practical constraints have not been considered so far.

10.10.2 Subset Cover Problem

The subset cover problem requires finding out the set of sets, the union of which covers all the elements of the given universal set.

   The following algorithm presents a greedy approach to solve the above problem.

Algorithm 10.9   Subset cover problem

Input
- Universal Set: U
- Family of sets: $S = \{s_1, s_2, ..., s_m\}$, such that $s_i \subseteq U$
- Number of sets: m

Output
To find a family of sets S_j such that

$$\bigcup_{i=1}^{m} Us_i = \bigcup_{i=j}^{p} Us_j$$

and

$$\bigcup_{i=1}^{m} Us_i \subset \bigcup_{i=j}^{p} Us_j$$

Moreover, there is a cost associated with each set. It is also desirable to minimize the cost. That is, if C_i the cost associated with S_i then $\sum_{j=1}^{j} C_i$ should be minimum.

```
Subset Cover(S, m) returns P
{
Arrange the sets S_i in descending order of their cost.
        Let the ordered sequence be T = {t_1, t_2, ..., t_m} where t_i = S_k for some
        I and k.
i = 0;
P = ϕ;
while ( _{j=1}^{m}Us_j ⊂ _{i=j}^{m}UT_i)
        {
        Pick T_i;
        P = P ∪ T_i;
        i++;
        }
Return P;
}
```

Problems of the above algorithm
- The algorithm minimizes the net cost; however, it may not be optimal in many cases. The aim of the algorithm was to find the set cover of a set using greedy approach.
- At times, the cost associated with a set is not known. In such cases, the above algorithm will not work.

Complexity: The complexity of the algorithm depends on the sorting algorithm used. It would be advisable to use heapsort. The heapsort has a complexity of $O(n \log n)$. Thus, making the complexity of the above algorithm $O(n) + O(n \log n)$, which is same as $O(n \log n)$.

The following algorithm presents another greedy approach to solve the above problem. The following algorithm does not require the cost of each set.

Algorithm 10.10   Greedy subset cover problem

```
Subset Cover(S, m) returns P
{
i=0;
P = φ;
while ( ⱼ₌₁ᵐUSⱼ ⊂ ᵢ₌ⱼᵐUTᵢ)
        {
        Pick Tᵢ, such that Tᵢ is a set which contains maximum number of uncovered
        elements at any instant.
        P = P ∪ Tᵢ;
        i++;
        }
Return P;
}
```

Here the cost of a set can be the reciprocal of the number of items, still not covered, in the set. The above algorithm tries to minimize the net cost.

10.10.3 Optimal Storage

In a computer, there is a memory having length n. N programs are to be stored in the memory each denoted by p_i, and having length l_i. The constraint is that the sum of lengths of all the programs that is to be stored in the memory must be less than n. i.e.,

$$\sum l_i \leq n$$

The programs will also be accessed at some point in time and assuming that each time the read write head starts from the initial position, the average access time of a program needs to be minimized.

In order to accomplish the task, greedy approach can be used. First of all, the programs should be arranged in the increasing order of lengths. The smaller program should be stored first followed by the next larger one. The concept is simple and also reduces the average access time.

Algorithm 10.11 Storage

Input
n: Number of programs
l_i: Length of ith program

Output
Array a[] which stores the sequence of programs to be stored in the disk.

Strategy
The programs are arranged in order of their lengths (increasing order).
Then the programs are stored one by one in the memory.

```
Storage (n, l[], N) returns a[]
        {
        Arrange the programs in increasing order of their lengths.
        Using the arranged sequence store the programs on the disk.
        }
```

Complexity: The complexity of the above algorithm depends on the sorting used. If heapify is used then the complexity will be $O(n \log n)$.

10.11 ANALYSIS AND DESIGN FOR GREEDY APPROACH

This chapter has presented the solutions for various problems via greedy approach. The problem, however, is that there are many cases in which greedy approach does not work. Moreover, it may not always give an optimal result. The problem needs to be analysed before applying greedy algorithm. Though it is difficult to say exactly if greedy approach works for a particular problem, there is a way to see when greedy algorithms give an optimal solution. The given problem can be converted into a weighted Matriod and then be solved.

For most of the problems, if not all, the optimality can be proved with the help of Matroids. A Matroid contains two things. The first being a set S and the second is T, the set of subsets of S, such that

$$\text{if } A \in T \text{ and } B \subseteq T$$

The set S, however, cannot be empty. There is another condition on sets of T. If there is a set, say M, whose cardinality is less than another set, say N, then there must exist $x \in N - M$, such that $M \cup \{x\} \in T$.

The above definition leads us to the concept of Graphic Matroid. An undirected graph G, generates a Matriod. This is because the set S is the set of edges, which cannot be empty. A set of edges is considered independent only if it is a forest. The set T satisfies the condition

$$\text{if } A \in T \text{ and } B \subseteq A \text{ then } B \subseteq T$$

Since the subset of a forest is a forest. The last property can easily be proved using mathematical induction.

10.12 CONCLUSION

Greedy approach helps us to survive in the world and to achieve our goals. Likewise, greedy algorithms help to achieve the goal in various optimization problems. In most of the cases, we get the best solution as in the case of Kruskal's algorithm. However, greedy approach may not always lead to the best situation. The case is depicted in the coin changing problem presented in Section 10.7. The topics discussed in this chapter give an idea of how to use greedy algorithms effectively and their problems. Chapter 11 on Dynamic Programming discusses the problems seen in this chapter with a different perspective. It may also be stated here that greedy algorithms are at times more efficient as compared to other approaches.

Points to Remember

- Greedy approach cannot be applied to every problem.
- The approach is not guaranteed to give an optimal result.
- The greedy approach is easy to implement.
- A spanning tree of a graph of *n* vertices has $(n - 1)$ edges.
- Prim's and Kruskal's algorithms generate minimum cost spanning tree using greedy approach.
- Huffman code use greedy approach to generate an efficient set of codes for the given set of symbols. In this code, the symbol with higher usage gets a shorter code.
- Job sequencing and knapsack are optimization problems.

KEY TERMS

Coin changing A set of currency notes of denomination $\{d_1, d_2, d_3, ..., d_m\}$, arranged in descending order, is given. The amount 'sum' is to be made out of the above notes in such a way that the number of currency notes used is minimum.

Greedy approach This method of solving a problem selects the most promising intermediate solution at that instance. It may be noted though, that the intermediate solution which seems promising at a point might not be that good in the long run.

Knapsack problem In the knapsack problem, a subset of items is to be selected from the given set of items. The subset should completely (or almost completely) fill the bag and the profit earned by the selected elements should be maximum.

Spanning tree A graph of spanning tree is a tree which covers all the vertices and does not contain any cycle.

EXERCISES

I. Multiple Choice Questions

1. Which of the following can be solved by greedy approach?
   - (a) Knapsack problem
   - (b) Job sequencing problem
   - (c) Minimum cost spanning tree
   - (d) All of the above

2. Which of the following sorting technique is used in Kruskal's algorithm?
   - (a) Heapsort
   - (b) Bubble sort
   - (c) Selection sort
   - (d) None of the above

3. Which of the following approaches is used in Kruskal's algorithm?
   - (a) Greedy
   - (b) Dynamic
   - (c) Backtracking
   - (d) None of the above

4. Which of the following is the complexity of Kruskal's algorithm?
   - (a) $O(n^2)$
   - (b) $O(n^3)$
   - (c) $O(n \log n)$
   - (d) None of the above

5. Which of the following sorting technique is used in Prim's algorithm?
   - (a) Heapsort
   - (b) Bubble sort
   - (c) Selection sort
   - (d) None of the above

6. Which of the following approaches is used in Prim's algorithm?
   - (a) Greedy
   - (b) Dynamic
   - (c) Backtracking
   - (d) None of the above

7. Which of the following is not the complexity of Prim's algorithm?
   - (a) $O(n)$
   - (b) $O(\log n)$
   - (c) $O(1)$
   - (d) None of the above

8. Coin changing problem can be solved via which of the following?
   - (a) Backtracking
   - (b) Neurogenetic
   - (c) Branch and bound
   - (d) Greedy algorithms

9. Which types of problems are generally handled by the greedy approach?
   - (a) Optimization
   - (b) Sociological
   - (c) Psychological
   - (d) None of the above

10. The knapsack problem can be solved by which of the following approaches?
    (a) Greedy
    (b) Branch and bound
    (c) Backtracking
    (d) All of the above

II. Review Questions

1. What is meant by greedy approach? Write an abstraction of greedy approach.
2. Write the algorithm for knapsack problem via greedy approach and derive the complexity of the algorithm.
3. Write the algorithm for job sequencing problem via greedy approach and derive the complexity of the algorithm.
4. What is a spanning tree? How many spanning trees can a graph of n vertices have? Explain the procedure of finding out the minimum cost spanning tree.
5. Write the algorithm for Prim's algorithm and derive the complexity of the algorithm.
6. Write the algorithm for Kruskal's algorithm and derive the complexity of the algorithm.
7. Amongst Kruskal's and Prim's which is a better approach, if any, and why?
8. Write an algorithm for coin changing problem and derive its complexity.
9. Does Prim's algorithm work for a graph having negative weights? If not why?
10. Does Kruskal's algorithm work for a graph having negative weights? If not why?

III. Application-based Questions

1. Apply Kruskal's algorithm to find the minimum cost spanning tree in the following graphs.

(a) Cost Matrix

$$
\begin{array}{ccccc}
0 & 1 & 2 & 4 & 8 \\
1 & 0 & 3 & 5 & \infty \\
2 & 3 & 0 & 6 & \infty \\
4 & 5 & 6 & 0 & 7 \\
8 & \infty & \infty & 7 & 0
\end{array}
$$
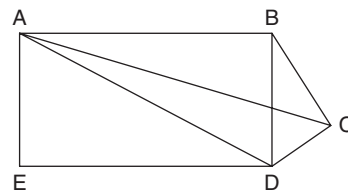
(b) Cost Matrix

$$
\begin{array}{ccccc}
0 & 1 & 3 & 5 & 7 \\
1 & 0 & 2 & 4 & 6 \\
3 & 2 & 0 & 9 & 11 \\
5 & 4 & 9 & 0 & 8 \\
7 & 6 & 11 & 8 & 0
\end{array}
$$
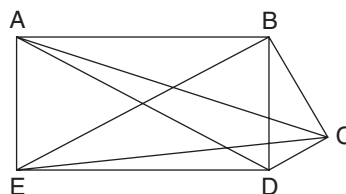
(c) Cost matrix

$$
\begin{matrix}
0 & 1 & 3 & 5 & 7 & 9 & \infty \\
1 & 0 & 2 & 4 & 6 & 8 & 10 \\
3 & 2 & 0 & 11 & 15 & \infty & 17 \\
5 & 4 & 11 & 0 & 12 & \infty & \infty \\
7 & 6 & 15 & 12 & 0 & 19 & \infty \\
9 & 8 & \infty & \infty & 19 & 0 & \infty \\
\infty & 10 & 17 & \infty & \infty & \infty & 0
\end{matrix}
$$

(d)

(e)

(f)

(g)

(h)

(i)

(j)
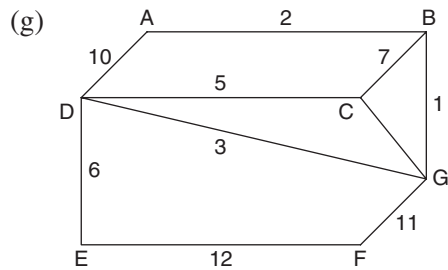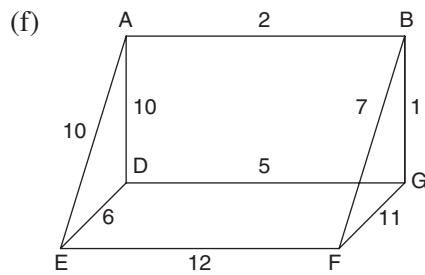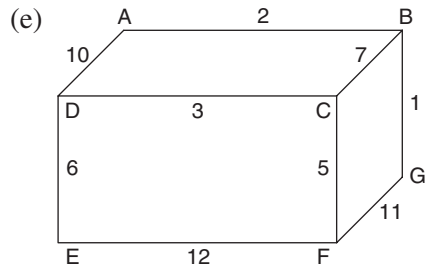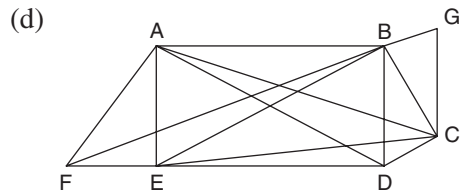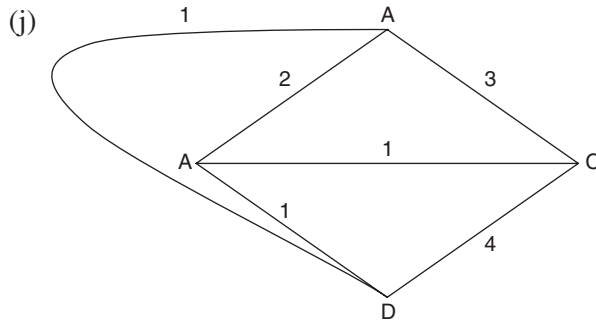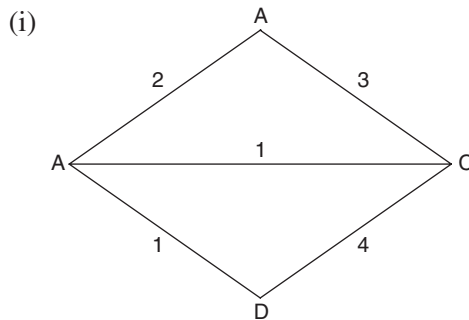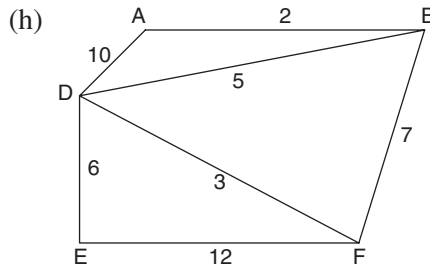
2. Apply Prim's algorithm to find the minimum cost spanning tree in the graphs shown in Question number 1(a–j).

3. Find for which of the following cases the greedy coin changing algorithm is optimal?

(a) {1, 2, 5} (f) {1, 3, 5, 9}
(b) {1, 4, 6} (g) {1, 2, 4, 6}
(c) {1, 5, 10} (h) {1, 2, 4, 6, 10}
(d) {1, 6, 10} (i) {1, 2, 3, 5, 10}
(e) {1, 2, 4, 8} (j) {1, 5, 25}

4. Trace coin changing problem for Question 1(a–j), when the value of amount is 137.

5. Prove that the greedy coin changing algorithm is optimal for {1, 2, 5, 10}.

6. Prove that the greedy coin changing algorithm is not optimal for {1, 8, 12}.

7. Indian rupee denominations are {1, 2, 5, 10, 20, 50, 100, 500, 1000}. Find whether the greedy coin changing algorithm works for the above set. If not, give a counter example.

8. The analysis of a text gives the following table(s), design a coding scheme for the symbols such that the number of bits required to store the text is minimum and at the same time, there should not be any ambiguity while decoding the text.

(a)

Symbol	Frequency
A	121
C	57
D	78
E	71
Space	45

(b)

Symbol	Frequency
A	12
C	5
D	32
E	18
Space	12

(c)

Symbol	Frequency
A	12
C	12
D	10
E	10
Space	12

Note that the above coding scheme is not that helpful for the part (c).

Answers to MCQs

1. (d)	3. (a)	5. (d)	7. (d)	9. (a)
2. (a)	4. (d)	6. (a)	8. (d)	10. (d)

CHAPTER 11

Dynamic Programming

OBJECTIVES

After studying this chapter, the reader will be able to
- Understand the concept of dynamic programming
- Recognize the difference between dynamic programming and divide and conquer approaches
- Apply dynamic programming to solve problems such as longest common subsequence, matrix chain multiplication, Floyd's algorithm, and travelling salesman problem
- Understand optimal substructure lemma
- Learn how to use dynamic programming to solve optimal binary search tree, coin changing, etc.

11.1 INTRODUCTION

As stated in the earlier chapters, the greedy approach does not give us a correct solution in many cases. The divide and conquer approach can only be applied if the sub-problems are symmetric and independent. For other problems, the above approaches would not work. Dynamic programming helps us to find the optimal solution of many such problems. There is another reason to use the dynamic approach. It was stated earlier that for every iterative procedure, we can have a corresponding recursive procedure. For example, the recursive procedure for calculating the nth Fibonacci term (given below), though easy to understand, results in calculating a particular value many times.

```
fib(int n)
        {
        if (n==1)
                return 1;
        else if (n==2)
                return 1;
        else
                return (fib (n-1) + fib (n-2));
        }
```

The calculation of 7th term, for instance, is done as shown in Fig. 11.1. From the figure, it can be seen that the third term, fib(3), is calculated 5 times. The approach is a top-down approach and is not capable of using values calculated earlier, so ends up evaluating the same value many times. As a matter of fact, the calculation of nth term would require $O(2^n)$ calculations, most of which would have been calculated earlier.
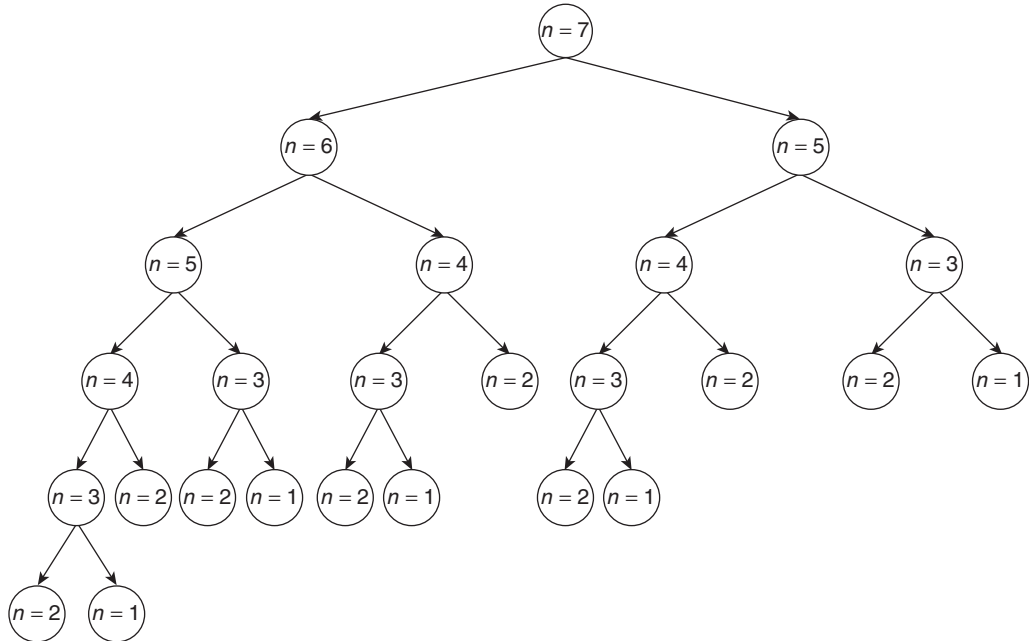
Figure 11.1   Calculating the nth Fibonacci term

The above recursive method can be made efficient by storing the earlier calculated values in a table. The following non-recursive procedure calculates the nth Fibonacci term in a bottom-up fashion.

```
fib (n)// non-recursive procedure to calculate n Fibonacci terms
{
//a[] is the global array
a[1] = 1;
a [2] = 1;
for ( i=3; i<n ; i++)
            {
                    a[i] = a[i-1] + a[i-2];
            }
}
```

This method of storing the values in a global array would make the calculation of the subsequent terms easy, as the values stored in the table would be used. Note that

the complexity of the above algorithm is just $O(n)$ as against $O(2^n)$ of the previous algorithm.

This approach of first designing a recursive procedure and then converting it into a form which makes the calculations easy is the gist of dynamic programming. This chapter explores the idea of dynamic programming and applies it to solve some of the most important problems such as longest common subsequence, matrix chain multiplication, and travelling salesman problem.

The chapter has been organized as follows. Section 11.3 discusses one of the most important applications of dynamic programming, that is, longest common subsequence. Section 11.4 discusses the matrix chain multiplication. Section 11.5 discusses the travelling salesman problem. This is followed by the discussion on the optimal sub-structure lemma. Section 11.7 explores optimal binary search trees and Section 11.8 discusses Floyd's algorithm. The last section discusses the miscellaneous problems.

11.2 CONCEPT OF DYNAMIC PROGRAMMING

Dynamic programming uses the results obtained in the previous steps to get the final answer. The idea is to use the sub-solutions obtained in solving larger sub-problems. This programming is different from the procedural or the object-oriented programming techniques. Though it is a difficult approach, the problems dealt within this chapter will enable us to understand it.

Dynamic programming might appear similar to the divide and conquer approach, but there is a fundamental difference in the two approaches. Both the divide and conquer approach and dynamic programming require division of the problems; in the former, the sub-problems are independent, whereas in the latter, they are not (Fig. 11.2). Moreover, the divide and conquer evaluates generally in a top-down fashion. The dynamic
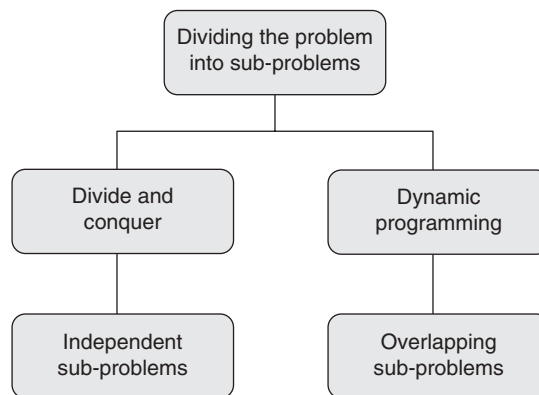
Figure 11.2 Dynamic versus divide and conquer

programming generally uses the bottom-up approach. These two approaches have been discussed as follows.

The use of the calculated values at a later stage is facilitated by storing the calculated values in a table. The paradigm, therefore, requires memorization as well. The approach can work in two ways: bottom-up and top-down. In the bottom-up approach, the solution starts with the basic input value and builds up the larger solution; the top-down approach, on the other hand, breaks the bigger problem into smaller sub-problems and then solves them (Fig. 11.3).
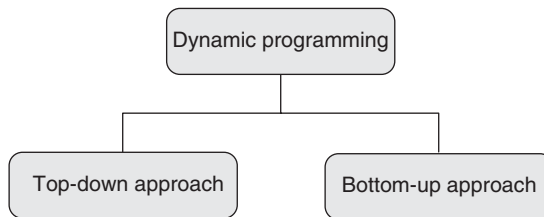
Figure 11.3   Classification of dynamic programming

11.2.1 Implementing the Dynamic Approach

The development of a dynamic algorithm for a problem requires the following steps:

- Conversion of the given problem into a search problem. The search would be from a large search space.
- The search space is generally segregated into various sub-spaces. This requires the development of a partition algorithm. This can be accomplished by developing a recursive algorithm.
- In the final step, the recursive calls are characterized and a non-recursive procedure is developed for filling the table. The values of the table are filled in a way so that the calculations can be used at a later stage.

Figure 11.4 depicts the approach.
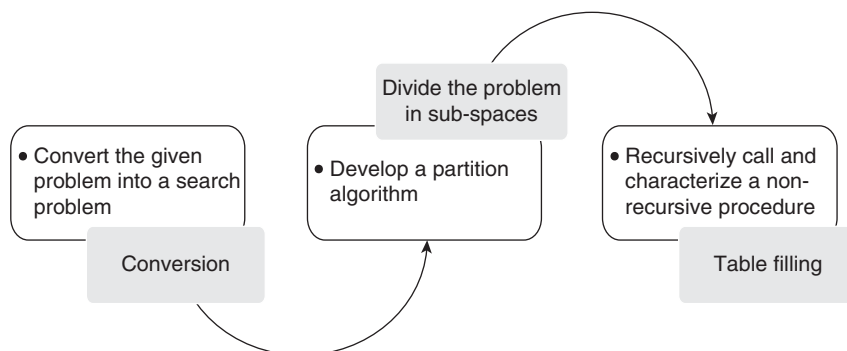
Figure 11.4   Implementing dynamic algorithms

11.3 LONGEST COMMON SUBSEQUENCE

An example of a problem that can be solved by dynamic approach is the longest common subsequence (LCS). LCS can be perceived as a measure of similarity between two given strings. That is, more the lengths of LCS, more similar are the given strings. Although LCS is not the only measure of similarity, it is also one of the most important approach. There are two approaches to find the longest common subsequence, namely brute force approach and dynamic approach. As stated in Section 11.2, the dynamic approach uses the values calculated earlier. The use of recursion and the above property makes it better than the brute force approach. The dynamic approach also has two versions. The second has been discussed with the help of illustrations that follow.

> **Definition** Two sequences X and Y having length M and N are given. It is required to find the longest subsequence $R[1…P]$. As the name suggests R is present in both X and Y. If nothing is common in X and Y then a NULL is returned.

11.3.1 Brute Force Approach

One of the easiest ways of achieving the task of finding the longest common subsequence would be as follows:

- Enlist all the subsequences of the given sequence
- Find the common subsequences
- Return the longest subsequences

There can be more than one subsequence of same length. However, in that case, we would be content in displaying all of them.

Let us assume that X has n characters and Y has m characters. The brute force approach calls for finding out all the possible subsequences of X and then checking for the existence of the longest common sequence that appears from left to right. However, the common subsequence may not be contiguous.

For example, if X is CTGCTGA and Y is TGTGT, then the longest subsequence is TGTG (Fig. 11.5).
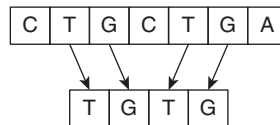
Figure 11.5 Longest common subsequence

Since there can be 2^n subsets of a set having n elements, the number of all possible subsequences of X would be 2^n.

For a sequence of 100 elements, this would be approximately 10^{32}. If the execution of a basic instruction takes 1 ms in a computer, then the execution of the whole procedure would take 3.1×10^{21} years. It is certain that no one has that much time. This is the

reason why a simpler approach is needed to accomplish the above task. The following sections discuss this problem and its various solutions.

Advantage

The brute force approach guarantees a solution or returns a NULL if none is found.

Disadvantage

The first step takes $O(n2^n)$ time. The following steps would make the total time of the process as $O(n2^n)$, which is too long.

The above approach therefore cannot be used in an application wherein the longest subsequence is desired. In the following sub-section, dynamic approach is discussed, which would greatly reduce our effort, not only because the complexity becomes too less, but also because we would be able to use the values calculated earlier, thus reducing the computation time.

11.3.2 Using the Dynamic Approach

In order to solve any problem via the dynamic approach, we need to convert that problem into a search problem. LCS is also, in fact, a search problem. We need to find the longest string from amongst the search spaces which contains all the common strings.

The search space will have all the possible strings, which are the subsequences of X or Y. The goal would be to find the strings that are common to both X and Y and have the longest length.

The above step should be followed by the formulation of a recursive procedure, which returns the best strings as per the constraints. This procedure would require the division of the search space into many search spaces. If the search space is S and the sub-spaces are S_1, S_2, …, etc., then the following condition must hold:

$$S = \bigcup_{i=1}^{k} S_i$$

To make things simpler, let us divide the search space into only two subspaces, S_1 and S_2. S_1 would contain all the strings beginning with X[1] and S_2 would contain the rest.

In order to use dynamic programming, we need to create a table which can use the values calculated earlier. This can be done by devising a non-recursive procedure, to fill the values in the table T. The procedure has been developed in the following discussion.

The recursive function would find the longest common subsequence by finding the longer of $R1$ and $R2$, where $R1$ is the result obtained by considering S_1 and $R2$ is obtained by considering S_2.

Here, we claim that if $R1$ is the result obtained by considering S_1, then the first element of $R1$ must be X[1], as was stated earlier. The first occurrence of X[1] in the second string would give us the index which would help in developing the recursive solution to the problem. Let this index be k. We can, therefore, say that R[2…p] is present in both X[2 … M] and Y[k + 1, …, N].

The above statement can be easily proved by the method of contradiction explained in Section 4.4.1 of Chapter 4.

The procedure can now be formalized as follows:

- For finding R1
  - o Determine smallest k such that $Y[k] = X[1]$, if there is no such k then return NULL.
  - o Find $L = \text{LCS}(A[2\ldots M], B[k+1\ldots N])$
  - o Return X[1]|L, where | denotes concatenation.

In the same way, $R2$ can be found.

The above concept needs to be implemented in a way that generates a table, which should be able to use the values generated in the earlier steps. This can be done as follows.

Suppose that the table is T. We need to fill the values of $T[i,j]$;

Algorithm 11.1 Longest common subsequence

Input: Two strings X and Y, the length of X is M and that of Y is N
Output: The longest common subsequence R of length p (which is as yet unknown)

```
LCS( X[1…M], Y[1…N])
{
for (i= 1; i<=M+1; i++)
        {
        T[i, N+1] = null;
        //Fill the last column with null's
        }
for (j=1; j<=N+1 ; j++)
        {
        T[N+1, j] = null;
        //fill the last row with null's
        }
for (i=M; i>=1; i--)
        {
        for (j=N ; J>=1;j--)
                {
                T[i, j]= fill_table(i, j);
                }
        }
return T[1, 1];
}
```

Complexity: The filling of the last row with NULLs takes $O(M)$ time. The filling of last column with NULL also takes $O(N)$ time. The next part of the algorithm would take $O(MN)$ time multiplied by the time taken to evaluate `fill_table(i,j)`. As we will see in

the following discussion that fill will take $O(\max(M, N))$ time. The overall time of the algorithm, therefore, becomes $O(MN \times \text{Max}(M, N))$.

The fill table algorithm would take i and j as inputs (row number and the column number) and returns the table entry as the output. The concept of the algorithm has been discussed in the section.

Algorithm 11.2   The fill_table(i, j) algorithm

```
{
If (i>N or j>M)
        {
        return null;
        }
k= smallest number such that Y[k]= X[i];
and k>=j;
        R=X[1]|T[i+1, k+1];
        R'=T[i+1, j];
return (longer (R, R'));
}
```

The implementation of the program along with the test cases has been included in the web resources of this book. The reader can run the program with the given test cases in order to analyse the behaviour of the above algorithm. There is another approach to find the LCS. Though similar, it helps to find the LCS easily, at least manually. The approach is given in Cormen [1]. In order to understand the approach, let us have a look at the following illustrations.

Illustration 11.1 Find the common subsequence of 'ACGT' and 'AGCTA'.

Solution The example presents another approach to calculate the longest common subsequence. The last row and the last column of the table would contain NULLs. The second last row is filled as follows. The symbol corresponding to this row is 'A'. 'A' does not match with any other symbol except that at the first column. Whenever a match of column and row occurs, the match (in this case 'A') is concatenated with the symbol at the south-east (SE) of the matched cell (in this case NULL). In order to indicate the match, an arrow pointing to the SE element is drawn. In the row having 'T', a match is found in the last column, in all other cases (starting from the right), the arrow points to the cell having more elements. In the row having 'C', the fourth column takes 'T' from the lower row, the third from the left takes 'T' from its right cell (here 'T' could have been taken from the bottom one as well). In the next cell, a match is found. The process is repeated in the row containing 'G' and 'A'. The final answer is 'AGT'. The reader should try to find another longest common subsequence for this example. The technique is used in DNA matching.

	A	C	G	T	
A	AGT↘	GT→	↓GT	↓T	NULL
G	GT→	GT→	GT↘	↓T	NULL
C	CT→	CT↘	T→	↓T	NULL
T	T→	T→	T→	T↘	NULL
A	A↘	NULL→	NULL→	NULL→	NULL
	NULL	NULL	NULL	NULL	NULL

Illustration 11.2 Find the longest common subsequence if the input strings are X = 'ATCGATGCA' and Y = 'GTACTACGA'.

Solution We proceed in the same way as the previous example. The last row and column is filled with NULLs. The second last row appends 'A' to the LCS. The third last row adds 'G' to it. The final answer is 'GACTA' (the first cell of the first row).

	A	T	C	G	A	T	G	C	A	
G	GATCA→	GATCA→	GATCA→	GATCA↘	↓ATCA	↓TCA	GCA↘	↓CA	↓A	NULL
T	TATCA→	TATCA↘	ATCA→	ATCA→	↓ATCA	TCA↘	CA→	↓CA	↓A	NULL
A	ACTCA↘	ATCA→	ATCA→	ATCA→	ATCA↘	↓TCA	CA→	↓CA	A↘	NULL
C	CTCA→	CTCA→	CTCA↘	TCA→	TCA→	↓TCA	CA→	CA↘	↓A	NULL
T	TACA→	TACA↘	TCA→	TCA→	TCA→	TCA↘	CA→	↓CA	↓A	NULL
A	ACGA↘	ACA→	ACA→	ACA→	ACA↘	CA→	CA→	↓CA	A↘	NULL
C	CGA→	CGA→	CGA↘	CA→	CA→	CA→	CA→	CA↘	↓A	NULL
G	GA→	GA→	GA→	GA↘	GA→	GA→	GA↘	A↘	↓A	NULL
A	A↘	A→	A→	A→	A↘	A→	A→	A→	A↘	NULL
	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

The reader is expected to develop an algorithm both for filling the table and putting the arrows. The corresponding program should be run for all the test cases given in the web resources of this book. The reader is also expected to compare the running time of the two algorithms stated above and find which gives better results in which cases.

Applications of LCS

The LCS finds applications in many fields. Some of them are as follows:
- *DNA matching*: It can be done with the help of the algorithm described in the following discussion. This book has a whole chapter dedicated to the fascinating branch of bioinformatics (Chapter 24). Sequencing and sequence matching are some of the prime tasks in bioinformatics.

- *Misspelt word*: Everybody must have observed the spellcheck facility in the word processor we work in or in our mobile. Have anyone ever wondered how it is done?

  One of the easiest ways of doing this is to use the algorithm that is discussed in this section.

11.4 MATRIX CHAIN MULTIPLICATION

Two matrices *A* and *B* are given. The order of *A* is $p \times q$ and the order of *B* is $q \times r$. The multiplication of A and B would require $p \times q \times r$ scalar multiplications. Though there are efficient ways of doing this, we will use this argument in the discussion that follows.

There is another issue in matrix multiplication. If there are more than two matrices, there can be more than one way of multiplying the matrices. Though the final result would be the same, the number of scalar multiplications would greatly vary. Take, for example, three matrices *A* of order 3×6, *B* of order 6×5, and *C* of order 5×4. The three matrices can be multiplied by in two ways. $A \times (B \times C)$ would require 120 scalar multiplications (multiplying *B* and *C*) and 72 scalar multiplications (multiplying the product of *B* and *C* with *A*). The total number of multiplications in this case would be 192.

The other way would be to multiply *A* and *B* (90 multiplications) and then multiplying the product with *C* (60 multiplications), thus resulting in 150 multiplications. The number of scalar multiplications has reduced by almost 30% by the latter method. Thus, it can be inferred that the order in which a matrix chain is multiplied determines the cost of the task, which is the number of scalar multiplications in this case.

The above problem can also be depicted by rooted trees, wherein the leaves would represent the matrices. In Fig. 11.6, M_1 is the first matrix (*A* in the above discussion), M_2 is the second (*B* in the above discussion), and M_3 is the third (*C* in the above discussion). This has been done as in the following discussion M_i has been used for the *i*th matrix.

(a)   (b)
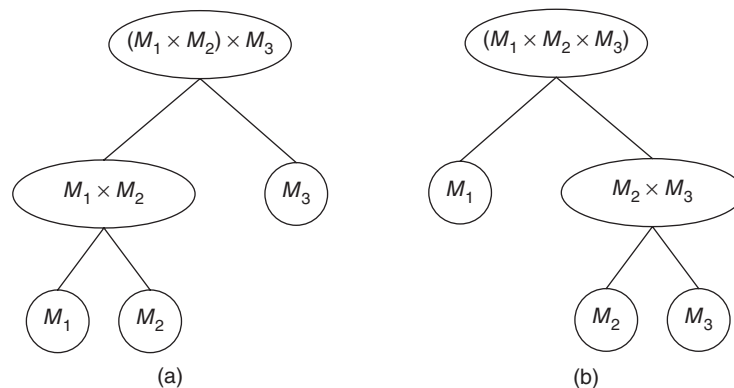
Figure 11.6   (a) Rooted tree for $(M_1 \times M_2) \times M_3$; (b) rooted tree for $M_1 \times (M_2 \times M_3)$

The various possible rooted trees of three matrices have been shown in Figs 11.6(a) and 11.6(b).

The problem of matrix chain multiplication, therefore, reduces to finding the optimal rooted tree. In the case of three or four matrices, it would not be difficult to make all the possible rooted trees and then selecting the best tree.

Suppose there are n matrices, M_1, M_2, …, M_n, the degree of the matrices is stored in an array $D[0 … n]$. The order of the ith matrix would be $D[i-1] \times D[i]$. The corresponding rooted tree of multiplication has, say, i matrices in the left sub-tree and $(n-i)$ matrices in the right sub-tree (Fig. 11.7). The cost of the tree would be the cost of the left sub-tree plus the cost of the right sub-tree plus $D[0] \times D[i] \times D[n]$.

Left sub-tree having matrices

M_1, M_2,…,M_i

Right sub-tree having matrices

M_{i+1},…,M_n
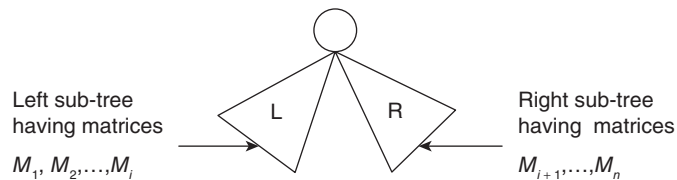
Figure 11.7 The rooted tree corresponding to matrix chain multiplication. The cost of the tree would be cost of the left sub-tree + cost of the right sub-tree + cost of multiplying the matrices formed by the left and the right sub-trees

The above method is easy. However, in the cases wherein the number of matrices is large, this technique would not work as the number of possible trees grows exponentially with the increase in the number of matrices. This brute force approach for the above problem can be summarized as follows.

Brute Force Algorithm for Matrix Chain Multiplication

- Create all the possible trees
- Find the cost of each tree
- Select one of the trees having minimum cost

Disadvantages

As stated earlier, the number of possible trees would grow exponentially with the increase in the number of matrices. Enlisting various possible trees, calculating their costs, and finding the minimum cost tree would simply not be possible; therefore, a better approach is needed.

Dynamic Approach

As stated in the introduction of this chapter, the development of a dynamic algorithm for a problem requires the following steps:

- Conversion of the given problem into a search problem. The search would be from a large search space.

- The search space is generally segregated into various sub-spaces. This requires the development of a partition algorithm. This can be accomplished by developing a recursive algorithm.
- In the final step, the recursive calls are characterized and a non-recursive procedure is developed for filling the table. The values of the table are filled in a way so that the calculations can be used at a later stage.

The objective function, in the case of matrix chain multiplication (MCM), is the cost. The cost, in this problem, needs to be minimized.

The division of a tree into two sub-trees requires at least one leaf in each part (left and right). If the tree corresponding to MCM is optimal, then its left tree must also be optimal. The statement can easily be proved by using proof by contradiction stated in Chapter 4.

The recursive algorithm can be stated as follows.

Algorithm 11.3   Matrix chain multiplication

Input: A one-dimensional matrix containing the orders of various matrices D[0…n]
Output: The optimal cost

```
MCM (D[0…n])
      {
      if (n==1)
            return (D[0] × D[1])
//The corresponding tree contains only a single vertex, as there is just one
matrix
      }
for (i=1; i< n-1; i++)
      {
      Cost_Left[i]= MCM D[0..i];
      Cost_Right[i]= MCM D[i+1…n];
      Cost[i] = Cost_Left[i]+ Cost_Right[i] + D[0] × D[i] × D[n];
//Cost_Left is the cost of the left subtree, Cost_Right is the cost of the
right sub-tree
      }
From amongst various Cost[i]'s find the minimum and return;
}
```

Corresponding Non-recursive algorithm for filling the entries in the table
```
MCM (D[j…k])
{
if (k==j+1)
        {
        return 0;
        }
// The diagonal of the table would have 0's.
```

```
else
    {
    return min MCM (D [j … i]) + MCM(D [j + 1 … k) + D[j] × D[k] × D[i];
    }
//The value of an element in the table would depend on the element below it and
on the left of it.
}
```

Complexity: As there are $O(n^2)$ entries in the table and each entry requires $O(n)$ time to evaluate, the complexity of the above algorithm becomes $O(n^3)$.

The web resources of this book contains program of matrix chain multiplication and the corresponding test cases. The reader is expected to execute the program and test the behaviour of the program with respect to the test cases. Another approach is given in Cormen [1], the reader is encouraged to implement and compare the two algorithms.

11.5 TRAVELLING SALESMAN PROBLEM

The travelling salesman problem (TSP) is an NP-hard problem. The problem can be solved using a number of approaches. The easiest approach to solve the problem would be enlisting all the possible paths, calculating their costs, and printing the path with the minimum cost. This is called the brute force approach and has been discussed in this section. The brute force approach is easier than the dynamic approach. However, using the dynamic approach gives the advantage of the earlier values being used in the present steps.

The problem can be stated as follows:

Given: A list of cities V and their pair-wise distances D such that D is a set where $x_i \in D$ is the distance between (l, m), where $l, m \in V$.

The aim of TSP is to find the shortest possible tour that can be made to each city exactly once and keeping the net cost minimum.

11.5.1 Using Brute Force Approach

The brute force approach to solve the travelling salesman problem would be as follows:
- Find all the possible paths from source back to the source. The path should contain all the vertices. The problem reduces to find all possible permutations of the paths. This step has a complexity of $O(n!)$.
- From the paths generated in the previous step, the path having least cost is selected.

It may be noted that the complexity of the above algorithm is $O(n \times n!)$ which is too high. Moreover, since $O(n!) > O(2^n)$, the above algorithm is not even as good as that having exponential complexity. In order to get an idea of the complexity of the brute force approach, let us consider the following example (Fig. 11.8).
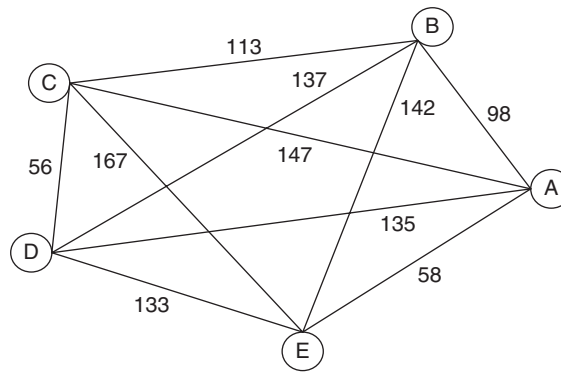
Figure 11.8 A weighted graph

The source is A and the destination is also A. All the various vertices are to be covered in a way so that no vertex is covered twice and the cost of the path traversed is minimum. The possible paths are shown in Table 11.1.

Table 11.1 Possible paths in the graph depicted in Fig. 11.8

Path	Cost
AECBDA	610
AECDBA	516
AEDBCA	588
AEDCBA	458
AEBDCA	540
AEBCDA	504
ABECDA	598
ABEDCA	576
ABDECA	682
ABCEDA	646
ACBEDA	670
ACEBDA	728

The minimum from amongst the above is 458. The corresponding path is AEDCBA. It may be noted here that had the number of vertices been 20, it would have taken (19!)/2 calculations to reach the solution. These many calculations would take colossal time and hence the approach is impractical for a graph that contains many vertices.

The dynamic approach discussed here will better the complexity of the above procedure, though not too much. The following algorithm is good owing to its ability to store the previous results.

11.5.2 Using Dynamic Approach

Step 1 Convert the problem to a search problem: travelling salesman problem is a search problem. The set of possible paths would form a search space. From this search space, the paths that cover all the vertices without having to traverse any vertex twice are to be chosen. From amongst the selected paths, the path(s) with minimum cost is chosen.

Step 2 Develop a recursive procedure to solve the problem: The following function of cost would help us to find the optimal cost of going from the source node to the destination,

$$\text{cost}(i, S) = \min_{j \in S} \left\{ c_{ij} + \text{cost}\left(j, S - \{j\} \right) \right\}$$

Step 3 Convert the above relation into a non-recursive procedure which should be able to fill the table values.

The procedure for filling the table can be understood by taking an instance of the problem with $n = 4$, where n is the number of cities.

Let the source node be 1.

Now the cost of going to any other node from the source node would be simply the cost of going from 1 to that node,

$$\text{Cost}(2, \varnothing) = c_{12}$$
$$\text{Cost}(3, \varnothing) = c_{13}$$
$$\text{Cost}(4, \varnothing) = c_{14}$$

That is, the first array would simply be the cost of going from the source node to that node.

In the next step, the cost of a node to another would be calculated. This step requires the results of the previous step, which have already been calculated and stored in the next array.

$$\text{Cost}\left(2, \{3\} \right) = c_{23} + \text{Cost}(3, \varnothing)$$
$$\text{Cost}\left(2, \{4\} \right) = c_{24} + \text{Cost}(4, \varnothing)$$
$$\text{Cost}\left(3, \{2\} \right) = c_{32} + \text{Cost}(2, \varnothing)$$
$$\text{Cost}\left(3, \{4\} \right) = c_{34} + \text{Cost}(4, \varnothing)$$
$$\text{Cost}\left(4, \{2\} \right) = c_{42} + \text{Cost}(4, \varnothing)$$
$$\text{Cost}\left(4, \{3\} \right) = c_{43} + \text{Cost}(4, \varnothing)$$

In the next step, the following costs would be calculated.

$$\text{Cost}\left(1, \{2, 3\} \right) = \min \begin{pmatrix} c_{12} + \text{Cost}\left(2, \{3\} \right), c_{13} + \text{Cost}\left(3, \{2\} \right) \text{Cost}\left(1, \{2, 4\} \right) \\ = \min(c_{12} + \text{Cost}\left(2, \{4\} \right), c_{14} + \text{Cost}\left(4, \{2\} \right) \end{pmatrix}$$

$$\text{Cost}\left(1, \{3, 4\} \right) = \min \left(c_{13} + \text{Cost}\left(3, \{4\} \right), c_{14} + \text{Cost}\left(4, \{3\} \right) \right)$$

$$\text{Cost}\left(2,\{1,3\}\right) = \min\left(c_{21} + \text{Cost}\left(1,\{3\}\right), c_{23} + \text{Cost}\left(3,\{1\}\right)\right)$$

$$\text{Cost}\left(2,\{3,4\}\right) = \min\left(c_{23} + \text{Cost}\left(3,\{4\}\right), c_{24} + \text{Cost}\left(4,\{3\}\right)\right)$$

$$\text{Cost}\left(2,\{1,4\}\right) = \min\left(c_{21} + \text{Cost}\left(1,\{4\}\right), c_{24} + \text{Cost}\left(4,\{1\}\right)\right)$$

$$\text{Cost}\left(3,\{1,2\}\right) = \min\left(c_{31} + \text{Cost}\left(1,\{2\}\right), c_{32} + \text{Cost}\left(2,\{1\}\right)\right)$$

$$\text{Cost}\left(3,\{2,4\}\right) = \min\left(c_{32} + \text{Cost}\left(2,\{4\}\right), c_{34} + \text{Cost}\left(4,\{3\}\right)\right)$$

$$\text{Cost}\left(3,\{1,4\}\right) = \min\left(c_{31} + \text{Cost}\left(1,\{4\}\right), c_{34} + \text{Cost}\left(4,\{1\}\right)\right)$$

$$\text{Cost}\left(4,\{1,2\}\right) = \min\left(c_{41} + \text{Cost}\left(1,\{2\}\right), c_{42} + \text{Cost}\left(2,\{1\}\right)\right)$$

$$\text{Cost}\left(4,\{1,3\}\right) = \min\left(c_{41} + \text{Cost}\left(2,\{4\}\right), c_{43} + \text{Cost}\left(4,\{3\}\right)\right)$$

$$\text{Cost}\left(4,\{2,3\}\right) = \min\left(c_{42} + \text{Cost}\left(1,\{4\}\right), c_{43} + \text{Cost}\left(4,\{1\}\right)\right)$$

The above costs would help us to reach the final answer, which is given by

$$\text{Cost}\left(1,\{2,3,4\}\right) = \min\begin{pmatrix} C_{12} + \text{Cost}\{2,\{3,4\}\}, C_{13} + \text{Cost}\{3,\{2,4\}\}, \\ C_{14} + \text{Cost}\{4,\{2,3\}\} \end{pmatrix}$$

Figure 11.9 depicts the above process.
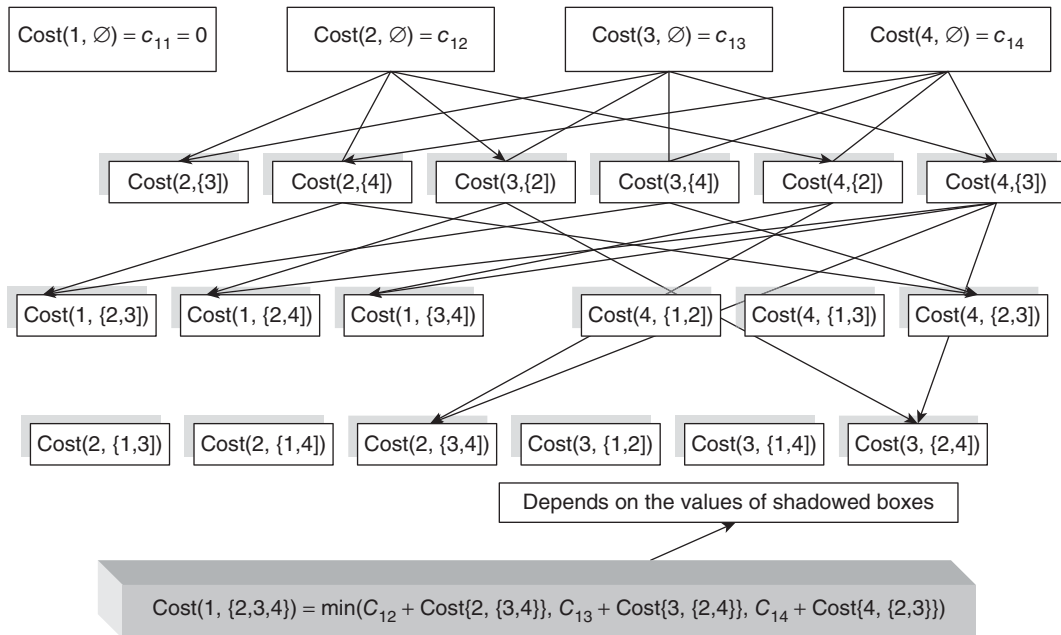
Figure 11.9 Travelling salesman problem, dynamic approach: the boxes using the cost of the first vertex have not been marked with an arrow

The following illustration would help the reader understand the above procedure.

Illustration 11.3 Solve the following travelling salesman problem for $n = 4$.

Solution

	A	B	C	D
A	0	5	2	3
B	5	0	4	1
C	2	4	0	7
D	3	1	7	0

$$\text{Cost}(2,\varnothing) = c_{12} = 5$$
$$\text{Cost}(3,\varnothing) = c_{13} = 2$$
$$\text{Cost}(4,\varnothing) = c_{14} = 3$$

$$\text{Cost}(2,\{3\}) = c_{23} + \text{Cost}(3,\varnothing) = 4 + 2 = 6$$

$$\text{Cost}(2,\{4\}) = c_{24} + \text{Cost}(4,\varnothing) = 1 + 3 = 4$$

$$\text{Cost}(3,\{2\}) = c_{32} + \text{Cost}(2,\varnothing) = 4 + 5 = 9$$

$$\text{Cost}(3,\{4\}) = c_{34} + \text{Cost}(4,\varnothing) = 7 + 3 = 10$$

$$\text{Cost}(4,\{2\}) = c_{42} + \text{Cost}(4,\varnothing) = 1 + 3 = 4$$

$$\text{Cost}(4,\{3\}) = c_{43} + \text{Cost}(4,\varnothing)$$

$$\text{Cost}(1,\{2,3\}) = \min\left(c_{12} + \text{Cost}(2,\{3\}), c_{13} + \text{Cost}(3,\{2\}) \text{Cost}(1,\{2,4\})\right)$$
$$= \min\left(c_{12} + \text{Cost}(2,\{4\})\right)c_{14} + \text{Cost}(4,\{2\}) = \min(5 + 4, 3 + 4) = 7$$

In the same way, the other costs can be calculated. The reader is expected to calculate all the values given in the above discussion and then find the value of $\text{Cost}(1,\{2,3,4\}) = \min\left(C_{12} + \text{Cost}\{2,\{3,4\}\}, C_{13} + \text{Cost}\{3,\{2,4\}\}, C_{14} + \text{Cost}\{4,\{2,3\}\}\right)$, which would be the final answer.

11.6 OPTIMAL SUBSTRUCTURE LEMMA

This section discusses the optimal substructure lemma, which is the base of dynamic approach. The statement and the proof of the lemma are as follows.

Statement: If T is an optimal binary search trees for keys $\{1, 2, 3, …, n\}$ with root $T0$, and $T1$ and $T2$ are the left and the right sub-trees, respectively, then $T1$ and $T2$ are the optimal binary search trees for the keys $\{1, 2, …, k - 1\}$ and $\{k = 1, …, n\}$. The key associated with $T0$ is k.

Proof *I: The left sub-tree is optimal:* It is given that the tree T is optimal. It implies that the cost associated with T is minimum. Let the cost of the left sub-tree be CL and that associated with the right sub-tree be CR. Let there be another tree T' with the right sub-tree $T2$ (same as that of T) and the left sub-tree $T1'$, having a better cost than $T1$.

Now, the cost of the tree depends on the sum of the cost of the left sub-tree and that of the right sub-tree. Since the cost of the right sub-tree of T' is same as that of T and that of the left sub-tree is better than that of T (less), the overall cost of T' becomes better than T. This contradicts our statement that T is the optimal tree. Hence, by contradiction, we can say that there cannot be a sub-tree that is better than the left sub-tree of T. Therefore, the left sub-tree of an optimal binary search tree is optimal.

II: The right sub-tree is optimal: It is given that the tree T is optimal. It implies that the cost associated with T is minimum. Let the cost of the left sub-tree be CL and that associated with the right sub-tree be CR. Let there be another tree T′ with the left sub-tree $T1$ (same as that of T) and the right sub-tree $T2'$, having a better cost than $T2$.

Now, the cost of the tree depends on the sum of the cost of the left sub-tree and that of the right sub-tree. Since the cost of the left sub-tree of T' is same as that of T and that of the right sub-tree is better than that of T (less), the overall cost of T' becomes better than T. This contradicts our statement that T is the optimal tree. Hence, by contradiction, we can say that there cannot be a sub-tree which is better than the right sub-tree of T. Therefore, the right sub-tree of an optimal binary search tree is optimal.

The above arguments are based on the formulae for finding the cost of a sub-tree and that of the overall tree, which are as follows:

The search time in T

$$\text{Time } Ct = \sum_{i=1}^{n} P_i \times \text{Time for searching } i \text{ in the tree } T \qquad \text{Formula 1}$$

The overall search time in an optimal binary search tree is

$$P_r \times 1 + \sum_{i=1}^{r-1} P_i \times \text{Time for searching } i \text{ in the tree } T + \sum_{i=r+1}^{n} P_i \times \text{Time for}$$

searching i in the tree T \hfill Formula 2

where P_i is the plausibility of searching i^{th} node and P_r is the plausibility of searching right sub-tree.

11.7 OPTIMAL BINARY SEARCH TREE PROBLEM

Searching is one of the fundamental problems in computer science. This section discusses the role of searching in dynamic approach in creating search trees. Linear search has been discussed in Algorithm 1.2, Chapter 1. The complexity of linear search is $O(n)$. Binary search is better than linear search, as its complexity is $O(n \log n)$. The problem of searching can be stated as follows.

A set of n items is given. It is desired to arrange them in such a way so that the cost incurred for searching a particular node is minimum. Designing a binary search tree

would be helpful. However, it might happen that the items with higher frequency are placed at the bottom of the tree, whereas those that are lesser number of times appear towards the top. This would increase the overall cost of searching all the elements. So an optimal binary search tree must be constructed to handle the above problem.

For example, if we intend to make a translator from English to Marathi, then the words that are most frequently used should be placed at a higher level and those that are less frequently used should be placed at a lower level.

11.7.1 Using Brute Force Approach

The brute force approach to form an optimal binary search tree (OBST) would be as follows:

- Enlist all the possible trees.
- Calculate their cost. Suppose the cost of the kth key is C_k, the cost of the keys in the left sub-tree would be $\sum_{i=1}^{k-1} C_i \times P_i$, where P_i denotes the probability of occurrence of the key i. Similarly, the cost of the right sub-tree would be $\sum_{i=k-1}^{n} C_i \times P_i$. The total cost of the tree would, therefore, become

$$C_k + \sum_{i=1}^{k-1} C_i \times P_i + \sum_{i=k-1}^{n} C_i \times P_i$$

- Select the tree with minimum cost.

The above approach guarantees the solution. However, the number of calculations would be exponential.

11.7.2 Using Dynamic Approach

The dynamic approach handles the problem in a better way. In fact, in this case, the complexity of the procedure to solve the OBST problem, through dynamic approach, would be cubic. A cubic complexity is not very good but is far better as compared to an algorithm that has exponential complexity.

The above problem can be solved through dynamic programming because of the following reasons:

- The problem can be converted into a search problem. The goal of the problem is, as stated earlier, to find the tree having least cost. The concept of cost has been discussed above.
- The problem can be stated as a recurrence relation for calculating the cost of the tree. The relation is as follows:

$$C_{ij} = \min r = i\,to\,j \left\{ \sum_{k=1}^{j} C_{i,r-1} + \sum_{k=r+1}^{n} C_{r+i,j} + C_r \right\}$$

- The above relation can be converted into a non-recursive table which fills in a way that the elements filled earlier help us to calculate the cost of the latter ones.
  The structure of the table is as follows (Fig. 11.10).
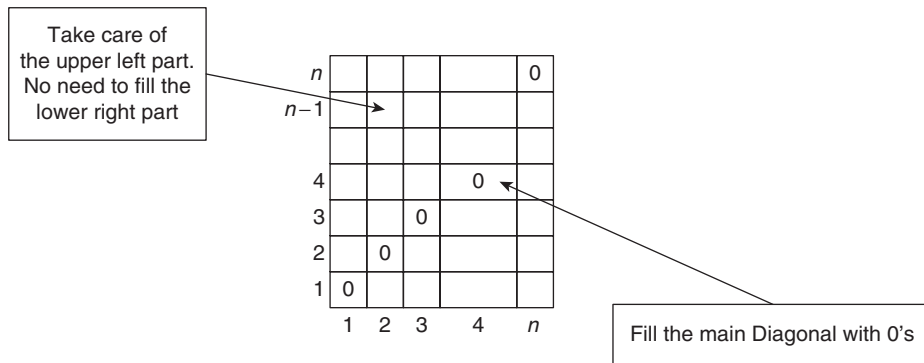
Figure 11.10 Structure of the table of optimal binary search tree

The non-recursive procedure to fill the table would be as follows:

o Let X be a two-dimensional array.

o Start filling the main diagonal of X with 0. The reason for doing so is that C_{ii} would always be 0.

o for (S=0; S<n−1; S++)

```
{
for( i=1 ; i< n; i++)
  {
    X[i, i+S ] =min min_{S=1}^{n}(\sum_{k=1}^{i+r}P_k + A[i, r − 1] + \sum_{k=r}^{n}P_k)
  }
}
```

The reader is advised to visit the web resources of this book. The resources contain the implementation of optimal binary search trees, and the corresponding test cases. The reader is advised to run same test cases in the implementation of binary trees and observe the difference in the running time.

11.8 FLOYD'S ALGORITHM

The shortest path problem has been discussed in Sections 10.2, 10.5, and 10.6. These methods used the greedy approach. As stated earlier, the problem finds its applications in packet routing, creating optimal paths, etc. The greedy approach, though produce good results. However, had an approach which could have created optimal sub-solutions been used, it would have been better. This section discusses one such algorithm which is Floyd's algorithm. The algorithm was given by Robert Floyd in 1962. The algorithm was published earlier by Roy in 1959 and Warshall in 1962. This is the reason that it is referred to as Floyd–Warshall algorithm in some books.

Floyd's algorithm finds out the shortest path. The algorithm uses a simple concept. The distance between any two nodes, in a given graph, is either the value of distance

given in the matrix or the sum of the value of the source vertex to a vertex 'x' and that from 'x' to the destination. It may be stated here that 'x' is neither the source nor the destination. The algorithm might appear brute force algorithm but it uses the optimal substructure lemma, and is therefore considered dynamic. Moreover, there is a scope of improvement in Algorithm 11.4. It may be inferred from the algorithm that the *i*th column and the *i*th row are not altered in the *i*th iteration. The algorithm can also be altered to print the values of the paths.

Algorithm 11.4 Floyd algorithm

Floyd (Weights, n)
Input: W, the matrix containing the weights, the weight of a node to itself is 0, and that of a node, directly connected to it, is infinity. '*n*' is the number of cities.
Output: A two dimensional matrix Distance, containing the shortest distance.

```
{
Distance=Weights;
for ( i=0; i< n; i++)
          {
          for (j=0; j<n; j++)
                  {
                  for( k=0; k<n ;k++)
                          {
                          Distance[i] [j] = minimum (Distance[i][j],
                          Distance[i][k]+Distance[K][J]);
                          }
                  }
          }
return Distance;
}
```

Complexity: The algorithm contains nested loops. The loop within two loops makes the complexity of the algorithm $O(n^3)$.

Explanation: The distance between any two points is either the direct distance or the sum of distance of the source to a node say 'x' and that from 'x' to destination, whichever is less.

Illustration 11.4 Trace the steps of Floyd's algorithm for the graph depicted in Fig. 11.11.

Solution The weights associated with various edges have been given in the table that follows.
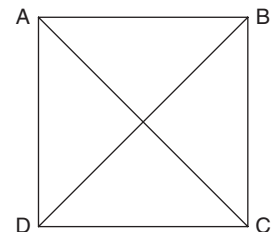
Figure 11.11 Graph in which the source is 'A' and the destination is 'D'

$$W =$$

	A	B	C	D
A	0	4	1	8
B	2	0	3	1
C	5	2	0	6
D	3	2	1	0

Step 1 The initial value of D is same as that of the weight matrix. That is,

$$D = W$$

$$D =$$

	A	B	C	D
A	0	4	1	8
B	2	0	3	1
C	5	2	0	6
D	3	2	1	0

Let the source be 'A'

$$x = \text{'B'}$$

Distance from A to C = min (1, 4 + 3) = 1;
Distance from A to D = min (8, 4 + 1) = 5; // in this case, the distance of A to D via B is less than the direct one. The matrix distance would now be updated. The new value of distance will be

.	A	B	C	D
A	0	4	1	5
B	2	0	3	1
C	5	2	0	6
D	3	2	1	0

$$x = \text{'C'}$$

Distance from A to B = min (4, 1 + 2) = 3; //The distance of A to B is less via C
Distance from A to D = min (5, 1 + 6) = 5;
The updated value of the distance matrix is therefore as follows.

.	A	B	C	D
A	0	3	1	5	← This row has the final distances
B	2	0	3	1
C	5	2	0	6
D	3	2	1	0

The reader is expected to apply the above steps to the rest of the rows as well.

11.9 MISCELLANEOUS PROBLEMS

The following discussion focuses on the application of the dynamic approach in solving common problems such as coin changing and the calculation of binomial coefficients. It may be stated here that the problems can be solved using other methods like greedy algorithms. However, the solution via the dynamic approach uses the values calculated earlier and hence is more efficient. The reader is requested to go through the solution of the problems using other approaches and then compare the results.

11.9.1 Coin Changing Problem

The coin changing problem has already been discussed in Section 10.7. The approach in that section was based on the greedy algorithms. The approach presented in this section is dynamic as it possesses an optimal substructure. It may be noted that if an array houses the various denominations in order to make the net value equal to 'C', then the array is divided into parts and each sub-array will in itself be optimal.

Proof If the left array is not optimal, and there exists a better array which can replace the left sub-array, then the argument of the optimality of the whole array will also not be true. Hence, the left sub-array must be optimal.

Similarly, if the right array is not optimal, and there exists a better array which can replace the right sub-array, then the argument of the optimality of the whole array will also not be true. Hence, the right sub-array must be also optimal.

The above arguments prove that the optimal array possesses optimal sub-arrays.

$$C[p] = 0 \text{ if } p = 0$$

$$\min i: {}_{di \leq p}\{1 + C[p - d_i]\} \text{ if } p > 0$$

11.9.2 Calculating Binomial Coefficients

The nth power of the sum of two variables is given by $(x + a)^n = \sum_{k=0}^{n} {}^nC_k x^k a^{n-k}$. This theorem is referred to as *Binomial Theorem*.

The coefficients ${}^n_rC$ are called *Binomial coefficients*. The coefficients can be calculated by using the following formula:

$$ {}^n_rC = \frac{n!}{(n-r)! \times r!} $$

However, using this formula would be computationally too expensive. It may be stated here that calculation of factorial of n, followed by that of $(n - r)$ and then dividing the two would be an $O(n!)$ task.

Another option would be using the recurrence relation for the calculation of ${}^n_rC$, which can be obtained as follows:

$$ {}^n_rC = \frac{n!}{(n-r)!\, r!} = \frac{n \times (n-1)!}{(n-r)! \times r \times (r-1)!} = \frac{n}{r} \times {}^n_{r-1}C $$

where $_0^nC = 1$.

This approach also takes $O(n)$ time. A better option would be to use dynamic programming to calculate the binomial coefficients. Here, dynamic approach can be used as it can be used to store the results of the calculations which would help us to calculate the rest of the values.

The table of binomial coefficients can be generated with the help of the following non-recursive procedure given in Algorithm 11.5.

Algorithm 11.5   Calculating binomial coefficients

Input: None
Output: X, a two-Dimensional Array
Let X be a two-dimensional array

```
for ( i=0; i<n; i++)
        {
        for( j=0; j<n j++)
                {
                if (i=1)
                        {
                X[i, j] =1;
                        }
        else
                        {
                        X[i, j] = X[i-1, j] + X[i, j-1];
                        }
                }
        }
```

Complexity: There are $(n^2/2)$ spaces to be filled in the array X.

For $n = 4$, the array has been shown in Fig. 11.12.
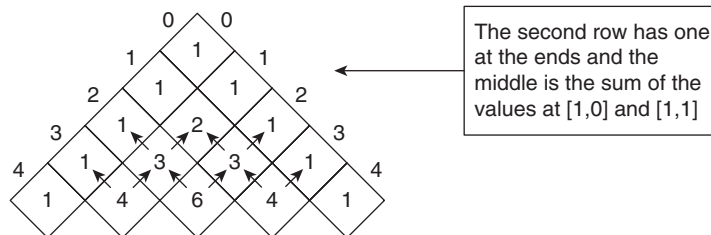
Figure 11.12   Array X, which stores the binomial coefficients

11.10 CONCLUSION

The chapter discusses the optimal substructure lemma approach to solve problems using dynamic approach. The above discussion exemplifies the approach. It can be seen in the

above examples that the values calculated in the earlier steps can be used in the present calculations. This is the reason for repeating the problems discussed in the previous chapters. The reader is requested to implement the algorithms and check the running time and the memory requirements of the algorithm implemented using dynamic approach and that using other approaches, in order to get a better insight of the advantages of this approach.

Points to Remember

- The development of a dynamic algorithm for a problem requires the following steps:
  - Conversion of the given problem into a search problem. The search would be from a large search space.
  - The search space is generally segregated into various sub-spaces which can be accomplished by developing a recursive algorithm.
  - In the final step, the recursive calls are characterized and a non-recursive procedure is developed for filling the table. The values of the table are filled in a way so that the calculations can be used at a later stage.
- The dynamic approach can work in two ways: bottom-up and top-down.
- In the bottom-up approach, the solution starts with the basic input value and builds up the larger solution.
- The top-down approach, on the other hand, takes the bigger problem into smaller sub-problems and then solves them.
- The divide and conquer approach requires division of the problems and so does dynamic programming; in the former, the sub-problems are independent, whereas in the latter they are not.

KEY TERMS

Longest common subsequence Two given strings can have more than one common subsequence. The LCS finds the largest from amongst them.

Optimal substructure lemma If T is an optimal binary search trees for keys $\{1, 2, 3, \ldots, n\}$ with root $T0$ and $T1$ and $T2$ are the left and the right sub-trees, respectively, then $T1$ and $T2$ are the optimal binary search trees for the keys $\{1, 2, \ldots, k-1\}$ and $\{k=1, \ldots, n\}$. The key associated with $T0$ is k.

EXERCISES

I. Multiple Choice Questions

1. The nth Fibonacci term is the sum of the $(n-1)$ and the $(n-2)$th Fibonacci term. The first and the second terms are 1 and 1, respectively. What would be the complexity of finding the nth Fibonacci term by a non-recursive procedure?
   (a) $O(n)$ (c) $O(n^3)$
   (b) $O(n^2)$ (d) $O(\varphi^n)$ where φ is the gold number

2. If the above procedure is implemented using a dynamic programming approach, what would be the time complexity of the algorithm?
   (a) $O(n)$ (c) O(n3)
   (b) O(n2)   (d) $O(\varphi^n)$ where φ is the gold number

3. In question 2, what would be the space complexity?
   (a) $O(n)$ (c) O(n3)
   (b) O(n2)   (d) $O(\varphi^n)$ where φ is the gold number

4. Hibonacci, an alleged relative of Fibonacci, devises the following series:

   $$f(n) = f(n-1) \times f(n-2), f(1) = 2 \text{ and } f(2) = 3$$

   He intends to calculate the sum of the first n terms of the above series. What would be the complexity of the recursive procedure, which accomplishes the task?
   (a) $O(n)$ (c) O(n3)
   (b) O(n2)   (d) None of the above

5. If dynamic programming is used to accomplish the above task, what would be the complexity of the best algorithm that can be implemented?
   (a) $O(n)$ (c) O(n3)
   (b) O(n2)   (d) None of the above

6. What would be the complexity of finding all the n binomial coefficients using dynamic programming approach?
   (a) $O(n)$ (c) $O(n^3)$
   (b) $O(n^2)$ (d) None of the above

7. What would be the complexity of finding a sequence of elements which has maximum sum from a given array?
   (a) $O(n)$ (c) $O(n^3)$
   (b) $O(n^2)$ (d) None of the above

8. If dynamic programming is used to solve the above problem, what would be the complexity of the best possible algorithm (which also makes use of the previously stored values)?
   (a) $O(n)$ (c) $O(n^3)$
   (b) $O(n^2)$ (d) None of the above

9. In the above problem, what would be the space complexity?
   (a) $O(n)$ (c) $O(n^3)$
   (b) $O(n^2)$ (d) None of the above

10. Which of the following is true?
    (a) The divide and conquer approach requires division of the problems so does dynamic programming.
    (b) In the divide and conquer approach, sub-problems are independent.
    (c) In the dynamic approach, the problems are not independent.
    (d) All of the above.

11. The dynamic approach can work in which of the ways?
    (a) Bottom-up   (c) Both
    (b) Top-down   (d) None of the above

12. Which of the following statements is true?
    (a) In the bottom-up approach, the solution starts with the basic input value and builds up the larger solution
    (b) In the top-down approach, we break the bigger problem into smaller sub-problems and then solve them
    (c) Both
    (d) None of the above

II. Review Questions

1. What is dynamic programming? Why is it called dynamic programming?
2. What are the elements of dynamic programming approach?
3. How does dynamic programming helps us to reduce the complexity of a problem?
4. The binomial coefficients are given by the formula $^nC_r = \dfrac{n!}{(n-r)!r!}$

    (a) Write a non-recursive algorithm for finding the above coefficients for $k = 1$ to n. Find the complexity of the procedure.
    (b) Write a recursive algorithm for finding the above coefficients for $k = 1$ to n. Find the complexity of the procedure.
5. Explain the dynamic approach to find the binomial coefficients.
6. The nth Fibonacci term is the sum of the $n - 1$th and $n - 2$th terms. The first and the second terms are 1 each.
7. Write a non-recursive algorithm for finding the above terms for $k = 1$ to n. Find the complexity of the procedure.
8. Write a recursive algorithm for finding the above term for $k = 1$ to n. Find the complexity of the procedure.
9. Explain the dynamic approach to find the nth Fibonacci term.
10. Prove optimal substructure lemma for the coin changing problem.
11. How is the dynamic approach better than the greedy approach for the coin changing problem?
12. Write a non-recursive procedure for calculating the values in the table for coin changing problem.
13. Prove optimal substructure lemma for the optimal binary search tree problem.
14. How is the dynamic approach better in terms of time complexity for the optimal binary search tree problem?
15. Write a non-recursive procedure for calculating the values in the table for the optimal binary search tree problem.
16. Explain dynamic programming approach for longest common sub sequence.
17. Explain dynamic programming approach for matrix chain multiplication.
18. Explain dynamic programming approach for Floyd–Warshall algorithm.
19. Explain dynamic programming approach for travelling salesman problem.
20. Explain the use of dynamic programming in calculating the nth Fibonacci term.

III. Numerical Problems

1. Find the longest common subsequence in the following strings:
   (a) ACACAACTGCACGAC and ACTGGCATG
   (b) ABHHKABADH and DGABHHGABAK
2. What would be the complexity of LCS if both the input strings are same? Also find the complexity if both the strings are completely different. Which of the above two cases, do you thing, constitute the best case.
3. Find optimal parenthesis structure for the following:

Matrix	Order
M1	2×3
M2	3×9
M3	9×4
M4	4×2

4. In the above question what happens if all the matrices have order 5×5? In such cases, do you think that the MCM algorithm explained in the text would be of any use?
5. Solve the following travelling salesman problem using dynamic approach. The cost matrix of the path is given as follows:

	A	B	C	D	E
A	0	2	3	5	32
B	2	0	45	2	12
C	3	45	0	12	17
D	5	2	12	0	23
E	32	12	17	23	0

6. Can the above problem be solved if the last values in the last three columns are infinity.
7. Using the cost matrix of problem 5 and take 'A' as the source. Apply Floyd's algorithm to find the shortest path from A to any other vertex.
8. In the above question, find the shortest distance of any vertex to any other vertex.
9. If in question 7, the second cell in the first row is (−2), would Floyd's algorithm work?
10. Compare the number of steps in question 7 with that on applying Prim's algorithm.

CHAPTER 12

Backtracking

OBJECTIVES

After studying this chapter, the reader will be able to

- Understand the concept of backtracking
- Apply backtracking to solve the subset sum problem
- Use backtracking to solve *N*-Queens problem
- Solve m-colouring problem and Hamiltonian cycle problem via backtracking
- Explain the problems in the approach

12.1 INTRODUCTION

The strategies such as divide and conquer and greedy approach, discussed in the earlier chapters, cannot solve all the problems. At times it is required to explore and analyse all possible outputs. In such cases, brute force can be used. Although it provides the solution but it is computationally expensive. In such situations, backtracking can be used. Backtracking is one of the most common strategies used for designing algorithms. It calls for the examination of each configuration of the solution sets, which satisfy the conditions imposed at the beginning of the problem. It may be noted that in backtracking, each configuration is generated once. When the final solution is produced, then the procedure is stopped; if not, we go back to the same point from where we started the step and examine the other configurations.

Section 12.2 examines the concept of backtracking with the help of illustrations. The example of mazes has been taken to explain the concept as it is one of the most common problems studied in algorithms as well as the implementation of game theory.

The chapter has been organized as follows. Section 12.3 discusses the subset sum problem. Section 12.4 examines the 8-Queens problem and a general variant of the problem called *N*-Queens problem. Section 12.5 discusses m-graph colouring. Section 12.6 deals with the Hamiltonian cycle problem and Section 12.7 deals with miscellaneous problems.

12.2 CONCEPT OF BACKTRACKING

The general backtracking algorithm is presented in the following abstraction. In the following algorithm, the first $(j-1)$ values have already been generated. The array $a[\ ]$ is a global array. It may also be noted that the following procedure uses recursion which makes the implementation of the concept easy.

Algorithm 12.1 Backtracking (j) Abstraction

```
for each a[1] perform the following steps:
  {
if B(a[i], a[2], … takes to the destination node

{
then print the answer
    if(j<n) then Backtracking (j+1);
}
  }
Backtracking Abstraction
```

> **Tip:** Backtracking can easily be implemented using recursion.

To understand the concept of backtracking, let us consider the following example. Consider a room shown in Fig. 12.1(a). The room has a door from which a person can enter (marked **In**) and a door from which he can come out (marked **Out**). There are many walls in the room depicted by lines. Now, he gets into the room and walks till he finds a path that can lead him to the door marked **Out**. Suppose the person starts walking along any turn that comes in the way.

The person then starts walking along the direction shown by the black arrow. He turns when he sees the first turn (Fig. 12.1(b)). It is quite possible that this path may lead to a dead end. This is what happens with the chosen path (Figs 12.1(c) and (d)).

When the person finds the dead end (Fig. 12.1(e)), then he backtracks and goes back along the path he came from (Figs 12.1(f)–(h)).
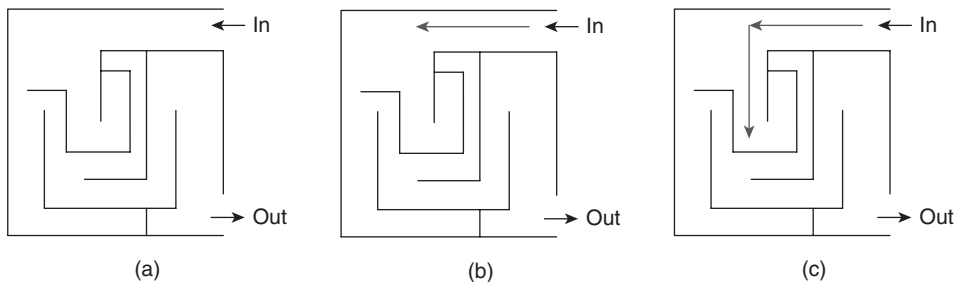
Figure 12.1 Maze problem
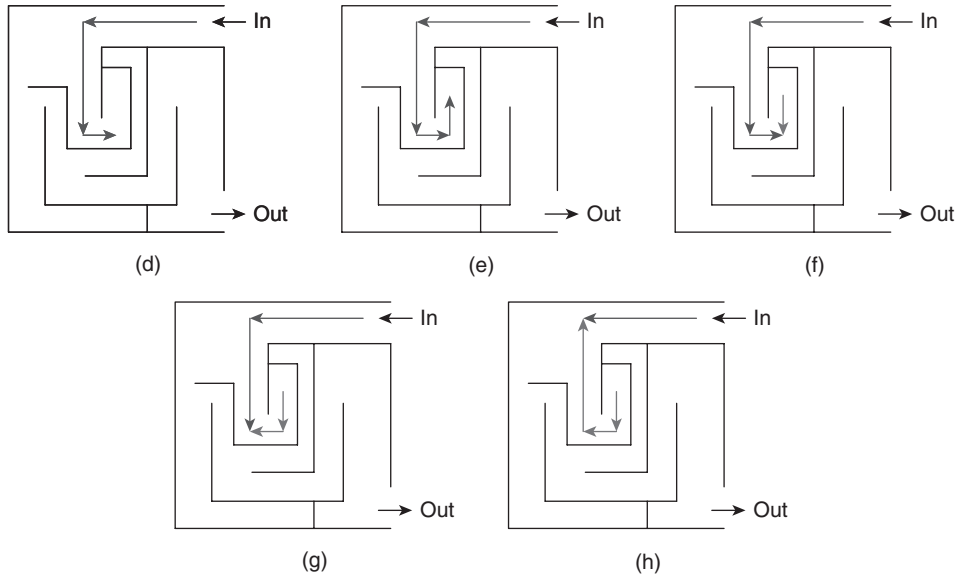
Figure 12.1 (contd)   Maze problem

When he comes back to the point from where he took that unfortunate turn, he decides moving forward (Fig. 12.1(i)) and takes the next turn (Fig. 12.1(j)).
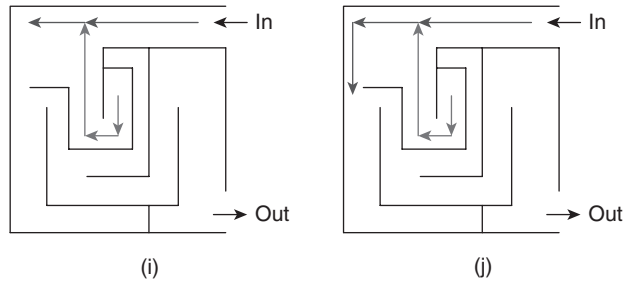
Figure 12.1 (contd)   Maze problem

He sees another turn and goes in the direction depicted by the arrow of Fig. 12.1(k). However, that turn also results in a dead end. So he decides to backtrack again (Fig. 12.1(l)).
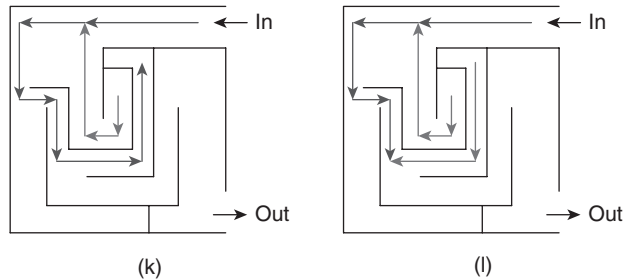
Figure 12.1 (contd)   Maze problem

The next path chosen by the person (Fig. 12.1(m)) takes him to the door (Fig. 12.1(n)) and he comes out.

(m)   (n)
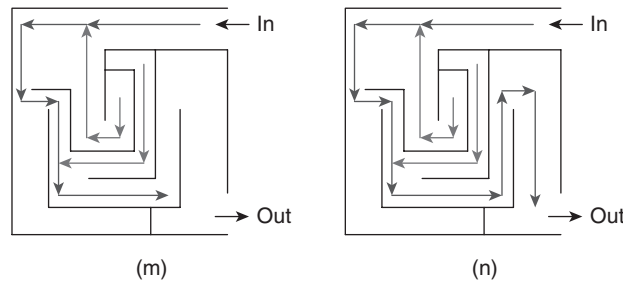
Figure 12.1 (contd)   Maze problem

The strategy followed by the person is more or less same as that followed by us in our life. For instance, we may start a business with someone when an appropriate opportunity comes and may not find it good enough at a later stage. Then, we come back to the same point and look for another alternative. This makes sense as we cannot say anything about a path until we travel on that path. The path may lead to the destination, or it may not. However, on encountering a dead end, we must backtrack and look for another path in order to achieve the goal. This is what backtracking is all about. So in a way it is inspired by nature.

In backtracking, there is a set of conditions that the values of x_i must follow in order to be a part of solution set: $s = \{x_1, x_2, ..., x_n\}$. These conditions are of two types: implicit and explicit.

The conditions that control the selection of elements are referred to as constraints. The elements x_i must take values only from the domain set. This is referred to as *explicit constraints*. The conditions that determine how various x_i's should be related to each other are referred to as *implicit constraints*. For example, in Fig. 12.1, the paths can be chosen from amongst the given paths. These paths must be such that for any two connective elements, x_i and x_{i+1}, the end of path x_i must be the beginning of the path x_{i+1}.

12.3 SUBSET SUM PROBLEM

The subset sum problem is one of the most important problems in algorithm analysis and design as it is used in many disciplines from computer science to computational biology. The problem calls for finding a subset of a given set which has the sum equal to the target sum. There are several ways to solve subset sum problem. The problem can be solved via greedy approach, dynamic approach, backtracking, and branch and bound. The present section presents the backtracking approach to solve the problem. Although the brute force algorithm would form all the subsets and then find the sum of each of the subsets, it is computationally expensive as the number of subsets of a set having n elements is 2^n. The backtracking approach solves the problem in much lesser moves. In order to understand

the concept, let us consider a set having four elements. The sum of the elements selected in the result set should be same as that given as the input to the problem (target sum). The state space tree depicted in Fig. 12.2 shows the various possible solutions.
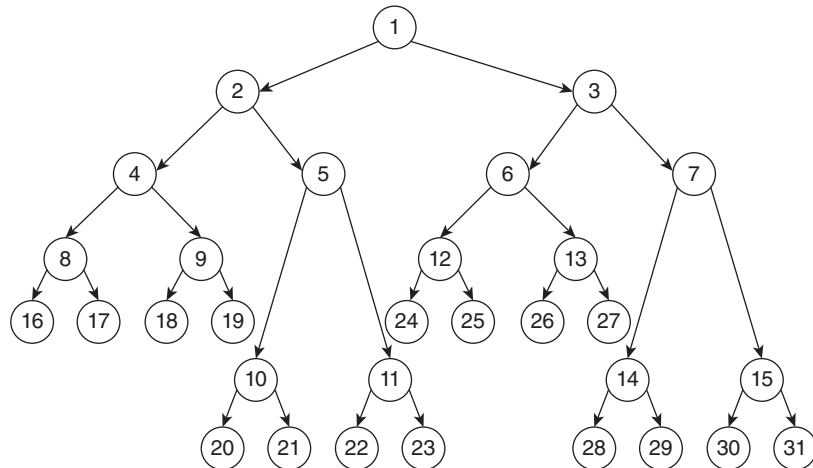
Figure 12.2 State space tree of a subset problem having four elements

The above state space tree represents the backtracking approach of solving the subset sum problem. The root node (number 1) depicts the decision regarding the inclusion or non-inclusion of the first element of the set in the result. If the first element is selected, then node 2 is processed; else node 3 is processed. The nodes at the next level (node number 2 and node number 3) depict the decision regarding the inclusion or the non-inclusion of the second element of the set in the result. The next level decides the inclusion or non-inclusion of element number 3 of the given set and the last level decides whether the fourth element of the original subset will be there in the solution or not. Table 12.1 summarizes the solution depicted by the last level of the tree (called the leaf nodes).

Table 12.1   Solutions depicted by
various leaf nodes of the state space tree

Nodes	Result sets
16	$\{x_1, x_2, x_3, x_4\}$
17	$\{x_1, x_2, x_3\}$
18	$\{x_1, x_2, x_4\}$
19	$\{x_1, x_2\}$
20	$\{x_1, x_3, x_4\}$
21	$\{x_1, x_3\}$
22	$\{x_1, x_4\}$

(*Contd*)

Table 12.1 Contd

23	$\{x_1\}$
24	$\{x_2, x_3, x_4\}$
25	$\{x_2, x_3\}$
26	$\{x_2, x_4\}$
27	$\{x_2\}$
28	$\{x_3, x_4\}$
29	$\{x_3\}$
30	$\{x_4\}$
31	$\{\ \}$

The above set is nothing but various subsets of the original set $\{x_1, x_2, x_3, x_4\}$. If 1 indicates the inclusion of the element and 0 indicates the non-inclusion of the element, then Table 12.2 depicts the above result.

Table 12.2 Binary depiction of the result

Node	$x_1\ x_2\ x_3\ x_4$
16	1111
17	1110
18	1101
19	1100
20	1011
21	1010
22	1001
23	1000
24	0111
25	0110
26	0101
27	0100
28	0011
29	0010
30	0001
31	0000

Backtracking Approach

The above problem requires finding the subset having the required sum. The solution starts with the processing of node 1. If the first node is selected then we move to node number 2; else to node number 3. In the process if the total sum of the values selected

becomes larger than the required sum then we move back to the parent node and then to an alternate path. However, the approach is computationally expensive. The process is depicted in Algorithm 12.2.

Algorithm 12.2 Subset sum problem by backtracking

Input: The set w[ ], the number of elements n, and the desired sum and the weight of the knapsack.
Output: The result set consisting of elements that can be selected to contain the desired weight and keeping the weight less than or equal to the weight of the knapsack.

```
ssp(sum, k, r)
        {
// left child
        flag[k] := 1;
        If(sum + wt[k] = m)
                Write (flag[i : k]);
        else if(sum + wt[k] + wt[k+1] <= m)
                ssp(sum + wt[k], k + 1, r - wt[k]);

// right child
        If((sum + r - wt[k] >= m) and (sum + wt[k + 1] <= m))
                {
                flag[k] := 0;
                ssp(sum, k + 1, r - wt[k]);
                }
        }
```

It may be noted that from a particular node we can either move to the left or to the right. At any instance when the sum becomes equal to the target sum, then the result is printed; else the next level is processed. The one-dimensional array flag indicates the inclusion or the non-inclusion of an element.

> **Tip:** The solution of subset sum using backtracking can also be used to solve sudoku problems.

12.4 *N*-QUEENS PROBLEM

The 8-Queens problem is one of the most common problems that can be solved by backtracking. The problem requires 8-Queens to be placed on a 8×8 chessboard in a way that no queen can attack each other. The problem was crafted by Max Bazzel in 1948. A general version of the problem which involved an $n \times n$ chessboard was given by Franz Nauck. Two queens can attack each other if they are in the same row or same column or same diagonal.

In order to explain the algorithm, a definite representation is required. A cell is the basic entity in a chessboard. A cell in a chessboard can be represented by two coordinates (x, y), where x represents the row number and y represents the column number. For example $(2, 1)$ represents the cell at the first column of the second row.

It may be deduced from the above discussion that 2-Queens if placed at (x_1, y_1) and (x_2, y_2) and, then the following constraints hold so that no 2-Queens can attack each other:

$$x_1 \neq x_2 \tag{12.1}$$

$$y_1 \neq y_2 \tag{12.2}$$

$$|y_2 - x_2| \neq |y_1 - x_1| \tag{12.3}$$

Equation (12.1) depicts the condition that 2-Queens cannot be placed in the same row. Equation (12.2) depicts the condition that 2-Queens cannot be placed in the same column and Eq. (12.3) depicts the condition that 2-Queens cannot be placed at the same diagonal.

We can begin the algorithm by placing a queen at the first column of the first row. Now the second queen must be placed in the second row. It cannot be placed at the first column and the second column as conditions depicted by Eqs (12.2) and (12.3) are violated by doing so. So, we place the second queen at the third column of the second row. The queen of the third row will be placed at the fifth column. The process continues till a particular queen cannot be placed in any of the columns of a particular row. In that case we backtrack and place the queen at the last row at the next best place. The solution of 8-Queens problem by backtracking has been discussed in Figs 12.3(a)–(h).

Figure 12.3(a) depicts the initiation of the algorithm. The first queen is placed at the first cell of the first row.
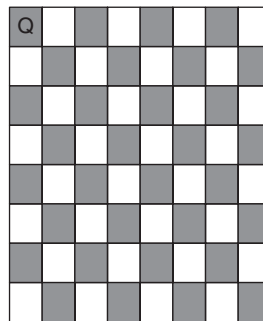
(a)

Figure 12.3(a)   8-Queens Problem

Now we move to the second row. The second queen cannot be placed in the first cell of the second row as it will be in the same column as the first queen. In addition, it cannot be placed in the second column since it will be in the same diagonal as the first queen. The second queen is therefore placed in the third cell of the second row (Fig. 12.3(b)).
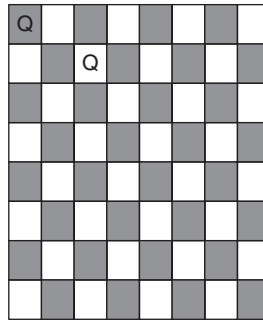
(b)

Figure 12.3(b)   8-Queens Problem

Now we move to the third row. The third queen cannot be placed in the first cell of the third row as it will be in the same column as the first queen. In addition, it cannot be placed at the second column since it will be at the same diagonal as the second queen. At the third and the fourth cells also it can be attacked by the second queen. The third queen is therefore placed at the fifth cell of the third row (Fig. 12.3(c)).
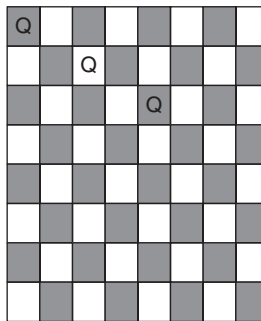
(c)

Figure 12.3(c)   8-Queens Problem

As per the fourth queen is concerned, it cannot be placed at the first cell as it would be attacked by the first queen. So, the fourth queen will be placed at the second cell of the fourth row (Fig. 12.3(d)).
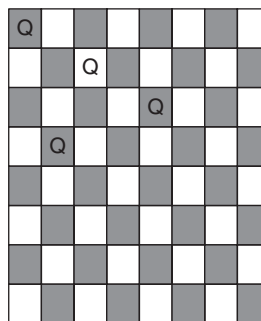
(d)

Figure 12.3(d)   8 Queens problem

Figures 12.3(e) and 12.3(f) depict the position of the fifth and the sixth queens that can be placed, respectively. Since a queen cannot be placed at the same row or column or same diagonal as any other queen, they will be placed respectively at the cell numbers 5 and 7, respectively.

(e)   (f)
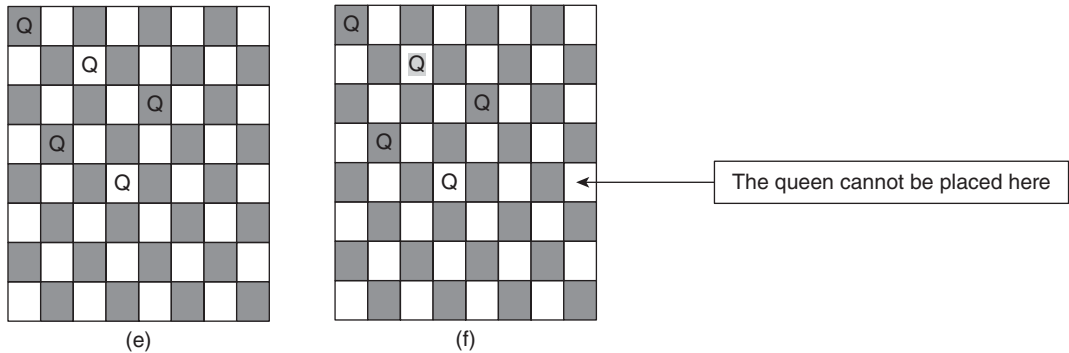
Figure 12.3(e)(f)   8 Queens problem

Here comes the point where backtracking is required. We will not be able to place the seventh queen at any of the cells. Now, we will have to go back one step and see if we can place the sixth queen in any other row. Since this is not feasible we will have to change the position of the fifth queen (Fig. 12.3(g)).

(g)
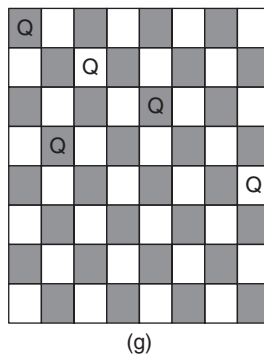
Figure 12.3(g)   8-Queens Problem

The reconfiguration will again lead us to a situation whereby the sixth queen cannot be placed. In order to proceed further, the fourth queen will have to be placed in the next best column. This results in the placement of the fourth queen in the seventh cell of the fourth row. Placing the rest of the queens in this fashion leads to a situation wherein the last queen cannot be placed (Fig. 12.3(h)).
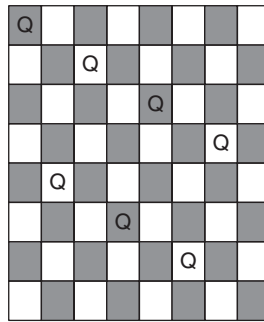
(h)

Figure 12.3(h)   8-Queens problem

The continuous application of the backtracking algorithm will lead us to the following situation. The solution can be represented by the set $\{1,5,8,6,3,7,2,4\}$. The solution set has the following properties:

- It has 8 elements (same as that of the number of queens in the problem)
- All 8 numbers appear in the set.
- The placement of the numbers is such that the element represents the cell number of a particular row.

For example, the first element of the set is 1 indicating that the first queen is to be placed in the first cell of the first row. Similarly, the second queen is to be placed at the fifth cell of the second row, and so on.

The solution is one from amongst the possible permutations of eight numbers.

The last point suggests that generation of permutations followed by the constraint checking can also lead us to the solution, but the computational complexity of the process would be too large. Backtracking can lead us to the solution in much lesser calculations. The formal algorithm to solve the above problem is presented in Algorithm 12.3. The algorithm makes use of the place algorithm which checks whether a queen can be placed at the position or not. Algorithm 12.4 describes the procedure.

Algorithm 12.3   *N*-Queens via backtracking

```
Algorithm N-Queens (k, n)
{
    for i := 1 to n step 1 do
    {
    If (place(k, i))
    {
        x[k] := i;
        if (k = n)
            Print(x[1 : n]);}
        else
```

```
            {
            {
            NQueens(k + 1, n);}
        }
        }
}
```

Algorithm 12.4   Place algorithm used by the *N*-Queens algorithm

```
Algorithm Place (k, i)
{
        for( j := 1 to k - 1 step 1 )
        {
        If ((x[j] = i) || (Abs(x[j] - i) = (Abs(j - k)))
        {
                    return false;
        }
        }
        return true;
}
```

Illustration 12.1   A 5 × 5 chessboard is such that the middle row and middle column have been marked inaccessible. That is, you cannot place a queen at these places. Use backtracking algorithm to place 4-Queens on this chessboard in a way that no queen attacks each other.

Solution Let us place the first queen at the first cell of the first row (Fig. 12.4(a)).
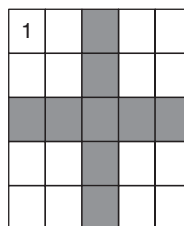
(a)

Figure 12.4(a)   Solution of Illustration 12.1

Now, the next queen is to be placed in the second row. Since we cannot place the queen at the first or the second cell and it is not allowed to place queen at the third cell, the next queen can be placed at the fourth cell of the second row (Fig. 12.4(b)).

(b)
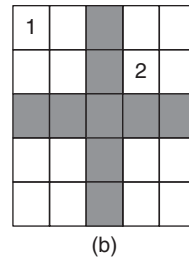
Figure 12.4(b) Solution of Illustration 12.1

The next queen cannot be placed in the third row owing to the conditions imposed at the beginning; therefore, it can be placed at the fourth row. The queen will be attacked by another queen, if it is placed at the first or the second cell. The third cell is already prohibited. Therefore, the queen can be placed at the fifth cell of the fourth row (Fig. 12.4(c)).

(c)

Figure 12.4(c) Solution of Illustration 12.1

As per the last queen is concerned, it cannot be placed at the first cell but can be placed at the second cell of the fifth row (Fig. 12.4(d)).

(d)

Figure 12.4(d) Solution of Illustration 12.1

Illustration 12.2 A 5 × 5 chessboard is such that the first row and first column have been marked inaccessible. That is, you cannot place a queen at these places. Use backtracking algorithm to place 4-Queens on this chessboard in a way that no queen attacks each other.

Solution Let us place the first accessible queen at the first accessible cell of the first row (Fig. 12.5(a)).
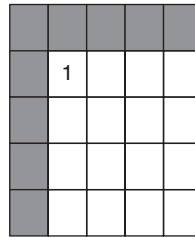
(a)

Figure 12.5(a)   Solution of Illustration 12.2

Now, the next queen is to be placed in the second row. Since we cannot place the queen at the first or the second cell and it is not allowed to place queen at the third accessible cell, the next queen can be placed at the fourth cell of the second row (Fig. 12.5(b)).
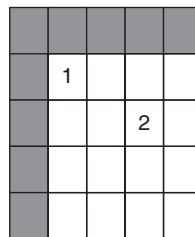
(b)

Figure 12.5(b)   Solution of Illustration 12.2

The next queen cannot be placed in the third accessible row owing to the conditions imposed at the beginning; therefore, it can be placed at the fourth row. The queen will be attacked by another queen, if it is placed at the first or the second accessible cell. The third cell is already prohibited. Therefore, the queen can be placed at the fifth cell of the third row (Fig. 12.5(c)).
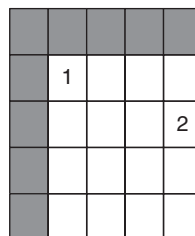
(c)

Figure 12.5(c)   Solution of Illustration 12.2

As per the last queen is concerned, it cannot be placed at the first cell but can be placed at the second cell of the fourth row (Fig. 12.5(d)).
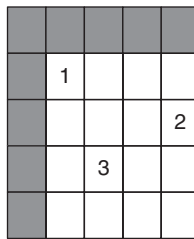
Figure 12.5(d) Solution of Illustration 12.2

The solution can be found by backtracking till the first level and placing the queens as per the Figs 12.5(e)–(h).
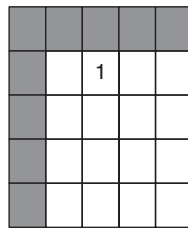
(e)
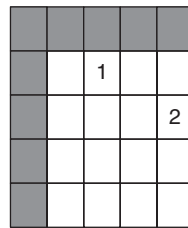
(f)
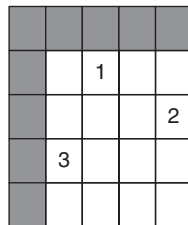
(g)
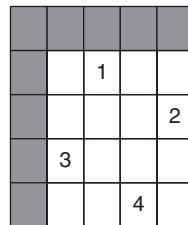
(h)

Figure 12.5(e)(f)(g)(h) Solution of Illustration 12.2

12.5 m-COLOURING PROBLEM

m-Colouring problem requires filling different colours in a planar map. To understand the problem, let us take the map shown in Fig. 12.6. The map is to be coloured in such a way that no two adjacent regions have the same colour. That is, the colour that is chosen for region 1 cannot be used to colour region 2 or 3 or 4. If colour C1 is selected for region 1, then C1 cannot be used to

Figure 12.6 Map in which different regions are to be assigned colours so that the number of colours is minimum and no two adjacent regions have same colour
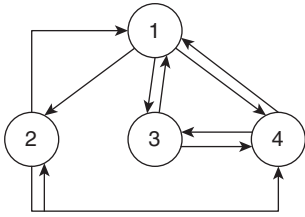
Figure 12.7 Graph corresponding to map shown in Fig. 12.6

fill the region 2. So, colour C2 is selected for the region 2. For the third region also C2 can be chosen as regions 2 and 3 are not adjacent. For region 4, C1 or C2 cannot be chosen, so colour C3 is selected. Therefore, the map shown in Fig. 12.6 can be filled with three colours.

It may be noted at this point that any map can be represented as a graph. The regions of the map can be mapped to the nodes of the corresponding graph. The map shown in Fig. 12.6 can be converted into a graph as shown in Fig. 12.7.

The graph corresponding to the given map can be stored in a two-dimensional matrix. The matrix will help us to find out the edges adjacent to a vertex. In order to accomplish the task of finding out whether the map can be filled by the given number of colours, the first vertex can be assigned a colour. Now the next vertex is given the first colour. However, if the second vertex is adjacent to the first vertex (to which colour 1 was initially assigned), the second vertex is assigned colour 2. In the above case, since the second vertex is adjacent to the first vertex, it is allotted the second colour. Now, the third vertex is also adjacent to the first vertex; therefore, it cannot be assigned the first colour, however, it can surely be allotted the second colour as it is not adjacent to the second vertex. Regarding the fourth vertex, it cannot be given any of the three colours allotted so far as it is adjacent to the rest of the three vertices.

The concept of the problem can be explained as follows using Algorithm 12.5.

Algorithm 12.5 Next vertex used by m-colouring

```
For each vertex
vᵢ, i from 1 to n, n being the number of vertices

   Assign cᵢ; i = 1 to vᵢ

      if cᵢ has been assigned to vⱼ, vⱼ is adjacent to vᵢ

            i = i + 1

            if no color is left then quit
```

The formal algorithm has two parts: next value and m-colouring. The next value algorithm assigns the colour to a particular vertex. The algorithm assumes that there are only k colours available from amongst the colours that can be assigned to the different regions of a map or to the vertices of the corresponding graph. The colours are stored in the array $x[\ ]$. $x[k] = 0$ indicates that no colour is left and hence the algorithm should terminate. Algorithm 12.6 presents the next vertex procedure and Algorithm 12.7 shows the main algorithm.

Algorithm 12.6 m-Colouring via backtracking

```
{
  while(true)
  {
    NextValue(k);
    if(x[k] = 0){
return;}
    if(k = n){
        Print(x[1 : n]);}
    else
    {
        mColoring(k + 1);}
    }
}
```

Algorithm 12.7 NextValue(k)

```
{
  while(true)
  {
    x[k] := (x[k] + 1) mod (m + 1);
    if(x[k ] = 0) {
        return;}
    for (j := 1 to n step 1 )
    {
        if((G[k,j] ≠0) && (x[k] = x[j])) {
        break;}
    }
    if(j = n + 1) {
        return;}
  }
}
```

12.6 HAMILTONIAN CYCLE

Before starting off with the application of backtracking in 'Hamiltonian cycle', let us go through the concept of Hamiltonian cycle. The Hamiltonian cycle can be defined as follows:

Given: A graph $G = (V, E)$; where V is the set of vertices and E is the set of edges, each element of E is in the form (x, y) such that $x, y \in V$.

To find: A sequence of vertices s $\{v_1, v_2, v_3, ..., v_n\}$ such that

- No two v_i's are same
- $\forall v_i,$ there is direct edge between v_i and v_{i+1}
- No edge is repeated
- v_n and v_1 are directly connected

There is, however, a possibility that in a particular graph, there is no cycle which satisfies all the above conditions. For example, the graph has a Hamiltonian cycle.

The graph shown in Fig. 12.8 is $G = (V, E)$ where V is the set $\{A, B, C, D\}$ and E is the set $\{(A, B), (A, C), (A, D), (B, D), (D, C)\}$.
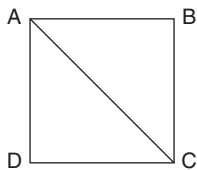
Figure 12.8 A graph which contains a Hamiltonian cycle

The cycle ABCDA is a Hamiltonian cycle as

- All the vertices in the set are different except for the first and the last
- It covers all the vertices
- No edge is repeated
- The first vertex is joined with the last vertex hence it is a cycle.

As stated earlier, a graph may not have a Hamiltonian cycle. Figure 12.9 depicts a graph with no Hamiltonian cycle.
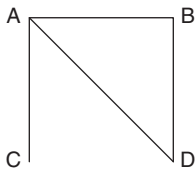
Figure 12.9 A graph that does not contain Hamiltonian cycle

Finding out whether a graph has a Hamiltonian cycle or not is computationally expensive task. The problem can be solved by backtracking as well. The other way out is to enlist all the permutations of the vertices and check which of them satisfies the above conditions. The number of permutations of a set having n vertices is ${}_{n}^{n}p = n!$. The generation of these permutations is not the only thing that needs to be done. Every possible permutation will have to be checked for the conditions stated above. The computational complexity of the above task is very large. Even if a graph contains 20 vertices, the generation of permutations will require 2.429×10^{14} calculations. A PC will take 78,002 years to carry out the calculations assuming that one calculation is performed in 10^{-6} seconds.

Doing the same task by backtracking requires much lesser calculations and hence helps us to reach to the solution.

12.6.1 Solution of Hamiltonian Cycle Using Backtracking

The solution set should contain n elements. These n elements represent the order in which the graph needs to be traversed. The last element of the set should be directly connected to the first element. In order to obtain the solution, the first vertex is to be selected. The next vertex is selected if the following three conditions are satisfied:

- There is still a vertex that can be selected.
- There is a direct edge from the selected vertex to the next vertex.
- The set formed so far has less than n vertices.

The 'next feasible vertex' algorithm tests the above conditions (Algorithm 12.8). The parameter r tests the above conditions for the $x[r]$th element of the set x being formed. The 'Hamiltonian algorithm' depicts the steps in order to generate the solution set (Algorithm 12.9). If all the n elements can be generated by the 'next feasible vertex' algorithm, then the final solution is obtained.

The inability of the 'next feasible vertex' algorithm to produce a set of n vertices means that the given graph does not contain a Hamiltonian cycle.

Algorithm 12.8   Hamiltonian problem

```
Algorithm Hamiltonian(r)
  {
     while(true)
     {
     NextValue(r);
     if(x[r] = 0) {
return;
           }
     if(r = n){
           Print(x[1 : n]);
           }
     else
           {
Hamiltonian(r + 1);
           }
  }
}
```

Algorithm 12.9   Next feasible vertex algorithm

Algorithm NextFeasible vertex(r)
```
{
  while(true)
  {
    x[r] := (x[r] + 1) mod (n + 1);
    if(x[r ] = 0) {
          return;}
    if(G[x[r - 1], x[r]] ≠ 0)
    {
          for j := 1 to r - 1 step 1 do
```

```
                    if(x[j] = x[r]) {
                            break;}
            if(j = r) then
                    if((r < n) || ((r = n) && G[x[n], x[1]] ≠ 0))
                            return;
        }
    }
}
```

12.7 MISCELLANEOUS PROBLEMS

This section discusses some miscellaneous problems that can be solved using backtracking.

12.7.1 Knapsack Problem

Knapsack problem is one of the most famous optimization problems. The problem is as follows:

A thief has to select a few things to put in his bag having capacity of m units. He can select the things from amongst the n things in the given set $\{x_1, x_2, x_3, ..., x_n\}$. The weights of the elements of the above set are given by the set $\{w_1, w_2, w_3, ..., w_n\}$ and the profits obtained by picking an item are $\{p_1, p_2, p_3, ..., p_n\}$. As in the case of subset sum problem, $x_n = 1$ denotes the inclusion of the item in the final set and $x_n = 0$ means that the item has not been selected. The problem is to select a subset of items (or perhaps all the items) in such a way that the total weight of the selected items is less than or equal to the weight of the bag (which is m in this case). The constraints are depicted in Eq. (12.4),

$$x_1 \times w_1 + x_1 \times w_2 + x_3 \times w_1 ... \leq m \tag{12.4}$$

x_i can either be 1 or 0.

In addition, we need to pick the items in such a way that the profit earned is maximum, i.e.,

$x_1 \times p_1 + x_1 \times p_2 + x_3 \times p_1 ...$ is maximum

The above problem is referred to as 0/1 knapsack problem.

Take, for example, a set of profits given by the set $p = \{10, 10, 12, 18\}$.

The set of weights is given by $w = \{2, 4, 6, 9\}$.

Let the value of m be 15.

The problem m can be solved by making a state space tree like in the case of subset sum problem (Fig. 12.2). The state space tree represents the backtracking approach of

solving the knapsack problem as well. The root node (number 1) depicts the decision regarding the inclusion or non-inclusion of the first element of the set in the result. If the first element is selected, then node 2 is processed; else node 3 is processed. The nodes at the next level (node number 2 and node number 3) depict the decision regarding the inclusion or the non-inclusion of the second element of the set in the result. The next level decides the inclusion or non-inclusion of element number 3 of the given set and the last level decides whether the fourth element of the original subset will be there in the solution or not. The backtracking comes into picture when the weight of the subset being selected exceeds the value m.

The problem can be solved much more efficiently by the branch and bound approach presented in Chapter 13. Some of the authors solve the problem by calling the bound procedure that restricts the processing of some of the nodes. The approach is similar to backtracking but is not strictly backtracking.

12.7.2 Other Problems

Job Assignment

The problem calls for assigning n jobs to n people in such a way that the net cost of assigning jobs is minimized. The cost of assigning ith job to jth person is given by $cost(i,j)$. This is an optimization problem. The backtracking approach calls for creating a state space tree of the problem and then processing each node and finding out the result. However, there is a catch if the value n, the number of jobs is large then the algorithm becomes computationally expensive. The problem has been discussed in Appendix A7.

Isomorphism

Two graphs G and G' are said to be isomorphic if there is a one-to-one correspondence between the vertices. The problem calls for finding out a backtracking-based solution of the problem which checks whether the two given graphs are isomorphic or not.

Maximum Clique in a Complete Graph

Finding a fully connected proper sub-graph from a graph is referred to as maximum clique problem. However, in a fully connected sub-graph having n vertices, the number of subsets of the vertices would be 2^n. All of the above subsets except for the n subsets which have only one vertex and one having 0 vertex. All the legal subsets can form their graph. However, whether those graphs form a clique or not should be checked. This must be followed by selecting the maximum clique.

The process can also be solved by backtracking. In order to do so, a state space tree of the problem needs to be constructed. However, a better solution of the problem is presented in Chapter 19 of the book.

12.8 CONCLUSION

This chapter presented one of the most interesting approaches of algorithm design—backtracking. The problems that require the final answer to be selected from amongst the various possible permutations are generally solved by the approach. There are two methods of implementing backtracking: recursive and iterative. However, the selection of the method of implementation depends on the problem at hand and the resources (such as the CPU and the amount of memory) available. Even though the approach solves the problems in a better way, there are approaches that are much better than backtracking. Chapter 13 discusses the concept of branch and bound that makes use of bound function, which reduces the number of nodes that need to be processed hence saving time. Having said that, the approach solves the *N*-Queens problem that would be almost impossible to handle via the approaches discussed in the previous chapters.

Points to Remember

- Backtracking can be easily implemented using recursion.
- There are techniques such as branch and bound that are more efficient than backtracking.
- The solution of *N*-Queens problem using backtracking requires two procedures. The first checks whether a queen can be placed at that position and the second is a recursive procedure for the main algorithm.
- The solution of Hamiltonian cycle and m-colouring also requires two procedures.

KEY TERMS

Backtracking Generating all the children in the state space tree and reverting back to the parent if the child is unable to lead to the solution.

Hamiltonian cycle Given: A graph $G = (V, E)$; where V is the set of vertices and E is the set of edges, each element of E is in the form (x, y) such that $x, y \in V$.

To find A sequence of vertices $s \{v_1, v_2, v_3, ..., v_n\}$ such that

- No two a $v_i's$ are same
- $\forall v_i$ there is a direct edge between v_i and v_{i+1}
- No edge is repeated
- v_n and v_1 are directly connected

m-Colouring problem Colouring an *m* vertex graph in such a way that no two adjacent vertices have same colour and the number of colours used is *m* is called m-colouring problem.

N-Queens problem The problem of placing *N*-Queens in an $N \times N$ chessboard so that no two queens attack each other is called *N*-Queens problem.

EXERCISES

I. Multiple Choice Questions

1. Who coined the term backtracking?
   - (a) D.H. Lehmer
   - (b) R.J. Walker
   - (c) Prim
   - (d) None of the above

2. If a problem requires searching from a set or ask for an optimal solution, then which of the following approaches would yield best results?
   - (a) Backtracking
   - (b) Greedy approach
   - (c) Dynamic approach
   - (d) None of the above

3. Which of the following problems can be solved by backtracking?
   - (a) 8-Queens
   - (b) Sum of subset
   - (c) Graph colouring
   - (d) All of the above

4. Which of the following are constraints of an N-Queens problem? If a queen has already been placed at (x, y) then the other is to be placed at (m, n) This can be done provided
   - (a) $x \neq m$
   - (b) $y \neq n$
   - (c) $|y - n| = |x-m|$
   - (d) All of the above

5. Which of the following is best suited for N-Queens problem?
   - (a) Backtracking
   - (b) Branch and bound
   - (c) Both
   - (d) None of the above

6. What is the complexity of N-Queens problem via backtracking?
   - (a) $O(n)$
   - (b) $O(n^2)$
   - (c) $O(1)$
   - (d) None of the above

7. What is the complexity of the subset sum problem via brute force?
   - (a) $O(2^n)$
   - (b) $O(n2^n)$
   - (c) $O(n^2)$
   - (d) None of the above

8. What is the number of subsets of a set having cardinality n?
   - (a) 2^n
   - (b) n^2
   - (c) N
   - (d) None of the above

9. If a graph has n vertices and there are m colours to be filled, which of the following is the correct complexity of graph colouring problem?
   - (a) $O(nm^n)$
   - (b) $O(mm^n)$
   - (c) $O(nn^n)$
   - (d) None of the above

10. Which of the following needs backtracking to be solved?
    - (a) Hamiltonian cycle
    - (b) Euler's cycle
    - (c) Both of the above
    - (d) None of the above

II. Review Questions

1. Explain the process of backtracking. What are the advantages of backtracking as against brute force algorithms?

2. Explain the solution of travelling salesman problem via backtracking.

3. Explain the solution of maze problem via backtracking.
4. Explain the solution of Hamiltonian cycle problem via backtracking.
5. Explain the solution of graph colouring problem via backtracking.
6. Explain the solution of knapsack problem via backtracking.
7. Can backtracking be implemented without recursion?
8. Analyse the solution of knapsack problem via backtracking and find its complexity.
9. Analyse the solution of Hamiltonian cycle via backtracking and find its complexity.
10. Analyse the solution of graph colouring problem via backtracking and find its complexity.

III. Application-Based Questions

1. Implement N-queens problem via backtracking.
2. Implement graph colouring problem via backtracking and find solutions of the following maps.
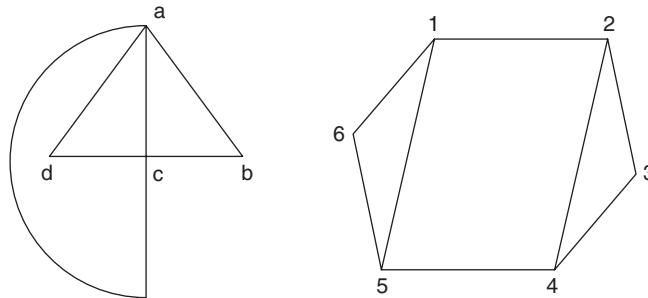
3. Implement Hamiltonian cycle via backtracking and find the solution of the following problems.
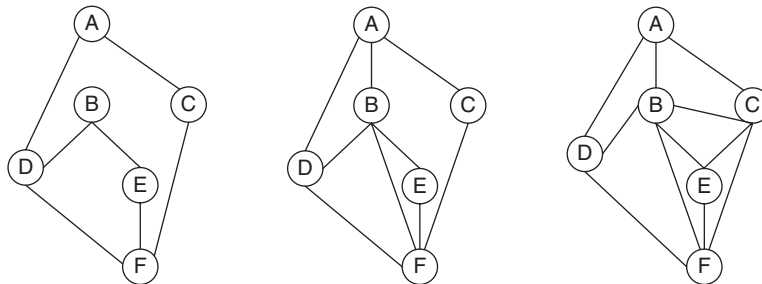
4. Implement maze problem given in Section 12.2 via backtracking.
5. Implement knapsack problem via backtracking.

Answers to MCQs				
1. (a)	3. (d)	5. (a)	7. (a)	9. (a)
2. (a)	4. (d)	6. (d)	8. (a)	10. (a)

CHAPTER 13

Branch and Bound

OBJECTIVES

After studying this chapter, the reader will be able to

- Understand the concept and importance of branch and bound
- Give the idea of optimization and relaxation
- Apply branch and bound to solve the following problems:
  - o Travelling salesman
  - o Knapsack
  - o 8-puzzle
- Recognize the complexity considerations of branch and bound

13.1 INTRODUCTION

The state space tree helps to find the solution using the backtracking approach, as discussed in Chapter 12. The branch and bound approach is similar to the backtracking approach in the sense that it also uses state space tree. However, there are some differences between the two approaches.

The branch and bound approach is also used for unconstrained non-convex optimization problems. Optimization has formally been discussed later in the chapter. Moreover, the way a state space tree is traversed is also different. Branch and bound gives us flexibility to traverse the tree in different ways. In this method, it is also seen whether the node, currently being processed, is the most promising. This task is accomplished by computing a BOUND at a node.

The rest of the chapter has been organized as follows. Section 13.2 discusses the concept of branch and bound and Section 13.3 discusses the travelling salesman problem. Section 13.4 discusses knapsack problem. Section 13.5 uses the approach to solve the 8-puzzle problem. Section 13.6 discusses the efficiency issues in branch and bound. Section 13.7 discusses optimization and relaxation and the last section concludes.

13.2 CONCEPT OF BRANCH AND BOUND

The idea is to put a bound on a node so that if the bound is not better than the best value obtained so far, then it is not considered promising. In this methodology, the children of a node are generated only if the node is considered promising. At times, the children of the best nodes are compared with the bounds of the nodes which are deemed promising.

As stated in Section 7.6 of Chapter 7, there are two major ways in which a given graph can be traversed. They are depth first search and breadth first search. Branch and bound technique is an extension of breadth first search.

The approach is better than backtracking as in this approach, the nodes, which are not considered promising, are not explored. The bound on a node guarantees that a solution obtained from expanding the node would be greater than a particular number. This is referred to as lower bound. In case of upper bound, the number generated by the bounding function should be less than a particular number. There are three types of traversals used to process the nodes of the state space tree. They are as follows.

13.2.1 FIFO Search

In the First In First Out (FIFO) search, the children of the root node are generated in the first iteration. In the next step, the children of the first child are generated, if it is not already killed using a bounding function. The children, if not killed, are put in a queue and the children of the second child of the root are explored. Except for those which are killed, the rest are put in a queue.

The strategy is similar to the breadth first search traversal of graphs.

For example, in Fig. 13.1, the nodes 2, 3, and 4 are the children of the root node, 1. The node 2 has children 5, 6, and 7. Say, node 5 is killed by the bounding function. The rest are put in a queue. The children of node 3 are then generated. The process continues till the goal state is found.
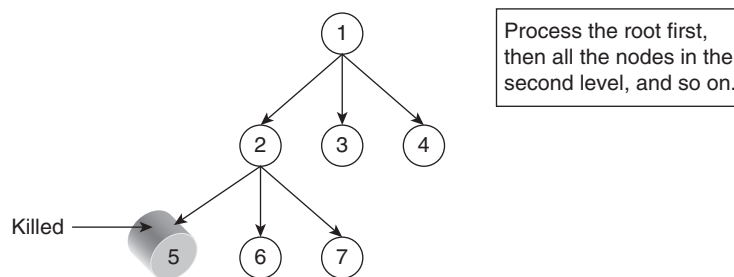
Figure 13.1 FIFO

13.2.2 LIFO Search

In the Last In First Out (LIFO) search, the children of the root node are generated in the first iteration. The first child is processed. In the next step, the children of the first child

are generated, if it is not already killed using a bounding function. The children, if not killed are put in a stack and the children of the second child of the root are explored. Except for those which are killed, the rest are put in a stack.

The strategy is similar to the depth first search traversal of graphs.

For example, in Fig. 13.2, the nodes 2, 3, and 4 are the children of the root node, 1. The node 2 has children 5, 6, and 7. Say, node 5 is killed by the bounding function. The rest are put in a stack.

The process continues till the goal state is found.

Process the root first, then the first child of the root, then the first child of the first node, and so on.
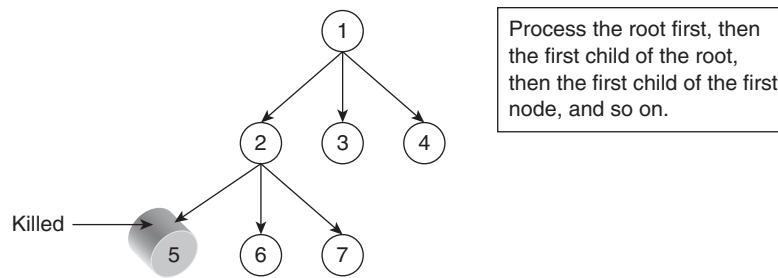
Figure 13.2 LIFO

13.2.3 Example of Branch and Bound: 0/1 Knapsack

The problem has already been stated in Section 10.3 of Chapter 10. A brief description of the problem has been stated as follows.

In the knapsack problem, a subset of items is to be selected from amongst the given set of items. The output of the algorithm should be a subset that completely (or almost completely) fills the bag and the profit earned by the selected elements should be maximized. The capacity of the bag is given as an input of the problem.

Input
- The set of items $x : \{x_1, x_2, x_3, ..., x_n\}$.
- The weights of the above items $W : \{w_1, w_2, w_3, ..., w_n\}$ and
- The profits earned by picking the items $P : \{p_1, p_2, p_3, ..., p_n\}$.

Output
- $x_n = 1$ denotes that the item has been picked and $x_n = 0$ means that the item has not been picked.

Constraints
- The total weight of the selected items is less than or equal to the weight of the bag, i.e.,

$$x_1 \times w_1 + x_1 \times w_2 + x_3 \times w_1 ... \le m \qquad (13.1)$$

where m is the weight of the bag

Goal The 'profit earned' is to be maximized, that is, $x_1 \times p_1 + x_1 \times p_2 + x_3 \times p_1...$ is to be maximized.

The concept of branch and bound can be best explained by 0/1 knapsack. The state space tree of 0/1 knapsack is easy to craft. The left sub-tree of the root node represents the inclusion of the first element of the given set and the right sub-tree of the root represents the exclusion of the first element. The root node has been numbered '1' in Fig. 13.3.
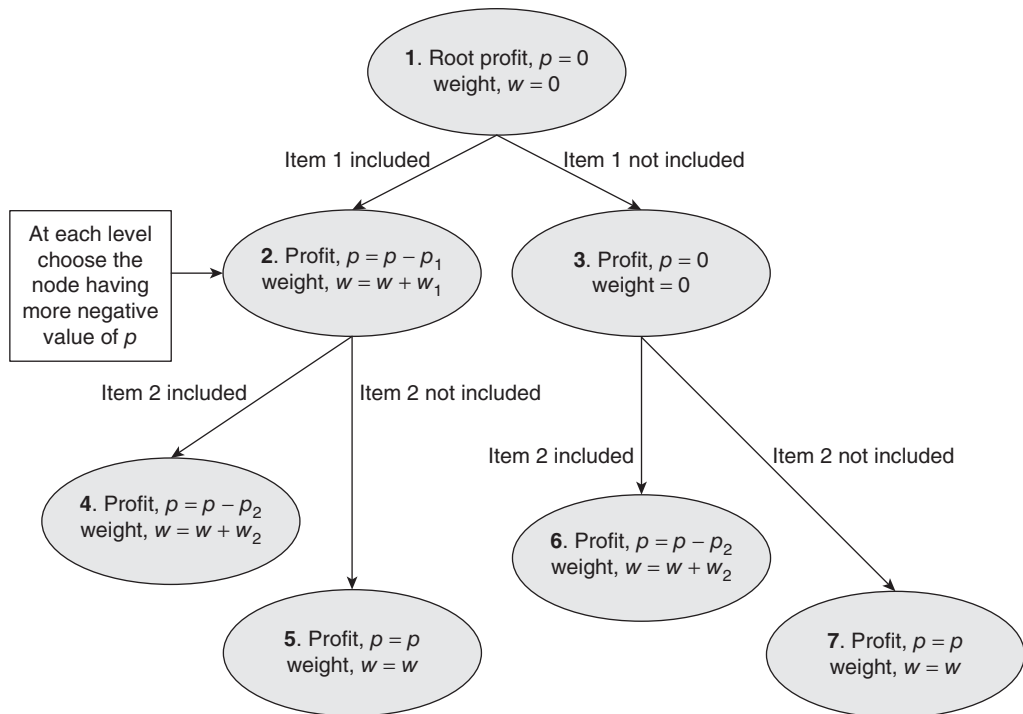
Figure 13.3 Applying branch and bound in knapsack problem

The left sub-tree of the node numbered '2' denotes the inclusion of the second element of the set and its right sub-tree denotes the inclusion of the second element of the given set. Same is the case with the node numbered 3. The difference, though, is that the left sub-tree of the node numbered '2' means leaving both the first and the second element have been included in the solution and the left sub-tree of '3' denotes the exclusion of the first element and inclusion of the second element of the given set in the solution set.

The branch and bound approach uses the breadth first search of the above tree and then tries to enhance it. In this approach, the children of nodes that are not deemed promising would not be generated. Breadth first search can be implemented using a queue. The nodes that are deemed promising would be placed in this queue.

The nodes are placed in the queue if the profit associated with them is better than the best so far. Otherwise, the node is not expanded.

The breadth first search is no better than depth first search; however, we can make it better using our bounding function. The above strategy is also referred to as best first search.

13.3 TRAVELLING SALESMAN PROBLEM

The travelling salesman problem is a combination of Hamiltonian cycle problem and minimum spanning tree problem. The Hamiltonian cycle is a path leading to the source node, without having to traverse any node more than once. The problem has already been discussed in Section 12.6 of Chapter 12.

Formally, if a graph $G = (V, E)$ is given, wherein the weight of an edge varies from from v_i to v_j, where v_i and $v_j \in V$ is given by w_{ij}. The aim is to find a path $v_1 v_{i1} \ v_{i2} \dots v_1$ where $v_{ij} \neq v_1$, such that the net cost of the path is minimum and all the v_{ij}'s are distinct.

The brute force approach of this algorithm would require the elicitation of all $(n-1)!$ paths followed by the calculation of the costs of those paths. Amongst these paths, the one having a minimum cost would be selected.

The above approach is computationally very expensive and is not feasible even for a moderately large value of n, n being the number of nodes.

A better approach was suggested in Chapter 10. The branch and bound algorithm would not drastically reduce the complexity, but would help us to reach the solution quickly. The algorithm is based on the concept of reduced matrix, which has been explained here.

The reduced matrix is obtained as follows:

- Subtract the minimum element of a row from each element of the row, thus making at least one element of the row equal to zero.
- Repeat the above for each column.

After the above process, there would be at least one zero in each row and each column.

13.3.1 Calculation of Cost

While carrying out the above steps, the value that is subtracted from each row and column is noted. The sum of the above values would give the cost of the reduced matrix.

13.3.2 Procedure

The given cost matrix is converted into a reduced matrix, as per the steps stated above. The cost of converting the matrix into reduced matrix becomes the value of the root node of the state space tree.

In the next step, the reduced cost matrix is taken as the input matrix and the following calculations are carried out.

As an example consider a graph having 4 vertices {1, 2, 3, 4}. If node 1 of the given graph is taken as the source node, then costs (1, 2), (1, 3), and so on are calculated. In order to calculate the cost of (1, i), the elements of the first row and the ith column are made ∞. Moreover, the element at the position (i, 1) is also made ∞. The resultant matrix is then converted into reduced matrix and the respective costs are noted. For example, if the cost of the reduced matrix is 20 and that of (1, 2), (1, 3), and (1, 4) are 20, 23, and 40, respectively, then the children of the root node will have labels 20, 23, and 40. Amongst these nodes, the one having minimum cost is selected. In the above example, the selected node has cost 20. It implies that from the source node, which is 1 in this case, one must move to 2.

Now, since 1 and 2 have been visited, the cost of (1, 2, 3) and (1, 2, 4) are calculated. The cost of (1, 2, 3) is calculated by making the elements of the third column of the matrix obtained for calculating (1, 2) equal to ∞ and then reducing the matrix so obtained. In the same way, the cost of (1, 2, 4) is calculated by making the elements of the third column of the matrix obtained for calculating (1, 2) equal to ∞ and then reducing the matrix so obtained. Assume that the two values obtained above are 40 and 32, the path leading to (1, 2, 3) would be selected.

Note that there were just four vertices in the graph. The last step would now be to calculate (1, 2, 3, 4) is selected.

The state space tree of the above has been shown in Fig. 13.4. The above concept can be understood by Illustration 13.1.
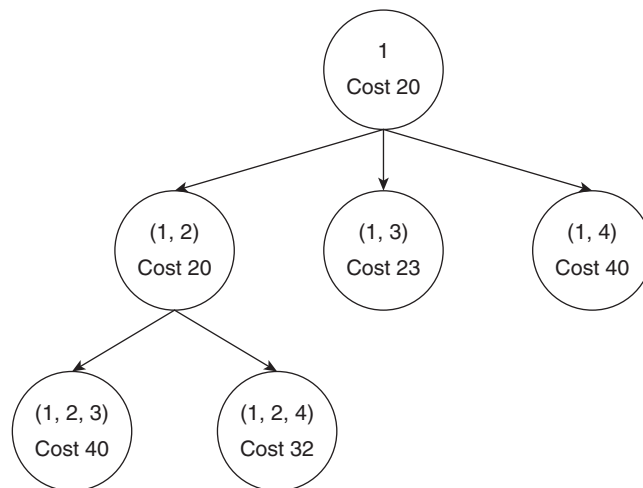
Figure 13.4   Selection of node by least cost

Illustration 13.1 The matrix of a graph is shown as follows. Find the Hamiltonian cycle, starting from the root node, having least cost.

Solution The cost matrix of a graph is as follows. The cost of going from node 1 to node 2 is 7, however, that of going from 2 to 1 is 2 (the element of the second row and first column is 2). All the elements of the main diagonal are infinity.

$$
\begin{pmatrix}
\infty & 7 & 2 & 10 & 5 \\
2 & \infty & 5 & 10 & 6 \\
4 & 7 & \infty & 5 & 12 \\
3 & 5 & 7 & \infty & 2 \\
12 & 7 & 3 & 4 & \infty
\end{pmatrix}
$$

The first step in solving the travelling salesman problem is to reduce the matrix such that each row and each column has a zero. This can be done by subtracting the least element of a row from each element of the row and then repeating the same process for the column. For example, in the given matrix, the minimum element of the first row is 2. So 2 is subtracted from each element of the first row. In the second row, the least element is 2, in the third it is 4, 3 in the fourth, and 3 in the fifth. These elements are subtracted from the respective rows. The process leads to the following matrix:

$$
\begin{pmatrix}
\infty & 5 & 0 & 8 & 3 \\
0 & \infty & 3 & 8 & 4 \\
0 & 3 & \infty & 1 & 8 \\
1 & 3 & 5 & \infty & 0 \\
9 & 4 & 0 & 1 & \infty
\end{pmatrix}
$$

The process is then repeated for the columns. The second and the fourth columns do not have a zero. So the minimum element of the second column, which is 5 and that of the fourth column, which is 1, is subtracted from the columns. The net value that is subtracted from the original matrix in order to convert it to the reduced matrix is 17. So the value associated with the root node of the state space tree is 17.

$$
\begin{pmatrix}
\infty & 2 & 0 & 7 & 3 \\
0 & \infty & 3 & 7 & 4 \\
0 & 0 & \infty & 0 & 8 \\
1 & 0 & 5 & \infty & 0 \\
9 & 1 & 0 & 0 & \infty
\end{pmatrix}
$$

The next step finds the node which must be visited after the first node in order to optimize the cost of the selected path. This is done as follows. The matrices (1, 2), (1, 3), (1, 4), and (1, 5) would be found and that having the least cost would become the node whose children would be generated. In order to find, say (1, 2) the first row, second

column, and the element (1, 2) are made infinity. The resultant matrix would then be reduced. The resultant matrix is as follows:

$$\begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 3 & 7 & 4 \\ 0 & \infty & \infty & 1 & 8 \\ 1 & \infty & 5 & \infty & 0 \\ 9 & \infty & 0 & 0 & \infty \end{pmatrix}$$

From the second row, 3 needs to be subtracted so that at least one element of the row becomes 0. The rest of the elements need not to be altered. The matrix (1, 2), therefore, is as follows. The cost associated with this matrix would be 17 (the cost of the reduced matrix) + 3 (the value subtracted in order to make the (1, 2) reduced). The value associated with the node depicting (1, 2) in the state space tree would be 20.

$$\begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & 4 & 1 \\ 0 & \infty & \infty & 1 & 8 \\ 1 & \infty & 5 & \infty & 0 \\ 9 & \infty & 0 & 0 & \infty \end{pmatrix}$$

In order to find, say (1, 3) the first row, third column, and the element (1, 3) are made infinity. The resultant matrix would then be reduced. The resultant matrix is as follows:

$$\begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & 7 & 4 \\ \infty & 0 & \infty & 1 & 8 \\ 1 & 0 & \infty & \infty & 0 \\ 9 & 1 & \infty & 0 & \infty \end{pmatrix}$$

In this case, the matrix is already in the reduced form, therefore, nothing needs to be done. The cost associated with the node depicting (1, 3) is 17.

In order to find, say (1, 4) the first row, fourth column and the element (1, 4) are made infinity. The resultant matrix is then reduced. The resultant matrix is as follows:

$$\begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & 3 & \infty & 4 \\ 0 & 0 & \infty & \infty & 8 \\ \infty & 0 & 5 & \infty & 0 \\ 9 & 1 & 0 & \infty & \infty \end{pmatrix}$$

In this case also the matrix is already in the reduced form and hence, the cost associated with the matrix is 17.

In order to find, say (1, 5) the first row, fifth column and the element (1, 5) are made infinity. The resultant matrix would then be reduced. The resultant matrix is as follows:

$$\begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & 3 & 7 & \infty \\ 0 & 0 & \infty & 1 & \infty \\ 1 & 0 & 5 & \infty & \infty \\ \infty & 1 & 0 & 0 & \infty \end{pmatrix}$$

In this case also the matrix is already in the reduced form and hence, the cost associated with the matrix is 17.

The state space tree formed up to this point is depicted in Fig. 13.5. The nodes depicting the path from (1, 3), (1, 4), and (1, 5) have the same cost. The first of these, that is, the node depicting the path (1, 3) is chosen.

The children of the node having least cost are generated at every level.
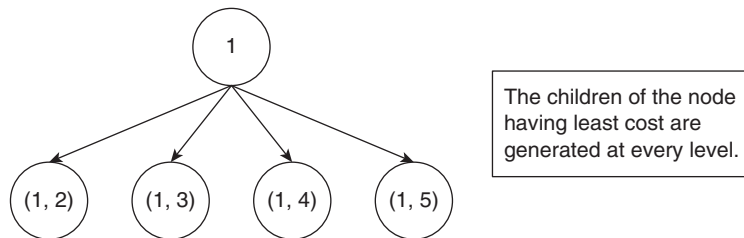
Figure 13.5 TSP using branch and bound

In the next step, the costs of (1, 3, 2), (1, 3, 4), and (1, 3, 5) are calculated. In order to calculate the cost of (1, 3, 2), the following steps are followed. The elements of the first row and the fourth column of the matrix (1, 3) are made infinity. In addition, the elements (2, 1) and (3, 1) are made infinity. The resultant matrix would then be reduced. The matrix (1, 2, 3) is as follows. The cost of this matrix is 22.

$$\begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & 3 & 0 \\ \infty & \infty & \infty & 0 & 7 \\ 1 & \infty & \infty & \infty & 0 \\ 9 & \infty & \infty & 0 & \infty \end{pmatrix}$$

This is followed by the crafting of matrices (1, 3, 4) and (1, 3, 5). From these three matrices whichever has the least cost is explored. The formation of the matrices (1, 3, 4) and (1, 3, 5) is left as an exercise for the reader. Moreover, the reader is also expected to find the matrices of the next two levels. There would be three such matrices. The final matrix not only gives the minimum cost of a node to another, but also the path from the root node to that matrix also tells us the nodes to be traversed in order to optimize the cost.

13.4 KNAPSACK PROBLEM

The problem can be stated as follows.

Let there are n items $\{1, 2, 3, \ldots, n\}$. The weight of the ith element is w_i. The weights are in the set $w_1, w_2, w_3, \ldots, w_n$. The profit earned on selecting the ith element in the solution set is p_i. The profits are the elements of the set $\{p_1, p_2, \ldots, p_n\}$. The selection or the rejection of an item is depicted by x_i. If the value of x_i is 1, then the ith element is deemed to be selected. On the other hand, if the value of the ith element is 0, then it is deemed to be rejected. The goal is to maximize the value of $\sum_{i=1}^{n} x_i p_i$, such that $\sum_{i=1}^{n} x_i w_i \leq m$, where m is the maximum capacity of the knapsack.

13.4.1 Knapsack Using Branch and Bound (Least Cost)

The problem can be solved using the branch and bound methodology. The procedure of doing so is as follows. The first node of the state space tree depicts the decision regarding the inclusion or the non-inclusion of the first item. The left sub-tree of the root indicates that the first item is selected and the left sub-tree indicates its non-inclusion. The inclusion of the first node means that the first node would be there in the solution set (except for the case wherein its weight exceeds the value of m). The left-hand side of the root contains the cost of selecting the first item. This cost is the minus of the profit earned if the items are selected in order of their occurrence, till the total weight selected is less than m. The right-hand side would contain the cost incurred in leaving the item 1.

The left sub-tree of the second level indicates the inclusion of the second item and the right, the non-inclusion of the second item. The children of only that node is generated which is better in terms of cost (more negative cost). The seemingly incompetent node is not expanded. When the complete state space tree is formed, the items selected are included in the solution set.

In order to understand the concept, let us explore Illustrations 13.2 and 13.3.

Illustration 13.2 Apply branch and bound to solve the following knapsack problem. The symbols have the usual meaning.

$n = 3$

$m = 6$

$w = \{2, 3, 4\}$

$p = \{1, 2, 5\}$

Solution If item 1 is included in the solution set, then item 2 can be selected but item 3 cannot be selected. The profit earned in selecting items 1 and 2 would be 3 units and hence the cost would be -3 units.

The exclusion of the first item would not make room for the inclusion of the third item. The profit earned would be 2 in this case and the cost would be −2. Since the least cost method is used, the node indicating the inclusion of the first item would be chosen.

The inclusion of the second item would result in −3 as the cost, whereas its exclusion would make room for the inclusion of the third item. The cost, in this case, would be −6.

In the next step, the node indicating the exclusion of item 2 is selected. This is followed by the decision regarding the inclusion or the non-inclusion of the third item. The non-inclusion of the third item leaves only the first term in the solution set and hence the cost becomes −1. The solution set would therefore be {1, 0, 1} meaning that the first item is selected, second is not selected, and the third is selected. The solution is depicted in Fig. 13.6.
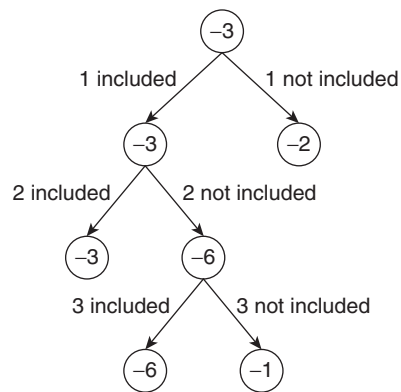
Figure 13.6 State space tree of Illustration 13.2

Illustration 13.3 Apply branch and bound to solve the following knapsack problem. The symbols have the usual meaning.

$n = 7$

$m = 15$

$w = \{2, 3, 5, 7, 1, 4, 1\}$

$p = \{10, 5, 15, 7, 6, 18, 3\}$

Solution If item 1 is included in the solution set, then items 2, 3, 5, and 6 can be selected but item 4 cannot be selected. The profit earned in selecting these items is 54 and hence the cost would be −54 units.

The exclusion of the first item would make room for the inclusion of the fourth item. The profit earned would be 27 in this case and the cost would be −27. Since the least cost method is used, the node indicating the inclusion of the first item would be chosen.

The inclusion of the second item would result in −54 as the cost, whereas its exclusion would make room for the inclusion of the fifth item. The cost, in this case, would be −38.

In the next step, the node indicating the exclusion of item 2 is selected. This is followed by the decision regarding the inclusion or the non-inclusion of the third item. On the non-inclusion

of the third item, the cost becomes −54. The process continues for the rest of the items also. The solution set would therefore be {1, 1, 1, 0, 1, 1, 0} meaning that the first item is selected, second is not selected, and the third is selected. The solution is depicted in Fig. 13.7.
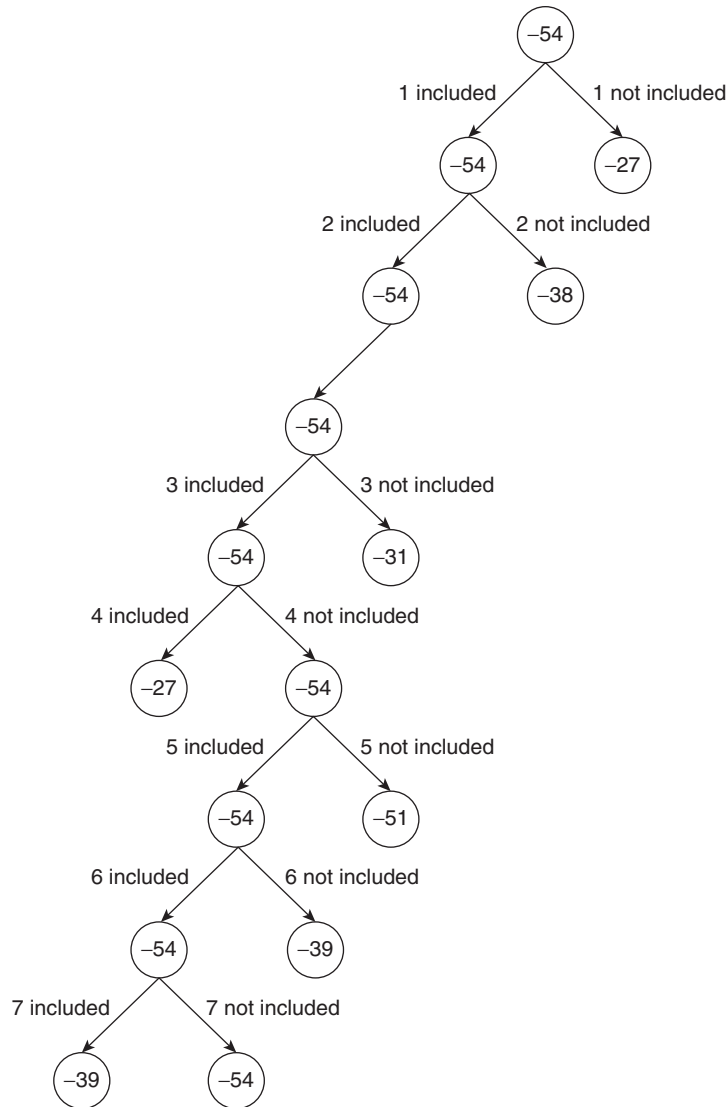
Figure 13.7 State space tree of Illustration 13.3

13.5 8-PUZZLE PROBLEMS

8-Puzzle problem is an instance of *N*-puzzle problem defined in Section 20.3.1. The solution of the problem is done using a process planning. However, a brief description of the problem and its solution by branch and bound has been discussed the following section.

13.5.1 First In First Out

The concept of First In First Out can be understood by considering the 8-puzzle problem. In the 8-puzzle problem, there are 9 cells and 8 numbers. One cell is empty so that the rest of the cells can be moved. The goal is to reach the solution configuration, which is the first cell having numbers 1, 2, and 3, in that order the second row having 4, 5, and 6 in that order and the third row having 7 and 8. Here, not all the initial configurations can lead to the solution. There are many ways of reaching the goal state. Of these, the First In First Out (FIFO), Last In First Out (LIFO), and Least Cost (LC) methods are discussed in this section.

In FIFO, the root node is expanded, that is, the children of the root node are generated. In this process, some of the children may be killed by the bounding function. The first child is then processed. The rest of the children, which are not killed, are put in queue. The process can be understood by Fig. 13.8. The root depicts the given configuration. The empty space, which is at the corner, can be moved either to the left or up. These two situations are depicted by the two children of the root. These two would be processed in that order. The first child of the left configuration (the left child of the root's child) gives the same configuration as that of the root, and is hence not processed. This may be considered as the first bounding condition, though there are many in this case. The second and the
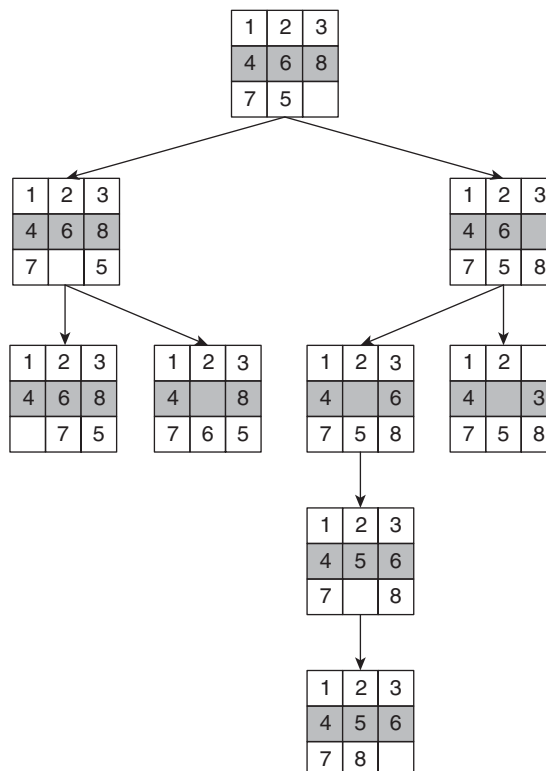
Figure 13.8 Applying FIFO to the 8-puzzle problem

third child represents the cases wherein the empty cell is moved to the left and up, respectively. After this, the two possible children of the right node of the root are generated.

The above process continues and the children at each possible level are generated. Though the figure does not show all the children but one of the paths leading to the goal state has been shown. The process is similar to the breadth first search traversal technique in graphs.

Though the method is easy to understand and implement, it does not find the solution efficiently. For instance, if the solution is at the tenth level, the above method will generate all the possible children of the first nine levels and then move to the tenth level.

13.5.2 Last In First Out

The concept of Last In First Out can be understood by considering the 8-puzzle problem. In LIFO, the root node is expanded, that is, the children of the root node are generated. In this process, some of the children may be killed by the bounding function. The first child is then processed. The rest of the children that are not killed are put in a stack. The process can be understood by Fig. 13.9.
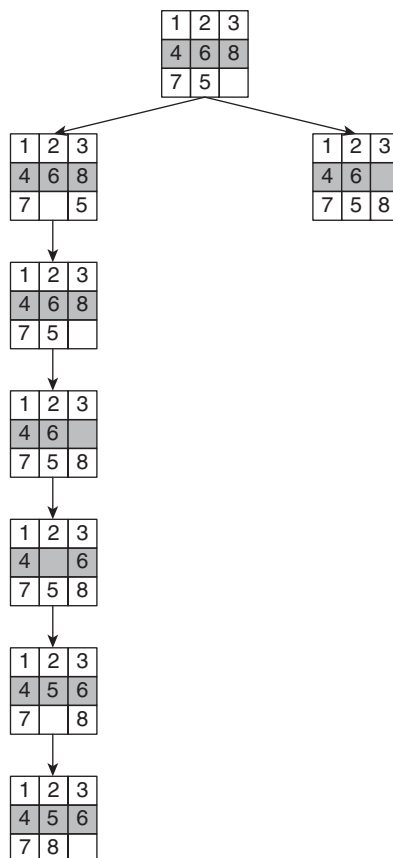
Figure 13.9 Applying LIFO in the 8-problem

The root depicts the given configuration. The empty space, which is at the corner, can be moved either to the left or up. These two situations are depicted by the two children of the root. These two would be processed in that order. The first child of the left configuration (the left child of the root's child) gives the same configuration as that of the root and is hence not processed. This may be considered as the first bounding condition, though there are many in this case. The second child represents the cases wherein the empty cell is moved to the right. After this, the child of this node is generated.

The above process continues and the first children at each possible level are generated. Though the figure does not show all the children but one of the paths leading to the goal state has been shown. The process is similar to the depth first search traversal technique in graphs.

Although the method is easy to understand and implement, it also does not find the solution efficiently. For instance, if the solution is at the first child of the last child of the root, the above method will generate all the possible children of the previous children and then would come to the requisite node.

13.5.3 Least Cost Search

The above methods are inefficient. Some of the researchers like Sahni [11] have termed these methods as blind. An efficient method is one which is able to tell us how good a generated child is. If the generated child is too poor then it need not be explored as in there is no need to generate its children. However, if the child is promising, its children would be generated. In the above example, if the net cost associated with each child is the cost in reaching that state plus the cost incurred in reaching the goal state from that state, the node having least cost would be explored and its children would be generated.

The method is similar to the A* method used in artificial intelligence. However, that is easier said than done. The most difficult part would be to design the cost function. The concepts of hamming distances, etc., can be used to find the effective cost of a node.

The least cost method would be used in text that follows. In the case of optimization problems where profit is given, the cost is the negative of the profit. This idea has been used to solve the knapsack problem.

13.6 EFFICIENCY CONSIDERATIONS

The following discussion focuses on the efficiency of the branch and bound method. The efficiency of a method, as stated earlier, is determined by the time and the space. The space would depend on the number of nodes of the state space tree.

As stated earlier, branch and bound can be implemented using the Least Cost (LC) method, Last In First Out (LIFO), and the First In First Out (FIFO) methods. The least cost method, however, has been an issue of contention explored in the following discussion. One of the goals of selecting the bound should be to reduce the number of nodes in

the state space tree. The reduction in the number of nodes can be done with the bounding function. The bound that is selected is generally initialized with a better value but it is hard to determine whether this 'better' value will really decrease the number of nodes of the state space tree. This, like most of the optimization problems, is a precarious issue. It cannot be said, as of now, if the value of bound is greater than our value and it would lead to reduction in the number of nodes.

The use of dominance relation has also come under the cloud of suspicion of not being better in terms of the number of nodes generated. As we will see in the following discussion, the answers to the above questions are not very much contrary to the expectations.

As stated earlier, there are three methods of generating the nodes in the state space tree. They are least cost, FIFO, and LIFO. These methods, when used, generate the minimum number of nodes when the value of the cost is the least upper bound.

Though there can be more than one upper bound, they do not outperform each other in terms of the number of nodes generated. The analysis of the trees point to the fact that the use of a better cost function does not increase the number of nodes if LIFO or FIFO is used; however, this might be the case if least cost method is used.

> **Tip:** If there are more than one upper bound then neither of them would outperform the other when space considerations are taken into account.

The nature of the dominance relation and the notion of its being strong also determine the number of nodes of the state space tree. From the above discussion, it becomes clear that the strategies for determining the dominance relation and the cost function are the important factors when efficiency is of concern.

The generalization of LIFO, FIFO, and LC gives rise to what is called a heuristic search. The search is based on the evaluation of a function called *heuristic function*. This method finds extensive applications in artificial intelligence. The topics like knapsack can be solved via heuristic search algorithms.

13.7 OPTIMIZATION AND RELAXATION

This section throws some light on the concepts of relaxation and optimization. The concepts are important both for understanding this chapter and the concepts in NP class (Chapter 19). Here, in most of the cases, an optimization problem that has a large domain is a contender of NP-hard problem. Moreover, optimization is also studied in Mathematics. Lately, methods based on artificial intelligence are being used to solve such problems. Some of these methods have been discussed in Chapter 23.

13.7.1 Optimization

Generally, the given circumstances can be described in terms of some constraints. Under the given circumstances, obtaining the best results is referred to as optimization. The

techniques invented by Newton (calculus), Bernoulli, Euler, etc., laid the foundation of calculus of variation. Both calculus and calculus of variation are important techniques of optimization. The optimization problems are used in engineering and non-engineering applications alike. Optimization finds its application in operations research as well. The goal of optimization is to maximize or minimize a function called *objective function*. As a matter of fact, maximizing $f(x)$ is same as minimizing $-f(x)$. So, the problems related to maximization can be converted to minimization and vice versa. We have explored many optimization problems in the previous chapters. For example, the travelling salesman problem, the knapsack problem, and the job scheduling are all examples of optimizing problems.

The optimization can be local or global. Local optimization methods are faster but do not guarantee a solution that is the best. The global optimization techniques are, though slow, but guarantee correct result.

There are many techniques for optimization. Different techniques are used as per the problem. The techniques employed to solve optimization problems can be categorized as follows.

In the first kind, the statistical methods are used to solve the optimization problems. Techniques such as regression analysis, cluster analysis, and factor analysis come under this class. The second kind of techniques uses stochastic process. The examples of techniques that use this method are queuing theory, simulation methods, reliability theory, etc. In the third type of techniques, mathematical programming is used. Techniques such as calculus methods, linear programming, integer programming, and dynamic programming come under this class (Fig. 13.10).
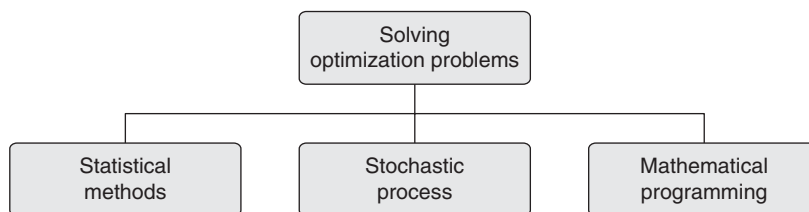
Figure 13.10 Classification of optimization problems

Statement

Maximize $f(x)$,

Subject to $g(x) \leq 0$

and $p(x) = 0$

$f(x)$ is referred to as objective function. $g(x)$ is the inequality and $p(x)$ is the equality constraints.

The answer is stored in a vector called *design vector*.

When the constraints are given, then the problem is called *constraint optimization problem*. When the constraints are not given, then the problem becomes an *unconstrained problem*. The existence of constraints and the nature of the design variables help classify the optimization problems. In the second classification, the parameters are not continuous functions that minimize or maximize the objective functions, whereas in the second case they are. The former are called *static optimization* problems and the latter are called *dynamic optimization* problems (Fig. 13.11).
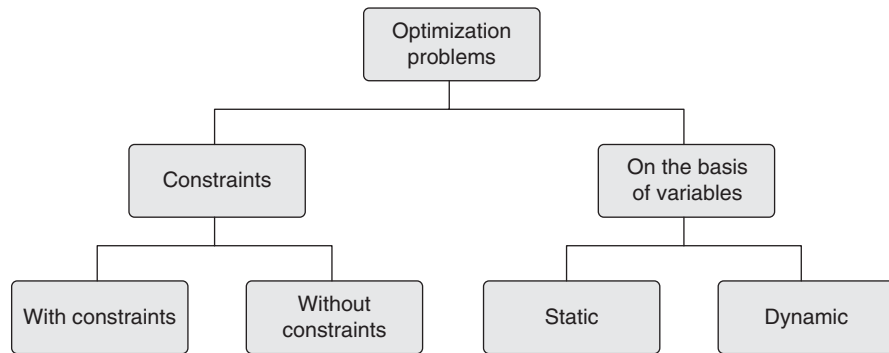
Figure 13.11 Classification of optimization problems

One such optimization where the functions (both objective and constraint) are convex is called *convex optimization*. Branch and bound is one of the methods of solving convex optimization problems.

13.7.2 Relaxation

During the literature review, it was found that many authors associate branch and bound with the concept of relaxation. The following discussion explores the idea of relaxation and its applicability to branch and bound.

It is clear from the above discussion that branch and bound is used for optimization problems. Any optimization problem has three components: the objective function that needs to be maximized or minimized, the constraints, and the variables, which are binary in nature and would be used in the solution. The types of relaxation have been shown in Fig. 13.12.
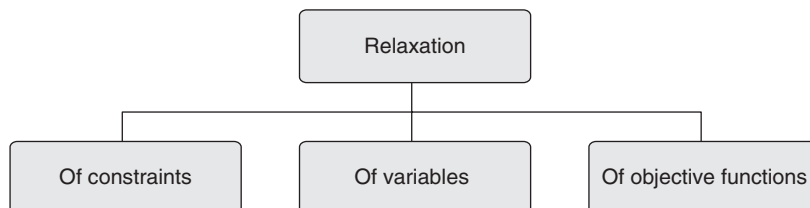
Figure 13.12 Types of relaxation

The relaxation would generally reduce the given problem in the form which has looser constraints. For example, obtaining answers in terms of variables that are non-binary, perhaps from a set that is larger than the set of binary numbers.

However, one may decide to relax the inequalities instead of constraints. In this case, the inequalities can be replaced by what is called *surrogate inequalities*. The idea is to replace

$$\sum_{i=1}^{n} a_i x_i$$

with

$$\sum_{i=1}^{n} a_i y_i$$

such that

$$\sum_{i=1}^{n} a_i x_i \leq \sum_{i=1}^{n} a_i y_i$$

One may also decide to relax the objective function. If the function, say, $f(x)$ is to be maximized, then instead of $f(x)$ the function $h(x)$ can be maximized which is always greater than $f(x)$.

The relation can also be a mixture of the above techniques. For instance, in the case of Lagrange's relation, both objective functions and constraints are modified.

13.8 CONCLUSION

The chapter discusses the branch and bound strategy. The idea is an extension of back-tracking discussed in the previous chapter. However, branch and bound reduces the nodes of the state space tree and is therefore more efficient than backtracking. The traversal of the tree can be done by either LIFO or FIFO or the least cost method. The examples of these methods have been discussed in the chapter. The chapter also provides food for thought for the formulation of the bounding functions. The reader is expected to implement the procedures given above and compare the running time with the dynamic approach.

Points to Remember

- The branch and bound approach is also used for unconstrained non-convex optimization problems.
- Branch and bound can be implemented using the Least Cost (LC) method, Last In First Out (LIFO), and the First In First Out (FIFO) methods.
- There are three methods of generating the nodes in the state space tree. They are least cost, First In First Out, and Last In First Out.

- When the constraints are given then the problem is called constraint optimization problem. When the constraints are not given then the problem becomes an unconstrained problem.
- Optimization can be local or global. Local optimization methods are faster but do not guarantee the best solution. The global optimization techniques, though slow, guarantee the correct result.

KEY TERMS

FIFO Search In the FIFO search, the children of the root node are generated in the first iteration. In the next step, the children of the first child are generated, if they are not already killed using a bounding function. The children, if not killed, are put in a queue and the children of the second child of the root are explored. Except for those which are killed, the rest are put in a queue.

LIFO Search In the LIFO search, the children of the root node are generated in the first iteration. The first child is processed. In the next step, the children of the first child are generated, if it is not already killed, using a bounding function. The children, if not killed, are put in a stack and the children of the second child of the root are explored. Except for those which are killed, the rest are put in a stack.

Optimization It refers to the technique which minimizes or maximizes an objective function, subject to given constraints.

Relaxation The relaxation would generally reduce the given problem in the form which has looser constraints.

EXERCISES

I. Multiple Choice Questions

1. Which of the following is used in FIFO search?
   (a) Queue   (b) Stack   (c) Both   (d) None
2. Which of the following is used in LIFO search?
   (a) Queue   (b) Stack   (c) Both   (d) None
3. Which of the following can be solved by the least cost method of branch and bound technique?
   (a) Travelling salesman problem   (c) Both
   (b) Knapsack problem   (d) None
4. Which of the following is considered as blind search?
   (a) LIFO   (b) FIFO   (c) Both   (d) None
5. Which of the following is a type of optimization?
   (a) Constrained   (c) Both
   (b) Unconstrained   (d) None

6. Which of the following is a type of relaxation?
   (a) Relation of constraints   (c) Both
   (b) Relaxation of variables   (d) None
7. Which of the following is a special case of backtracking?
   (a) Branch and bound
   (b) Divide and conquer
   (c) Dynamic programming
   (d) None of the above
8. Which of the following is not used in 8-puzzle problem?
   (a) LIFO   (c) Least cost search
   (b) FIFO   (d) Divide and conquer
9. Which of the following is true according to branch and bound techinique?
   (a) It requires a bounding function.
   (b) It reduces the space requirement of the state space tree.
   (c) It is a special case of backtracking.
   (d) All of the above.
10. Which of the following can be considered as branch and bound?
   (a) A*   (c) Linear search
   (b) B*   (d) None of the above

II. Review Questions

1. Discuss the concept of branch and bound. Compare it with backtracking.
2. Explain how to solve travelling salesman problem using branch and bound.
3. Explain how to solve the Knapsack using branch and bound.
4. Explain how to solve 8-puzzle problem using branch and bound.
5. Discuss the efficiency considerations in branch and bound.
6. Explain the concept of optimization.
7. What is relaxation? What is its importance with respect to branch and bound?

III. Numerical Problems

1. The cost matrix of the input graphs has been given as follows. Solve the travelling salesman problem in the following cases using branch and bound:

(a) $\begin{pmatrix} \infty & 7 & 1 \\ 3 & \infty & 7 \\ 9 & 8 & \infty \end{pmatrix}$
(b) $\begin{pmatrix} \infty & 7 & 1 \\ 7 & \infty & 7 \\ 1 & 7 & \infty \end{pmatrix}$

(c) $\begin{pmatrix} \infty & 10 & 3 & 7 \\ 2 & \infty & 9 & 12 \\ 9 & 2 & \infty & 11 \\ 5 & 4 & 8 & \infty \end{pmatrix}$
(d) $\begin{pmatrix} \infty & 10 & 3 & 7 \\ 10 & \infty & 9 & 12 \\ 3 & 9 & \infty & 11 \\ 7 & 12 & 11 & \infty \end{pmatrix}$

$$
\text{(e)} \quad \begin{pmatrix} \infty & 7 & 2 & 10 & 5 \\ 2 & \infty & 5 & 10 & 6 \\ 4 & 7 & \infty & 5 & 12 \\ 3 & 15 & 7 & \infty & 2 \\ 12 & 7 & 31 & 42 & \infty \end{pmatrix}
\qquad
\text{(f)} \quad \begin{pmatrix} \infty & 7 & 2 & 10 & 5 \\ 7 & \infty & 5 & 10 & 6 \\ 2 & 5 & \infty & 5 & 12 \\ 10 & 10 & 5 & \infty & 2 \\ 5 & 6 & 12 & 2 & \infty \end{pmatrix}
$$

2. Can branch and bound be applied to solve Illustration 9.1 of Chapter 9.
3. The initial configuration of 8-puzzle problem is as follows. Apply the following methods to find the solution (if possible).

(a) LIFO   (b) FIFO   (c) Least cost

1	6	3
4	2	8
7	5	

1	6	7
2	4	8
3	5	

1	6	7
2	4	8
5		3

4. Solve the following knapsack problem:

$n = 5$ $\qquad\qquad\qquad\qquad\qquad$ $w = [10, 4, 3, 9, 8]$

$P = [20, 10, 5, 4, 3]$ $\qquad\qquad\qquad$ $m = 22$

where n is the number of terms, P denotes the profit gained on selecting each item, w is the weight of each item, and m is the capacity of knapsack.

Answers to MCQs

1. (a)	3. (c)	5. (c)	7. (a)	9. (d)
2. (b)	4. (c)	6. (c)	8. (d)	10. (a)

CHAPTER 14

Randomized Algorithms

OBJECTIVES

After studying this chapter, the reader will be able to

• Understand the concept of randomized algorithms
• Differentiate between Monte Carlo and Las Vegas approach
• Explain the complexity classes of randomized algorithms
• Use randomized algorithms to solve problems

14.1 INTRODUCTION

Randomization is a way of life. We make use of randomization in our daily life. The concept is extensively used in gaming theory, generating the initial population of genetic algorithm, assigning weights in neural networks, and so on. The use of randomization in genetic algorithms and neural networks has been dealt with in Chapter 23. On the face of it, the use of randomization in algorithms may seem preposterous, illogical, or even irrational. One might not find it logical to use random algorithms after studying the design techniques discussed in this section. One may think that even if the answer to the problem is found using the concept of randomization, then it would be difficult to explain how the results have been obtained.

At times most of the things that cannot be explained have their reasons hidden in simple mathematics. The probability theory along with statistical techniques has the capacity to answer most of the things that are deemed inexplicable by us, such as raining, weather forecasting, migration of birds, and even the evolution of the earth. These things might appear incomprehensible but the reason might lie in simple probability theory.

This chapter explores the concept of randomized algorithms. Randomized algorithms generally improve the worst case time complexity. As an example, the sixth section of this chapter introduces randomized quick sort, in which the worst case complexity has been improved by incorporating randomization. The idea is that the reader should be able to tackle difficult problems using the concept. The chapter intends to cover the basics of randomization explanation of the various approaches used therein.

The problems have been given but an involved mathematical analysis is not the purpose of this chapter.

The chapter examines the two approaches used in the randomized algorithms and discusses the methods. It also discusses the complexity classes and examines some of the basic problems that can be tackled using this approach.

14.2 RANDOMIZATION

According to the Oxford dictionary, a random process is one which does not follow any specific pattern. The randomization process is an experiment that gives each item an equal chance to be considered.

Before proceeding any further, let us try to understand what a random sequence is. A good random sequence has generally two attributes:
- One cannot guess the next number as in the sequence generated by a random number generator does not form a pattern.
- As far as possible, the probability of turning up of a particular number is the same.

For example, if a random number generator produces numbers from 1 to 10. Five random numbers are generated from the generator and the sequence produced is 1, 1, 2, 1, 3; then the given generator cannot be considered good as the sequence contains a '1' many more times as compared to other numbers. If the sequence is 5, 4, 3, 2, 1; even then it is not a good random sequence as the output forms a pattern. However, if the sequence is 2, 4, 1, 5, 3, then it can be considered a better sequence than the above two. Though the above discussion in no way intending to state that non-repetition is the only criteria for deciding the goodness of a random sequence, there are many tests for finding out the goodness of a random sequence. Tests such as chi-square, coefficient of auto-correlation, etc., help to judge the quality of a random sequence.

In order to generate a random number, we generally use a random number generator. The pseudorandom number generator generates a random number when the requisite function is invoked. Table 14.1 gives the functions/methods that need to be invoked in order to generate a random number.

If the random number generated is either 0 or 1, then the output would henceforth be referred to as randomized bits.

Table 14.1 Generating a random number in various languages

Language	Procedure to generate a random number
C	Use rand() or random() after randomize()
C++	Use rand() or random() after invoking randomize()
C#	Instantiate the random class and call the next function
JAVA	Instantiate the random class and call the next method

14.3 MONTE CARLO vs LAS VEGAS ALGORITHMS

Randomized algorithms are those that use randomized bits as input. The algorithm is random in the sense that either it produces a correct output in random amount of time or it produces an output (which may or may not be correct) in a fixed amount of time. So, either the time to produce a correct output is infinite (in the maximum case) or the output obtained may not be the best.

The first type of algorithms always produces a correct answer. These are called Las Vegas algorithms. The second type of algorithms terminates in a limited period of time but the output produced may or may not be correct. Such algorithms are called Monte Carlo algorithms. Figure 14.1 and Table 14.2 depict the classification.
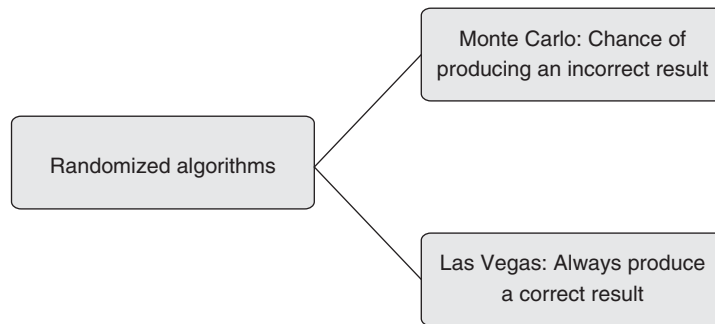
Figure 14.1 Classification of randomized algorithms on the basis of running time

Table 14.2 Las Vegas vs Monte Carlo

Las Vegas	Monte Carlo
Eventually succeeds with a probability 1	Not the case
Running time may be infinity	Running time is constant

14.3.1 Selection of Appropriate Technique

Normally, an algorithm designer would want his algorithm to succeed. In that case, he would use a Las Vegas algorithm. However, in some cases, where time is a major constraint, Monte Carlo algorithms are used. For example, in the case of search, a programmer would want his algorithm to succeed with a probability 1. This can, however, be done only if the number of elements in the list is not colossal. For such situations, the Las Vegas algorithms are suitable. In the other cases, wherein the number of elements is too large and there is a fitness associated with each item, we would want the selection algorithm to produce the result with a good fitness but in a finite time. In such cases, the Monte Carlo algorithms may be used. Such cases may arise in the designing of a crawler, which is a part of a search engine. The search procedures, both using Las Vegas and Monte Carlo, have been discussed as follows.

Algorithm 14.1   Randomized_Search (a[], x)

```
//The algorithm finds the position at which the element 'x' is present in the
array a
{
while(true)
        {
        i=rand()%n;
        if (a[i]==x)
                {
                Print "Found at "+x;
                exit();
                }
        }
}
```

Case 1 Example of Las Vegas Algorithm
The algorithm exits only if the required element is found. While designing the algorithm, it was assumed that the element x is present in the array. Interestingly, the best case complexity of the algorithm is $O(1)$ and the worst case complexity is $O(n)$, if there are n elements in the given array. In the case of linear search also the best-case complexity is $O(1)$ and that of worst case is $O(n)$. So, computationally, the Randomized_Search algorithm is equivalent to the linear search algorithm. And more so, there is a probability that it might perform better than the linear search.

Case 2 When the algorithm terminates in a constant time.
In this case, a minor change in the algorithm would help us to achieve the task. The infinite `while` loop in the above algorithm, if replaced with a loop which terminates after k iterations, would help us to achieve the task.

Algorithm 14.2   Randomized_Search_Modified (a[], x)

```
//The algorithm finds the position at which the element 'x' is present in the
array a
{
j=o;
while(j<k)
        {
        i=rand()%n;
        if (a[i]==x)
                {
                 Print "Found at "+x;
```

```
                        exit();
                        }
              J=j+1;
                }
   }
```

It may be noted that the algorithm exists if the required element is found or the number of iterations exceed k.

Some of the applications of Monte Carlo algorithms are as follows:

- The algorithms are used in physics and mathematics. The areas where these algorithms used are computational physics, physical chemistry, etc.
- The algorithms have successfully been used in weather forecasting.
- Studying the evolution of galaxies is also one of the fascinating areas where these algorithms are used.
- The correlated variations in digital ICs are studied using Monte Carlo algorithms.
- The algorithms have also been used in robotics and even in localization algorithms.

14.4 USES OF RANDOMIZED ALGORITHMS

The concept of randomization gives an idea of creation of structures with a non-zero value of desired property. This is used in many applications of randomization, some of which have been discussed as follows.

One of the most important applications of randomization is sampling. A sample selected to carry out a particular task should be such that the ethos of the sample is same as that of the population. The goodness of the sample would decide the accuracy of the experiment.

Samples are generally selected to carry out polls. A closer look at the problem would reveal something interesting. While sampling, we would have no idea whether the sample, being selected, is the best or not. We would get the answer only if the result is known. In an opinion or an exit poll, a pollster is never sure of the accuracy of the result, till the final outcome is known. That is, a good sample can only be formed by combining deterministic methods with the randomized ones.

An important consideration while selecting the sample can be the statistical analysis of the sample. For example, an opinion poll is conducted to gauge the mood of the nation on a particular issue. As stated earlier, the goodness of the sample would decide the accuracy of the result. If the population of the country consists of 80% Hindus, 12% Muslims, and 8% people of other religions; the sample must also be of the same composition (as far as possible). Further, in these 80% Hindus, there are 25% people of caste A, 25% of caste B, 12.5% of caste C, and the rest of other castes; then the sample must also have the same sub-composition in the people who are Hindus. Furthermore, the sample must be chosen to reflect the gender segregation, the economic segregation, and the educational segregation of the population. This is determinism.

This determinism is sure to make our results better. However, the subclasses of the sample must be selected randomly. Here comes the concept of randomization. For example, there are 6.2% educated women of caste C, in the population of a 1000 million. Because of constraints (economic and time), only 324 such women may be selected in the sample. This can be done by using randomization. The selection must be completely random and free from any pattern. Sampling technique can also be used to solve problems like quick sort.

The randomized algorithms have many other applications. One of the most important applications is to be able to hide the details. In a deterministic algorithm, the steps to be executed are known. Most of the algorithms studied so far are deterministic. For example, in the linear search discussed in Section 5.3.1 of Chapter 5, the steps are known and hence, it can be considered as deterministic. All the algorithms discussed in Chapter 8 are deterministic as their steps can be traced. These steps in turn can be used to determine the output of the algorithm, which in turn can be an input to some other algorithm. For example, the input to the merge algorithm discussed in Chapter 9 is sorted lists, which in turn can be generated by using sorting algorithms.

On the other hand, if one intends to hide the output, then instead of deterministic algorithms, randomized algorithms can be used. The randomization makes the backtracking virtually impossible.

Hashing is one of the most important applications of randomization. The following discussion briefly introduces the concept:

Hashing

A hash function is a function that computes the index of an array, from what is referred to as a 'key'. Ideally, a hash function should map to different locations, each time it is executed. If the hash function returns the same index next time, then collision occurs. In this case, solutions such as 'bucket' come to our rescue.

Theoretically, we cater to situations where either of the following are true:
- There is no space constraint; however, the time is limited.
- There is no time constraint; however, the space is limited.

Hashing takes care of both of the above. In real life situations, one neither has infinite time nor infinite space.

14.5 COMPLEXITY CLASSES OF RANDOMIZED ALGORITHMS

The complexity classes discussed in Chapter 19 of this book have a randomized counterpart. The following section discusses the complexity classes of randomized algorithms. The reader can leave this section in the first reading or else can go through Chapter 19.

Randomized P or RP Problems

The set contains languages L such that if an input belongs to the language, then the probability of the Turing machine accepting that input is greater than or equal to half.

If, on the other hand, the input does not belong to the language, then the probability of the corresponding Turing machine accepting that input would be 0. The formal definition of the RP problems is as follows:

$$p(x) = \begin{cases} \geq \dfrac{1}{2}, & x \in L \\ 0, & x \notin L \end{cases}$$

Co-randomized P or Co-RP Problems

The set contains languages L such that if an input does not belong to the language, then the probability of the Turing machine accepting that input is greater than or equal to half. If, on the other hand, the input belongs to the language, then the probability of the corresponding Turing machine accepting that input would be 0,

$$p(x) = \begin{cases} \geq \dfrac{1}{2}, & x \notin L \\ 0, & x \in L \end{cases}$$

Bounded-error Probabilistic Polynomial Algorithms (BPP)

These are a class of algorithms in which if the input does not belong to the language, then the probability of corresponding machine accepting that input is less than or equal to p, whereas in the other case, the probability is greater than $(1 - p)$,

$$p(x) = \begin{cases} \leq p, & x \notin L \\ \geq (1 - p), & x \in L \end{cases}$$

Probabilistic P Class (PP)

In this class of algorithms, if the input is accepted by the machine, then the probability is less than p and in the other case, the probability is greater than or equal to p,

$$p(x) = \begin{cases} \leq p, & x \notin L \\ \geq p, & x \in L \end{cases}$$

Class Zero-error Probabilistic P (ZPP)

The class of languages, which are both RP and Co-RP are referred to as ZPP. The algorithms are, in fact, the Las Vegas algorithms, in which the probability of getting an answer is 1 (Fig. 14.2).

Figure 14.3 depicts the above classification.

Co-BPP may also be defined on the same lines and owing to its symmetry Co-BPP = BPP. The proof of this is beyond the scope of this book. However, the reader can refer to the paper by Impagliazzo et al.
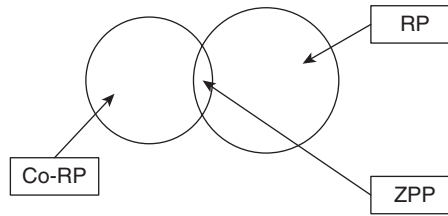
Figure 14.2 ZPP class
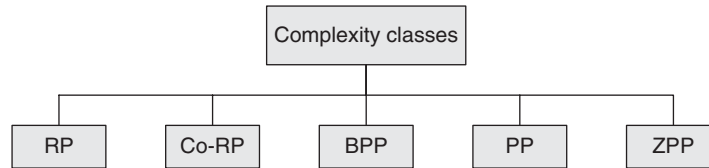
Figure 14.3 Complexity classes of randomized algorithms

14.6 APPLICATIONS OF RANDOMIZED ALGORITHMS

In this section, we will discuss some of the applications of randomized algorithms.

14.6.1 Book Problem

A group of students studying in an engineering college, in the National Capital Region (NCR) of India, live in a PG. All of them are from the same class. A day before the exam of an elective paper, .NET, they decide to study from the book of their professor. They find that there is just one copy of the book in the PG. Moreover, the city is facing acute load shedding, therefore their mobiles and laptops are not charged. Hence, they cannot click the photos of the requisite pages. The only option left now is to share the book. In addition to the above facts, they know that they cannot study with each other. So each one must have an exclusive access of the book, for the time the person is studying. One of the boys, named Hari, is worried about his chance of getting the book. So, he decides to develop a randomized algorithm to solve the above problem. Suppose there are n people in the PG. Each person, therefore, can access the book with a probability p. Since Hari's goal would be to analyse the situations that greatly increase his chances of getting the book in the least number of turns. He performs the following calculations:

$$p = \frac{1}{n}$$

Now, the probability of Hari getting the book in the first turn $= p$.

The probability of Hari getting book in the second turn would be $p \times (1 - p)$. (The probability of that person, not getting the book in the first turn would be $(1 - p)$. Getting a book in the first turn and getting it in the second turn are independent events (see Appendix A6). Therefore, the two probabilities would be multiplied.)

The probability of Hari getting book in the mth turn would be $(1 - p)^{m-1} p$.

Substituting the value of p in the above equation, we get $\left(1 - \dfrac{1}{n}\right)^{m-1} \times \left(\dfrac{1}{n}\right)$.

The minimum number of turns to make the probability of Hari's getting the book $\geq p$,

$$\left(1 - \frac{1}{n}\right)^{m-1} \times \left(\frac{1}{n}\right) \geq p$$

$$(m-1)\log\left(1 - \frac{1}{n}\right) \geq \log np$$

$$m \geq \frac{\log np}{\log\left(1 - \dfrac{1}{n}\right)} + 1$$

For example, if the number of people in the PG is 10, then the number of turns which would make the value of m maximum is infinity. Even if, n is not infinity, the requisite value of m from the given value can be easily calculated from the above formula.

14.6.2 Load Balancing

There are n processors and m jobs. The jobs need to be catered immediately, so when a job arrives, it is allotted to one of the processors. In order to maintain an even balancing, the jobs must be allocated to a processor judiciously. The problem becomes easy if the jobs are all equal and the distribution of jobs is done in such a way that each job is assigned to $(i + 1)\%n$ processor, i starting from 1. The approach is similar to the Round-Robin scheduling in an operating system. However, the approach does not always work. The approach works only if there is a central arbitrator. For this, there is a central controller which is assigned the responsibility of allocating things and taking them back.

There is another approach to tackle the problem. In this approach, the processes are allocated in a randomized fashion to a processor. Surely, the evenness of Round Robin cannot be matched using this approach. What can be, though, ensured is the maximum probability of the distribution being almost even. It is like the army of one country firing missiles on another, randomly, in a hope that at least some terrorists would be killed. So let us discuss the solution of the above problem. Ideally, when the number of processors is same as the number of jobs, each processor should get one job. Although this can rarely be achieved using a randomized approach.

Analysis

Let the α_{ij} be a binary variable that denotes whether a job j has been assigned to the processor i. α_{ij} would be 1 if the job j is assigned to the processor i, otherwise it would be 0. Let β_i be the number of jobs assigned to the processor i.

The value of β_i would therefore be $\sum_{i=1}^{n} \alpha_{ij}$. The expectation of α_{ij} would be $1/n$ and that of β_i would be 1. The important part though is to assess the deviation from the ideal behaviour.

The mathematical analysis of the scheduling is involved; however, it can be proved that the probability of no processor receiving $\theta\left(\dfrac{\log n}{\log(\log n)}\right)$ jobs is minimum $(1-(1/n))$.

14.6.3 Quick Sort

Quick sort is one of the most efficient algorithms for sorting. The following discussion assumes that all the elements in the array being discussed are different. The algorithm has already been discussed in Section 9.4 of Chapter 9. The reader is advised to go through that section, if he has not already done so.

Quick sort requires a pivot element to be selected. The pivot element, in case of randomized algorithms, is picked at random. If the array has just one element, then it is deemed to be sorted, otherwise the array is divided into two parts. The first part would contain all the elements lesser than the pivot and the second will contain all the elements greater than the pivot.

The two arrays, thus formed, are sorted recursively thus forming the resultant array. This type of algorithm, though randomized, ensures the probability of correct answer being produced is 1.

Conventionally, the segregation of the array into two requires $O(n)$ complexity. Here, one can choose the pivot from amongst the n elements present in the array. The probability of the correct pivot being selected is $(1/n)$ if this selection leads to the formation of two arrays, one containing k elements and the other containing $(n - (k - 1))$ elements.

The recursive equation of the above discussion would be as follows:

$$T(n) = \frac{1}{n}\sum_{k=0}^{n} T(k) + T(n-k-1)$$

The solution of the above equation has been discussed in Chapter 3. The complexity comes out to be $O(n\log n)$. It can also be inferred from the above relation that $T(n) \leq 2n\log n$, if the value of n is greater than or equal to 1.

The randomized quick sort is one of the most important applications of randomized algorithms. The procedure reduces the worst case time for quick sort. It may be stated that the average case running time for the algorithm is $O(n \log n)$ and the worst case running time is $O(n^2)$. That is, there is a large gap between the two running times. Although,

most of the times, the average case running times are considered for comparing the goodness of algorithms, the average case analysis would not be helpful in all the scenarios. Moreover, it is difficult to find whether the next input would be one of the typical inputs (as considered in the average case analysis). The scenario in which the input to quick sort is an output of some other component is also a contentious case. In such situations also, the input might not be typical and hence the average case analysis would fall flat.

The higher level algorithm for quick sort is as follows:

The procedure consists of two parts: `Random_Partition()` and `Quick_Sort()`

$A[]$ is the array which is to be sorted, *low* is the first index of the array and *high* is the last index of the array.

Algorithm 14.3   Random_Partition ()

Input: A, array; low: the first index; high: the last index
Random_Partition(A, low, high) returns s
```
{
r = (Random()+low)%high; //randomly pick an integer between low and high
swap(A[low], A[r]);
Random_Partion(A, low, high);
}
```

Discussion: The swap function exchanges the element at low with that at the position generated randomly.

The number of calls of the above algorithm in the `Random_Quick_Sort` is at most n.

Algorithm 14.4   Random_Quick_Sort ()

Input: A, array; low: the first index; high: the last index
Random_Quick_Sort(A, low, high) returns sorted array
```
{
pos=Random_Partition(A, low, high);
Random_Quick_Sort(A, low, pos-1);
Random_Quick_Sort(A, pos+1, high);
}
```

Discussion: The function calls the partition function which finds the position of the pivot, referred to as pos.

Complexity: In an array of length n, the random quick sort has a complexity $O(n \log n)$.

The position returned by the partition algorithm would determine the goodness of the algorithm. It is possible that we might get the median every time. In this case, the

equation of the time complexity would be $T(n) \leq 2T\left(\dfrac{n}{2}\right) + \theta(n)$. The solution of this can be found by using the Master theorem (refer to Chapter 9). This case would result in the running time of $O(n \log n)$.

Consider, for example, a position that divides the array in the ratio 1:9. In this case, the tree method for finding complexity can be used (Fig. 14.4).
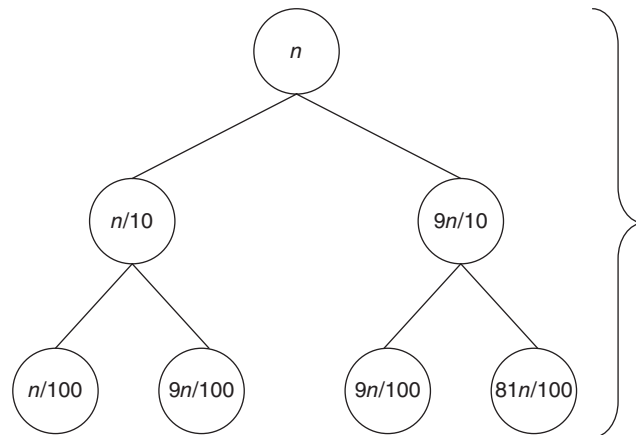
Figure 14.4 Complexity when the array is divided in the ratio 1:9

In this case, the complexity comes out to be $O(n \log n)$, as the base of logarithm does not affect the asymptotic complexity. Now consider a more general case, in which position the array is divided in the ratio $a{:}b$. In this case also, the tree method for finding complexity can be used (Fig. 14.5).
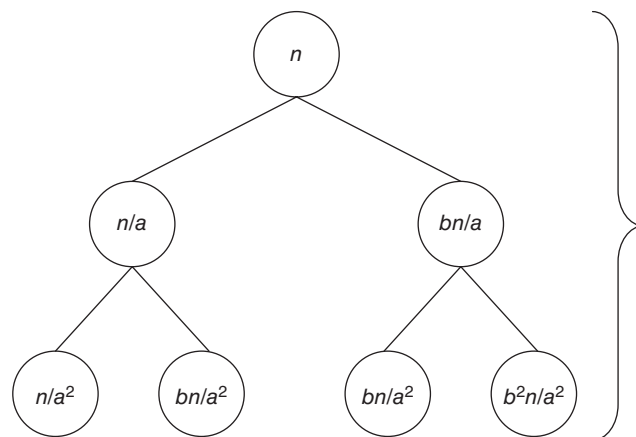
Figure 14.5 Complexity when the array is divided in the ratio $a{:}b$

In this case also the complexity comes out to be $O(n \log n)$, as the base of logarithm does not affect the asymptotic complexity. It may be concluded from the above discussion that every division results in an $n \log n$ complexity.

> **Tip:** For any input, the randomized quick sort has an expected running time of $O(n \log n)$.

14.6.4 Equality of Polynomials

A polynomial is algebraic expression consisting of many (poly) terms (nomial). As per the dictionary a polynomial is defined as follows.

Suppose there are two polynomials of order n. The first being $a_0 x^n + a_1 x^{n-1} + a_2 x^{n-2} + \cdots + a_n x^0$ and the second being $a_0 (x - \alpha)(x - \beta) \cdots (x - \xi)$ [n factors]. If one intends to check whether the above polynomials are equal or not, without having to multiply the polynomials, then the following strategy may help:

- Randomly generate a number and check whether the value obtained is same in both cases.
- If we are able to find a number for which the two are not equal, then we can surely state that the two polynomials are not equal. If, however, after some runs, one is not able to find a number for which the two polynomials are not equal, then there is a probability that the two are equal. However, even after many runs, the equality of two polynomials cannot be declared with certainty.

The algorithm for the above problem is as follows.

Algorithm 14.5 Check

Input: Two polynomials f and g, where $f = a_0 x^n + a_1 x^{n-1} + a_2 x^{n-2} + \cdots + a_n x^0$ and $g = a_0 (x - \alpha)(x - \beta) \cdots (x - \xi)$
Output: If the polynomials are equal, then the algorithm returns TRUE
Check (f, g)

```
{
x = random();// random() is a pseudo random number generator
if(f(x) = g(x))//if the value of f and g at 'x' are equal then return TRUE
        {
        return TRUE;
        }
}
```

Complexity: The algorithm runs a constant number of times, therefore, the complexity is $O(1)$.

The above algorithm is an example of recursively enumerable machine. There are two types of machines namely recursive and recursively enumerable. The first answers in a 'yes' or a 'no', as in if the machine accepts the input, it answers in a yes, else it answers in a 'no'.

The second type of machine answers in a 'yes', if the input is accepted by the machine, in the other case it does not do anything. The machines have been depicted in Fig. 14.6.
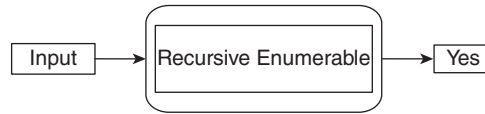
Figure 14.6

The reason why one may not want to multiply the factors of the second polynomial to establish the equality of the two polynomials is simple. If one multiplies the factors one by one, the complexity would be $O(n^2)$. The randomized approach, though not certain, is a better solution.

This technique makes sure that the algorithm ends after a definite time. However, there is no assurance that the task would be accomplished with a probability 1. This is, therefore, an instance of Monte Carlo algorithms.

The good part is that with every iteration, the probability of error decreases. For example, for a polynomial having degree n and the root lies in the range $[0, m]$. The probability of error in the first iteration would be (n/m), in the second iteration, it becomes $(n/m)^2$. In the kth iteration, the probability of error reduces to $(n/m)^k$. Note that if k approaches infinity, the probability of error becomes 0.

14.7 CONCLUSION

The chapter explores the fascinating world of randomized algorithms. The algorithms are being used to carry out tasks that were considered difficult at one point of time, owing to being computationally expensive. In order to understand the concept, one must appreciate the difference between the Monte Carlo and Las Vegas algorithm (refer Section 14.3). An example of each of the above has been given so that the reader can design the requisite algorithms as per the requirements.

The section on methods has specifically been included so that the topics studied in data structures can be used in randomized algorithm design as well. The reader is advised to explore 'hashing', in order to fine-tune the development skills.

Lastly, the topic on the complexity classes of randomized algorithms establishes the fact that the study of these algorithms have grown to the level of conventional algorithms. In order to appreciate this topic, as stated earlier, the reader is advised to go through Chapter 19.

The reader is expected to go through the web resources of this chapter, where the randomized approach for SAT 3 problem has been given.

Finally, nature also works randomly, but produces wonderful results. On the other hand, we work in a deterministic fashion and mostly end up messing up the things. So, despite not being deterministic, things do work out, mostly!

Points to Remember

- The type of algorithms that always produces a correct answer is called Las Vegas algorithms.
- The type of algorithms that terminates in a limited period of time but the output produced may or may not be correct are called Monte Carlo algorithms.
- In order to generate a random number in C#, the random class is used. To do so in C, the rand() function is used.
- Randomized algorithms improve the worst case complexity of some algorithms such as quick sort.
- The complexity classes for randomized algorithms are RP, Co-RP, BPP, PP, and ZPP.
- Probability theory helps in the analysis of randomized algorithms.

KEY TERMS

Bounded-error probabilistic algorithms (BPP) These algorithms are a class of algorithms in which if the input does not belong to the language, then the probability of corresponding machine accepting that input is less than or equal to p, whereas in the other case, the probability is greater than $(1 - p)$.

Class zero-error probabilistic P (ZPP) The class of languages that are both RP and Co-RP are referred to as ZPP. The algorithms are, in fact the Las Vegas algorithms, in which the probability of getting an answer is 1.

Co-RP problems The set contains languages L such that if an input does not belong to the language, then the probability of the Turing machine accepting that input is greater than or equal to half. If, on the other hand, the input belongs to the language, then the probability of the corresponding Turing machine accepting that input would be 0.

Las Vegas algorithms This type of randomized algorithms always produces a correct answer.

Monte Carlo algorithms This type of algorithms terminates in a limited period of time but the output produced may or may not be correct.

Probabilistic P class (PP) In this class of algorithms, if the input is accepted by the machine, then the probability is less than p and in the other case, the probability is greater than or equal to p.

RP problems The set contains languages L such that if an input belongs to the language, then the probability of the Turing machine accepting that input is greater than or equal to half. If, on the other hand, the input does not belong to the language, then the probability of the corresponding Turing machine accepting that input would be 0.

EXERCISES

I. Multiple Choice Questions

1. Which of the following are the types of randomized algorithms?
   - (a) Monte Carlo
   - (b) Las Vegas
   - (c) Both of the above
   - (d) None of the above

2. Which of the following eventually terminates with a probability 1?
   (a) Monte Carlo   (c) Both of the above
   (b) Las Vegas   (d) None of the above
3. In which of the following the running time is constant?
   (a) Monte Carlo   (c) Both of the above
   (b) Las Vegas   (d) None of the above
4. In which of the following the output produced may not be correct?
   (a) Monte Carlo   (c) Both of the above
   (b) Las Vegas   (d) None of the above
5. Randomized quick sort is an example of which of the following?
   (a) Monte Carlo   (c) Both of the above
   (b) Las Vegas   (d) None of the above
6. Randomized search is an example of which of the following?
   (a) Monte Carlo   (c) Both of the above
   (b) Las Vegas   (d) None of the above
7. In which of the following, Monte Carlo algorithms are not used?
   (a) Physics
   (b) Simulation of evaluation of galaxies
   (c) For finding correlated variations in digital ICs
   (d) For firing missiles
8. In which of the following, the probability of the Turing machine accepting the string that belongs to the language is greater than half, whereas in the other case, the probability is 0.
   (a) RP   (b) Co-RP   (c) BPP   (d) PP
9. In which of the following, the probability of the Turing machine accepting the string that does not belong to the language is greater than or equal to half, whereas in the other case, the probability is 0.
   (a) RP   (b) Co-RP   (c) BPP   (d) PP
10. In which of the following, the probability of machine accepting the string that belongs to the language is p, whereas in the other case, the probability is $(1 - p)$.
    (a) RP   (b) Co-RP   (c) BPP   (d) PP

II. Review Questions

1. What are randomized algorithms?
2. What is the difference between Las Vegas and Monte Carlo approaches?
3. What are the various complexity classes of randomized algorithms?

III. Application-based Questions

1. Design a Monte Carlo algorithm for searching an element in an array.
2. Design a Las Vegas algorithm for searching an element in an array.
3. Design a randomized algorithm for quick sort.

4. Design a randomized algorithm for the vertex cover problem (VCP). Find after how many runs the probability of error would become less than 1/128.
5. Design a randomized algorithm for the set cover problem (SCP). Find after how many runs the probability of error would become less than 1/512.
6. Design a randomized algorithm for the travelling salesman problem (TSP). Find after how many runs the probability of error would become less than 1/1024.
7. Design a randomized algorithm for the equality of polynomials. Find after how many runs the probability of error would become less than 1/128.
8. Design a randomized algorithm finding whether a number is prime or not. Find after how many runs the probability of error would become less than 1/128. If there are 16 processors what would be the probability of no processor receiving more than 2 jobs?
9. If there are 65,536 processors, then what would be the probability of no processor receiving more than 4 jobs?
10. Design a randomized algorithm for an even balancing in a hash table, so that minimum number of collisions occurs. Analyse the algorithm developed.
11. In networking, packet routing is one of the most important problems. The reader is requested to go through Reference [20] for a detailed explanation of this problem and then develop a randomized algorithm to solve it. Analyse your algorithm.
12. Classify the algorithms developed above into various complexity classes.

Answers to MCQs				
1. (c)	3. (a)	5. (b)	7. (d)	9. (b)
2. (b)	4. (a)	6. (a)	8. (a)	10. (c)

SECTION IV

ADVANCED TOPICS

Look deep into nature, and then you will understand everything better.

— *Albert Einstein*

Transform and Conquer

OBJECTIVES

After studying this chapter, the reader will be able to
- Understand the concept of transform and conquer
- List the applications of presorting
- Explain the concept of Gauss elimination
- Use Horner's rule for evaluating polynomials
- Find the lowest common multiple of two numbers

15.1 INTRODUCTION

How can a difficult problem be dealt with in the real life? The problem can be converted into a simpler form, it can be represented in a different way, it can be reduced to some other form, or it can be simply ignored. Though the last option is seldom used, there are instances where the last option can also be used. For instance, when many processes are competing for the same set of resources while holding some, the progress of processes can come to a halt. This situation is deadlock. Those who are familiar with operating systems must be knowing that neglecting the deadlock is one of the most common ways of handling deadlocks. However, in algorithms, we generally do not neglect a problem. The rest of the options constitute what is called *transform and conquer*.

Transform and conquer refers to changing the form of a problem into a simpler one, in order to solve it. The transformation, as per the literature review, can be of three types. Here, Levitin's interpretation of the topic is generally considered as the most credible one (Levitin, 2009). According to him, in the first type of transformation, a difficult instance of a problem is converted into its easier instance. The second type of transform and conquer involves the change in the representation of the given problem and the third is the reduction of the problem. The classification is depicted in Fig. 15.1.

This chapter discusses briefly the three types of transform and conquer method. Problem reduction will be dealt with in detail in Chapter 19. Moreover, we have been using this approach knowingly or unknowingly. For example, AVL trees discussed in Appendix A8 is an example of transform and conquer.
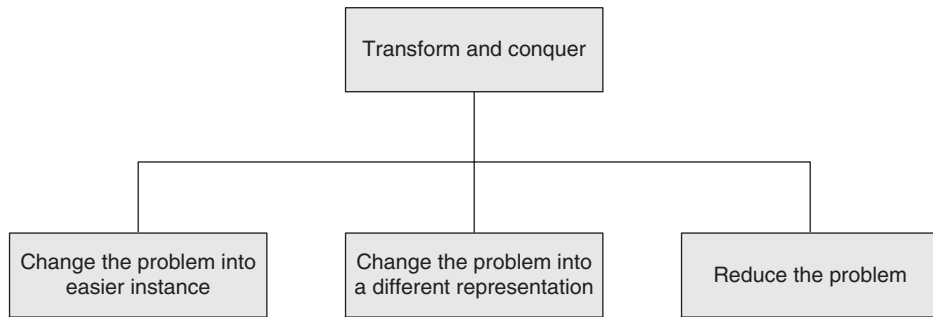
Figure 15.1 Types of transform and conquer

The following problems also come under the purview of transform and conquer. These are discussed in other chapters of the book—Heap creation (Sections 6.9 and 6.10), AVL trees (Appendix A8), and Reduction (Chapter 19).

The chapter has been organized as follows. Section 15.2 introduces the concept of presorting. Section 15.3 discusses an important method of solving linear equations: Gauss elimination, Section 15.4 presents LU decomposition, Section 15.5 explores Horner's rule and exemplifies it. This is followed by a section on finding the LCM of two numbers. The last section in the chapter explains how this method has been/can be applied to other problems.

15.2 PRESORTING

One of the most common examples of the transform and conquer technique is presorting, which is a special case of preprocessing. When a list is sorted before allowing it as an input to some algorithm, it is referred to as presorting. Presorting is required in many applications discussed as follows.

15.2.1 Applications of Presorting

Chapter 8 discussed the sorting in linear and quadratic time and Chapter 6 discussed the concept of binary search trees. This method helps us to sort a given list. What do we do with these sorted lists? How are they better than the unsorted lists we have? The answer is easy. Sorting helps us in binary search. Binary search has been explained in Chapter 6. It was stated that binary search is much more efficient as compared to the linear search and it becomes possible only because the list has already been sorted.

There is another advantage of presorting: finding repeated elements. If the list is sorted, then the repeated elements would appear at consecutive places, thus reducing the time complexity of finding them. Figure 15.2 depicts the above idea.

Another important application of presorting is finding the median of a given sequence. If the number of elements in a list is odd (say n) and the list is sorted, then

| 2 | 5 | 7 | 7 | 8 | 9 | 10 |

The sorted list will have the repeated elements at the consecutive locations
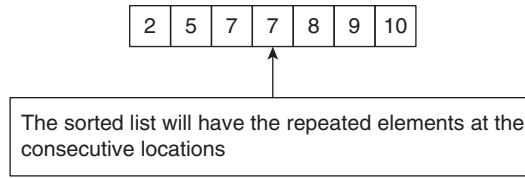
Figure 15.2 Finding repeated elements using presorting

the $(n + 1)/2$ element would be the median element. In the case of a list having even number of elements, the mean of $(n/2)$th and $((n + 2)/2)$th element is the median of the given sequence. Figure 15.3 depicts the concept and Fig. 15.4 depicts the applications of presorting.

| 2 | 5 | 7 | 8 | 9 | 10 | 11 |

$n = 7$, therefore, median is at the fourth position.
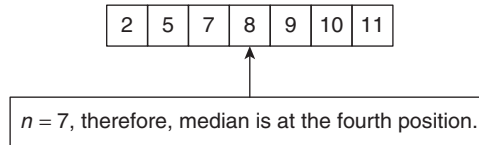
Figure 15.3 Finding median using presorting

Applications of presorting

Finding an element

Finding the number of occurences of an element

Finding medians
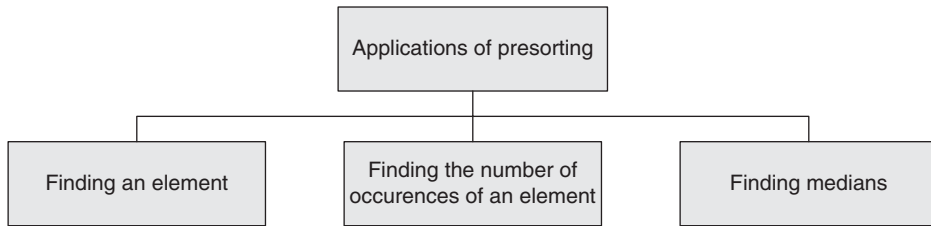
Figure 15.4 Applications of presorting

15.3 GAUSS ELIMINATION METHOD

In solving linear equations with two variables, one of the most common methods is elimination. Carl Friedrich Gauss suggested a similar method for solving linear equations with three or more variables.

Let there be three linear equations with variables x, y, and z. In order to solve a set of linear equations, the coefficients of x in the second and the third equations can be made zero by using the first equation. This is followed by making the coefficient of y zero in the third equation, by using the second equation.

Given equations

$$a_1x + b_1y + c_1z = d_1 \tag{15.1}$$

$$a_2x + b_2y + c_2z = d_2 \tag{15.2}$$

$$a_3x + b_3y + c_3z = d_3 \tag{15.3}$$

Apply the following operations:

- Multiply Eq. (15.1) by a_2 and Eq. (15.2) by a_1 and subtract the first equation from the second.
- Multiply Eq. (15.1) by a_3 and Eq. (15.3) by a_1 and subtract the first equation from the third.

The equations become

$$(a_1b_2 - b_1a_2)y + (c_2a_1 - c_1a_2)z = (d_2a_1 - d_1a_2) \tag{15.4}$$

$$(b_3a_1 - b_1a_3)y + (c_3a_1 - c_1a_3)z = (d_3a_1 - d_1a_3) \tag{15.5}$$

Now, multiply Eq. (15.4) by the coefficient of y in Eq. (15.5), and Eq. (15.5) by the coefficient of y in Eq. (15.4) and subtract Eq. (15.4) from Eq. (15.5).

The equations become

$$\begin{aligned} ((c_3a_1 - c_1a_3)(a_1b_2 - b_1a_2) - (c_2a_1 - c_1a_2)(b_3a_1 - b_1a_3))z \\ = ((d_3a_1 - d_1a_3)(a_1b_2 - b_1a_2) - (d_2a_1 - d_1a_2)(b_3a_1 - b_1a_3)) \end{aligned} \tag{15.6}$$

From Eq. (15.6), the value of z can be obtained. This value, when substituted in Eq. (15.4), will give the value of y. The values of z and y thus obtained can be substituted in Eq. (15.1) to obtain x.

There is another way to interpret the above method.

The coefficient matrix of the given set of equations along with the constants is as follows:

$$\begin{matrix} a_1 & b_1 & c_1 & d_1 \\ a_2 & b_2 & c_2 & d_2 \\ a_3 & b_3 & c_3 & d_3 \end{matrix}$$

The first operation, that is, making the coefficients of x in the second and the third equations zero by using the first equation, and then making the coefficient of y zero in the third equation, by using the second equation is basically a set of row operations, wherein the coefficients of the first column, second row, and the third row are made zero with the help of the first row. The corresponding matrix becomes

$$\begin{matrix} a_1 & b_1 & c_1 & d_1 \\ 0 & (a_1b_2 - b_1a_2) & (c_2a_1 - c_1a_2) & (d_2a_1 - d_1a_2) \\ 0 & (b_3a_1 - b_1a_3) & (c_3a_1 - c_1a_3) & (d_3a_1 - d_1a_3) \end{matrix}$$

The next step would be to make the second column of the third row as zero. This can be done by multiplying the second row by the coefficient of y in the third row, and the third row by the coefficient of y in the second row, followed by subtracting the second row from the third. This step leads to the creation of a triangular matrix,

$$\begin{matrix} a_1 & b_1 & c_1 & d_1 \\ 0 & (a_1b_2 - b_1a_2) & (c_2a_1 - c_1a_2) & (d_2a_1 - d_1a_2) \\ 0 & 0 & c_4 & d_4 \end{matrix}$$

The value of z becomes $\dfrac{d'_4}{c'_4}$. This value of z would be used to find the value of y from the second row and finally the value of x from the first row.

The following Illustrations will enable us to understand the concept:

Illustration 15.1 Solve the following set of equations using Gauss elimination method:

$$2x + 3y + 4z = 9$$
$$3x + 2y + z = 6$$
$$x + 2y + z = 4$$

Solution

Step 1 Eliminate x by multiplying the second equation by 2 and the first by 3, followed by subtracting the second equation from the first. Similarly, x can be eliminated from the third equation by multiplying the third equation by 2 and subtracting it from the first equation. The new set of equations, thus obtained, is as follows:

$$2x + 3y + 4z = 9$$
$$5y + 10z = 15$$
$$y - 2z = -1$$

Finally, the variable y can be eliminated from the third equation by multiplying the third equation by 5 and subtracting it from the second. The new set of equations, thus obtained, is as follows:

$$2x + 3y + 4z = 9$$
$$5y + 10z = 15$$
$$-20z = -20$$

The value of z, therefore, becomes 1 and the value of y becomes 1 (on substituting $z = 1$ in the second equation). These values of y and z, when substituted in the first equation gives $x = 1$.

Illustration 15.2 Solve the following set of equations using Gauss elimination method:

$$2x - y - z = 3$$
$$x + y + 2z = 3$$
$$x + 2y - 7z = 4$$

Solution

Step 1 Eliminate x by multiplying the second equation by 2 and subtracting the second from the first. Similarly, x can be eliminated from the third equation by multiplying the third equation by 2 and subtracting it from the first equation. The new set of equations, thus obtained, is as follows:

$$2x - y - z = 3$$
$$3y + 5z = 3$$
$$5y - 13z = 5$$

Finally, the variable y can be eliminated from the third equation by multiplying the third equation by 3 and the second by 5, followed by subtracting it from the second. The new set of equations, thus obtained, is as follows:

$$2x - y - z = 3$$
$$3y + 5z = 3$$
$$+64z = 0$$

The value of z, therefore, becomes 0 and the value of y becomes 1 (on substituting $z = 0$ in the second equation). These values of y and z, when substituted in the first equation gives $x = 2$.

How is Gauss elimination transform and conquer?

In the method, the given augmented matrix is converted to the triangular form, thus reducing the effort to calculate the value of variable z. Figure 15.5 depicts the steps.
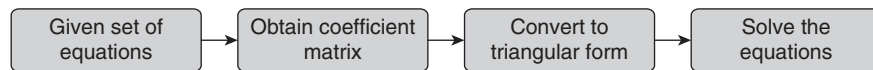
Figure 15.5 Transform and conquer in Gauss elimination

This method can also be used for solving a set of n equations. The process of solving the equations remains the same. The coefficient matrix obtained would be converted into triangular matrix, followed by the evaluation of the last variable and back substitution. The algorithm for conversion of a given matrix into the triangular form is given in Algorithm 15.1.

Algorithm 15.1   Matrix reduction for Gauss elimination

Input: A, the coefficient matrix of the given set of equations and n, the number of equations.
Output: The reduced matrix (mostly in triangular form), which helps us to find the values of the variables.

```
{
for (i=1; i<=n ;i++)
    {
    add the constants to the last column of the coefficient matrix, thus making
    it augmented.
    }
for(i=1; i<=n-1; i++)
```

```
    {
    for(j=(i+1); j< n ; j++)
        {
        for(k=i ; k<=(n+1), k++)
            {
                A[j, k] = A [j, k] - A [i, k] *A[j, i] / A[i, i];
            }
        }
    }
}
```

Complexity: The algorithm has a loop within a loop and a loop also in the inner loop, thus making the complexity of the algorithm $O(n^3)$.

A similar method, in which the coefficient matrix of the given set of equations is reduced to its diagonal form, is called Gauss–Jordan method. The method is similar to the above, except that after obtaining the triangular matrix, the process does not stop. The value of the last row's last column is used to make all the elements of the last column, except for the last row, zero. The same procedure is repeated, till a diagonal matrix is obtained.

The above method is far efficient than the Cramer's rule and the solution of given set of equations by finding the inverse of the coefficient matrix, which has been discussed in Appendix A3.

15.4 LU DECOMPOSITION

Gauss elimination method, however, becomes incompetent in some cases. For example, consider the case when the coefficients are same but the constants differ. In such cases, another method called LU decomposition comes to our rescue. The concept of LU decomposition is simple. Any matrix can be written as a product of upper triangular matrix U and a lower triangular matrix L, that is,

$$A = L \times U$$

where A is the coefficient matrix, L is the lower triangular matrix, and U is the upper triangular matrix.

$$
\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \begin{bmatrix} 1 & u_{12} & u_{13} \\ 0 & 1 & u_{23} \\ 0 & 0 & 1 \end{bmatrix}
$$

$$
\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} l_{11} & (l_{11}u_{12}) & (l_{11}u_{13}) \\ l_{21} & (l_{21}u_{12}+l_{22}) & (l_{21}u_{13}+l_{22}u_{23}) \\ l_{31} & (l_{31}u_{12}+l_{32}) & (l_{31}u_{13}+l_{32}u_{23}+l_{33}) \end{bmatrix}
$$

The values of the unknowns can be found by equating the corresponding coefficients. The values thus obtained are as follows.

$$\begin{cases} l_{11} = a_{11}; l_{21} = a_{21}; l_{31} = a_{31} \\[2mm] l_{11}u_{12} = a_{12}, \text{ therefore, } u_{12} = \dfrac{a_{12}}{l_{11}} = \dfrac{a_{12}}{a_{11}} \\[2mm] l_{21}u_{12} + l_{22} = a_{22}, \text{ therefore, } l_{22} = a_{22} - l_{21}u_{12} \\[2mm] l_{31}u_{12} + l_{32} = a_{32}, \text{ therefore, } l_{32} = a_{32} - l_{31}u_{12} \\[2mm] l_{11}u_{13} = a_{13}, \text{ therefore, } u_{13} = \dfrac{a_{13}}{l_{11}} = \dfrac{a_{13}}{a_{11}} \\[2mm] l_{21}u_{13} + l_{22}u_{23} = a_{23}, \text{ therefore, } u_{23} = \dfrac{a_{23} - l_{21}u_{13}}{l_{22}} \\[2mm] l_{31}u_{13} + l_{32}u_{23} + l_{33} = a_{33}, \text{ therefore, } l_{33} = a_{33} - l_{31}u_{13} - l_{32}u_{23} \end{cases}$$

The above method, as a matter of fact, is also transforming the coefficient matrix into the product of a lower and an upper triangular matrix and thus obtaining the result. The steps in the process are summarized in Fig. 15.6.
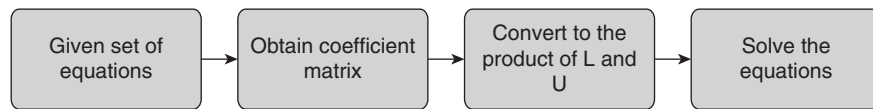
Figure 15.6 Transform and conquer in LU decomposition

15.5 HORNER'S METHOD

The following discussion explores the efficient ways of evaluating polynomials at given values. The evaluation of polynomials is important in many algorithms. If an efficient method of the task is available, all such algorithms would become efficient. Now, the reader must be wondering that if such a method was known to them in their high school, it would have been greatly benefitted. This method is attributed to William George Horner (1837–1976), who was a lecturer in mathematics in University of Sydney.

In fact, it is widely believed that the following method was known before him. In fact, the felicitations given to him during his lifetime have been a source of contention.

According to the Horner's rule or scheme, a given polynomial $a_0x^n + a_1x^{n-1} + a_3x^{n-2} + \cdots + a_n$ can be written as $x(a_0x^{n-1} + a_1x^{n-2} + a_3x^{n-3} + \cdots) + a_n$, which in turn can be reduced to the form $xP_{n-1} + a_n$, where P_{n-1} can be expressed as $xP_{n-2} + a_{n-1}$ and so on. At any instance, the value of P_{n-k} can be expressed as $P_{n-(k+1)} + a_{n-(k)}$.

The concept can be understood by the following examples.

The quadratic polynomial $ax^2 + bx + c$ can be written as $x(ax + b) + c$. The method requires evaluation of $(ax + b)$ and then multiplying it by x, followed by the addition

with c. There are two advantages of such reduction. First, the number of multiplications is greatly reduced (generally) and second is that the above method eliminates the need of calculating the powers of a number.

The cubic polynomial $ax^3 + bx^2 + cx + d$ can be written as $x(ax^2 + bx + c) + d$. The above procedure of converting a quadratic polynomial to a linear one is then followed. The repeated application of this method reduces the given cubic polynomial to $x(x(ax + b) + c) + d$. This reduces the number of multiplications from 7 to 3 and also eliminates the need of calculating the powers of x.

It is left for the reader to evaluate the reduction in the number of multiplications, when a bi-quadratic polynomial is reduced using the above method.

Evaluating Expression Using Brute Force

In evaluating $a_0 x^n + a_1 x^{n-1} + a_3 x^{n-2} + \cdots + a_n$ using brute force, the powers of x need to be calculated. The number of multiplications involved in the calculation of $x^n, x^{n-1}, ..., x^2$ would be $\frac{1}{2} \times (n(n-1))$ (refer to the formula for the sum of an AP in Section 2.3). The evaluation of powers is followed by the multiplication of these results with the coefficients. There would be n such multiplications. The total number of multiplications involved in the evaluation of a polynomial by brute force is therefore $\frac{1}{2} \times ((n+1) \times (n+2))$.

> **Tip:** The complexity of polynomial evaluation by brute force is $O(n^2)$.

The above procedure can be expressed by the following algorithm.

Algorithm 15.2   Horner's scheme

Input: The coefficients of the polynomial $a_0 x^n + a_1 x^{n-1} + a_3 x^{n-2} + \cdots + a_n$ and the value of x_0 for which it needs to be evaluated.
Output: The value of $a_0 x_0^n + a_1 x_0^{n-1} + a_3 x_0^{n-2} + \cdots + a_n$

```
{
i=n.
Pᵢ = aᵢ
do
    {
    Pᵢ₋₁ = aᵢ₋₁ + PᵢX₀
    i=i-1;
    }
while (i ≥ 0)
}
```

Complexity: $O(n)$, where n is the degree of the polynomial.

Illustration 15.3 Evaluate the value of $3x^3 + 7x^2 + 37$ at $x = 3$.

Solution

Method 1 The given polynomial can be written as $x(3x^2 + 7x^1) + 37$.
The polynomial $3x^2 + 7x^1$ can in turn be written as $x(3x^1 + 7)$.

- The value of $3x^1 + 7$ at $x = 3$ is $3 \times 3 + 7 = 16$.
- Therefore, the value of $3x^2 + 7x^1 = x(3x + 7)$ becomes $3 \times 16 = 48$.
- The value of the original polynomial becomes $x(3x^2 + 7x^1) + 37 = 3 \times 48 + 37 = 181$.

Method 2 There is another method of calculating the above expression. The method involves tracing of the steps of Algorithm 15.1.

The coefficients of the given polynomial are 3, 7, 0, 37 (in the order of exponents of x). The expression is to be evaluated at $x = 3$. The second row of Table 15.1 shows the iterations of the procedure.

Table 15.1 Evaluating an expression using Horner's method

	3	7	0	37	Coefficients
3	3	$3 \times 3 + 7 = 16$	$16 \times 3 + 0 = 48$	$48 \times 3 + 37 = 181$	

Illustration 15.4 Evaluate the value of $4x^3 + 7x^2 + 5x + 3$ at $x = 2$.

Solution The coefficients of the given polynomial are 4, 7, 5, 3 (in the order of exponents of x). The expression is to be evaluated at $x = 2$. The second row of Table 15.2 shows the iterations of the procedure.

Table 15.2 Evaluating an expression given in Illustration 15.4 using Horner's method

	4	7	5	3	Coefficients
2	4	$4 \times 2 + 7 = 15$	$15 \times 2 + 5 = 35$	$35 \times 2 + 3 = 73$	

15.6 LOWEST COMMON MULTIPLE

As stated earlier, a number x may be expressed as the product of the powers of prime numbers, that is, $x = p_1^{i1} \times p_2^{i2} \times \cdots \times p_k^{ik}$. For example, 2134 may be expressed as $2 \times 11 \times 97$.

If two prime numbers are expressed as stated above, that is, two numbers say, 'x' and 'y' are written as

$$x = p_1^{i1} \times p_2^{i2} \times \cdots \times p_k^{ik} \text{ and}$$
$$y = p_1^{j1} \times p_2^{j2} \times \cdots \times p_k^{jk}, \text{ then the number}$$
$$z = p_1^{l1} \times p_2^{l2} \times \cdots \times p_k^{lk}$$

where $l_m = \begin{cases} i_m, \text{if } i_m > j_m \\ j_m, \text{otherwise} \end{cases}$ is called the least common multiple of 'x' and 'y'.

The idea of an LCM is simple. As stated earlier, since a number may have numerous factors, two numbers may also have common multiples. The least of these common multiples is referred to as the GCD.

Though the above method appears simple, it is computationally expensive. The C code given in the web resource of this book finds the LCM of two numbers using the above approach. It can be inferred from the code that the computational complexity of the process involved is $O(n^4)$.

The following illustrations explain the above process.

Illustration 15.5 Find the LCM of 102685968 and 103733784.

Solution Since any number can be expressed as the powers of prime numbers, first of all the two numbers should be expressed as the powers of prime numbers.

$$102685968 = 2^4 \times 3^5 \times 7^4 \times 11^1$$
$$103733784 = 2^3 \times 3^7 \times 7^2 \times 11^2$$

In the next step, the greater of the exponents of a common prime number is to be extracted.
- The powers of 2 in the two numbers are 4 and 3, out of which 4 is greater.
- The powers of 3 in the two numbers are 5 and 7, out of which 7 is greater.
- The powers of 7 in the two numbers are 4 and 2, out of which 4 is greater.
- The powers of 11 in the two numbers are 1 and 2, out of which 2 is greater.

So, the LCM of the two numbers is $2^4 \times 3^7 \times 7^4 \times 11^2$.

The above method, though easy, is computationally expensive. In Chapter 17, we will see a more efficient method of calculating the GCD. Since

$$LCM(p,q) \times GCD(p,q) = p \times q$$

Therefore, the LCM can be calculated by computing the GCD of the numbers and dividing the answer by the product of the two numbers. Figure 15.7 depicts the process of calculating the LCM of two numbers.
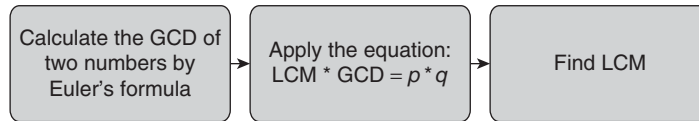
Figure 15.7   Transform and conquer in LCM calculation

15.7 NP-HARD PROBLEMS

Transform and conquer technique is used for solving NP-hard problems discussed in Chapter 19. It is also used for solving NP-hard problems that are not NP-complete (discussed in Appendix A7). Section 19.5 of Chapter 19 can be perceived as conversion of a problem into some other form and then handling it. The strategy involved is transform and conquer. For example, if it is required to find the minimum cost Hamiltonian cycle in a weighted, directed graph (travelling salesman problem), then the problem does not remain NP-complete, as will be seen in Section 19.6. The reason is that until all the possible paths have been processed, there is no way of knowing whether the solution that we have obtained is the minimum or not. We will also learn in Chapter 19 that the maximum clique problem and minimum cover also belong to this class. Though it is easy to verify that the clique obtained by a particular algorithm contains k nodes, it is almost impossible to state whether the clique obtained is the maximum.

Appendix A7 discusses some of the ways to deal with such problems. Though, not a part of most of the UG programs, the algorithms surely give an insight to an efficient way of dealing with such problems. Figure 15.8 depicts the process.
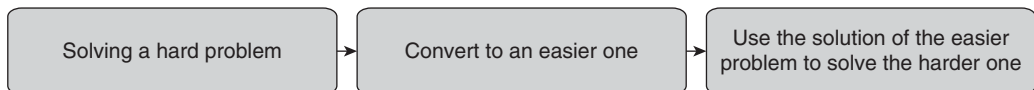
Figure 15.8   Transform and conquer in hard problems

Although the concept transform and conquer was being used in earlier chapters, this chapter formally establishes it. It is one of the most natural concepts used in problem solving.

15.8 CONCLUSION

The chapter discussed one of the most important methods of solving a set of linear equations that is Gauss elimination. The section also introduced LU decomposition. Another important topic covered in the chapter was Horner's rule. The applications of these methods can be appreciated in modelling and simulation. The next section discussed the concept of LCM, though not very important, the topic was given to establish the concept of transform and conquer.

Heap and heapsort discussed in Chapter 6 can also be considered as an example of transform and conquer. The reader is strongly encouraged to go through Appendix A7 on NP-hard scheduling problems and Chapter 19, in order to fully understand the concept.

Finally, as it is said 'what's in the name', once we are well versed with algorithm designing, we would automatically use the technique, even unknowingly.

Points to Remember

- Presorting is used in many applications like finding repeated elements and finding medians.
- Gauss elimination method is one of the efficient ways of finding the solution to linear equations.
- LU decomposition helps in finding the solution to a set of linear equations.
- The complexity of Gauss elimination is $O(n^2)$.
- Horner's method is used to find the value of a polynomial at a given value of the independent variable.
- The complexity of Horner's algorithm is $O(n)$.

KEY TERMS

Presorting When a list is sorted before giving it as an input to some algorithm, it is referred to as presorting.

Least common multiple If two prime numbers are expressed as stated above, that is, two number say, 'x' and 'y' are written as

$$x = p_1^{i1} \times p_2^{i2} \times \cdots \times p_k^{ik} \text{ and}$$
$$y = p_1^{j1} \times p_2^{j2} \times \cdots \times p_k^{jk}, \text{ then the number}$$
$$z = p_1^{l1} \times p_2^{l2} \times \cdots \times p_k^{lk}$$

where $I_m = \begin{cases} i_m, \text{if } i_m > j_m \\ j_m, \text{otherwise} \end{cases}$ is called the least common multiple of 'x' and 'y'.

EXERCISES

I. Multiple Choice Questions

1. Which of the following are the advantages of presorting?
   (a) Finding an element
   (b) Finding median
   (c) Finding the number of occurrences of an element
   (d) All of the above

2. Which of the following are not a type of transform and conquer?
   (a) Changing instance
   (b) Changing representation
   (c) Reduction
   (d) Backtracking
3. Which of the following can be solved using transform and conquer?
   (a) Finding LCM of two numbers
   (b) AVL
   (c) Both
   (d) None of the above
4. Which of the following cannot be solved using transform and conquer?
   (a) Searching a number in a list
   (b) Balancing trees
   (c) Both
   (d) None of the above
5. Which of the following is the advantage of binary search?
   (a) Reduces time complexity
   (b) Reduces space complexity
   (c) Both
   (d) None
6. Which of the following is correct as regards Gauss elimination?
   (a) In this method the coefficient matrix is converted into diagonal form
   (b) In this method the determinant of coefficient matrix is evaluated
   (c) In this method the inverse of coefficient matrix is evaluated
   (d) All of the above
7. What is L in LU decomposition?
   (a) Lower triangular matrix
   (b) Upper triangular matrix
   (c) Diagonal matrix
   (d) None of the above
8. What is U in LU decomposition?
   (a) Lower triangular matrix
   (b) Upper triangular matrix
   (c) Diagonal matrix
   (d) None of the above
9. Which method is used in LU decomposition?
   (a) Transform and conquer
   (b) Backtracking
   (c) Both
   (d) None of the above
10. Which method is used in Gauss–Jordan?
    (a) Transform and conquer
    (b) Backtracking
    (c) Both
    (d) None of the above

II. Numerical Problems

1. Solve the following using Gauss elimination:

   (a) $2x - y + z = 4$
   $x + y + z = 1$
   $x - 3y - 2z = 2$

   (b) $x + y + z = 6$
   $x - y + 2z = 5$
   $3x + y + z = 8$

   (c) $x + 2y + z = 3$
   $2x + 3y + 2z = 5$
   $3x - 5y + 5z = 2$
   $3x + 9y - z = 4$

2. Find the LCM of the following:
   (a) 2348 and 1234
   (b) 326474439 and 264284
   (c) 247395305243240 and 242940294247

3. Find the value of the following polynomials at $x = 7$.

   (a) $3x^4 + 2x^3 + x^2 + 7$

   (b) $31x^4 + 12x^3 + x^2 + 17x + 7$

   (c) $7x^5 + 2x^2 + x - 7$

   (d) $323x^2 + 2x + 734$

Answers to MCQs

1. (d)	3. (c)	5. (a)	7. (a)	9. (a)
2. (d)	4. (a)	6. (b)	8. (b)	10. (a)

CHAPTER 16

Decrease and Conquer

OBJECTIVES

After studying this chapter, the reader will be able to
- Understand the concept of decrease and conquer (D&C)
- Explain various types of decrease and conquer
- Recognize how insertion sort, DFA, and BFS are related to decrease and conquer
- Find out how to find permutations of a number

16.1 INTRODUCTION

A problem can be easily solved if there is a way to solve the smaller version of the problem. This smaller version can be obtained by decreasing the number of inputs by a constant number, or decreasing the number of inputs by a constant or variable factor. The interesting part of this chapter is that there is practically nothing new to learn. The reader is expected to revisit the algorithm which he has already read. The difference, however, would be in the perspective, the way of seeing things. As per the literature review, the approach is used along with other approaches to accomplish a goal. For instance, the brute force analysis, which we have been studying all along, is nothing but the first variant of decrease and conquer wherein the input size is decreased by one step every time. This can be understood by taking the following example. Suppose a teacher expects to write 15 books in his teaching career. In the first year, he gets one book published. This takes him towards his goal and the remaining number of books reduces to 14. In the next year, he publishes one more book. This takes him one more step towards his goal. In the same way, he continues and in the next 15 years achieves his goal.

If we analyse the above strategy, we might not find anything new in it. Most of us would find it a natural way of doing a task. The difference, however, is in the way of looking at it. The strategy can be viewed as a decrease and conquer approach of the first type. Here, the decrease in the input size is by a constant number (1). Had the above person been writing three books a year, even then it would have been a decrease and conquer approach wherein the factor is 3 instead of 1. The approach has been depicted in Fig. 16.1.
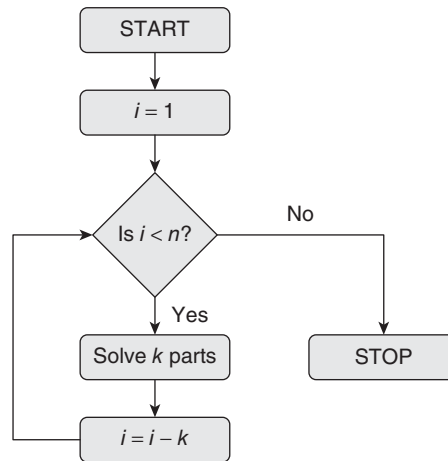
Figure 16.1   Decrease and conquer, decrease by a constant number

The second way of implementing decrease and conquer is to reduce the size by a constant factor. Dividing the problem into two sub-problems of equal size, as was done in merge sort and quick sort, also falls under this category. According to some authors like Levitin, the decrease and conquer approach is more efficient than the divide and conquer approach. However, the definition of decrease and conquer approach suggests something else. The implementation of decrease and conquer approach by a constant factor is in fact same as divide and conquer.

For instance, merge sort divided the array into two equal (in almost all the cases) parts and then solves the individual parts. So, the problem is being divided into two sub-problems of equal sizes and handled and is hence decrease and conquer wherein the decrease is by a constant factor. The approach has been depicted in Fig. 16.2.
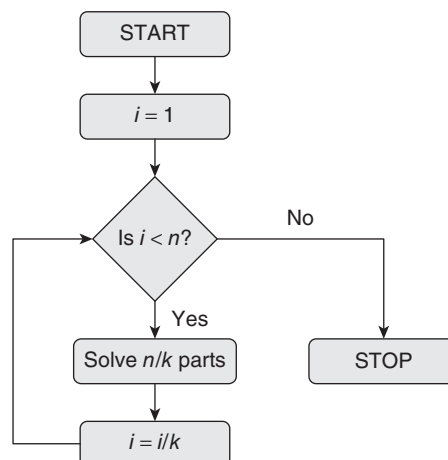
Figure 16.2   Decrease and conquer, decrease by a constant multiple

There is another way of implementing decrease and conquer. This is using a recursion formula that passes different input sizes at successive calls. The sizes are generally not related to each other. One of the most common examples of this is the GCD algorithm. The three approaches have been depicted in Fig. 16.3.
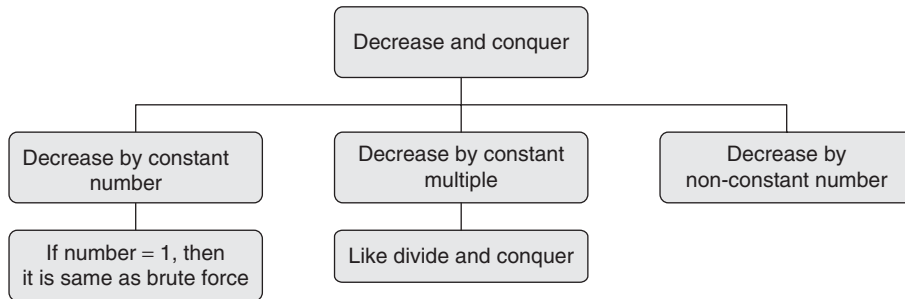
Figure 16.3 Classification of decrease and conquer

The above approaches have been explained in the sections that follow. The second section is about finding power set of a given set. A new perspective of looking at this problem has been presented in the section. The third section of the chapter examines the applicability of decrease and conquer in breadth first search and depth first search. The fourth section deals with permutation generation and the following section presents variable decrease.

16.2 FINDING THE POWER SET OF A GIVEN SET

At times all the possible subsets of a given set need to be enlisted. For example, in order to find all possible relations of a given set A, we need to find all the possible subsets of $A \times A$. In such cases, power set comes to our rescue.

A power set is defined as the set of all the subsets of a given set. For example, the power set of $\{x\}$ would be $\{\{\},\{x\}\}$. The power set of $\{1,2\}$ is $\{\{\}, \{1\}, \{2\}, \{1,2\}\}$. Here, a set having n elements has 2^n subsets and hence the number of elements in its power set would be 2^n. In order to find the subset of a set consisting of more than one element, the following recursive equation can be used:

$$\text{Power set of}\left(\{A\}\cup\{x\}\right) = \text{Power set of}\{A\}\cup\{A,x\}\cup\{x\}$$
$$= \{\{\ \},\{A\},\{A,x\},\{x\}\} \tag{16.1}$$

$$\text{Power set of}\left(\{\ \}\cup\{x\}\right) = \{\{\ \},\{x\}\} \tag{16.2}$$

In order to understand the above point, let us find the power set of $\{1, 2, 3, 4\}$. The power set can be found by finding the power set of $\{1, 2, 3\}$ and then taking union of each element with $\{4\}$ and then taking the union of the result with $\{4\}$.

The power set of {1, 2, 3} can also be calculated in the same way. The process has been depicted in Fig. 16.4.
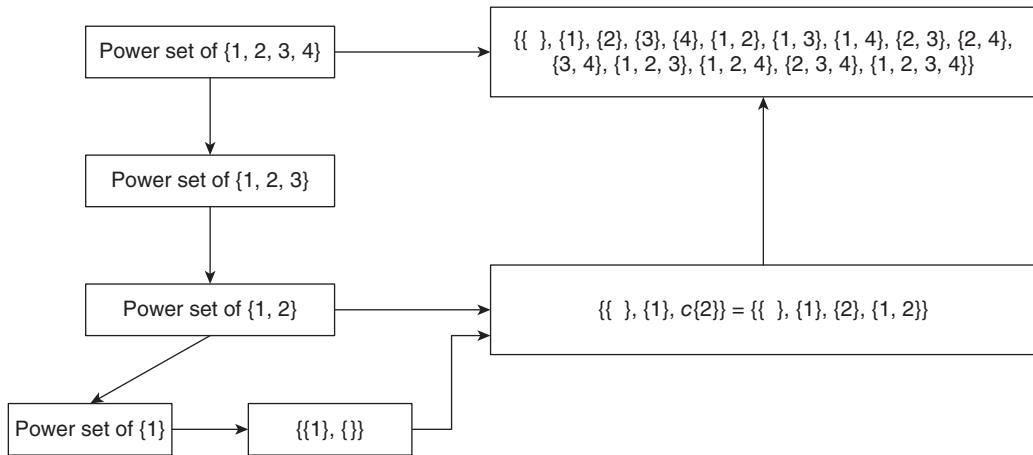
Figure 16.4 Calculating power set by decrease and conquer

The method followed to calculate the power set of a given set decreases the number of elements by one in each step. Since we know that the power set of a set containing a single element has the element and a NULL set as its elements. The method, therefore, falls under the category of decrease and conquer. The sub-type of algorithm followed here is decrease by a constant number (1, in this case). Actually, the strategy is not much different from a brute force approach.

The insertion sort, explained in Section 8.6 of the book, also makes use of this strategy. In the insertion sort also, one element is processed at a time thus decreasing the number of input elements by one in the next iteration. The formal algorithm of the above process is given as follows.

Algorithm 16.1 Power set

Input: Set A
Output: Set B, containing the subsets of A
Function used: con(A, x). The function generates a set having all the elements of set A and the sets formed by taking union of each element of A with x.

```
Power_Set(A) returns set B
{
      Split A into two parts H|T, T is a single element and H is a set
      if(H==NULL)
            {
            B= {{}, T};
            }
      else
```

```
                {
                B=con(Power_Set(H), T);
                }
        return B;
    }
```

Complexity: The number of recursive calls of the above algorithm would be n. Each call performs the $O(\log n)$ operation. Therefore, the complexity of the above algorithm would be $O\left(\sum n \times \log(n)\right)$.

16.3 BREADTH FIRST SEARCH AND DEPTH FIRST SEARCH

The algorithms of depth first search and breadth first search have already been described in Section 7.6. However, a brief overview of both has been given in this section.

In depth first search, one visits a node, processes it, and goes to its adjacent vertex and repeats the process. When there is no node to process, then using backtracking, we travel to a node whose adjacent nodes have not been visited. On reaching that node the process starts again. Figure 16.5 depicts the process.
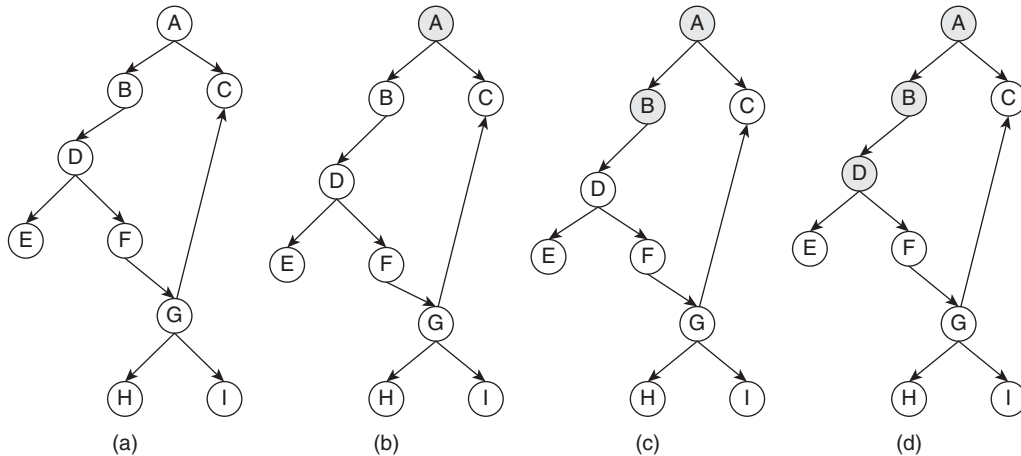
Figure 16.5 Steps in depth first search (*Contd*)

Hence, it is evident from the figure, the algorithm processes one node at a time thus decreasing the remaining nodes by 1. Therefore, it is a type of decrease and conquer algorithm.

In breadth first search, one visits a node, processes it, and goes to its adjacent vertices. Here, all the adjacent vertices are processed first and then the control moves to the node that was processed at the second position. The whole process is repeated for that node also. The process stops when there is no node left to be processed.
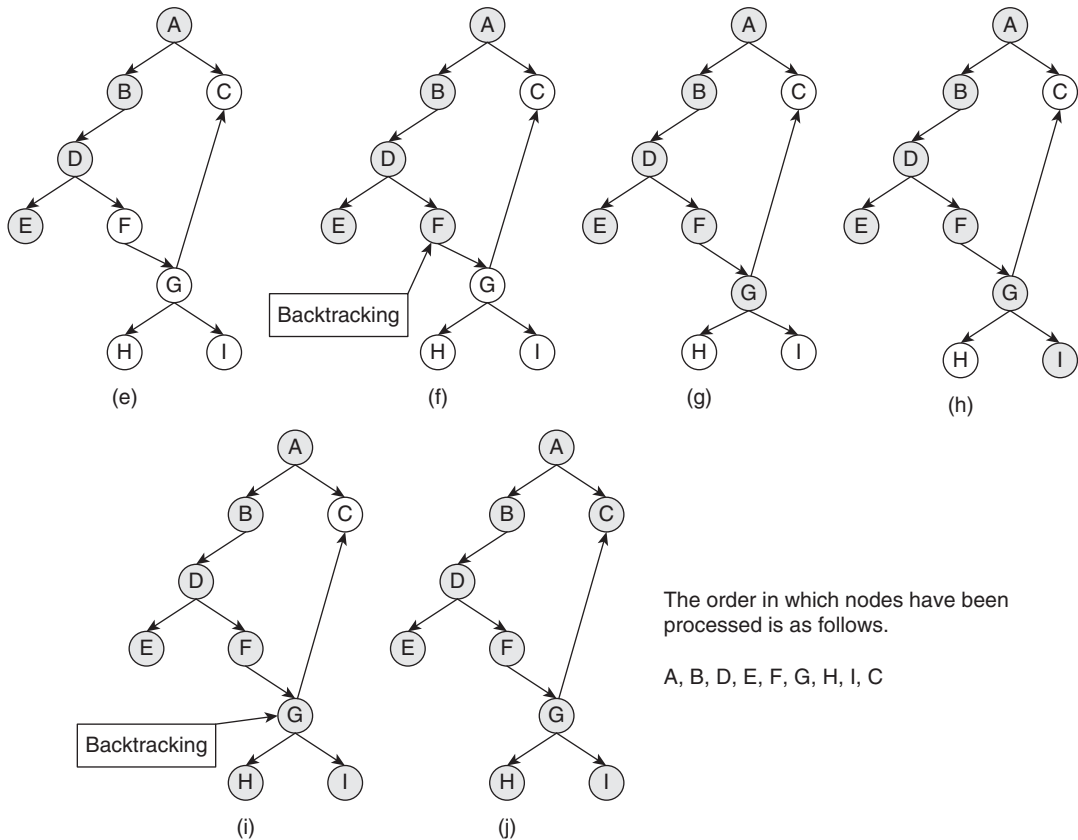
Figure 16.5 (*Contd*) Steps in depth first search

Figure 16.6 depicts the process. As is evident from the figure, the algorithm processes one node at a time thus decreasing the remaining nodes by 1. Therefore, it is a type of decrease and conquer algorithm.
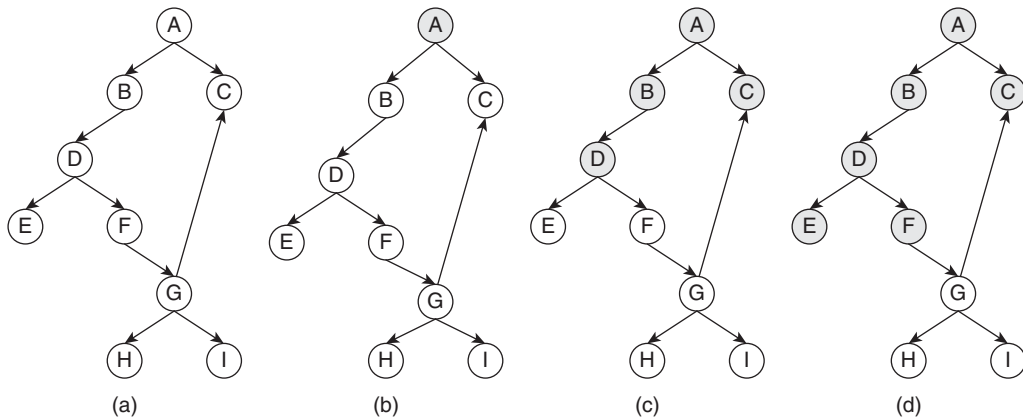
Figure 16.6 Steps in breadth first search (*Contd*)

(e)　　　　　　　　(f)　　　　　　　　(g)
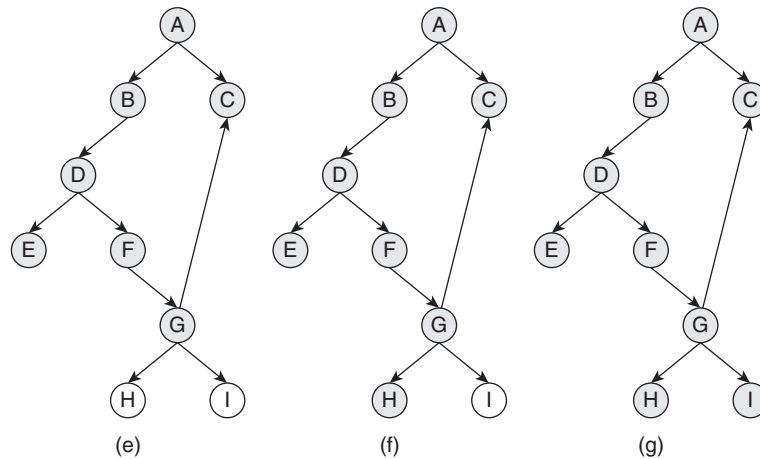
The order in which nodes have
been processed is as follows

A, B, C, D, E, F, G, H, I

Figure 16.6 (*Contd*) Steps in breadth first search

16.4 PERMUTATION GENERATION

Suppose there are three items say A, B, and C and we need all the possible arrangements of two items, selected from these three numbers such that the repetition is not allowed and the order is important. In this case, the solution would be {AB, BA, AC, CA, BC, CB}. That is, there are six permutations.

Permutations, in fact, refer to arrangements of *r* elements from the given set such that repetition is not allowed and order is important.

In order to generate permutations of a given sequence, many algorithms are used. One of the most common algorithms is Johnson–Trotter algorithm. In this algorithm, a pointer points to the largest element of the sequence and the element is contently swapped with its immediate neighbour on the left. The process has been depicted by an example shown in Fig. 16.7. The process has been explained in Algorithm 16.2.

Algorithm 16.2 Johnson–Trotter algorithm

Input: A sequence of n elements, stored in array a
Output: All the permutations of that sequence

```
ALGORITHMPerm(a[])
```

```
{
Perm = null;
for (i=0;i<n;i++)
        {
        for(j=n-1 ; j>i+i ; j--)
                {
                swap(a[j], a[j-1]);
                print: a;
                }
        }
Perm = Perm + a;||+is concatenation
Return Perm in the reverse order
}
```

Complexity: The number of swaps in the above procedure would be n, in the second iteration, the number of swaps would be $(n-1)$ and so on. So, the total number of movements of the arrow would be

$$1+2+\cdots+n=n\times\frac{n+1}{2}$$

which is $O(n^2)$.
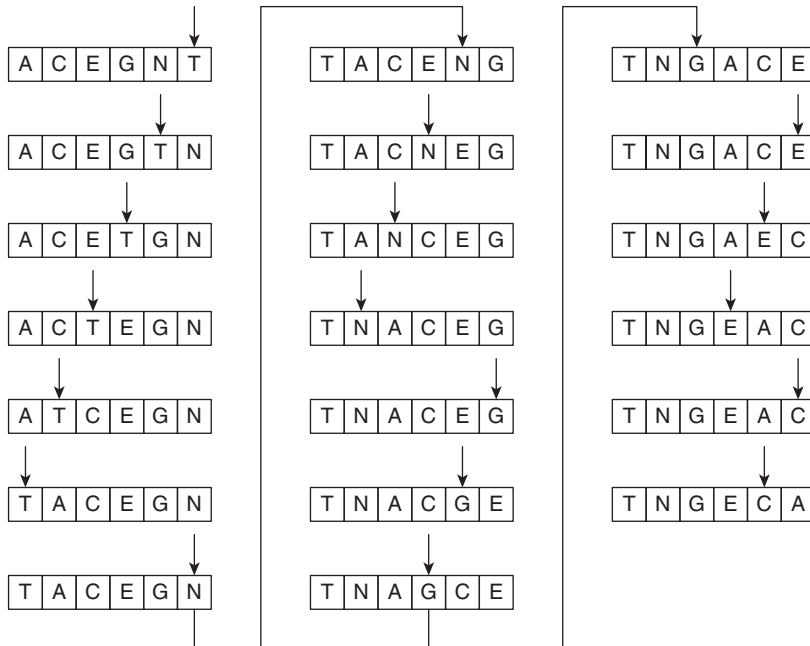
Figure 16.7 A few iterations of Johnson–Trotter algorithm

However, the algorithm does not stop here. After the process shown in Fig. 16.7 ends, the arrow starts moving from right to left. The process is repeated recursively thus making the total number of swaps equal to *n*! Now, one may wonder whether it is a brute force algorithm. As a matter of fact it is, but an organized one. Some of the iterations of the algorithm are shown in the following figure. The total number of iterations is 720, since the number of items in the array is 6.

Topological sorting is another example of decrease and conquer algorithm. The topic has already been discussed in Section 7.8. Thus, it can be said that any algorithm that decreases the input size by a constant number (generally 1) falls in this category.

16.5 DECREASE AND CONQUER: VARIABLE DECREASE

Chapter 17 deals with the concept of greatest common divisor and the algorithms required to find the GCD of two numbers. However, in order to explain the idea of variable decrease, the implementation of GCD has been taken as an example.

While calculating the greatest common divisor of a number using recursion, the argument passed in the 'if part' (refer to the code given below) is the modulus of the first number by the second. This technique helps us to calculate the GCD of two numbers effectively and efficiently. This procedure not only demonstrates the power of recursion but also depicts what is called the *technique of variable decrease*. In order to understand the concept, let us see the arguments that are generated in the various calls of the program which calculates GCD using recursion. To show the arguments at each call, a minor change has been made in the program. An additional print statement has been added in the 'if part' of the program. The output of the program for various inputs has been shown in Fig. 16.8. As stated earlier, here we conquer a problem by diminishing the arguments neither by a constant number, nor by a constant factor but by a variable number. This is the third type of decrease and conquer approach.

Figure 16.8 Calls of GCD using recursion

Algorithm 16.3   GCD using recursion

```
intgcd(int num1, int num2)
    {
    if((num1%num2)==0)
        {
        return num2;
        }
    else
        {
        return (gcd(num2, num1%num2));
        }
    }
void main()
    {
    int num1, num2, answer;
    clrscr();
    printf("\nEnter the first number\t:");
    scanf("%d",&num1);
    printf("\nEnter the second number\t:");
    scanf("%d", &num2);
    answer=gcd(num1, num2);
    printf("\nThe greatest common divisor of %d and %d is %d", num1, num2, answer);
    getch();
    }
```

16.6 CONCLUSION

The chapter explores the concept of decrease and conquer. There are three ways of implementing decrease and conquer, first is decrease by fixed size, second is decrease by a constant factor, and the third is decrease by a variable factor. Though, most of the algorithms discussed in this chapter have already been visited before, they have been revisited in order to put things in perspective. As per the definition, there is not much difference between decrease by constant factor and divide and conquer; therefore, the topics falling in that category have been covered in Chapter 9. It may also be stated here that though brute force is sometimes considered the simplest technique, it requires deep understanding and the ability to decode the problem. The decrease by constant number also requires the same level of understanding. The reader is advised to explore the web resources of the book to see the implementation of Johnson–Trotter algorithm and some more algorithms related to permutation generation.

Points to Remember

- Decrease and conquer is of three types.
- In decrease and conquer approach of the first type, the decrease in the input size is by a constant number.
- In decrease and conquer approach of the second type, the decrease in the input size is by a constant factor.
- In decrease and conquer approach of the third type, the decrease in the input size is by a different unrelated number.
- Here, it may be stated that a set having n elements has 2^n subsets and hence, the number of elements in its power set would be 2^n.

KEY TERMS

Permutation The arrangement of elements such that the order is important and repetition is not allowed is called permutations.

Power set It is the set of all the subsets of a set and is referred to as power set of a set.

EXERCISES

I. Multiple Choice Questions

1. Which one of the following is a method for generating permutations?
   - (a) Johnson–Trotter
   - (b) Johnson–Gamble
   - (c) Procter–Gamble
   - (d) Johnson–Procter

2. Which of the following is not a technique of decrease and conquer?
   - (a) Decrease by constant number
   - (b) Decrease by constant factor
   - (c) Variable decrease
   - (d) Decrease by harmonic factor

3. Which of the following is closest to decrease by constant number, when the number is 1?
   - (a) Brute force
   - (b) Divide and conquer
   - (c) Both
   - (d) None of the above

4. Which of the following is closest to decrease by constant factor?
   - (a) Brute force
   - (b) Divide and conquer
   - (c) Both
   - (d) None of the above

5. Which of the following does not employ decrease by constant number?
   - (a) Breadth first search
   - (b) Depth first search
   - (c) Topological sorting
   - (d) All of the above

6. Which of the following uses decrease by constant number?
   - (a) Merge sort
   - (b) Quick sort
   - (c) Topological sorting
   - (d) All of the above

7. Which of the following uses decrease by constant number?
   - (a) Selection
   - (b) Insertion sort
   - (c) Both
   - (d) None of the above
8. Which of the following uses variable decrease?
   - (a) Breadth first search
   - (b) Greatest common divisor
   - (c) Topological sorting
   - (d) All of the above
9. A fake coin problem can be solved using which of the following?
   - (a) Decrease by constant number
   - (b) Decrease by constant factor
   - (c) Both
   - (d) None of the above
10. Which of the following can be considered a part of decrease and conquer?
    - (a) Divide and conquer
    - (b) Backtracking
    - (c) Branch and bound
    - (d) None of the above

II. Review Questions

1. Explain the concept of decrease and conquer using insertion sort.
2. What are the different types of decrease and conquer?
3. How does decrease and conquer be used to multiply two numbers?
4. Find the GCD (greatest common divisor) of two numbers using decrease and conquer.
5. How would you generate all permutations of a given set of number using decrease and conquer?
6. Find a^n using decrease and conquer.

Answers to MCQs

1. (a)	3. (a)	5. (d)	7. (b)	9. (c)
2. (d)	4. (b)	6. (c)	8. (b)	10. (a)

CHAPTER 17

Number Theoretic Algorithms

OBJECTIVES

After studying this chapter, the reader will be able to

- Understand the techniques of finding the GCD of two numbers
- Know the application of Euclid theorem
- Use the extended Euclid theorem
- Learn the techniques of solving linear modular equations
- Explain the idea of cryptography and digital signatures

17.1 INTRODUCTION

The chapter deals with the concept of greatest common divisor, that is GCD. Before starting the discussion, let us understand the meaning of divisor. 'a' is a divisor of 'b' if 'a' divides 'b' that is $b|a$. For example, 4 divides 44444, therefore, 4 is a divisor of 44444. Two numbers may have common divisors. There can be situations in which there is more than one common divisor. In such cases, the greatest of these common divisors is referred to as the GCD of two numbers. This chapter discusses the algorithms used for finding the GCD of two numbers. The brute force approach of finding the GCD of two numbers would require the enlisting of all the possible factors of the two given numbers and then finding the common numbers from the two result sets. Though finding GCD of small numbers is not that difficult using the above approach, the GCD of two large numbers is a little difficult to find. In such cases, recursion comes to our rescue. Though recursion increases the space requirement, it reduces the time complexity to a great extent.

Finding GCD is an important task as it is also the basis of modular linear equations and Chinese remainder theorem discussed in Sections 17.5 and 17.6, respectively. The theorem helps to express the GCD of two numbers as their linear combination. This theorem also helps us in cryptography algorithms. Finally, the concept of cryptography and digital signatures has been introduced and the use of the above techniques in cryptography has been discussed. The concept of symmetric and asymmetric keys has also been discussed in the chapter. The chapter also discusses one of the most important algorithms used in cryptography, RSA. The chapter, though highly mathematical, forms the basis of the application of algorithms in the field such as network security. However,

the aim of this chapter is not to teach cryptography or network security. The goal is to use the concepts studied in this book so far, such as recursion and complexity considerations in the field of security. In fact, the concepts form the backbone of the school of cryptography which believes in the invincibility of primes.

The rest of the chapter has been organized as follows. Section 17.2 introduces the concept of GCD. The next section explains the use of Euclid theorem in finding the GCD of two numbers. Extended Euclid theorem has been discussed in Section 17.4. Sections 17.5 and 17.6 discuss the modular linear equations and the Chinese remainder theorem, respectively. Section 17.7 deals with cryptography. Section 17.8 examines the concept of digital signatures and the last section explores the RSA algorithm.

17.2 GCD OF TWO NUMBERS

A number x may be expressed as the product of powers of prime numbers, that is, $x = p_1^{i1} \times p_2^{i2} \times ... p_k^{ik}$. For example, 2134 may be expressed as $2 \times 11 \times 97$.

If two prime numbers are expressed as stated above, that is, two numbers, say, 'x' and 'y' are written as

$$x = p_1^{i1} \times p_2^{i2} \times ... p_k^{ik} \text{ and}$$

$$y = p_1^{j1} \times p_2^{j2} \times ... p_k^{jk}, \text{ then the number}$$

$$z = p_1^{l1} \times p_2^{l2} \times ... p_k^{lk}$$

where $l_m = \begin{cases} i_m, \text{if } i_m < j_m \\ j_m, \text{otherwise} \end{cases}$ is called the greatest common divisor of 'x' and 'y'.

The concept of GCD is simple. As stated earlier, since a number may have numerous factors, two numbers may also have common factors. The greatest of these common factors is referred to as the GCD.

Though the above method appears simple, it is computationally expensive. The C code given in the web resource of this book finds the GCD of two numbers using the above approach. It can be inferred from the code that the computational complexity of the process involved is $O(n^4)$.

Tip: The calculation of GCD by brute force is computationally expensive.

The following illustrations explain the above process.

Illustration 17.1 Find the GCD of 102685968 and 103733784.

Solution Since any number can be expressed as the powers of prime numbers, first of all the two numbers would be expressed as the powers of prime numbers.

$$102685968 = 2^4 \times 3^5 \times 7^4 \times 11^1$$

$$103733784 = 2^3 \times 3^7 \times 7^2 \times 11^2$$

In the next step, the smaller of the exponents of a common prime number is to be extracted.

- The powers of 2 in the two numbers are 4 and 3, out of which 3 is smaller.
- The powers of 3 in the two numbers are 5 and 7, out of which 5 is smaller.
- The powers of 7 in the two numbers are 4 and 2, out of which 2 is smaller.
- The powers of 11 in the two numbers are 1 and 2, out of which 1 is smaller.

So, the GCD of the two numbers is $2^3 \times 3^5 \times 7^2 \times 11^1 = 1047816$.

In the following illustration, the case wherein the two numbers have some powers of different primes are dealt with.

Illustration 17.2 Find the GCD of 487265625 and 66150.

Solution The given numbers can be expressed as the product of the following:

$$487265625 = 3^4 \times 5^7 \times 7^1 \times 11^1$$

$$66150 = 2^1 \times 3^3 \times 5^2 \times 7^2$$

The minimum of the two exponents are chosen to form the GCD. In this case, the GCD is the product of the following:

$$2^0 \times 3^3 \times 5^2 \times 7^1 \times 11^0$$

which makes 4725. Therefore, the GCD of 487265625 and 66150 is 4725. The process is depicted in Table 17.1.

Table 17.1 GCD of 487265625 and 66150

Prime numbers	Exponents in the first number	Exponents in the second number	Minimum exponent
2	0	1	0
3	4	3	3
5	7	2	2
7	1	2	1
11	1	0	0

17.3 EUCLID THEOREM

The above expensive method of calculating the GCD may be replaced by a cheaper method, which uses recursion. The method goes as follows:

$$GCD(a,b) = \begin{cases} GCD(b, a\%b), & \text{if } b \neq 0 \\ a, & \text{if } b = 0 \end{cases}$$

Tip: Euclid theorem has lesser complexity than the brute force method of calculating GCD.

Illustrations 17.3–17.6 explain the process.

Illustration 17.3 Find the GCD of 10214 and 2366.

Solution In the first iteration, the value of 'a' is 10214 and that of 'b' is 2366. In the next iteration, 'b' becomes 'a' and (a mod b) becomes 'b'. The process continues till b becomes 0. The process is depicted in Table 17.2.

Table 17.2   GCD of 10214 and 2366

Iteration number	a	b
1	10214	2366
2	2366	750
3	750	116
4	116	54
5	54	8
6	8	6
7	6	2
8	2	0

The value of 'a', when 'b' becomes 0 is the GCD of the two numbers. In this case, the GCD is 2.

Illustration 17.4 Find the GCD of 4562 and 2134.

Solution In the first iteration, the value of 'a' is 4562 and that of 'b' is 2134. In the next iteration, 'b' becomes 'a' and (a mod b) becomes 'b'. The process continues till b becomes 0. The process is depicted in Table 17.3.

Table 17.3   GCD of 4562 and 2134

Iteration number	a	b
1	4562	2134
2	2134	294
3	294	76
4	76	66
5	66	10
6	10	6
7	6	4
8	4	2
9	2	0

The value of 'a', when 'b' becomes 0 is the GCD of the two numbers. In this case, the GCD is 2.

Illustration 17.5 Find the GCD of 12124 and 4452.

Solution In the first iteration, the value of 'a' is 12124 and that of 'b' is 4452. In the next iteration, 'b' becomes 'a' and (a mod b) becomes 'b'. The process continues till, b becomes 0. The process is depicted in Table 17.4.

Table 17.4 GCD of 12124 and 4452

Iteration number	a	b
1	12124	4452
2	4452	3220
3	3220	1232
4	1232	756
5	756	476
6	476	280
7	280	196
8	196	84
9	84	28
10	28	0

The value of 'a', when 'b' becomes 0 is the GCD of the two numbers. In this case, the GCD is 28.

Illustration 17.6 Find the GCD of 21215 and 7995.

Solution In the first iteration, the value of 'a' is 21215 and that of 'b' is 7995. In the next iteration, 'b' becomes 'a' and (a mod b) becomes 'b'. The process continues till b becomes 0. The process is depicted in Table 17.5.

Table 17.5 GCD of 21215 and 7995

Iteration number	a	b
1	21215	7995
2	7995	5225
3	5225	2770
4	2770	2455
5	2455	315
6	315	250
7	250	65
8	65	55
9	55	10
10	10	5
11	5	0

The value of 'a' when 'b' becomes 0 is the GCD of the two numbers. In this case, the GCD is 5.

The above process of calculating the GCD of two numbers is called Euclid theorem. The process calculates the GCD of two numbers using recursion and is more efficient, in terms of implementation, as compared to the method discussed in the previous section.

17.4 EXTENDED EUCLID THEOREM

The GCD of two numbers can also be expressed as the linear combination of the two numbers in question. That is, $GCD(a,b) = x \times a + y \times b$.

The values of x and y can be found by extended Euclid theorem.

> **Tip:** Extended Euclid theorem helps to express the GCD as the linear combination of the two given numbers.

The theorem can be stated as follows:

Let the GCD of two numbers be g, then g can be written as $ax + by$ where the values of x and y can be found by the following procedure.

Algorithm 17.1 Euclid_GCD

Input: Two numbers a and b.
Output: The GCD of the two numbers

```
EE(a,b)
        {
           if( b=0)
                        {
                        g=a;
                        x=1;
                        y=0;
                        }
           else
                        {
                        g,x',y'=EE(b, a mod b);

                        g, x, y = g, y',x' - ⌊a/b⌋;

                        }
        }
```

The following illustrations exemplify the above procedure.

Illustration 17.7 Find the values of x and y of extended Euclid for the numbers 10214 and 2366.

Solution The GCD is calculated using the same process followed in Illustrations 17.3–17.6. In the last step, while calculating GCD, x becomes 1 and y becomes 0. After which the above algorithm is followed. The calculation of x and y is done in the bottom-up fashion. The process is depicted in Table 17.6.

Table 17.6 Values of x and y for 10214 and 2366

Iteration number	a	b	x	y
1	10214	2366	−306	1321
2	2366	750	97	−306
3	750	116	−15	97
4	116	54	7	−15
5	54	8	−1	7
6	8	6	1	−1
7	6	2	0	1
8	2	0	1	0

The values of x and y in this case are −306 and 1321.

Illustration 17.8 Find the values of x and y of extended Euclid for the numbers 4562 and 2134.

Solution The GCD is calculated using the same process followed in Illustrations 17.3–17.6. In the last step, while calculating GCD, x becomes 1 and y becomes 0. After which Algorithm 17.1 is followed. The calculation of x and y is done in the bottom-up fashion. The process is depicted in Table 17.7.

Table 17.7 Values of x and y for 4562 and 2134

Iteration number	a	b	x	y
1	4562	2134	421	−900
2	2134	294	−58	421
3	294	76	15	−58
4	76	66	−13	15
5	66	10	2	−13
6	10	6	−1	2
7	6	4	1	−1
8	4	2	0	1
9	2	0	1	0

The values of x and y in this case are 421 and −900.

Illustration 17.9 Find the values of x and y of extended Euclid for the numbers 12124 and 4452.

Solution The GCD is calculated using the same process followed in Illustrations 17.3–17.6. In the last step, while calculating GCD, x becomes 1 and y becomes 0. After which the Algorithm 17.1 is followed. The calculation of x and y is done in the bottom-up fashion. The process is depicted in Table 17.8.

Table 17.8 Values of x and y for 12124 and 4452

Iteration number	a	b	x	y
1	12124	4452	−47	128
2	4452	3220	34	−47
3	3220	1232	−13	34
4	1232	756	8	−13
5	756	476	−5	8
6	476	280	3	−5
7	280	196	−2	3
8	196	84	1	−2
9	84	28	0	1
10	28	0	1	0

The values of x and y in this case are −47 and 128, respectively.

Illustration 17.10 Find the values of x and y of extended Euclid for the numbers 21215 and 7995.

Solution The GCD is calculated using the same process followed in Illustrations 17.3–17.6. In the last step, while calculating GCD, x becomes 1 and y becomes 0. After which the Algorithm 17.1 is followed. It may be stated here, that the calculation of x and y is done in the bottom-up fashion. The process is depicted in Table 17.9.

Table 17.9 Values of x and y for 21215 and 7995

Iteration number	a	b	x	y
1	21215	7995	736	−1953
2	7995	5225	−481	736
3	5225	2770	255	−481
4	2770	2455	−226	255
5	2455	315	29	−226
6	315	250	−23	29
7	250	65	6	−23
8	65	55	−5	6
9	55	10	1	−5
10	10	5	0	1
11	5	0	1	0

The values of x and y in this case are 736 and −1953.

17.5 MODULAR LINEAR EQUATIONS

A modular linear equation is an equation of degree one of the form $ax \equiv b \bmod n$, which means that $(ax - b)$ is divisible by n. The following discussion would focus on the techniques of solving the modular linear equations of the type $ax \equiv b \bmod n$.

The equation requires that $ax - b$ should be divisible by n. Here, the value of a and n should be greater than 0. The solution of the above helps in RSA algorithm, described in Section 17.9. Though an important point is that the above equation may not even have a solution. There may be cases where the number of solutions is more than one.

Since $ax \equiv b \bmod n$, $ax - b$ is divisible by n, that is, $(ax - b) = kn$ or $ax - kn = b$.

Since a and k are constants, one can say that b is the linear combination of a and n. It was stated that even GCD (a, n) is a linear combination of a and n (see Section 17.5). Therefore, it may be concluded that b is a multiple of the GCD(a, n). Though it can be proved that the proportionality constant in this case is 1.

> If $ax \equiv b \bmod n$ and b is divisible by the GCD(a, n) then the given equation has solution.

The technique of solving the modular linear equation is attributed to Cormen and is as follows.

Algorithm 17.2 Extended Euclid theorem

Input: A and b, the two given numbers.
Output: The GCD of the two numbers and the values of x and y which would help us to express the GCD as the linear combination of the given numbers.

```
{
Use Extended Euler's formula to find the value of (g, x, y) for the pair (a, n);
If (g|b)
        {
```
$$x_0 = x_1 \left(\frac{b}{g} \right) \bmod n;$$
```
                i=1;
        while(i< g-1)
                {
```
Print: $x_0 = i \left(\dfrac{n}{g} \right) \bmod n$
```
                i++;
                }
else
        {
        print: "No solution";
        }
}
```

17.6 CHINESE REMAINDER THEOREM

Sun Tzu, a Chinese mathematician, published a book called *The Mathematical Classic of Sunzi*. The third chapter of the book contains the Chinese Remainder Theorem (CRT). This is perhaps the earliest known reference of the CRT. However, in some Indian scriptures, such as *Brahma-Sphuta-Siddhanta*, questions related to CRT have been found. The theorem is particularly helpful in finding out the minimum number which when divided by the given set of numbers $\left(\text{say } \{n_1, n_2, ..., n_n\}\right)$ gives remainder $\left(\text{say } \{a_1, a_2, ..., a_n\}\right)$. The formal statement of the theorem is as follows.

If n_1, n_2, …, n_k are pair-wise relatively prime positive integers and if a_1, a_2, …, a_k are any integers, then the simultaneous congruences $x \equiv a_1 (\text{mod } n_1)$, $x \equiv a_2 (\text{mod } n_2)$,..., $x \equiv a_k (\text{mod } n_k)$ have a solution, and the solution is unique modulo n, where $n = n_1 n_2 \dots n_k$. The modulo is given by $x \equiv a_1 \times N_1 \times y_1 + a_2 \times N_2 \times y_2 + \dots + a_k \times N_k \times y_k$. The value of $N_i = (n_1 \times n_2 \times ... n_k)/n_i$ and the value of y_i is obtained by the equation $y_i \equiv N_i^{-1} \text{mod } n_i$.

The explanation of the above theorem is as follows (Cormen):

Since there is no common factor between N_i and n_i the $\text{GCD}(N_i, n_i) = 1 \forall i, 1 \le i \le r$. Therefore, $y_i \equiv N_i^{-1} \text{mod } n_i$ exist.

The value of y_i may be found by the extended Euler theorem:

$$N_i \times y_i = 1 \text{ mod } n_i$$

This implies that $a_i \times N_i \times y_i = a_i \text{ mod } n_i$

On the other hand, $a_i \times N_i \times y_i = 0 \text{ mod } n_j$, if $j \ne i$.

Therefore, $x \equiv a_i \text{ mod } n_i$, for $1 \le i \le r$.

If x_0 and x_1 were the solutions, then $x_0 - x_1 \equiv 0 \text{ mod } (n_i)$, for all i, so that $x_0 - x_1 \equiv 0 \text{ mod } (M_i)$.

17.6.1 Applications

The theorem has many applications. Some of them are as follows.
- The theorem is used to construct a sequence of Golden numbers for RSA algorithm.
- According to the book *Microwave Radar: Imaging and Advanced Concepts*, the Chinese remainder theorem is used to solve many problems in radar systems also.
- Even interpolation technique such as 'Lagrange's interpolation' is considered as a special case of Chinese remainder theorem.
- The theorem can be used to find the numbers whose square ends in them.
- For those who are interested in mathematics, the CRT can be used to prove Gauss' Quadratic Reciprocity.

In order to understand the theorem, let us consider the following example:

A student while playing with a file of her father made the following observations.
- When she clubbed the documents in groups of three, one document remained.
- When she clubbed the documents in groups of two, one document remained.

- When she clubbed the documents in groups of five, one document remained.
- When she clubbed the documents in groups of seven, one document remained.
- When she clubbed the documents in groups of eleven, no document remained.

She designed the equations corresponding to the above situations. In the following equations, the number of documents in the given file is assumed to be x. The equations obtained were as follows:

$$x \equiv 1 \bmod 3$$

$$x \equiv 1 \bmod 2$$

$$x \equiv 1 \bmod 5$$

$$x \equiv 1 \bmod 7$$

$$x \equiv 0 \bmod 11$$

The value of N, in this case is $2 \times 3 \times 5 \times 7 \times 11 = 2310$

The value of N_i's were as follows:

$$N_1 = \frac{N}{n_1} = 1155$$

$$N_2 = \frac{N}{n_2} = 770$$

$$N_3 = \frac{N}{n_3} = 462$$

$$N_4 = \frac{N}{n_4} = 330$$

$$N_5 = \frac{N}{n_5} = 210$$

This was followed by the calculation of y_i's,

$$y_1 \equiv 1155^{-1} \bmod 2, \text{ that is, } 1155 \times y_1 = 1 \bmod 2, \ y_1 \text{ is therefore } 1$$

$$y_2 \equiv 770^{-1} \bmod 3, \text{ that is, } 770 \times y_2 = 1 \bmod 3, \ y_2 \text{ is therefore } 2$$

$$y_3 \equiv 462^{-1} \bmod 5, \text{ that is, } 462 \times y_3 = 1 \bmod 5, \ y_3 \text{ is therefore } 3$$

$$y_4 \equiv 330^{-1} \bmod 7, \text{ that is, } 330 \times y_4 = 1 \bmod 7, \ y_4 \text{ is therefore } 1$$

$$y_5 \equiv 210^{-1} \bmod 11, \text{ that is, } 210 \times y_5 = 0 \bmod 11, \ y_5 \text{ is therefore } 1$$

The final step required the calculation of *x* which is

$$x \equiv a_1 \times N_1 \times y_1 + a_2 \times N_2 \times y_2 + \cdots + a_k \times N_k \times y_k$$

i.e.,
$$x \equiv 1 \times 1155 \times 1 + 1 \times 770 \times 2 + 1 \times 462 \times 3 + 1 \times 330 \times 1 = 4411$$

17.7 CRYPTOGRAPHY

Have you ever thought which is the easiest means of sharing information? Perhaps sending e-mails! We send e-mails to carry out our most important tasks. We send it to our friends, employers, employees, family, and foes. What if the contents of these mails are made public or are changed in the channel? Same is the case with the data stored in our computer or laptop. We would like it to be as safe as possible, both in terms of unauthenticated access and that it should not be altered.

The discipline of information security takes care of the above. The two most important parts of this discipline are cryptography and cryptanalysis. Let us begin our discussion with the concept of communication and then understand what the two terms are.

In digital communication, a message is transmitted from a sender to a receiver. The abstraction of the medium, via which the message is transmitted, is referred to as a channel. The process of communication can be shown as in Fig. 17.1.

| Sender | Channel | Receiver |

| Hacker trying to read the message |
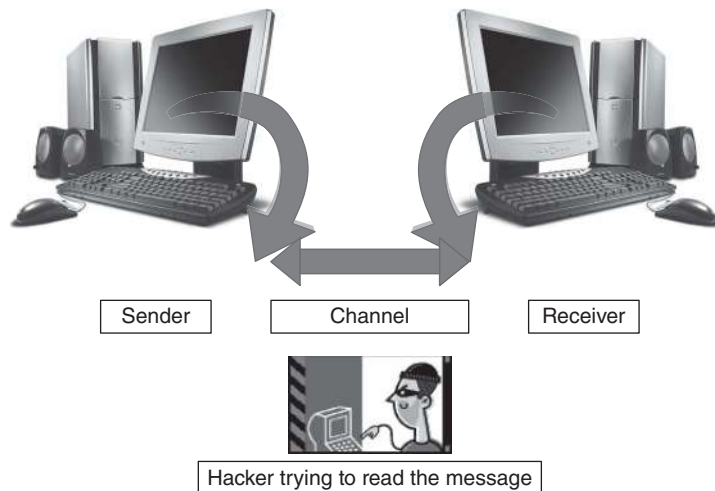
Figure 17.1 Communication process

However, while sending the message, there is always a possibility of someone listening to the communication. This is called eavesdropping. In order to prevent this, the original text is converted into some other form. This can be done in two ways: the symmetric key cryptography and the asymmetric key cryptography.

17.7.1 Symmetric Key Cryptography

As stated earlier, the encryption of a message into ciphertext, is generally done with the help of a key. For example, the letter 'A', which has ASCII value 65 (binary equivalent of 65 is 01000001) is to be transmitted. In order to make sure that if the data is read in the channel, then no one should be able to decode that the binary equivalent of 'A' is XOR-ed with the key produced by some key-generating algorithm. The key generated is, say, 10101011. The XOR-ing of 10101011 and 01000001 is 11101010. The number '01000001', that is, the given text, is called the plaintext. The data obtained by XOR-ing the two numbers is the ciphertext. In such cases, the following equations hold:

$$P \oplus K = C$$

P is the plaintext, K is the key, and C is the ciphertext. At the receiver's end, the plaintext is obtained by XOR-ing the key, K with the ciphertext, C, that is $C \oplus K = P$. The process of converting the plaintext to ciphertext is called cryptography. The technique(s) of finding the key so that the plaintext can be obtained from the ciphertext is called 'cryptanalysis'.

17.7.2 Asymmetric Key Cryptography

In this type of cryptography, the key(s) are of two types: public key and the private key. The public key is available to all, whereas the private key is secret. The private key is known only to the person. It may be stated here that the two keys are, generally, mathematically related. These keys are used to encrypt a message, that is, to convert a message into ciphertext.

The idea of asymmetric key cryptography is to encrypt a text with a different key and to decrypt it with a different key.

The above discussion can be summarized as follows.

If a person 'A' sends a message to 'B', then he uses the public key of 'B'. The encrypted message can now only be decrypted using the private key of 'B', which is known to 'B'. If anyone tries to decrypt the message, he will not be able to do so owing to the fact that the secret key is similar to an inverse function of the public key. In Fig. 17.2, P_B represents the public key of B, S_B represents the secret key of B. Document D when
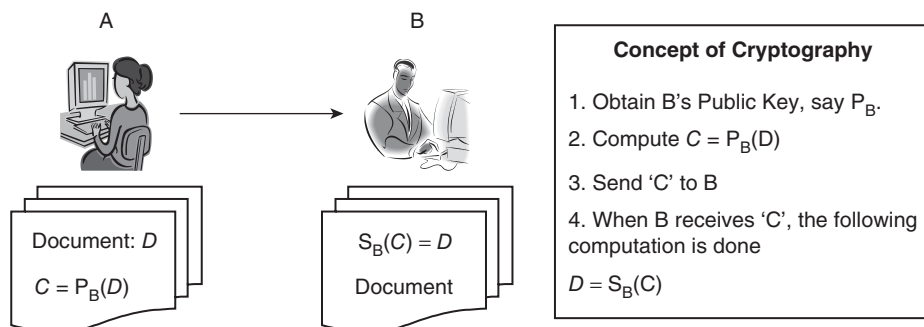
Figure 17.2 Cryptography

encrypted with the public key of B becomes C. Now, C can be converted back to D only using the secret key of B. An eavesdropper would not be able to decode C as she would not be having S_B.

17.8 DIGITAL SIGNATURES

The second goal, stated at the beginning of this section, is achieved by digital signatures. Digital signatures are used to verify whether the text was sent by a particular sender. It helps demonstrate the authenticity of a message. The concept is similar to the analog signature. Generally, a person is asked to give his signatures in a bank so that he can be verified. This serves two purposes. It not only helps in authentication but also the person in question cannot deny that the cheque was issued by him. The digital signatures serve the same purpose. Digital signatures help in both authentication and non-repudiation. One more task that can be accomplished using the technique is to demonstrate that the message has not been violated.

The concept, generally, requires two keys: a public key and a private key. The private key is used to create a digital signature and the public key is used to verify the digital signature.

For example, when B sends a message M to A, then he computes $S_B(M)$ and sends $(S_B(M), M)$ it to A. A, on receiving the message, uses the public key of B to compute $P_B(S_B(M))$; if the result obtained is same as that of M, then the authenticity of the message can be verified. The process is depicted in Fig. 17.3.

A   B

Concept of Digital Signature

1. B computes $S_B(M)$, S_B is the secret key of B.

2. B sends $(S_B(M), M)$ to A

if, $M = P_B$ $(S_B(M))$, then the message is end by A.

$S_B(M), M$

Send to A.

3. A computes $P_B(S_B(M))$.

4. If the result obtained in step 3 is same as that of M, the authenticity of M is estabilished.
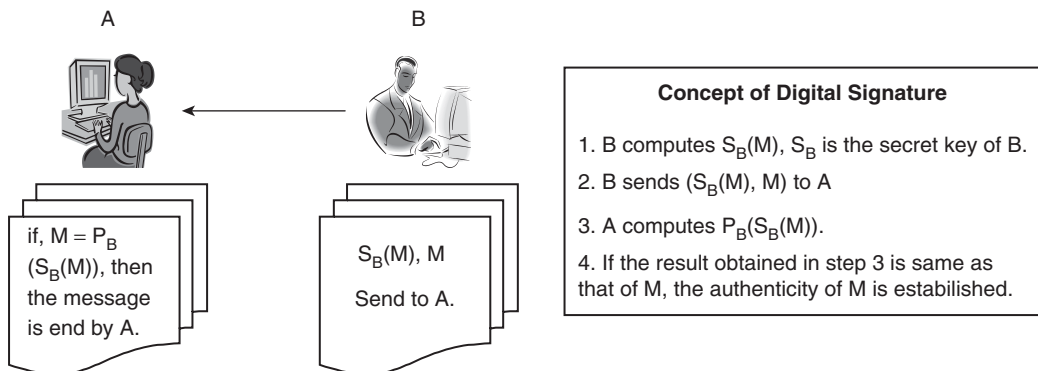
Figure 17.3 Digital signatures

Definition Digital signatures is a type of electronic signature that has three components. The first is the algorithm that generates key. Two types of keys are generated by this algorithm: the public key and the private key. The second component is the signing algorithm that takes the message and the private key as its input and the third component is the signature verifying the algorithm that uses the public key to verify the authenticity of the message.

The advent of digital signature helped the computer scientists to prevent forgery and tampering. Digital signatures also helped in handling the issues such as security of data and e-transactions, which were a point of contention during the development of Internet. Digital signatures have legal validity in many countries including United States. The concept can be summarized as follows.

17.9 RSA ALGORITHM

The need of secure data transmission was realized as early the late 1970s, and hence one of the first public key cryptosystems, RSA was introduced. The encryption, as explained earlier in the section is public, whereas the decryption can only be done using the secret key. The algorithm is based on the practical difficulty of factoring the product of large prime numbers. The algorithm was given by Ron Rivest, Adi Shamir, and Leonard Adleman and is hence called RSA. The public key is produced by two large prime numbers. The algorithm is as follows:

1. Choose two distinct prime numbers, say p and q.
2. Compute $n = p \times q$
3. Compute $\varphi(n) = (p-1)(q-1)$
4. Choose e such that e and $\varphi(n)$ are co-primes.
   e becomes the public key.
5. Compute $d = e^{-1} \mod (\varphi(n))$
   d becomes the secret key.

The process of encryption and decryption can be explained as follows.

If the message is m, public key is e, then the encrypted text is computed using the equation:

$$c \equiv m^e \mod n$$

The decrypted text is given by the following equation:

$$m \equiv c^d \mod n$$

The following steps explains the above process:

Step 1 $p = 163$ and $q = 167$

Step 2 $n = p \times q = 27221$

Step 3 $\varphi(n) = (p-1)(q-1) = 26{,}892$

Step 4 A number co-prime to the value obtained in the previous step is $e = 7$.

Step 5 Now $d \times e \equiv 1 \mod (26{,}892)$, the calculation of the value of d has been left as an exercise for the readers.

17.10 CONCLUSION

The concepts such as GCD, Euclid theorem, extended Euclid theorem, and linear modular equations discussed in this chapter form the foundation of 'applied algorithms'.

These concepts find the application not only in algorithms but also in subjects such as network security, etc. For example, while implementing RSA, it is important to generate prime numbers, solve modular equations, etc. These concepts also form the basis of mathematical approaches to more difficult problems. The reader is requested to go through the web resources of this book that contains programs pertaining to the above concepts, an introduction to the powers of prime numbers, and some extra questions.

Points to Remember

- A number may have numerous factors, two numbers may also have common factors. The greatest of these common factors is referred to as the GCD.
- The two most important parts of information security are cryptography and cryptanalysis.
- While sending the message, there is always a possibility of someone listening to the communication. This is called eavesdropping.
- The idea of asymmetric key cryptography is to encrypt a text with a different key and to decrypt it with a different key.
- Digital signatures help in both authentication and non-repudiation.
- The private key is used to create a digital signature and the public key is used to verify the digital signature.

KEY TERMS

Chinese remainder theorem If $n_1, n_2, \ldots, n_k$ are pair-wise relatively prime positive integers, and if $a_1, a_2, \ldots, a_k$ are any integers, then the simultaneous congruences $x \equiv a_1 \pmod{n_1}$, $x \equiv a_2 \pmod{n_2}$, $\ldots$, $x \equiv a_k \pmod{n_k}$ have a solution, and the solution is unique modulo n, where $n = n_1, n_2, \ldots, n_k$. The modulo is given by $x \equiv a_1 \times N_1 \times y_1 + a_2 \times N_2 \times y_2 + \cdots + a_k \times N_k \times y_k$. The value of $N_i = (n_1 \times n_2 \times \ldots n_k)/n_i$ and the value of y_i is obtained by the equation $y_i \equiv N_i^{-1} \bmod n_i$.

Cryptography Conversion of plaintext into ciphertext is referred to as cryptography.

RSA It was one of the first public key cryptosystems. In cryptography using RSA, the encryption is public, whereas the decryption can only be done using the secret key.

EXERCISES

I. Multiple Choice Questions

1. Which of the following can be used to find the GCD of two numbers?
   - (a) Euclid theorem
   - (b) Fermat's theorem
   - (c) RSA
   - (d) None of the above

2. Which of the following is a technique of cryptography?
   (a) RSA
   (b) BSA
   (c) DSA
   (d) NSA

3. The GCD of two numbers can be expressed as a linear combination of the two numbers. Which of the following helps us to find the corresponding constants?
   (a) Euler's theorem
   (b) Extended Euler
   (c) Fermat's theorem
   (d) None of the above

4. How many prime numbers are there between 1 and 10^{20}?
   (a) $\approx 5 \times 10^{18}$
   (b) $\approx 23 \times 10^{15}$
   (c) $\approx 24 \times 10^{15}$
   (d) $\approx 25 \times 10^{16}$

5. Which of the following is incorrect as regards prime number P?
   (a) $GCD(p, x_i) = 1 \forall x_i \in N$
   (b) $c_1 p + c_2 x_i = 1$
   (c) x_i does not divide $P \forall a_i \in N$
   (d) All of the above

6. Which of the following is used to find a number which leaves a given set of remainders when divided by a given set of numbers?
   (a) Chinese remainder theorem
   (b) Euclid theorem
   (c) Both
   (d) None of the above

7. Who introduced CRT?
   (a) Sun Tzu
   (b) Xing Sang
   (c) Sun Leo
   (d) En Tzu

8. How many solutions can following modular linear $ax \equiv b \pmod{n}$, where $(a, n)/b$ equation have?
   (a) $b - 1$
   (b) $b - 2$
   (c) b
   (d) 0

9. Which of the following best describes cryptography?
   (a) The conversion of plaintext into ciphertext
   (b) The conversion of ciphertext to plaintext
   (c) Both
   (d) None of the above

10. For which of the following digital signatures are used?
    (a) Measurement
    (b) Optimization
    (c) Authentication
    (d) Virtualization

II. Review Questions

1. Define cryptography.
2. Explain the concept of digital signature.
3. Explain and exemplify RSA.
4. Prove Euclid formula.
5. Prove extended Euclid formula.
6. State and prove CRT.
7. Write an algorithm to find 10 prime numbers greater than 10^7.

8. Write a program in C which implements RSA.
9. Write a program to solve modular equation.

III. Numerical Problems

1. Solve the following if possible:
    (a) $3 \times \equiv 5 \bmod (27)$
    (b) $2 \times \equiv 3 \bmod (5)$
    (c) $9 \times \equiv 2 \bmod (10)$
    (d) $7 \times \equiv 1 \bmod (3)$
    (e) $3 \times \equiv 2 \bmod (21)$

2. Find the GCD of the following:
    (a) 2134, 1124
    (b) 1324, 4321
    (c) 11728, 82187
    (d) 11781, 1118
    (e) 8181, 1818
    (f) 2323, 23
    (g) 11111, 11
    (h) 31312, 213
    (i) 117217, 17
    (j) 123456, 12

3. For all of the above values given in Problem 2, find the value of x and y, so that $ax + by = \text{GCD}(a,b)$ where a and b are given numbers.

Answers to MCQs

1. (a)	3. (b)	5. (d)	7. (a)	9. (a)
2. (a)	4. (a)	6. (a)	8. (d)	10. (c)

CHAPTER 18

String Matching

OBJECTIVES

After studying this chapter, the reader will be able to

- Understand the importance of string matching
- Learn and implement naïve string matching
- Explain rabin–karp algorithm
- Define deterministic finite acceptor and its use in string matching
- Understand Knuth–Morris–Pratt automata
- Understand the concept of tries and suffix trees

18.1 INTRODUCTION

String matching, generally, finds a smaller string and its position within a larger string or all occurrences of a pattern in a given document. The formal definition of string matching is given in Section 18.2. String matching finds its applications in most important tasks such as text editors, intrusion detection systems, DNA matching, etc. The implementation of string matching also uses the longest common subsequence algorithm discussed in Chapter 11. The topic is important not only for a student pursuing computer engineering but also for a student pursuing biochemistry or biotechnology. The reason for matching the strings may not always be to correct errors in a text editor. The purpose may be finding noise in communication, intrusion detection in a system using trees, or even in bioinformatics.

The first part of the chapter discusses the most basic method of string matching that is naïve string matching, followed by a more efficient method called Rabin–Karp method, which reduces the number of comparisons. The chapter then introduces deterministic finite acceptors and explains the difference between a deterministic finite acceptor and a non-deterministic finite acceptor. These accept regular languages. This is followed by the most important method of this chapter: (Knuth–Morris–Pratt) KMP algorithm. The chapter also introduces tries and explains the algorithm for the creation of a trie. Section 18.8 discusses the concept of suffix trees, which are used in directories.

18.2 STRING MATCHING—MEANING AND APPLICATIONS

Formally, string matching may be defined as follows.

> **String Matching** Given a text T[0…(n − 1)] and a pattern P[0…(m − 1)], it is required to find the beginning index of all the occurrences of P in T.

For example, if the text T is 'harsharsh' and the pattern P is 'arsh', then the pattern occurs at the first position in T ('h*arsh*arsh') and at the fifth position ('harsh*arsh*'). So the answer to the string-matching problem in this case is 1 and 5.

The starting index in the text that follows is taken as 0. Had the starting index been '1', the answer would have been 2 and 6. The important point is that the pattern occurs in the given text, two times and at the indices guided by the starting index.

18.2.1 Applications

It is due to string matching that we easily find strings while working in MS Word or in a web page. Matching of strings is also important for a web crawler as keyword searching also requires the algorithms described in this chapter. Some of the applications of string matching are as follows:

- The string matching algorithms are used in text editors. They help us in finding occurrences of a particular pattern in a text. One of the most common algorithms that is used to find the difference between two files is the longest common subsequence as discussed in Chapter 11.
- These algorithms are used in computational biology, in DNA matching (largest match of the shortest common superstring), in finding close mutation, and so on.
- The algorithms are also used in intrusion detection in a network and even in retrieving musical pattern from a multimedia database. An intruder tries to get into a network, generally, by trying out some common combinations which would let him in. It turns out that this attempt has some pattern attached to it. The intrusion detection is found by matching patterns. These systems take help of the pattern-matching algorithms discussed in the following sections.

18.2.2 Algorithms and Data Structures

The following discussion uses many important algorithms for finding a particular pattern in a given text. There are various algorithms that are used in string matching; those that have been explained in the chapter are listed in Table 18.1:

Table 18.1 String-matching algorithms

Name of the algorithm	Complexity
Naïve string matching	$O(m(n − m + 1))$
Rabin–Karp	$O(m(n − m))$ in the average case and $O(n + m)$ in the best case
Finite automata	$O(n)$
Knuth–Morris–Pratt	$O(n)$

Some of the most important data structures used in string matching are as follows.
- Suffix tree
- Tries
- Deterministic finite automata

The above data structures have been explained and exemplified in the following sections.

18.3 NAÏVE STRING MATCHING ALGORITHM

This technique looks for each occurrence of the string $P[0\ldots(m-1)]$ in $T[0\ldots(n-1)]$, by comparing all possible substrings of the original string. The task can be accomplished by running a loop from 0 to $(n-m-1)$ times. The counter of this loop would indicate the position of the first character in T. m characters after this index would be compared with P; if a match is found then the value of the counter of the outer loop would be printed (denoted by 'i' in the algorithm). As is evident from the algorithm that follows, the inner loop must run from 0 to $(m-1)$. Algorithm 18.1 represents the course of action.

Algorithm 18.1   Naïve string matching

Input: T[0..(n–1)], P[0…(m–1)]

Naïve_string_Matching (T, P) returns position of P in T, if the string is found otherwise returns –1.

```
{
flag1=0;
for(i=0; i<(n-m-1); i++)
        {
        flag=0;
        for(j=0;j<m;j++)
                {
                {if(T[j]!=P[j])
                        {
                        flag=1;
                        }
                }
        if(flag==0)
                {
                return j;
                }
        }
return -1;
}
```

Note that "return –1" is executed only if the procedure does not return any time between.

Problem with the above algorithm The algorithm returns as soon as it finds the first occurrence of the string. However, there can be more than one occurrence of P in T. The following modification helps to tackle this problem.

Naïve_string_Matching (T, P) prints position of P in T, if the string is found otherwise prints "Not Found".

```
        {
        flag1=0;
        for(i=0; i<(n-m-1); i++)
                {
                flag=0;
                for(j=0;j<m;j++)
                        {
                        {if(T[j]!=P[j])
                                {
                                flag=1;
                                }
                        }
                if(flag==0)
                        {
                        Print i;
                        flag1=1;
                        }
                }
if(flag1==0)
        {
        print "Not Found";
        }
        }
```

Complexity: The complexity of the above algorithm can be found easily. The outer loop runs $(n - m - 1)$ times and the inner m times. The complexity, therefore, becomes $O((n-m-1)(m))$. If n is constant, then the complexity becomes $O(m^2)$, otherwise $O(mn)$.

Here, the above method is not the most efficient way of matching strings. The following sections explore better ways of accomplishing the above task. The reader can find the C code of the above algorithm in the web resources of this book.

18.4 RABIN–KARP ALGORITHM

The string matching procedure described in the earlier section requires $m(n - m - 1)$ comparisons, where m and n denote the length of P and T, respectively. The number of comparisons can be reduced by a simple modification to the above algorithm. The substrings having the same length as that of P and give the same modulus with a number, say q, and can be isolated from T.

These strings can then be compared with P. If the strings match, then the initial index of the substring in T can be printed. For example, if the substring is '5623', q is 13, and T is '789562378563267'. '5623' mod 13 is 7. The first step would be to find all strings of length m from T. This is followed by taking mod 13 of all the substrings. In the above case, the various substrings and their mod 13 are depicted in Table 18.2.

Table 18.2 Substrings of length m from T, and their mod 13

Substrings of length m	Substring % 13
7895	4
8956	12
9562	7
5623	7
6237	10
2378	12
3785	2
7856	4
8563	9
5632	3
6326	8
3267	4

Contenders

The strings with the value 7 are the contenders for the correct match. In this case, there are two such strings. These strings are then compared with P. The second string does not match P and hence would be dubbed as a spurious match. The matching has been depicted in Fig. 18.1. Algorithm 18.2 represents the course of action.

Tip: Rabin–Karp is more efficient than naïve string matching.

mismatch |9|5|6|2| |5|6|2|3| Match

|5|6|2|3| |5|6|2|3|
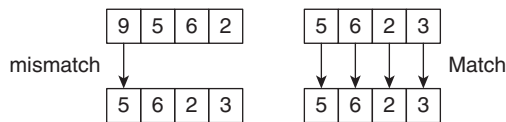
Figure 18.1 String matching

Algorithm 18.2 Rabin–Karp algorithm

Input: T[0…(n−1)], P[0…(m−1)]
Output: The starting index of the matching substring, if string is found else −1.

Rabin–Karp Algorithm(T[0…(n–1)], P[0..(m–1)]

```
modT := mod(T[0..(m-1)], q); modP := mod(s[0..(m-1)], q)

    for i from 1 to n-m+1
        if modP = modT
            if P[i..i+m-1] = P
            return i
        modP := mod(s[i+1..i+m], q)
    return not found
```

Explanation: The modT variable holds the value of mod of the substring of T (equal in length to P) and modP stores the value of mod of P with q. If these are equal, the characters are compared, otherwise the control moves one step forward.

Complexity: The contentious point is to find the value of the substring in order to calculate its mod. This, if done in the conventional way, would lead to quadratic complexity. In order to make the things efficient, recursion can be used. The complexity which becomes $O(m(n - m + 1))$ in the worst case. It may be noted though that the number of comparisons are reduced as compared to naïve string matching.

18.5 DETERMINISTIC FINITE AUTOMATA

This section briefly discusses the concept of deterministic finite automata (DFA) and non-deterministic finite automata (NFA). Moreover, the knowledge of DFA is also necessary to understand the first phase of the compiler, which is lexical analysis. Both DFA and NFA accept regular language. The first phase of compiler, for example, forms tokens. This is done with the help of a regular expression. This regular expression is then converted into an NFA. The NFA is converted to the corresponding DFA and finally the DFA is minimized. The formal definition of a DFA is as follows.

A deterministic finite acceptor is a five-tuple calculus having the following components:

- A finite, non-empty set of states, denoted by Q
- A finite, non-empty set of input symbols, denoted by Σ
- An initial state, denoted by q_0. The state $q_0 \in Q$
- A finite, non-empty set of final states, denoted by F, $F \subseteq Q$
- A transition function, generally denoted by δ, which maps $Q \times \Sigma \to Q$

For example, the DFA depicted in Fig. 18.2 can be described as follows.

The initial state of the automata, depicted in Fig. 18.2 is q_0. The final state is $\{q_2\}$. The set of states is the set $= \{q_0, q_1, q_2, q_3\}$. The set of states is given by $\Sigma = \{a, b\}$. The transition is given by
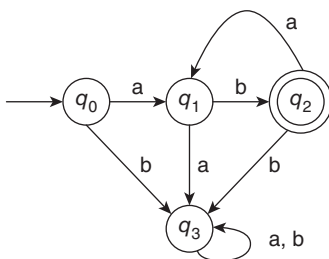
Figure 18.2 An automata that accepts (ab)$^+$

Table 18.3 Transition table of automata depicted in Fig. 18.2

State	a	b
q_0	q_1	q_3
q_1	q_3	q_2
q_2	q_1	q_3
q_3	q_3	q_3

$$\delta(q_0, a) = q_1$$

$$\delta(q_0, b) = q_3$$

$$\delta(q_1, a) = q_3$$

$$\delta(q_1, b) = q_2$$

$$\delta(q_2, a) = q_1$$

$$\delta(q_2, b) = q_3$$

$$\delta(q_3, a) = q_3$$

$$\delta(q_3, b) = q_3$$

The transition functions can also be depicted in Table 18.3.

A string is said to be accepted by an automata if by giving an input it reaches the final state. In the above case, the strings ab, abab, ababab, and so on (ab, any number of times) are accepted by the automata, whereas any other input leads us to the state q_3, referred to as *a trap state*. A trap state is one, wherein giving any input does not take us forward. Moreover, a trap state is always a non-final state.

In the above discussion, the output of a given automata has been discussed. However, in many situations, it is required to design an automata that accepts a given string. The concept can be understood by the illustrations that follow.

Illustration 18.1 Design an automata that accepts a string over {0, 1}, in which the first symbol is 1 and the third symbol is 0.

Solution The required automata is depicted in Fig. 18.3.
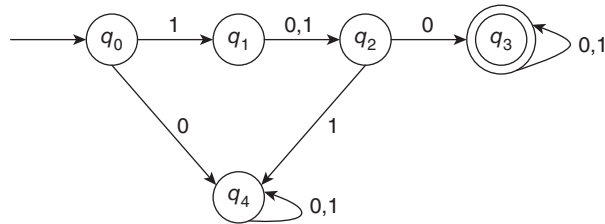
Figure 18.3 An automata that accepts a string having first symbol 1 and the third symbol 0

If the first symbol is not 1 then the transition takes us to the trap state. In the same way, if the third symbol is not 0 then the transition takes into the trap state. The second symbol can either be 0 or 1, since nothing is given regarding the second input.

The automata can be defined as follows.
- Initial state q_0
- Final state $\{q_3\}$
- Set of states $\{q_0, q_1, q_2, q_3, q_4\}$
- $\Sigma = \{0, 1\}$
- Transition diagram: Depicted in Fig. 18.3

Illustration 18.2 Design an automata that accepts a string over {0, 1}, in which the number of 1's are multiples of 4.

Solution The required automata is depicted in Fig. 18.4.
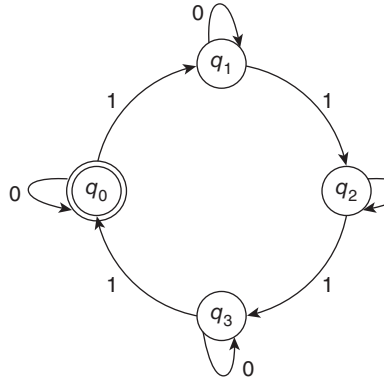
Figure 18.4 An automata that accepts a
string having number of 1's multiple of 4

The initial state of this automata is also the final state. If the number of 1's are 4 or 8 or 12 etc., then the automata reaches the final state. Since nothing is given regarding 0 is, any state can have any number of 0's.

The automata can be defined as follows:

- Initial state q_0
- Final state $\{q_0\}$
- Set of states $\{q_0, q_1, q_2, q_3\}$
- $\Sigma = \{0, 1\}$
- Transition diagram: Depicted in Fig. 18.4

18.5.1 Non-deterministic Finite Automata

A non-deterministic finite acceptor is a five-tuple calculus having the following components:

- A finite, non-empty, set of states, denoted by Q
- A finite, non-empty, set of input symbols, denoted by Σ
- An initial state, denoted by q_0. The state $q_0 \in Q$
- A finite, non-empty, set of final states, denoted by F, $F \subseteq Q$
- A transition function, generally denoted by δ, which maps $Q \times \Sigma \rightarrow$ any of the 2^n subsets of Q

Difference between a deterministic finite acceptor (DFA) and a non-deterministic finite acceptor (NFA)

- A DFA goes to a particular state on giving a particular input, that is, to a particular state. An NFA, on the other hand, can move to more than one state.
- There are no NULL moves in a DFA. An NFA, on the other hand, may contain a NULL move.

For example, the automata depicted in Fig. 18.5 is a non-deterministic finite acceptor as on giving 'a' to q_0, the machine goes to both q_1 and q_2.
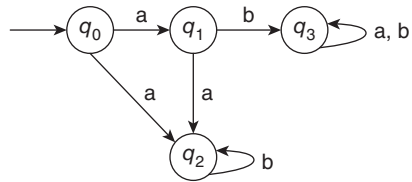
Figure 18.5 An NFA

18.6 KNUTH–MORRIS–PRATT AUTOMATA

The KMP algorithm finds a substring in a given string in a way that is more efficient than the naïve pattern-matching algorithm. The algorithm has two parts. The first part has a complexity of $O(m)$, m being the length of the string which is to be searched in a bigger string. The second part has complexity $O(n)$, n being the length of the given string, generally a bigger one (or at least equal). The complexity of the complete algorithm is, therefore $O(m + n)$, which is linear and hence better than the naïve string matching, which has a quadratic complexity.

> **Tip:** The KMP algorithm is the most efficient amongst the string matching algorithms discussed.

In the procedure explained in the following discussion, we would craft a KMP DFA. The number of states in the DFA would be one more than the number of characters in the pattern. The last state would represent the acceptor state. It should be obvious that the maximum number of characters that would match is m (the number of characters in P). Moreover, each state will be denoted by a number that would indicate the number of characters matched. For example, if we are at state 1, it means that 1 character has matched. Similarly, being in state 2 would mean that 2 characters have matched.

The approach relies on finding the maximum prefix of the pattern that is also the suffix of the given text (till that instant). As stated earlier, the second part of the algorithm requires $O(n)$ time. This is because if the requisite DFA is constructed then the only thing left would be to look for the corresponding entry in the DFA table. However, the construction of this DFA table is a difficult task.
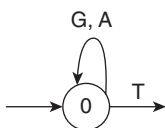
Figure 18.6 Designing transitions for the initial state

The construction of DFA starts with the transitions of the start state. Suppose, the first character that is to be matched is 'T' and the characters in the pattern are 'T', 'G', 'A', and 'C'. Now if the state encounters a 'T', then the control moves to the next state; otherwise it stays there. The situation is depicted in Fig. 18.6.
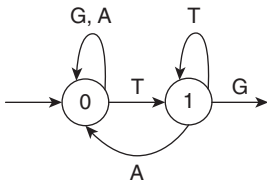
Pattern: TGC

In the next transition, the desired character, which in this case is 'G', takes the control forward, whereas a 'T' would keep the control to the state 2. Any other character would take the control back to state 0 (Fig. 18.7).

Figure 18.7 Designing transitions for state 1

Tip: The memory of the pattern to be searched is built in the KMP DFA

In the next state, if the machine encounters a 'G', then it moves forward, encountering any other character would take it back to the state 0 (Fig. 18.8). Figure 18.9 depicts the configuration of the DFA. Resulting after designing state 3.
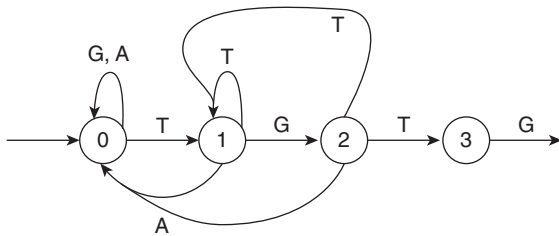
Figure 18.8 Designing transitions for state 2
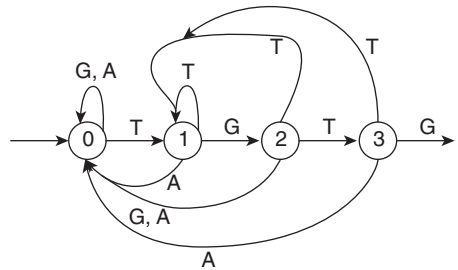
Figure 18.9 Designing transitions for state 3

In the same way, the transitions for the rest of the states can also be designed. The final DFA is depicted in Fig. 18.10.
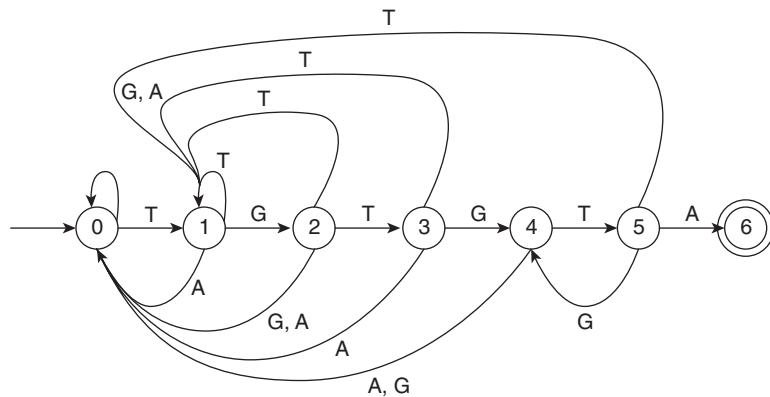
Figure 18.10 Final DFA

Let us create the transition table of the DFA also.

1. As stated earlier, the automata would have one state more than the number of alphabets in the pattern. The last state would be the accepting state.

2. If we are in the jth state and the next character matches then we move to the $(j + 1)$th state. This is because in the transition table, one must try to fill the states that would take the machine forward. For example, if we are at a state 0 and encounter a T, then we would want to go to state 1; in state 1 if we encounter a G we would go to state 2. In state 2, if we encounter a C we would go to state 3. That is, if the state is j and the next character is same as that in the pattern then move to the state $j + 1$ (Table 18.4).

3. If, on the other hand, the pattern does not match, then the following procedure must be followed. For example, if 1 is in the 0th state and encounters a G or A, it is not expected to move and therefore, the corresponding transition should take us to T (Table 18.5).

Table 18.4 Designing transition table

	T	G	T	G	T	A
T	1		3		5	
G		2		4		
A						6

Table 18.5 Designing transition table for mismatch transition

	T	G	T	G	T	A
T	1		3		5	
G	0	2		4		
A	0					6

Similarly, in the second state if the input is G then one should move forward, in the case where the input is T the state should remain same. In the case of A the control should move to state 0 (Table 18.6).

Table 18.6 Designing transitions for state 1

	T	G	T	G	T	A
T	1	1	3		5	
G	0	2		4		
A	0	0				6

In the case of the third state, if one encounters a 'G' or an 'A', one must go back to the state 0 (Table 18.7). The transitions for the next state are shown in Table 18.8, for the state 4 in Table 18.9, and that for state 5 in Table 18.10.

Table 18.7 Designing transitions for state 2

	T	G	T	G	T	A
T	1	1	3		5	
G	0	2	0	4		
A	0	0	0			6

Table 18.8 Designing transitions for state 3

	T	G	T	G	T	A
T	1	1	3	1	5	
G	0	2	0	4		
A	0	0	0	0		6

Table 18.9   Designing transitions for state 4

	T	G	T	G	T	A
T	1	1	3	1	5	
G	0	2	0	4	0	
A	0	0	0	0	0	6

Table 18.10   Designing transitions for state 5

	T	G	T	G	T	A
T	1	1	3	1	5	1
G	0	2	0	4	0	4
A	0	0	0	0	0	6

18.7 TRIES

A trie is a tree in which the root point to various sub-trees. The sub-trees depict the strings that are accepted by the tree. Every edge in a trie is marked with an alphabet and each node, which is not a root is a depiction of a state. The string formed while travelling from the root to any leaf is deemed to be accepted by the trie. The leaf node can, therefore, be considered as the accepting state of the automata. The root is, obviously, the starting state. It may be stressed that a trie, unlike a binary search tree, does not have just two children.

In order to create a trie from a given set of strings that must be accepted by the trie, the steps given in Algorithm 18.3 must be carried out.

Algorithm 18.3   Creation of trie

Input: Set of strings, a string in this set would be denoted by w_i
Output: A Trie

```
        TrieCreation(Set of strings) return Trie
{
    Create an empty state, the root node, which would point to the various sub-
    trees.
    The first string w, from the set of given strings is taken. For each character
    x ∈ w, create a transition labelled x. The string w would be the path from
    the root to the only leaf at this time in the tree.
    Pick the next string from the given set. Follow the path from the root till
    the characters match after which create a sub-tree which would be joined to
    the node till which wᵢ matched.
    Repeat the above step till all the strings have their sub-trees or a part
    of it.
    return Trie;
}
```

Illustration 18.3   Create a trie if the set of strings is {apple, ape, ampere, ample}.

Solution The creation of the trie would be as follows:
- Create a root node.
- Pick the first string from the set. In the given set, it is apple. Create a sub-tree having edges labelled a, p, p, l, and e (Fig. 18.11).
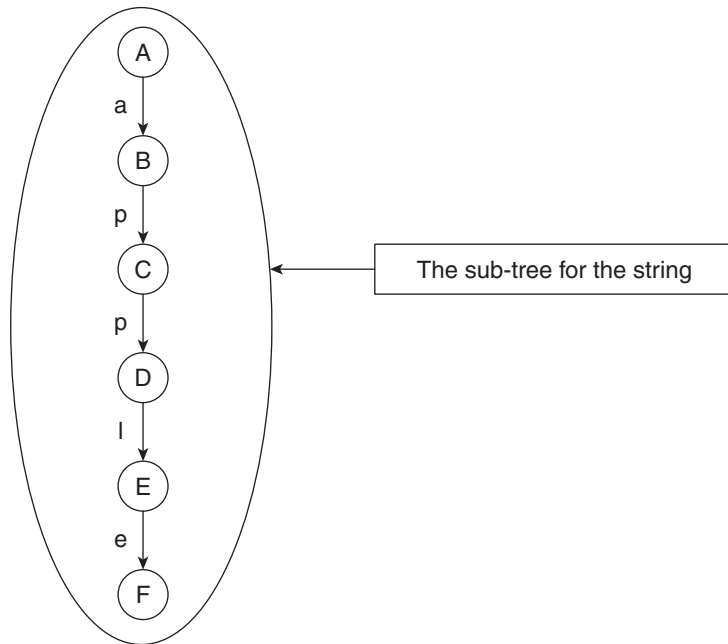
Figure 18.11 Creation of the root and sub-tree for the first string

- This is followed by the creation of the uncommon part of the second substring and joining it with the first sub-tree (Fig. 18.12).
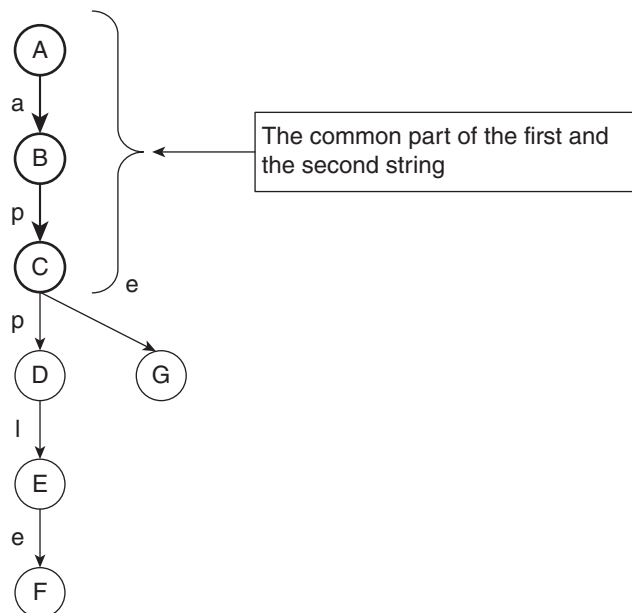
Figure 18.12 Expanding the trie for the second string

- The third substring has only 'a' in common with the first one, the rest of the part is created and joined with the first one (Fig. 18.13).
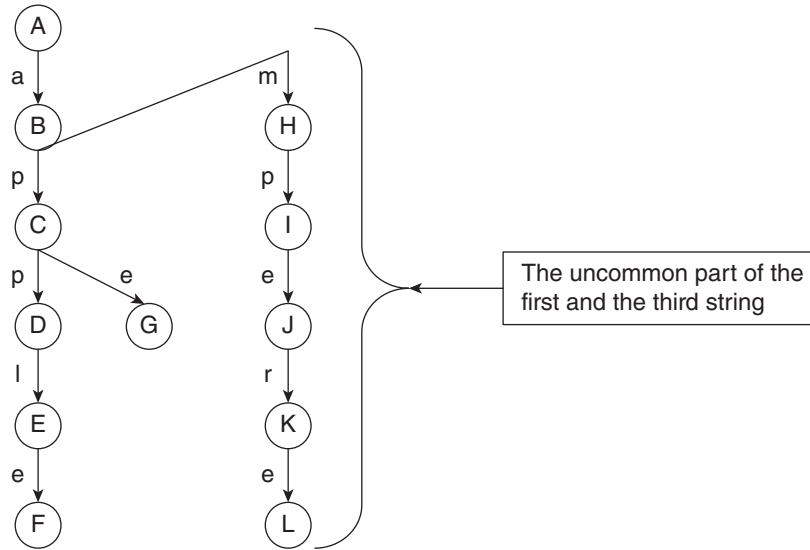
Figure 18.13 Expanding the trie for the third string

- The same process is followed for the fourth one (Fig. 18.14).
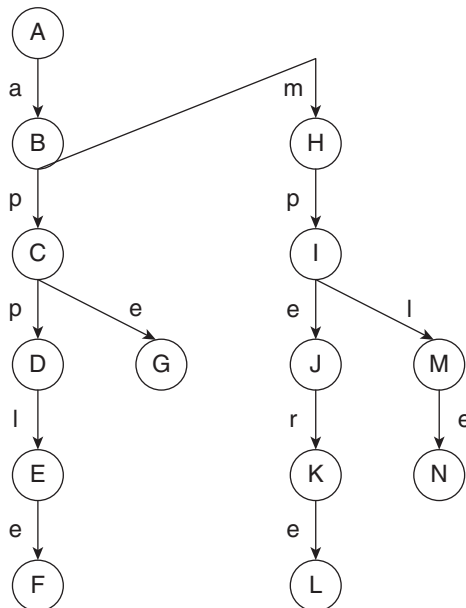
Figure 18.14 Expanding the trie for the fourth string

- The final tree is depicted in Fig. 18.15.
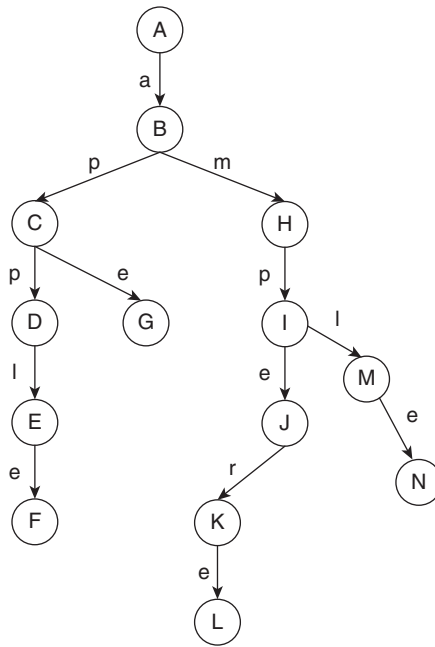
Figure 18.15 The final trie

18.8 SUFFIX TREE

A suffix tree is a trie formed by taking all the suffixes of a given string. For example, consider the string 'Harihan'. The various suffixes of this string are as follows:

arihan	rihan	ihan	han
an	n		

The trie of the above strings is depicted in Fig. 18.16. Such a trie is referred to as a suffix tree. The suffix tree is formed by taking all the *n* suffixes of a string of length *n*. Here, the leaves depict the accepting state. In the worst case, the complexity of creating such a tree would be $O(n^2)$. The better option, in such cases, would be to club together the characters, which would form the labels of the efficient version of the tree.

The suffix tree has many applications. For example, the formation helps in finding the longest common subsequence. The longest common subsequence has already been explained in Chapter 11. The reader is advised to explore the work of some of the authors who have proposed efficient versions of suffix trees. The versions are, however, beyond the scope of the book.
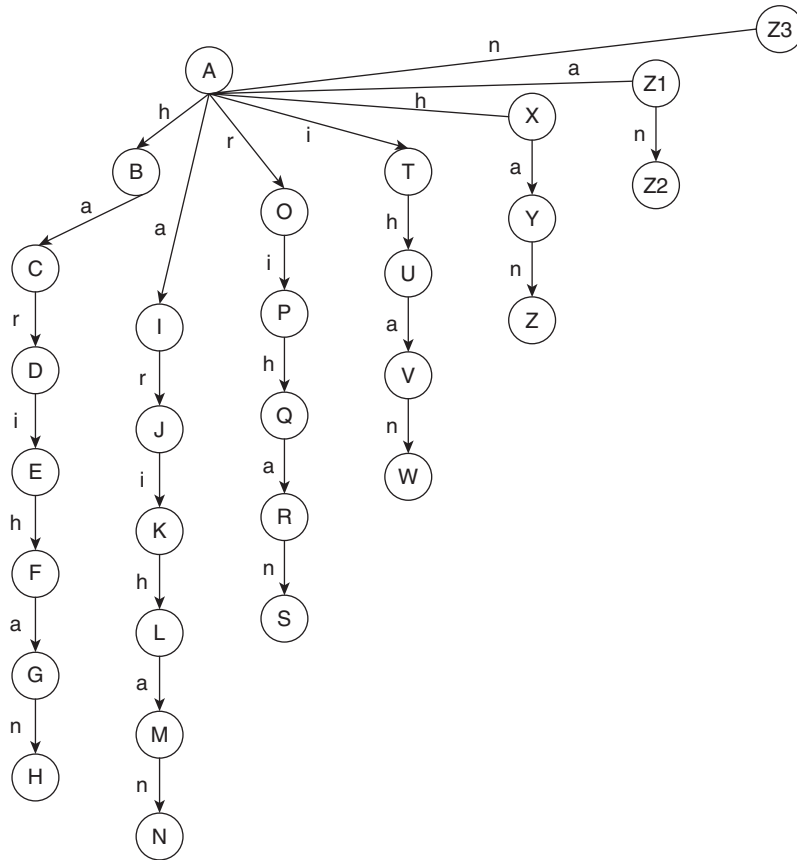
Figure 18.16 An example of suffix tree

18.9 CONCLUSION

The chapter explores an immensely important topic: string matching. The first algorithm presented in the chapter (naïve string matching) though computationally expensive, is easy to implement. The number of comparisons can be reduced by a minor modification suggested in the Rabin–Karp method. The method along with an illustration has been explained in the chapter.

The concept of DFA has also been dealt with in the text. The section also differentiates between a DFA and an NFA and introduces the concept of regular expressions. The power of a DFA can be judged by the fact that the most important method of string matching, that is, the KMP method is based on the construction of an efficient DFA, which is capable of pattern matching.

The chapter also introduces the trie and suffix tree. The construction of a trie has been dealt with in detail, by taking an appropriate example.

The chapter is not only important to those who want to pursue their career in computer science but also to those who intend to work in computational biology.

The topic is used in areas such as DNA matching, which makes it all the more important. The reader is advised to go through Jones & Pevzner and Attwood et al., (2009) for a deeper understanding. Some of the programs have been included in the web resources of this book.

Points to Remember

- The complexity of naïve string matching is $O(m(n − m + 1))$.
- Rabin–Karp algorithm has the best case complexity as $O(n + m)$.
- The worst case complexity of Rabin–Karp is $O(m(n − m))$.
- KMP is the most efficient algorithm for string matching.
- For every regular language, there is an NFA.
- Corresponding to every NFA there is a DFA.
- An NFA can have a NULL move, a DFA cannot.
- An NFA can go to more than one state on giving an input, a DFA cannot.

KEY TERMS

Deterministic finite acceptor A deterministic finite acceptor is a five-tuple calculus having the following components:
- A finite, non-empty set of states, denoted by Q.
- A finite, non-empty set of input symbols, denoted by Σ.
- An initial state denoted by q_0. The state $q_0 \in Q$.
- A finite, non-empty set of final states, denoted by F, $F \subseteq Q$.
- A transition function generally denoted by δ, which maps $Q \times \sum \to Q$.

Non-deterministic finite acceptor A non-deterministic finite acceptor is a five-tuple calculus having the following components:
- A finite, non-empty set of states, denoted by Q.
- A finite, non-empty set of input symbols, denoted by Σ.
- An initial state denoted by q_0. The state $q_0 \in Q$.
- A finite, non-empty set of final states, denoted by F, $F \subseteq Q$.
- A transition function, generally denoted by δ, which maps $Q \times \sum \to$ any of the 2^n subsets of Q.

String matching It refers to finding a small string and its position in a larger string. Given a text T[0…(n − 1)] and a pattern P[0…(m − 1)], it is required to find the beginning index of all the occurrences of P in T.

Suffix tree A trie of all suffixes of a particular string is called a suffix tree.

Tries A trie is a tree in which the root points to various sub-trees. The sub-trees depict the strings that are accepted by the tree. Every edge in a trie is marked with an alphabet and each root, which is not a node, is a depiction of a state.

EXERCISES

I. Multiple Choice Questions

1. In which of the following 'string matching' is used?
   (a) DNA matching
   (b) Text editors
   (c) Finding musical patterns from multimedia databases
   (d) All of the above

2. What would be the complexity of 'naïve string matching', if the text T is of length 'n' and the pattern P is of length 'm'?
   (a) $O(m^2)$ (c) $O(n^2)$
   (b) $O(m(n-m))$ (d) None of the above

3. What is the best case complexity of Rabin–Karp algorithm?
   (a) $O(n)$ (c) $O(m^2)$
   (b) $O(n+m)$ in the best case   (d) None of the above

4. What is the worst case complexity of Rabin–Karp algorithm?
   (a) $O(m(n-m))$ (c) $O(n^2)$
   (b) $O(m^2)$ (d) None of the above

5. What is the complexity of string matching using deterministic automata?
   (a) $O(n)$ (c) $O(n^2)$
   (b) $O(m)$ (d) None of the above

6. What is the complexity of Knuth–Morris–Pitt algorithm?
   (a) $O(n)$ (c) $O(n^2)$
   (b) $O(nm)$ (d) None of the above

7. Which of the following data structures is best suited when many of the strings to be matched have the common prefix with the first string?
   (a) Trie   (c) Do not try
   (b) Try   (d) Trial

8. Which of the following is the trie of all the possible suffixes of a string?
   (a) Suffix tree   (c) Suffix–Prefix
   (b) Suffix array   (d) Prefix tree

9. Which of the following is also called a prefix tree?
   (a) Tries   (c) Binary search tree
   (b) Suffix tree   (d) None of the above

10. Which language is accepted by a deterministic finite acceptor?
    (a) Regular   (c) English
    (b) Irregular   (d) Spanish

11. Which of the following is accepted by a non-deterministic finite acceptor?
    (a) Regular   (c) English
    (b) Irregular   (d) Spanish

12. Which of the following is true?
    (a) Corresponding to every DFA, there is an NFA
    (b) Corresponding to every NFA, there is a DFA
    (c) Both
    (d) None of the above

13. Which of the following is true?
    (a) An NFA can have a NULL transition
    (b) An NFA can go to more than one states on giving an input
    (c) Both of the above
    (d) None of the above

14. Which data structure is used in string matching?
    (a) DFA   (c) Prefix tree
    (b) Trie   (d) All of the above

15. Which is the most efficient string matching algorithm?
    (a) Knuth–Morris–Pitt
    (b) Rabin–Karp
    (c) Naïve
    (d) All of the above are equally efficient

16. Which of the following is not used in string matching?
    (a) Longest common subsequence   (c) Naïve string matching
    (b) Master theorem   (d) All of the above

II. Review Questions

1. Define string matching. What are the various applications of string matching?
2. Explain the naïve string-matching algorithm. What is its complexity?
3. Explain the Rabin–Karp algorithm. What is its complexity?
4. Explain the Knuth–Morris–Pratt algorithm. Discuss its complexity.
5. What are tries? Explain the algorithm for their formation.
6. What are suffix trees? What are the applications of suffix trees?
7. In the string '78956237856327', if the value of q in Rabin–Karp is 7, how many spurious matches are there?
8. Design a DFA that accepts
    (a) All the strings in {0, 1} in which the third symbol is 1 and the fifth is 0
    (b) The number of 1's are multiple of 4 and that of 0's are multiple of 5
    (c) The string starts with a '01' and ends with a '01'
    (d) The string starts with a '01' or ends with a '01'
    (e) The number of 1's are 2 more than the number of 0's
    (f) Having odd number of zeros and even number of ones
9. Formulate an algorithm to convert an NFA to a DFA.
10. Formulate an algorithm to convert a regular expression to an NFA.

11. Form the trie of all the suffixes of the word 'banana'.
12. Form a suffix tree of all the prefixes of 'banana'.
13. Explain how Knuth–Morris–Pitt algorithm has complexity $O(n)$.

Answers to MCQs

1. (d)	5. (b)	9. (a)	13. (c)
2. (b)	6. (a)	10. (a)	14. (d)
3. (b)	7. (a)	11. (a)	15. (a)
4. (a)	8. (a)	12. (a)	16. (b)

CHAPTER 19

Complexity Classes

OBJECTIVES

After studying this chapter, the reader will be able to

- Define P-type problems
- Distinguish between P and NP problems
- Define NP-complete problems and NP-hard problems
- Understand that all the NP-hard problems are not NP-complete

19.1 INTRODUCTION

Algorithms are designed to solve a problem or to accomplish a particular task. The problem in question can be of many types. Broadly, the problems can be classified into two types: decision problems and optimization problems. The decision problems determine whether an algorithm accepts an input, say *w*. If the input is accepted, the algorithm answers in a 'yes', otherwise it answers in a 'no'. The corresponding machine is a recursive machine. There is another type of machine called recursive enumerable machine, which answers in a 'yes' (Fig. 19.1). In such machines, one gets an answer only if the input is accepted, in other cases, neither the machine produces an error message nor does it produce a result.
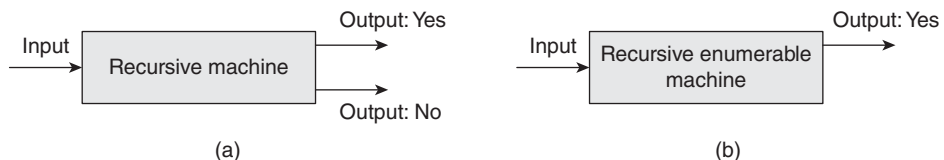
Figure 19.1   Types of machines: (a) recursive machine,
(b) recursive enumerable machine

The other type of problems, generally referred to as optimization problems, maximizes or minimizes the value of the objective function. The latter are a bit difficult to solve, as discussed in the sections that follow. Figure 19.2 summarizes the discussion.
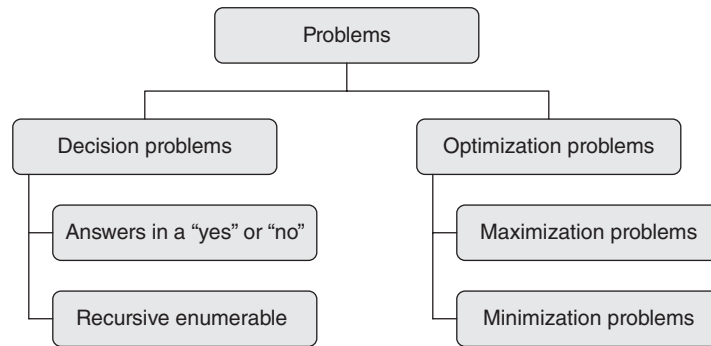
Figure 19.2 Types of problems

Definition:

Decision problem A problem that answers in a 'yes' or a 'no' is referred to as a decision problem.

Optimization problem A problem that maximizes or minimizes an objective function is called an optimization problem.

The basic definitions related to automata theory are presented in Fig. 19.3. Formally, a decision problem (M) can be defined as a language (L) which returns a '1', if the given string (w) is accepted by the language,

$$L = \{w \in L : M(w) = 1\}$$

Alphabet: $x : x \in \Sigma$, Σ being the finite, non-empty set of input symbols. In the discussion that follows, Σ contains at least two alphabets.

Strings: If Σ is the finite, non-empty set of input symbols, then Σ is the set of strings over Σ.

Language accepted by machine: If a machine accepts all $w \in \Sigma^*$, then L is said to be accepted by M.

Figure 19.3 Basic definitions

In the discussion that follows, the classes P, NP, etc. would be discussed in terms of Turing machine. The Turing machines are one of the most powerful automata. In fact, all computational functions can be defined in terms of Turing machines. The machine consists of a tape and a read–write head. Formally, it consists of the following components:

- Finite, non-empty set of input symbols, Σ.
- Finite, non-empty set of states, Q.

- Initial state, q_0, which belongs to the set of states
- Finite non-empty set of tape symbols, T.
- Transition function that maps the combination of input and state to that of state and tape symbol.
- Set of final states, which is a subset of the set of states.

Alan Turing introduced these machines in 1936. It is worth noting that though the machines were introduced before the physical computers (the first computer, ENIAC, was completed in 1946), these are still the best bet to define a computable function (Sipser, 1997). A string is deemed as accepted by a machine if the corresponding computation of the machine terminates in accepting state. Otherwise, the given string is not accepted by the machine. The number of steps in the computation of a given string by a given machine would be denoted by t_m and the worst case running time by T_m.

M runs in polynomial time if $T_m \leq n^k$. Correspondingly, the class P can be defined as follows (Sipser, 1997):

> **Definition** $P = \{L \mid L = L(M)$ for a Turing machine that runs in polynomial time$\}$
>
> The corresponding Turing machine should terminate in finite number of steps, bounded by polynomial time.
>
> Informally, the class P may be defined as the class of decision problems which can be solved in polynomial time, by some deterministic algorithm.

The informal definition is attributed to the work of Jack Edmonds (1965). The Turing machine mentioned above is a one-tape Turing machine. Examples of some of the problems that belong to the P class are as follows (Table 19.1).

Table 19.1   Some examples of problems that belong to the P class

Problem	Concept given by	Reference
SAT2	Cook in 1971	Section 19.3
Minimum spanning tree	Kruskal in 1956	Section 10.5
Shortest path	Dijkstra in 1959	Section 10.1
Solvability of linear equations		Appendix
Minimum cut	Edmonds in 1972	Section 19.3
Edge cover or arc cover	Edmonds in 1965	Section 19.3
Bipartite matching	Hall in 1948	Web resources
Sequencing with deadlines		Section 10.4

The class NP, on the other hand, contains problems that can be solved by non-deterministic algorithms in polynomial time. The problems were defined in terms of non-deterministic machines by Cook (1971). The non-deterministic machines have been defined in Section 20.4. Formally, NP can be defined as follows (Karp, 1972):

Definition $NP = \{L \mid L = L(M)$ for a non-deterministic machine that runs in polynomial time$\}$

Informally, the class NP may be defined as the class of decision problems which can be solved in polynomial time, by some non-deterministic algorithm.

Examples of some of the problems that belong to the NP class are as shown in Table 19.2.

Table 19.2 Some examples of problems that belong to the NP class

Problem	Reference
SAT2	Section 19.3
Clique problem	Section 19.3
Chromatic number problem	Section 19.3

Here, all the problems that belong to the class P are in fact in the class NP as well.

The NP-complete problems are those for which no polynomial time algorithm is known. However, the solution of these problems, if given, can be verified in polynomial time.

Then, there are problems for which no polynomial time algorithm is known and even their solutions, if given, cannot be verified in polynomial time. Such problems are called *NP-hard problems*. For example, the travelling salesman problem (TSP) is an optimization problem and hence is an NP-hard problem. There are two versions of the TSP; in one of the versions, the cost is given and it is required to find a path having a given cost from the given graph. For this variant, though there is no polynomial time solution, the answer can be verified in polynomial time. Such problems would be referred to as *NP-complete problems*. In the second variant of TSP, it is required to find the minimum cost Hamiltonian cycle of the given graph. There is no polynomial time algorithm that accomplishes the given task. Moreover, the solution obtained by a non-deterministic algorithm cannot be deemed as correct until and unless we have all the possible paths and the corresponding costs. This variant is one of the problems which is NP-hard but not NP-complete.

A problem that is NP-hard and NP is called an NP-complete problem. The TSP (in which the desired cost is given) belongs to this class.

Important points regarding complexity classes

- A problem that belongs to the class P also belongs to the class NP.
- All NP problems are not NP-hard.
- A problem for which the solution can be verified in polynomial time is NP-complete.
- There are some NP-hard problems, which are not NP.
- The problem that belongs to both NP-hard class and the NP class are called NP-complete problems.

A problem can be solved by reducing the problem to some other problem whose solution is known. The solution of the known problem can then be used to solve the given instance of the former. However, this method works only if the reduction can be done in polynomial time. In the discussion that follows if a problem A can be converted into B, in polynomial time, then A reduces to B or $A \propto B$.

> **Definition**
>
> **Reduction** If a problem A can be solved by a deterministic polynomial time algorithm, then that solves B in polynomial time.

The pioneers of the subject, like Richard M. Karp, have categorically stated the following points regarding reducibility:
- If $A \propto B$ and $B \propto C$, then $A \propto C$.
- If $A \propto B$ and B belongs to the class P, then A belongs to the class P.

19.2 CONCEPT OF P AND NP PROBLEMS

The problems of the class P are also called tractable, as they are solvable in polynomial time. For the problems of the class NP, on the other hand, there is no polynomial time algorithm and are hence intractable. According to literature review, a problem that belongs to the class P, also belongs to the class NP (Fig. 19.4).
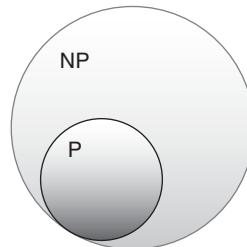
Figure 19.4 P and NP problems

The question whether a given problem is tractable or not is a precarious one. For instance, some of the authors believe that finding the shortest path in a graph is tractable. The complexity of the requisite algorithm being $O(n \log n)$, whereas the problem of finding the longest path is intractable, as the corresponding algorithm is NP. However, it is also possible that till now we have not been able to find the requisite algorithm and hence we are calling it intractable. The day an efficient polynomial time algorithm is designed for the longest path problem, the problem would become tractable.

There is another famous problem one instance of which is tractable, whereas the other is intractable. The problem is the CNF satisfiability (conjunctive normal form) problem. The CNF is the conjunction of clauses. A clause, in turn, is a disjunction of

literal, and literal is a variable or its negation. A Boolean function is one that maps $\{0, 1\}^n$ to $\{0, 1\}$. For example, for two variables 'a' and 'b', the literals are as follows:

- a
- $\sim a$
- b
- $\sim b$

The possible clauses are as follows:

- $a \wedge b$
- $a \wedge \sim b$
- $\sim a \wedge b$
- $\sim a \wedge \sim b$

In addition, the possible number of CNFs in the above case is infinite.

The CNF satisfiability problem is to find the values of literals which satisfy a given CNF. A 2CNF form has two literals in each term and a 3CNF form has three literals in each term. The 2CNF problem is tractable whereas the 3CNF is intractable (The problem has been defined in the following discussion.). Though the problem cannot be solved in polynomial time, whether it can be verified in polynomial time.

Definition

Verifies An algorithm 'ALG' verifies language L if $L = \{w \in \{0,1\}^* : \text{there exists } y = \{0,1\}^*, \text{so that } A(x,y) = 1\}$

19.3 IMPORTANT PROBLEMS AND THEIR CLASSES

Path Problem (PP)

Given a graph $G = (V, E)$ to find out whether there is a path from u to v, $u, v \in V$, which is of length l, such that $l \leq n$ for some n.

This problem answers in a 'yes' or a 'no', and is hence a decision problem. Moreover, since there exists an algorithm that accomplishes the above task in polynomial time, the PP is a problem that belongs to the P class.

Shortest Path Problem (SPP)

Given a graph $G = (V, E)$ and the corresponding matrix, the problem is to find the shortest path from u to v, $v \in V$.

This problem is an optimization problem, since it requires us to find a path that minimizes the distance between the given vertices. However, there are many algorithms which accomplish the above task in polynomial time. Hence, the problem belongs to the class P.

Longest Path Problem (LPP)

Given a graph $G = (V, E)$ and the corresponding matrix, the problem is to find the longest path from u to v, $v \in V$.

This problem is an optimization problem, since it requires us to find a path that maximizes the distance between the given vertices. However, the problem is an NP-complete problem, as against the shortest path problem.

Subset Sum Problem

Given a set S and a number k, the problem is to find out the subset of the given set having sum of its elements equal to k.

For example, if the set S is $\{1, 2, 3, 4, 5\}$ and the value of desired sum is 6, then the possible subsets that have the sum of its elements as 6 are $\{1, 5\}$ and $\{2, 4\}$. The problem seems simple. However, if a set has n elements and the desired sum is, say m, then the brute force approach would require the crafting of all possible 2^n subsets, calculating their sum, and finding out which subset gives the desired sum. The number of subsets of a set having n elements is 2^n, the problem is therefore an exponential one. However, the above problem requires enlisting of all the subsets of a given set, finding the sum of all of them, and then checking whether the sum of elements of that subset is same as the given number or not. The problem, though not unsolvable, has exponential time complexity. There is no algorithm for the problem that runs in polynomial time. However, if the solution is known, it is easy to verify the solution in polynomial time. Therefore, the problem can be categorized as an NP-complete problem.

0–1 Integer Programming

Given a set of equations of the form $A \times X = B$, where A and B are integer vectors, then there exists a vector X in $\{0, 1\}$, which satisfies the above equation.

The problem does not have a polynomial time algorithm. However, if the answer is given, it would be easy to find whether it is correct or not. Therefore, the problem can be categorized as an NP-complete problem.

Clique Problem

A clique is a complete sub-graph of a given graph. The problem is to find whether a clique of a given number of nodes exists, in the given graph, or not.

Suppose there are n nodes in a graph. There would be 2^n subsets of the set of nodes. Each graph would be checked (if it is a complete graph). The procedure is of exponential complexity. However, if the solution to the above problem is found given, then it would be easy to see whether the solution is correct or not. The problem is, therefore, an NP-complete problem.

Maximum Clique Problem

As stated earlier, a clique is a complete sub-graph of a given graph. The problem is to find the biggest clique of a given number.

The problem is not just NP-complete. Until we have the cost of all the paths, it would not be possible to find which is the maximum clique. The problem, therefore, can be categorized as NP-hard problem.

Set Packing

Given a family of sets, say $\{S_j\}$ and a number k. The problem is to find out whether there are k mutually disjoint sets.

Again finding the solution of the problem has exponential complexity. However, if the solution of the problem is given, it would be easy to check whether it is correct.

Node Cover

Given a graph $G = (V, E)$, it is required to find a set of vertices V' such that $V' \subseteq V$ and $\forall x \in E$, if $x = (u, v)$ then either u or v or both should belong to the set V'.

The problem is not solvable in polynomial time. However, if the solution of the problem is given it can be verified in polynomial time and hence, it is an NP-complete problem.

Set Cover

Given a finite family of sets $\{S_1, S_2, ..., S_k\}$ and an integer k. It is required to find whether there are k sets such that the union of the selected sets yield a set with all the elements present in the given family.

In order to solve the problem, all possible sub-facilities would have to be considered, this would be followed by applying union operations to all the groups. The complexity of this task would be exponential. However, if the solution is given, it would be easy to find whether it is the correct solution or not. This is the reason why this problem is considered as an NP-complete problem.

Tip: The following problems are also considered as NP-complete problems. The problems also mentioned in the paper titled 'Reducibility among Combinatorial Problems' by Richard M. Karp. The problems are related to the problems discussed above. The reader can find the definitions of the following problems in Karp (1972).
- Feedback node set
- Feedback arc set
- Undirected Hamiltonian cycle
- Exact cover
- Hitting set
- Steiner tree
- 3-D Matching

Directed Hamiltonian Cycle

A Hamiltonian cycle has already been defined in Section 12.6 of Chapter 12. The problem to find whether a Hamiltonian cycle occurs for a given graph is an NP-complete problem.

Satisfiability

The discussion that follows uses $\wedge$ for conjunction, $\vee$ for disjunction, and $\sim$ for negation. A Boolean variable is one which can have value either 'true' or 'false'. A Boolean expression is one which returns a 'true' or a 'false' for a given assignment of literals x_i's. A literal here refers to a Boolean variable or its negation. A Boolean formula is satisfiable if it is possible to find a set of values of literals for which it is true. If a Boolean formula

is true for all possible values of x_i's, then it is said to be valid. If a formula is valid, then the satisfiability of that formula follows. That is, validity is reducible to satisfiablity. Opposite of satisfiability is unsatisfiablity and that of validly is invalidity. A formula is unsatisfiable if there is no set of values which make the formula true. A formula f is not satisfiable if ~f is valid. In the same way, if a formula f is valid then ~f is not satisfiable.

For example, the following formula is satisfiable as $x_1 =$ True, $x_2 =$ False, and $x_3 =$ True makes the formula True. However, it is not valid, as there are some values of x_i's that make the expression False,

$$(x_1 \wedge x_3) \vee x_2$$

Now, let us come to the computational complexity of the above problem. If there are n literals in an expression, then there would be 2^n possible set of values that could be assigned to literals. This would be followed by finding if there is any set for which the expression is true. The task, though not impossible, is that of exponential complexity.

We henceforth restrict our discussion to a special class of expressions that have literals combined in pairs of two, associated by a conjunction. Between these pairs would be a disjunction. There can be many such pairs. For example, $(x_1 \wedge x_3) \vee (x_2 \wedge x_1) \vee (x_1 \wedge x_3)$ is an expression of the said type. Such problems that involve assignment of truth values to the x_i's in the above type would be referred to as SAT2. SAT2 is a problem that belongs to the class P.

There is another class in which the literals form parts connected by disjunction, wherein each part has three literals and these literals are connected by conjunctions. Here, the condition is that the parts should not have a literal and its negation. Moreover, no two parts should be exactly same. It is a harder decision problem. The corresponding problem finds whether there is a set of values of x_i's for which the given expression is true. The problem is generally referred to as SAT3 problem. The SAT3 problem is an NP-complete problem. There is no known algorithm that can solve the SAT3 problem in polynomial time. However, if the solution of the problem is given, it can be verified in polynomial time.

The list of NP-complete problems has been presented in Fig. 19.5.

List of Karp's NP-Complete Problems

- Satisfiability
- Integer programming
- Clique
- Set packing
- Node cover
- Set cover
- Feedback node set
- Feedback arc set
- Directed Hamiltonian cycle
- Undirected Hamiltonian cycle
- Chromatic number
- Clique cover
- Exact cover
- Hitting set
- Steiner tree
- 3-D Matching
- Knapsack

Figure 19.5 List of NP-complete problems

Related to SAT3 we have an important theorem. The theorem, referred to as Cook's theorem, is as follows.

19.4 COOK'S THEOREM

The theorem is one of the most basic theorem in the NP theory. It can be stated as follows:

Satisfiability is in P if $P = \mathrm{NP}$

The premise of the above is a simple fact that every NP-complete problem can be reduced to satisfiability. If someone would be able to come up with an algorithm that solves the SAT3 problem, then all the NP-complete problems (that can be reduced to SAT3) would be solved. The concept can be understood with the help of Fig. 19.6.
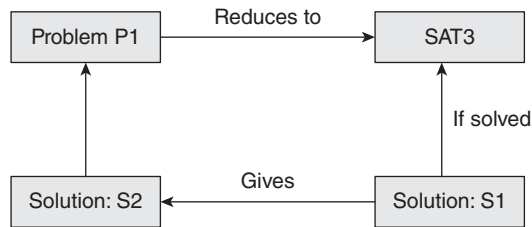
Figure 19.6   Reducibility to SAT3

19.5 REDUCIBILITY

The concept of reducibility has been discussed in Section 19.1. The application of reducibility to solve problems that are NP type has been discussed in this section. The concepts discussed here would help the user to convert a similar NP problem to one which can be solved and hence tackle the problem.

19.5.1 How to Convert a CNF into an AND-OR Graph?

The conversion of a CNF into an AND-OR graph is described in the following discussion. The conversion algorithm is a polynomial time algorithm. As stated earlier, since the reduction is polynomial time, and a CNF is known to be an NP-complete problem, the AND-OR graph becomes an NP-complete problem. The procedure requires the creation of a root node. The root node will have at least two children. The first child, generally from the left, depicts the node, referred to as E, will have a number of children equal to the number of $\wedge^s$ in the CNF. For example, in the expression $(x_1 \wedge x_2 \wedge x_3) \vee (\sim x_2 \wedge x_2 \wedge \sim x_3)$, there would be two children of E. In the expression $(x_1 \wedge x_2 \wedge x_3) \vee (\sim x_1 \wedge x_2 \wedge \sim x_3) \vee (\sim x_1 \wedge x_2 \wedge \sim x_3)$, there would be three children of E. The rest of the children of the root node would be the literals in the given expression. In the above cases, there would be three such nodes x_1, x_2, and x_3. Each literal, in turn would have two nodes on denoting the case wherein the value of that literal is true and
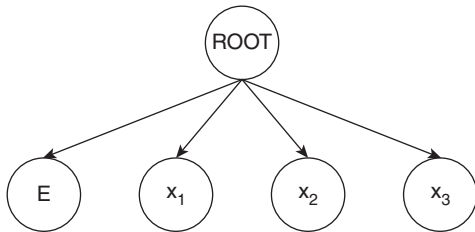
Figure 19.7 Creation of the ROOT and its children
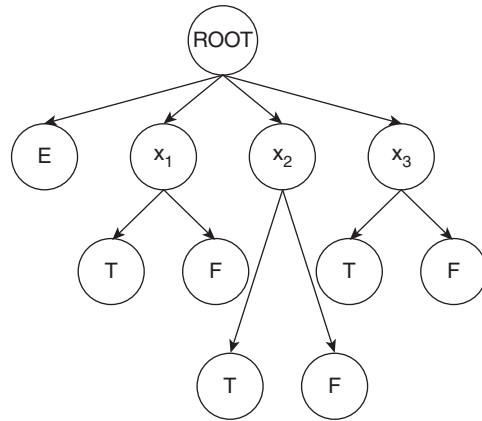
Figure 19.8 Creation of the children of literals

the other denoting the cases wherein the value of that literal would be false. The children of E would be connected to the requisite nodes, after that. In order to understand the process, let us consider the conversion of the following expression into an AND-OR graph.

$$(x_1 \wedge x_2 \wedge x_3) \vee (\sim x_1 \wedge x_2 \wedge \sim x_3) \vee (x_1 \wedge \sim x_2 \wedge x_3)$$

Step 1 Create a root node, say ROOT. The root node would have four children E, x_1, x_2, and x_3 (Fig. 19.7).

Step 2 Each literal would have two children, one depicting the value when it is true and the other depicting the value when it is false (Fig. 19.8).

Step 3 The next step requires the children of E. The number of children of E would be same as the number of conjunctions in the CNF. In the given expression, there are three such conjunctions, E would therefore have three children (Fig. 19.9).
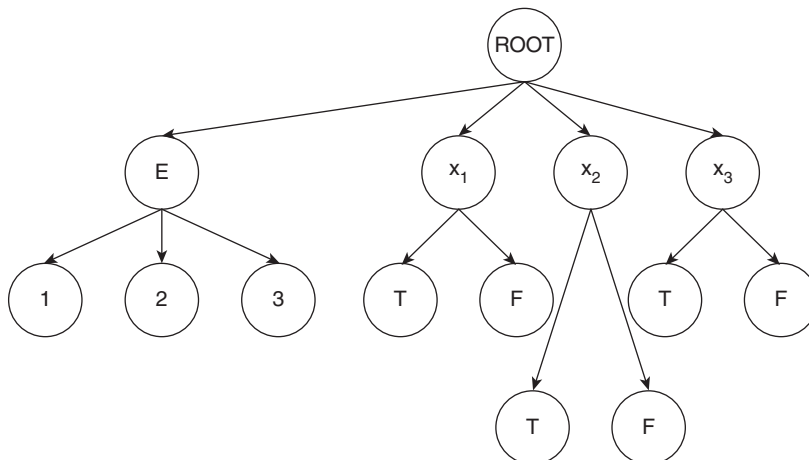
Figure 19.9 Creation of children of E

Step 4 In the last step, the children of E would be connected to the appropriate nodes (children of x_i's). In the above case, 1 would be connected to x_1, x_2, and x_3's true value; 2 would be connected to the false value of x_1, true of x_2, and false of x_3; 3 would be connected to true of x_1, false of x_2, and true of x_3 (Fig. 19.10).
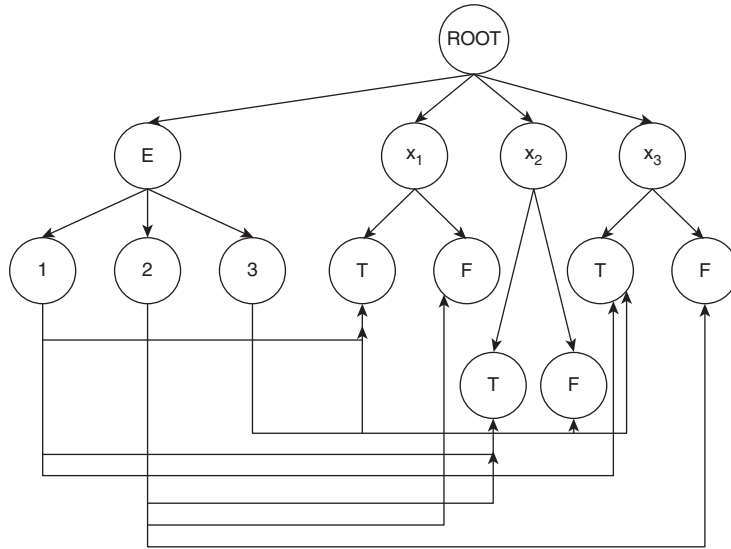
Figure 19.10 Connecting the nodes of E to the appropriate nodes

Step 5 Finally, the root node and the node E would represent ANDing (shaded circles in Fig. 19.11).
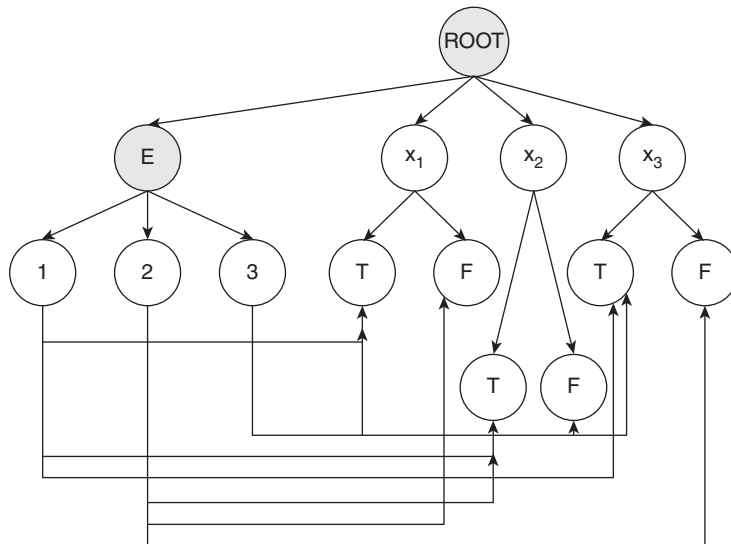
Figure 19.11 Final answer

19.5.2 Maximum Clique from SAT3

The SAT3 problem can be reduced to maximum clique using the following process. Here, the reduction presented is a polynomial time reduction, which makes maximum clique an NP-hard problem. A clique is a complete sub-graph of a given graph. The problem is to find the maximum clique of a given graph. The graph shown in Fig. 19.12 has many cliques. The maximum clique is of size 5. The discussion explores the conversion of SAT3 into the corresponding decision problem of maximum clique (whether a clique of order k exists in a graph).
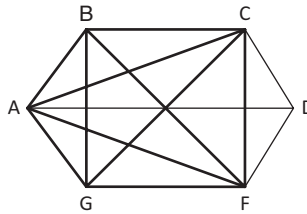
Figure 19.12 A graph
with MAX clique of size 5

The problem, as discussed earlier, is NP-hard. The brute force requires the enlisting of all possible subsets of nodes. If there are n nodes in a graph, the total number of subsets of these nodes would be 2^n. This follows the enlisting of subsets which form a complete graph. This is followed by finding the clique having maximum size. The process has, therefore, exponential complexity.

In order to carry out the reduction, we first start with a 3CNF. Let the number of clauses in a CNF be k. Since there are three literals in each clause, there would be $3 \times n$ literals. A graph is formed corresponding to this CNF. For each literal, a node is created in the corresponding graph. The value of the node would be same as that of the literal. A literal would be joined to all others, which are not in the same clause. This way a literal would be connected to $3 \times n - 3$ nodes.

For example, for the CNF

$$(x_1 \wedge x_2 \wedge \sim x_3) \vee (x_1 \wedge \sim x_2 \wedge x_3) \vee (\sim x_1 \wedge x_2 \wedge x_3) \vee (x_1 \wedge x_2 \wedge x_3).$$

The value of $k = 4$, since there are 4 clauses.
- There would be $4 \times 3 = 12$ nodes in the corresponding graph.
- Each node would be connected to $4 \times 3 - 3 = 9$ other nodes.

The corresponding graph is shown in Fig. 19.13.

Note that each node is connected to all other nodes except for those depicting the literals in the same clause.

Claim: If the graph formed has a clique of size k, then the formula is satisfiable.

In order to accomplish the task of finding the requisite clique, each of the literal in the clique should come from a different clause. One such clique has been depicted in the graph in Fig. 19.13 (see bold line).
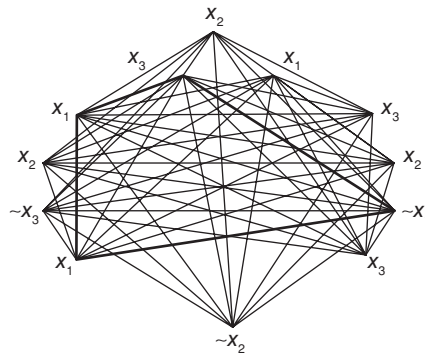
Figure 19.13   Reduction of CNF to maximum clique

Now, literals that form the clique come from different clauses. The proof proceeds by assigning one literal per clause 'true'. Since there is no contradiction, the assignment makes sense (It was earlier stated that a clause cannot contain both literal and its negation). The literals would form a clique if the formula is satisfiable and vice versa.

The above discussion points towards the fact that a 3CNF can be converted to clique. Therefore, clique is an NP-complete problem. The corresponding optimization problem is NP-hard.

19.5.3 Independent Set

An independent set is the set of nodes with no edge between them. The decision problem is to find whether an independent set of order k exists in the graph. For example, in Fig. 19.14, the nodes {A, D, F} form an independent set, as there is no edge between A and D, A and F, D and F.
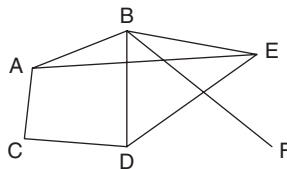
Figure 19.14   An example of an
independent set

The corresponding optimization problem is to find the maximum independent set in a graph. The former is NP-complete, whereas the latter is though NP-hard but not NP-complete.

If one is able to find the largest clique in a graph, then the maximum independent set can be found by simply taking the complement of that graph. It may be stated here that every graph has a corresponding complementary graph.

19.5.4 Vertex Cover

The vertex cover of a graph is the set of all the vertices such that each edge has at least one of the vertices in the set.

It was shown in the last sub-section that the independent set can be deduced from the max clique problem. Now, having got the independent set, one can find the vertices that are not in the largest independent set. The solution would give the minimum vertex cover of the graph. The vertex cover problem has been dealt with in Chapter 21 on Approximation Algorithms.

Now, it is clear that 3CNF can be converted into max clique, max clique to independent set, and independent set to vertex cover. It is left for the reader to deduce that SAT3 can also be converted to graph colouring. Figure 19.15 summarizes the discussion.
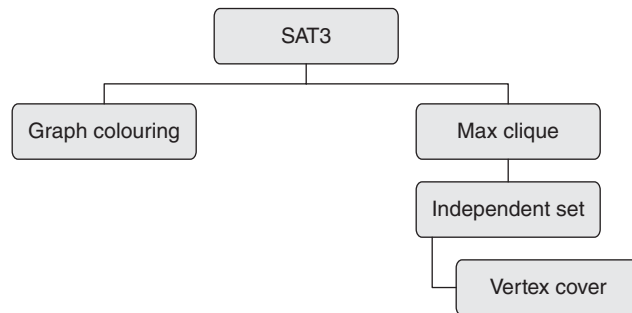
Figure 19.15 Reductions explored in Section 19.5

The earlier section gives an idea of reducibility. The concept can be applied to many problems. The 0–1 integer problem, for instance, can be reduced to satisfiability. As per Richard Karp, the reduction of a problem into another can be understood by Fig. 19.16. Here, some of the most common problems have been included. Karp (1972), in one of his pioneering works, presented the graph depicting the concept for all the NP-complete problems.
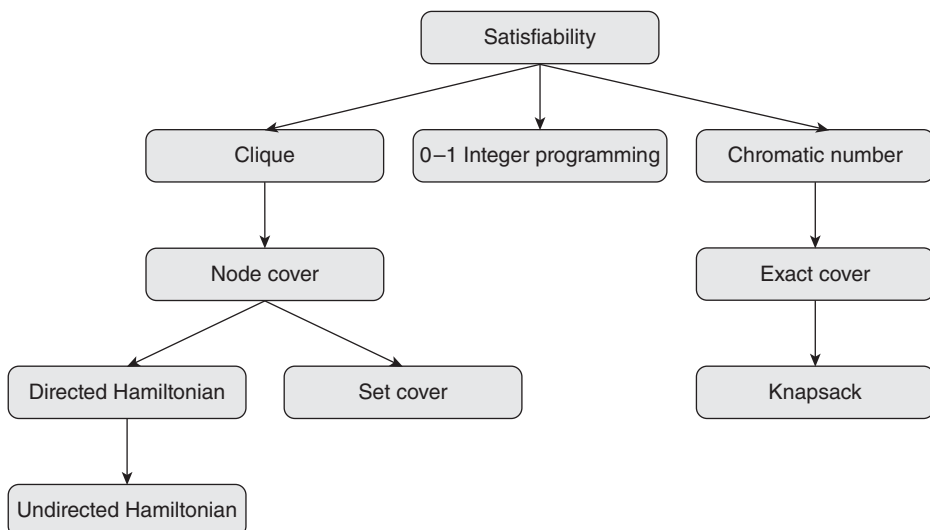
Figure 19.16 Reducibility

19.6 PROBLEMS THAT ARE NP-HARD BUT NOT NP-COMPLETE

The problems discussed in the earlier sections are NP-complete. However, some problems have a harder version. For example, if it is required to check whether a TSP of given cost exists in a graph or not, then though the solution would require a non-polynomial algorithm but the verification can easily be done. If, on the other hand, one is required to find the minimum cost Hamiltonian cycle in a weighted, directed graph, then the problem does not remain NP-complete. The reason being that until all the possible paths have been processed, there is no way of knowing whether the solution that we have got is the minimum or not. Such problems are NP-hard problems, which are not NP-complete.

Another example of such problem is that of a maximum clique. Though it is easy to verify that the clique obtained by a particular algorithm contains k nodes, it is almost impossible to state whether the clique obtained is the maximum.

Minimal set cover is another problem, which is an NP-hard problem that is not NP-complete. Now, if one is able to obtain the solution of the harder version of the problem, some of the complete versions would automatically be solved. For example, if one is able to obtain the maximum clique, it would be obvious that a clique exists. If one is able to obtain the minimum cost Hamiltonian cycle in the given graph the existence of Hamiltonian cycle would automatically be proved.

It may therefore be concluded that all the NP-hard problems are not NP-complete (Fig. 19.17).
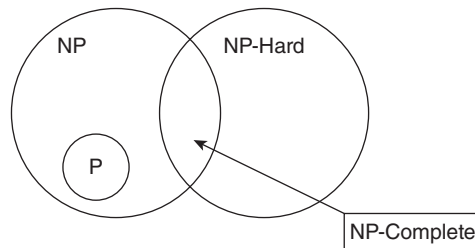
Figure 19.17 Relation between P, NP, NP-complete and NP-hard problems

The complete description of such problems along with a tool that helps to solve such problems can be found in the web resources of this book.

19.7 CONCLUSION

In Chapter 5, various algorithms were discussed, which were used to solve a variety of problems. The algorithms were of the type $O(1)$ like the addition of an element in a linked list. As far as complexity is concerned, these are the cheapest algorithms. The linear time algorithms, such as linear search, take $O(n)$ time. The time taken by a quadratic

algorithm is much more than that by a linear algorithm. Examples of such algorithms are selection sort and bubble sort. The cubic algorithms such as conventional matrix multiplication take a longer time as compared to the above algorithms.

All the above are of the type P, as the time taken by the above are $O(n^p)$. The algorithms that take $O(k^n)$ algorithm such as TSP are much harder to solve. The problems belonging to the above class can, however, be solved by non-deterministic algorithms. This class is called NP. The NP algorithms that can at least be verified in polynomial time are called NP-complete problems. The harder ones are referred to as NP-hard problems. The NP-complete problems have also been dealt with in Chapter 23, where the problems have been handled by using a heuristic search process called genetic algorithms.

Points to Remember

- Each P-type problem is an NP problem.
- Not every NP-complete problem is NP-hard.
- Problems that are both NP and NP-hard are NP-complete.
- 2CNF is satisfiable, 3CNF is not.
- Finding Euler cycle is not an NP-complete problem, however Hamiltonian cycle is.
- Finding longest path between two nodes is a NP-complete problem whereas finding out the shortest path is not.
- Generally, most of the NP-complete problems can be converted to SAT3.
- If solution of SAT3 is found, then all the NP-complete problems would become P type.
- The concept of reducibility can be used to solve NP-complete problems, once SAT3 is solved.
- Each computational function has a corresponding Turing machine.

KEY TERMS

Decision problem A problem that answers in a 'yes' or a 'no' is referred to as a decision problem.

Optimization problem A problem that maximizes or minimizes an objective function is called an optimization problem.

P Class $P = \{L \mid L = L(M)$ for a Turning machine that runs in polynomial time$\}$.

The corresponding Turing machine should terminate in a finite number of steps, bounded by polynomial time. Informally, the class P may be defined as the class of decision problems that can be solved in polynomial time, by some deterministic algorithm.

Reduction If a problem A can be solved by a deterministic polynomial time algorithm, then that solves B in polynomial time. The reducibility is generally written as A $\propto$ B (A reduces to B).

NP-complete problems The NP-complete problems are those for which no polynomial time algorithm is known. However, the solution of these problems, if given, can be verified in polynomial time.

EXERCISES

I. Multiple Choice Questions

1. A problem that can be solved by a deterministic machine in polynomial time is
   (a) P   (c) NP-complete
   (b) NP   (d) NP-hard

2. A problem that can be solved by a non-deterministic algorithm in polynomial time is
   (a) P   (c) NP-complete
   (b) NP   (d) NP-hard

3. A problem that has output as 'Yes', if the given input is accepted, otherwise 'No' is
   (a) Decision problem   (c) Both
   (b) Optimization problem   (d) None of the above

4. A problem that requires maximization or minimization of the objective function is a
   (a) Decision problem   (c) Both
   (b) Optimization problem   (d) None of the above

5. A problem that cannot be solved by a polynomial time algorithm but is verifiable in polynomial time is
   (a) P   (c) NP-complete
   (b) NP   (d) NP-hard

6. An NP-hard problem is
   (a) NP   (c) P
   (b) NP-complete   (d) None of the above

7. An NP-complete problem is
   (a) NP   (c) Both
   (b) NP-hard   (d) None of the above

8. SAT2 is
   (a) P   (c) NP-hard
   (b) NP   (d) NP-complete

9. SAT3 is
   (a) NP   (c) P
   (b) NP-complete   (d) None of the above

10. Longest path problem is
    (a) P   (c) NP-hard but not NP-complete
    (b) NP-complete   (d) None of the above

11. TSP is
    (a) P
    (b) NP-complete
    (c) NP-hard but not NP-complete
    (d) None of the above

12. Which of the following is correct?
    (a) All the problems can be solved in polynomial time
    (b) $P = NP$
    (c) If SAT3 can be solved in polynomial time, then $P = NP$
    (d) None of the above
13. A 3 clique problem is
    (a) NP
    (b) NP-hard but not NP-complete
    (c) NP-complete
    (d) None of the above
14. A 2 clique problem is
    (a) NP
    (b) NP-hard but not NP-complete
    (c) NP-complete
    (d) None of the above
15. If vertex cover is NP-complete then which of the following is NP-complete?
    (a) Independent set
    (b) SAT3
    (c) Both
    (d) None of the following

II. Review Questions

1. Define complexity class P, NP, NP-complete, and NP-hard.
2. Discuss the following problems and prove that they are NP-complete:
    (a) SAT3
    (b) 3CNF
    (c) 3 COLOR
    (d) Hamiltonian cycle
    (e) Travelling salesman problem
    (f) Vertex cover
    (g) Clique problem
    (h) Independent set
    (i) Subset sum
    (j) Integer programming
    (k) Partition
    (l) Knapsack
    (m) Scheduling
3. Prove that SAT3 can be reduced to vertex cover.
4. Prove that vertex cover can be reduced to independent set.
5. Prove that independent set can be reduced to set cover.
6. Prove that SAT3 can be reduced to graph colouring.

Answers to MCQs

1. (a)	5. (c)	9. (b)	13. (c)
2. (b)	6. (d)	10. (b)	14. (a)
3. (a)	7. (c)	11. (b)	15. (c)
4. (b)	8. (a)	12. (c)	

CHAPTER 20

Introduction to PSpace

OBJECTIVES

After studying this chapter, the reader will be able to
- Understand the concept of PSpace
- Explain the relation between different classes of problems
- Define the concept of QSAT and why is it considered a PSpace problem
- Learn the concept of regular expressions
- Understand planning problems

20.1 INTRODUCTION

In the earlier chapters, we have explored the concept of P and NP classes. P problems are those that are solvable in polynomial time. In fact every P problem is also an NP problem. For NP problems, the point of contention is whether the solution of the problem can be verified in polynomial time or not. These issues were discussed in Chapter 19.

The focus, till now, has been on time. However, in the first chapter, we had seen that space is an equally important resource as time. In fact, making a program efficient both in terms of time and space has been a goal of the algorithms that we design.

So the complexity classes must also be extended to the concept of space. This chapter explores the concept of space as a resource and defines some classes with respect to the above premise. The decision problems solvable in polynomial space are called PSpace problems. In fact, every problem that can be solved in polynomial time requires polynomial space.

> **Tip:** Each problem that requires polynomial time requires polynomial space.

For example, linear search requires $O(n)$ space, so does bubble sort and selection sort. Even if the algorithm takes a longer time, it is possible that the space requirement is not too high. In dynamic programming, we have seen cases wherein the space requirement is large $O(n^2)$ but the time requirement is not that large $O(n)$. Every polynomial time algorithm that we would see in the book would take polynomial space. So the claim is that a polynomial time algorithm would require polynomial space, that is, $P \subseteq PSpace$ (Fig. 20.1).
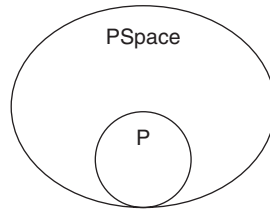
Figure 20.1 P is a subset of PSpace

Interestingly, it has not been established that $P \neq PSpace$. So it might be the case that the relation depicted in Fig. 20.1 is not absolutely true.

Definition A problem that is solvable by a Turing machine via polynomial space is called a PSpace problem.

One of the most common examples found during the literature review was that of counting till an exponential number. In order to do that, we need a linear space possibly n bits, if the base is 2. Even if the base is not 2, then the number of bits would increase but the space requirements would still remain polynomial.

Moreover, even if a problem is NP, then the space requirements also remain polynomial. In Chapter 19, it was discussed that SAT3 is an NP-complete problem. Furthermore, many problems can be converted into SAT3. The reductions of problems such as maximum clique and subset sum were also discussed in this chapter.

SAT3 requires values to be assigned to n variables and checking if the given Boolean expression is true. These values can be stored in an n bit array. A '1' would indicate that the variable is true and a '0' would indicate that the variable is false.

That is, SAT3 despite being an NP-complete problem requires a polynomial space. Since SAT3 is NP-complete and problems discussed in Chapter 19 could be converted into SAT3, the next claim is that NP is also a subset of PSpace, that is, $NP \subseteq PSpace$ (Fig. 20.2). The relation between the various classes is shown in Fig. 20.3.
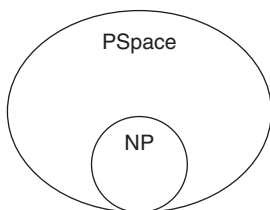
Figure 20.2 NP is a subset of PSpace
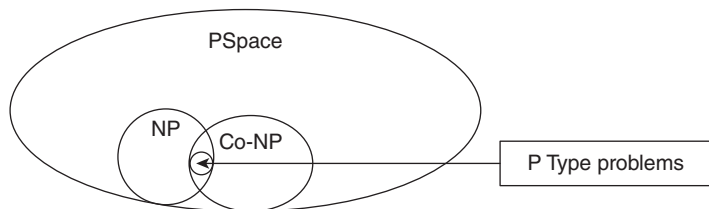
Figure 20.3 P is the intersection of NP and Co-NP

The above argument leads to the conclusion that $P \subseteq NP \subseteq PSpace$, though PSpace is perceived to contain problems that are not P or NP.

Those who have studied the "Theory of Computation" must be familiar with Turing machines. It was found that adding non-determinism to a Turing machine does not greatly increase its power, in fact not even by a small amount. This premise leads to the conclusion that PSpace is same as NPSpace.

$$PSpace = NSpace$$

20.2 QUANTIFIED SATISFIABILITY

SAT3 is one of the most amazing problems in complexity classes. It was discussed in Chapter 19 that almost all the NP-complete problems can be easily converted into SAT3. The question that arises is whether we have a corresponding problem in PSpace which despite its hardness can be used to understand all such problems.

The answer is yes. The problem that is as amazing and interesting in PSpace is quantified satisfiability (QSAT). The problem can be stated as follows.

QSAT

Let f be a CNF formula consisting of n variables $\{x_1, x_2, ...\}$. For what assignments of $\{x_1, x_2, ...\}$, the following formula is true:

$$\exists x_1 \forall x_2 \exists x_3 \forall x_4 ... x_n f$$

One of the brute force approaches to solve the problem would be to try all possible solutions recursively. If we recursively try all the possibilities for the given variables we end up getting a tree. The root of the tree would contain x_1. The left part of the tree would be traversed if the value of x_1 is 0, if the value of x_1 is 1 then the right part of the tree would be traversed.

In the same way, the next level would have x_2. If the value of x_2 is true then the left part of the tree would be processed; otherwise the right part would be processed. The creation of the whole tree would require exponential number of spaces.

The solution of the above problem has been depicted in Fig. 20.4.
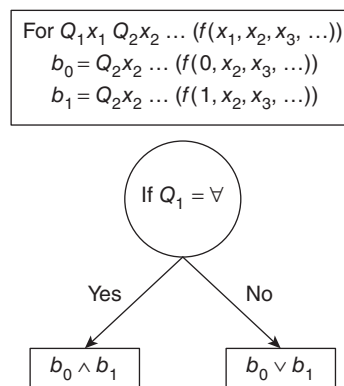
Figure 20.4 Solution of QSAT by Valentine Kabanets;
here Q is a quantifier.

The QSAT has two versions, one with the varying number of alternating quantifiers and the other with a fixed number of alternating quantifiers. The second version is obviously simpler than the first one and hence can be perceived as a subset of the first.

The conclusion of the earlier discussion is that the QSAT problem is a PSpace problem. We now move to another class of PSpace problem, which is planning problems.

20.3 PLANNING PROBLEMS

Suppose if we have been asked to design an artificial life system. In the system, there are various species. We must decide the rules that govern their evolution and interaction. Hence, the task is complex. Although there are artificial life systems such as the variants of Langton's loop (Langton designed a self-replicating system in which a new loop is generated on the completion of one cycle Berry and Ravindra (1999)) which carry out the above task, in a fascinating way, the concept of planning is still a contentious issue in such systems. For example, in developing a system that has two species A and B. B are bad and their company can even corrupt A's. The designing of such a system requires the enlisting of many possibilities.

The above problem is an instance of planning problem is one in which we find all the possible configurations of the environment, if a change is made in the environment. Some of the problems that require planning are Rubik's cube, *N*-puzzle problem, artificial life simulations, etc.

20.3.1 *N*-Puzzle Problem

An *N*-puzzle problem has an $m \times m$ board and tiles numbered from 1 to *N*. There is also a blank space so that the tiles can move. The value of *N* is related to *m* as $N = m^2 - 1$. There are many versions of *N*-puzzle problem wherein the values of *N* are 8 (in a 3×3 board), 15 (in a 4×4 board), 24 (in a 5×5 board), and so on. As stated earlier, the blank space allows tiles to move. The goal state is an instance like that shown in Fig. 20.5.

Figure 20.5 The goal state of a 8-, 15-, and 24-puzzle problem

The formal definition of the automata is as follows.

The automata has a set of states (*Q*), the initial state (q_0), the goal state (*F*), the path cost (*C*), and successor function (*f*) (Bedau, 2003).

Formally,

$$P = (Q, q_0, F, f, C)$$

where

Q = Set of states

q_0 = Initial state

F = Final state. This can be one of the goal states already defined.

f = Function called successor function which generates the next state. This state can be described by a move left, right, up, or down.

C = Path cost. It is the number of steps in the path considering each move to be of unit cost.

The problem is a planning problem and it requires the elicitation of transition rules, the corresponding costs, and the heuristics used. In a planning problem, a move leads to a new configuration.

8-Puzzle Problem

An 8-puzzle problem has a 3×3 board and 8 tiles numbered from 1 to 8. One cell of the frame is empty which helps in the movement of tiles. The problem is to change the initial state to goal state by sliding the tiles, one at a time, in minimum moves. One of the instances of the initial and the final states are depicted in Fig. 20.6.

4	5	7
2	1	8
3	6	

Initial state

1	2	3
4	5	6
7	8	
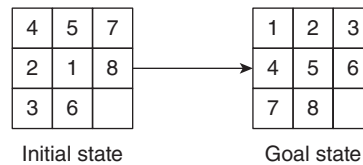
Goal state

Figure 20.6   Example of a 8-puzzle problem

15-Puzzle Problem

A 15-puzzle problem has a 4×4 board and 15 tiles numbered from 1 to 15. One cell of the frame is empty which helps in the movement of tiles. The problem is to change the initial state to goal state by sliding the tiles, one at a time, in minimum moves. One of the instances of the initial and the final state are depicted in Fig. 20.7.

4	5	7	10
3	1	2	12
8	6	9	11
13	14	15	

Initial state

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

FInal state
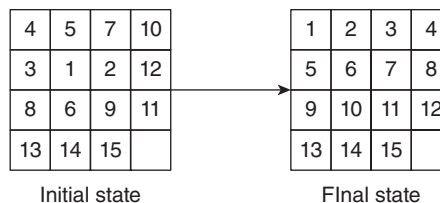
Figure 20.7   Example of a 15-puzzle problem

20.3.2 Solution

There are various techniques of solving the *N*-puzzle problem. The brute force algorithm is one of the most obvious. Considering the fact that there can be at maximum 3 moves, if the empty space is somewhere in between the board and 1, if the empty space is at corner of the board.

If a state space tree is constructed and depth-limited search is applied, then at the *n*th level, a solution can be found. In this case, the complexity would be $O(3^n)$.

The above solution is simply not implementable. Assume that we stop at the 18th level of the tree; the total number of moves comes out to be 1162261466. If we take a processor that can perform 10^3 instructions per second, then the above process takes 1162261 seconds, that is, 13.4 days. Now, if we process the state space tree till the 25th level, then the above processor would take 26.8 years. How many of us would be ready to spare this much part of your life to see the solution?

It is evident from the above discussion that the above problem is an NP-complete problem. This has been proved by Kendall (2005). According to the literature review, in order to solve a 3×3 problem, which is solvable, 0.01 seconds are required, whereas for a 24-puzzle problem, 12 billion years are required.

Those who are familiar with the artificial intelligence can appreciate the use of A* in solving such problems. However, genetic algorithms, described in Chapter 23 can also be of help. The state space tree of a 3×3 board is shown in Fig. 20.8.
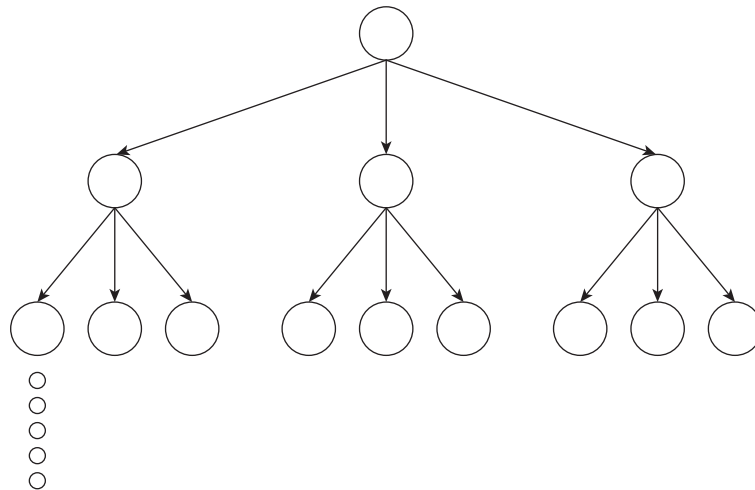
Figure 20.8   State space tree of a 8-puzzle problem

In *N*-puzzle problem, there are total of $N + 1$ tiles that contain distinct numbers and a blank space. These $N + 1$ tiles can result in $(N + 1)!$ initial configurations. Out of these many configurations, only half of the configurations are solvable and others are not. Thus, only $(N + 1)!/2$ initial configurations can lead to goal configuration using limited number of moves.

Given an initial configuration, it can be checked whether the configuration can be solved or not. The steps for determining solvability are as follows:

Step 1 Shift the blank tile at the bottom right corner of the grid. This can be easily done.
Step 2 Calculate permutation inversion for each tile. An inversion is when a tile precedes another tile with a lower number on it.

Let us now come to the formal definition of planning problems. As per Kleinberg, the planning problems can be defined as follows.

> **Planning Problem** Is it possible to apply sequence of operators to get from initial configuration to goal configuration?

20.4 REGULAR EXPRESSIONS

In Chapter 18, the concept of deterministic and non-deterministic finite acceptors (DFA and NFA) was discussed. The language accepted by a DFA or an NFA is a regular language. The regular language stems from a regular expression. A regular expression is defined as follows:

- Any symbol that belongs to Σ; the non-empty, finite, set of symbols is a regular expression.
- If r is a regular expression, then the following are also regular expressions:
  o $r_1 + r_2$
  o $r_1 r_2$
  o r_1^* or for that matter r_2^*
- Any expression that is formed with the help of the above is a regular expression.
  In order to understand the concept, let us consider the following examples.

Illustration 20.1 Design a regular expression over {0, 1}, in which the third symbol is 1 and the fifth is 0.

Solution The first symbol of the expression can be 0 or 1. Same is the case with the second and the fourth symbol. All these can be expressed as (0 + 1), meaning that these symbols can either be 0 or 1. Moreover, there can be any number of symbols after the fourth symbol and they can either be 0 or 1. The rest of the symbols are, therefore, represented by (0 + 1)*. The required expression, therefore, is as follows:

$$(0+1)(0+1)1(0+1)0(0+1)^*$$

Illustration 20.2 Design a regular expression, over {0, 1}, in which there are no consecutive 0's or 1's.

Solution The required expression would either be of the form (0101010101…) or (1010101010…). Therefore, either (01)* or (10)* would generate the expression.

The required expression is, therefore, as follows:

$$(01+10)^*$$

Illustration 20.3 Design a regular expression over {0, 1} wherein the number of 1's are even.

Solution The question is similar to Illustration 18.2 of Chapter 18. Consider the following DFA (Fig. 20.9) which accepts all the strings over {0,1} wherein the number of 1's are even.
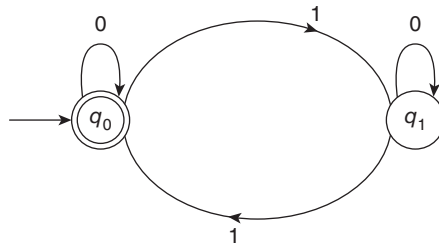
Figure 20.9 A DFA that accepts all the strings in which the number of 1's are even

The regular expression corresponding to the DFA is as follows. Note that there is no constraint as regards the number of 0's. So, we can have any number of zeros at any position,

$$(0^*10^*10^*)^*$$

Illustration 20.4 Design a regular expression over {0, 1} wherein the number of 0's is a multiple of 3's.

Solution Consider the following DFA (Fig. 20.10) which accepts all the strings over {0,1} wherein the number of 0's are a multiple of 3's.

The regular expression corresponding to the DFA is as follows. Note that there is no constraint as regards the number of 1's. So, we can have any number of ones at any position,

$$(1^*01^*01^*01^*)^*$$
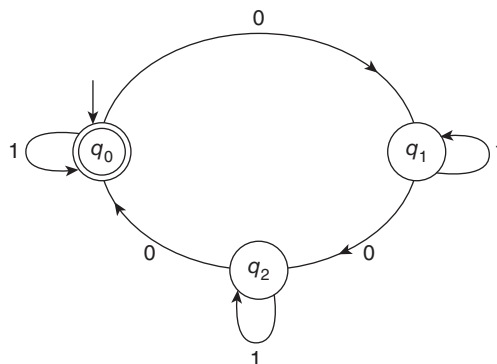
Figure 20.10 A DFA that accepts all the strings in which the number of 1's are even

As stated in Chapter 18, any NFA can be converted into an equivalent DFA. Moreover, a regular expression exists for any NFA. The following figure shows the corresponding NFA for a regular expression. Figure 20.11 shows the basic rules that would help to convert a regular expression to an NFA.
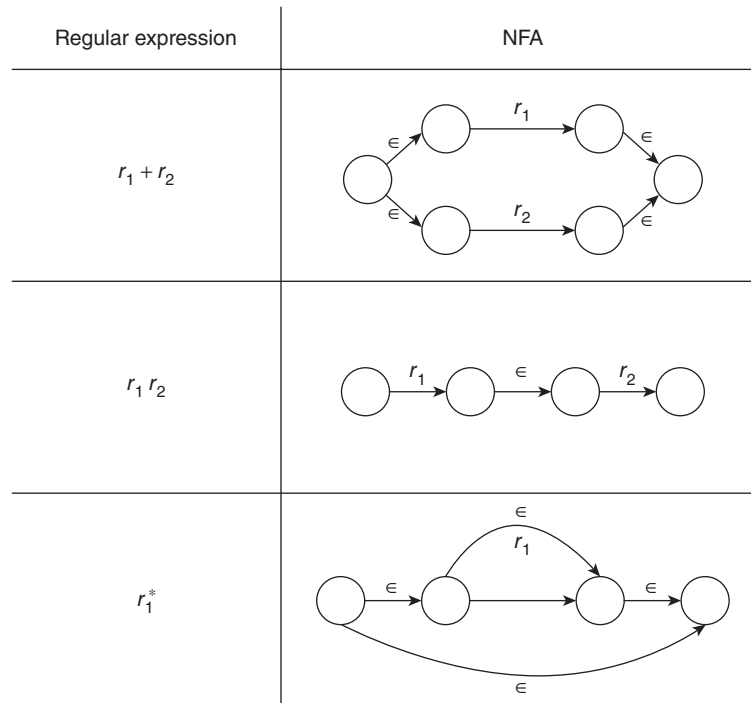
Figure 20.11   Regular expression to NFA

Using the above rules, any regular expression can be easily converted into an NFA. As a matter of fact, this concept is used by the first phase of the compiler also. The first phase of the compiler is called a *lexical analyser*. This phase converts the regular expressions into corresponding NFAs. These NFAs are then converted into DFAs and finally, these DFAs are minimized. This phase of compiler generates tokens that are then used by the second phase to form parse trees. In fact, the concept of regular expressions and finite acceptors are also used in word processors.

Having understood the concept of regular expressions, let us now move to the point of contention. We are provided with a string and a regular expression and we are required to find whether that string is accepted by the given expression or not.

To be able to do that via the brute force approach, one needs to enlist all the possible strings that can be formed by the given regular expression and then check whether the given string is one of those generated by the regular expression.

In order to accomplish the above task, an infinite number of strings will have to be generated and stored. The task will not only require infinite time but also infinite

space. A better option would be to create an NFA and then convert it to a DFA. This technique is computationally better than the previous one.

Having discussed the concept of regular expressions and the formation of finite acceptors from these expressions, it is left for the reader to see whether he can develop an algorithm to check whether the languages accepted by two regular expressions are same.

Try developing the algorithm; you will reach the conclusion that the problem is a PSpace problem.

20.5 CONCLUSION

The chapter discussed the concept of PSpace. The concept is an extension of what has already been discussed in Chapter 19. The reader is therefore requested to go through Chapter 19 before starting with this concept. It is also important to understand the importance of saving space in the design of an algorithm. The reader is advised to go through the works of Stockmeyer and Savage in order to get an insight of QSAT and its relation with PSpace. Adi Shamir also has to his credit a work which proved that the proofs that can be verified in polynomial time are exactly those that can be generated in polynomial space.

Points to Remember

- P is a subset of PSpace
- NP is a subset of PSpace
- PSpace is same as NPSpace

KEY TERMS

PSpace A problem that is solvable by a Turing machine via polynomial space is called a PSpace problem.

QSAT Let f be a CNF formula consisting of n variables $\{x_1, x_2, ...\}$. For what assignments of $\{x_1, x_2, ...\}$, the following formula is true:

$$\exists x_1 \forall x_2 \exists x_3 \forall x_4 ... x_n f$$

Finding the sequence of operations in order to reach final state.

Regular expression
- Any symbol that belongs to Σ, the non-empty, finite, set of symbols, is a regular expression.
- If r is a regular expression then the following are also regular expressions:
  - $r_1 + r_2$
  - $r_1 r_2$
  - r_1^* or for that matter r_2^*
  - Any expression that is formed with the help of the above is a regular expression.

EXERCISES

I. Multiple Choice Questions

1. Which of the following is true?
   (a) $P \subseteq PSpace$
   (b) $NP \subseteq PSpace$
   (c) $PSpace = NPSpace$
   (d) All of the above

2. Which of the following is correct?
   (a) Each problem that requires polynomial time requires polynomial space.
   (b) Each problem that requires polynomial space requires polynomial time.
   (c) Each problem that requires polynomial time requires non-polynomial space.
   (d) None of the above

3. The problem of counting till an exponential number is obtained is
   (a) P
   (b) NP
   (c) PSpace
   (d) None of the above

4. A PSpace-complete problem is
   (a) NPSpace
   (b) PSpace hard
   (c) Both
   (d) None of the above

5. QSAT is
   (a) Only PSpace
   (b) Only PSpace hard not PSpace complete
   (c) Both
   (d) None of the above

6. NPSpace is same as PSpace can be argued owing to
   (a) Savitach's theorem
   (b) Turing theorem
   (c) Both
   (d) None of the above

7. The proofs that can be verified in polynomial time are exactly those that can be generated in polynomial space. This statement is true because of the work by
   (a) Adi Shamir
   (b) Adi Chopra
   (c) Adi Puri
   (d) None of the above

8. If one can increase not decrease the length of a sentence using the given grammatical transformations and find out if the given sentence can be produced by these transformations is a
   (a) P
   (b) PSpace
   (c) NPSpace
   (d) Both b and c

9. The above problem is called
   (a) Word problem
   (b) Sentence problem
   (c) Tense problem
   (d) None of the above

10. If chess is converted into polynomial time, then it would be
    (a) Only PSpace
    (b) PSpace complete
    (c) Both
    (d) None of the above

II. Review Questions

1. Define PSpace problem. Discuss why P is a subset of PSpace.
2. Discuss why PSpace is same as that of NPSpace.
3. What is meant be a PSpace-complete problem?
4. Prove that QSAT is PSpace complete.
5. Define planning problems and discuss why 15-puzzle is PSpace.
6. Search some of the solitaire games that are PSpace complete.

Answers to MCQs

1. (a)	3. (c)	5. (b)	7. (a)	9. (a)
2. (a)	4. (a)	6. (a)	8. (d)	10. (c)

CHAPTER 21

Approximation Algorithms

OBJECTIVES

After studying this chapter, the reader will be able to

- Understand the concept of approximation algorithms
- Explain the idea of ρ-approximation algorithms
- Use approximation algorithms to solve problems such as load balancing, vertex cover, and subset cover
- Learn the difference between heuristic and approximation algorithms
- Use linear programming in approximation algorithms

21.1 INTRODUCTION

In the previous chapters, the terms NP, NP-hard, and NP-complete have been discussed and explained. The chapter on NP also discussed the problems whose polynomial time algorithms have either not been found or are only valid for a subset of the inputs. The present chapter would help us to tackle such problems. The way of handling such problems is the same as that used in daily life.

Imagine that it is the first day of your job and you have been entrusted upon the responsibility of finding the solution to a very hard problem. Now the problem is that if you refuse to do the work, your job would be in danger. On the other hand, the problem is so hard that you are almost sure that you would not get an exact solution. In such cases, what will you do? Either you will take a lot of time to find the solution of the above problem, thus endangering your job. Or you will find an approximate solution to that problem in a limited time. There is another possibility, though. You might come up with a solution that works correctly in almost all the cases. The last type of solution is not that reliable. The reason is that you might solve those instances of the problem that come in your mind at that point in time. So, you might end up solving a small subset of the problem at hand, not the problem itself. Therefore, it is better to be able to develop an algorithm that gives an approximate solution to the problem every time, rather than giving a 'good' solution in some of the cases. Once you start using your algorithm for an unknown case, it is difficult to know whether the solution which you have got is 'good'

or not. So, your algorithm might be giving you 'good' solution only in some of the cases, while in most of the cases it may give a 'not good' solution.

The previous chapters have discussed the concept of NP-hard problems. In such problems also, the same issue prevails. There is no polynomial time algorithm to solve such problems. At the same time, these algorithms are too important to be ignored. So, there must be a way to solve these problems 'approximately' or to be able to develop an approximation algorithm for such problems. Some of the clues that might help us in our journey of finding approximate solutions to the NP-hard problems will be unfolded in the following discussion.

If the input domain of NP-hard problems is small, then they can be handled. Moreover, if we are able to find a sub-exponential algorithm for the above problems, even then the time would be considerably reduced as compared to a conventional exponential algorithm. The concept of reducibility, discussed in the earlier chapters, might also help us to tackle such problems. In other cases, it is practically impossible to find an optimal solution, so it is advisable to adopt a method which guarantees a solution that is near to the optimal one. It is definitely difficult to argue about the closeness to an optimal solution when it is not possible to find it. This chapter would give an idea so as to how to tackle this issue.

> **Approximation Algorithms** These are the algorithms which find an approximate solution to computationally hard problems.

The techniques studied so far such as greedy and dynamic approach would also help in finding the approximation solution to a problem. The approximation methods would use one of the following techniques:

- *Greedy approach*: The decision at a particular point is taken considering profit or loss at that time.
- *Dynamic programming*: Produce better results than the greedy algorithms.
- *Linear programming*: An introduction has been given in this chapter. The technique has also been applied to vertex cover problem. However, Appendix A4 of this book explores this technique.

Before proceeding further, one is expected to revise the following concepts and terms. The idea of 'optimal solution' introduced in Section 19.1, the concept of 'feasible solution' given in Section 19.1, and the list of NP-hard problems given in Section 19.4 are required to proceed further. Before proceeding further, one is expected to shed the inhibitions developed in the mind owing to our study of this course up to now. Here, there are some problems, such as maximum clique problem, for which good approximation algorithms are not known.

The chapter is organized as follows. Section 21.2 of the chapter introduces the basic terms. Section 21.2 introduces taxonomy. Section 21.4 of the chapter presents an approximate solution to the vertex cover problem, Section 21.6 of this chapter discusses

approximation algorithms, and the last section presents the conclusions. The chapter is sure to change the way one looks at the theory of algorithms.

21.2 TAXONOMY

The convention followed in the chapter is same as that in Cormen [1]. The terms optimal solution, feasible solution, and cost have same meanings as that discussed in the previous chapters. A solution that satisfies all the given constraints is referred to as a feasible solution. An optimal solution is one which is good. The meaning of ρ can be inferred from the following inequality:

- The problem would be denoted by P
- The instance of P would be denoted by I
- The cost of the optimal solution of P would be denoted by C^*
- The cost of the solution obtained by the algorithm would be denoted by $C(I)$

The following constraints govern the system:

$$\text{maximum}\left(\frac{C(I)}{C^*}, \frac{C^*}{C(I)}\right) \leq \rho, \text{ where } \rho \text{ is the maximum value of the ratio of the optimal}$$

and the effective cost.

There are two types of optimization problems—maximization and minimization. For the maximization problems,

$$\rho \leq 1$$

For the minimization problems,

$$\rho \geq 1$$

The importance and concept of these terms have been discussed in the following sections. The chapter discusses the concept of ρ-approximation problems. The application of this concept tells us how close the approximation algorithm is to the optimal solution (C^*). For example, the optimal solution of a 2-approximation algorithm would be at maximum 2 times better than the approximation algorithm. As proved in Section 21.4, the approximate solution to a vertex cover problem is a 2-approximation. It means that the optimal solution will have at maximum double the number of vertices as compared to that obtained using approximation algorithm. In the same way, the optimal solution of a 3-approximation algorithm (say of an instance I' of a travelling salesman problem (P)) has at maximum 3 times the number of edges as compared to the optimal solution. The goodness of an approximation algorithm has been discussed along with the corresponding algorithms.

21.3 APPROXIMATION ALGORITHM FOR LOAD BALANCING

This section discusses the load balancing problem and its solution using an approximation algorithm.

Problem: There are m processors and n processes. The processes are to be assigned to the processors in a way that none of the processors is either overburdened or idle. If the time taken by the first job is t_1, that by the second is t_2, and so on. The time required to complete a task by the ith machine would be denoted by T_i. The value of T_i would be the sums of t_j's, which are the task allocated to the machine i, that is,

$$T_i = \sum_{\substack{j \in \text{ jobs that have been} \\ \text{assigned to the machine}}} t_j$$

The makespan is defined as the maximum of T_i's. In order to achieve the task, we start with all the processors idle. We then take one job at a time and assign it to the least occupied processors. The approach is a greedy one as we are trying to minimize the makespan.

The above problem is referred to as load balancing as the aim is to balance the loads assigned to the processors.

> **Load Balancing Problem** The value of maximum T_i is to be minimized, that is minimize (maximum T_i), where T_i is the time taken by a machine to complete all the jobs that have been assigned to processor i.

The algorithm for the above procedure is as follows.

Algorithm 21.1 Greedy approximate load balancing

Input: m, the number of processors; n, the number of processes; the array $t = \{t_1, t_2, ...\}$, containing the times required by the ith process to complete.
Output: A 2-dimensional matrix D, wherein the ith row depicts the ith machine and the values in that row are the processes that have been assigned to that machine.

```
{
for(i=0; i<m; i++)
        {
        T  = 0;
         i
        ith row of matrix D = NULL;
        }
for (j = 0; j<n; j++)
        {
        Find the processor which has minimum T ;
                                                j
        Assign the task, j to the processor i.
        T  = T  + t ;
         j    j    i
        D[i][ ] = D[i][ ]∪{Job };
                                i
        }
return D;
}
```

Complexity: Consider the second loop. The loop has a search procedure in between, the worst case complexity of the above algorithm would therefore be $O(nm)$.

Correctness of this approach: The approach does not always give correct results. The results are near to the optimal, which would be proved in the discussion that follows.

Theorem 21.1

$$\text{Optimal Makespan} \geq \frac{1}{m} \sum_{\forall j} t_j$$

Proof The total time taken to complete all the tasks, had there been just one processor, would have been $\sum_{\forall j} t_j$. The balanced division of all the tasks, to m processors would result in average time $\frac{1}{m} \sum_{\forall j} t_j$. Since makespan is the maximum time taken by any of the processors, the makespan would be greater than (or equal to) $\frac{1}{m} \sum_{\forall j} t_j$. The above argument can be summarized as follows.

p = The balanced division of all the tasks, to m processors makes the average time $\frac{1}{m} \sum_{\forall j} t_j$.

q = Makespan is the maximum time taken by any of the processors.

r = Makespan is greater than average,

$$q \rightarrow r$$

Substituting $\frac{1}{m} \sum_{\forall j} t_j$ for average (by p), we get

$$\text{Makespan} \geq \frac{1}{m} \sum_{\forall j} t_j$$

Theorem 21.2 The time taken by the above algorithm is less than twice the optimal timespan.

Proof Consider a situation in which the last job leads to a situation wherein the scene becomes the most optimal. The removal of that job would lead to a situation wherein the timespan is less than the optimal timespan, that is,

$$T_i - t_i \leq \text{Optimal } T$$

$$T_i \leq t_i + \text{Optimal } T$$

$$\text{Since, } t_i \leq \text{Optimal } T$$

$$T_i \leq 2 \times \text{Optimal } T$$

Scope of improvement: Had the jobs been arranged in ascending order of time, the greedy algorithm can be altered to make the timespan $\leq 3/2$ (optimal time).

21.4 VERTEX COVER PROBLEM

The vertex cover problem (VCP) selects the set of vertices V' from a graph $G = (V, E)$, such that $V' \subseteq V$ and $\forall (x, y) \in E$, either $x \in V'$ or $y \in V'$. It may be noted that even $V \subseteq V$ and $\forall (x, y) \in E$ either $x \in V$ and $y \in V$. So, even V is the vertex cover of a given set. Although it is easy to find the vertex cover of a given graph, it is surely difficult to find the minimum vertex cover of a set. The problem has already been discussed in Chapter 19. The following method gives a new overview of the problem.

The following method selects minimal set of vertices (approximately) which satisfy the above property.

21.4.1 Vertex Cover Problem Using Approximation Algorithm

The solution of the VCP using approximation algorithm has been explained using the graph shown in Fig. 21.1
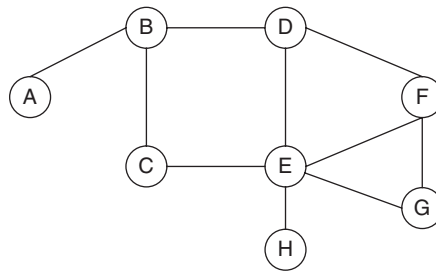
Figure 21.1 Graph 1

The following sections discusses two approaches for the VCP. The first approach is that followed in Cormen and the second approach is a modified version of the former, which adds a greedy paradigm alongwith. The input to this problem is a graph $G = (V, E)$, where V is the set of vertices and E is the set of edges.

21.4.2 Cormen Approximation Approach

The following procedure uses S, the temporary set which stores the edges that can be selected in the next step. The set R stores the selected edges and the set F gives the set of vertices.

Step 1 Initialize the set S with E. Randomly select an edge (say, BD), remove it from the set S, and put it in the set R. The set R now becomes {(B, D)} and the set S becomes S = {(A, B), (B, C), (D, E), (D, F), (C, E), (E, F), (E, G), (E, H)} (F, G) (Fig. 21.2(a)).
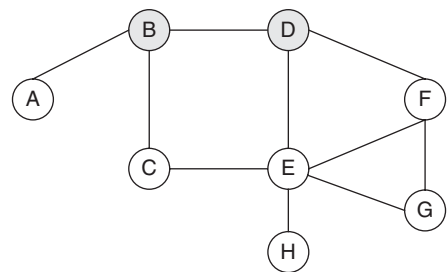
Figure 21.2 (a) Step 1, approximation algorithm for VCP

Step 2 The edges, except for the selected edge, adjacent to the vertices selected, are removed from the set S. Now, the set S becomes {(C, E), (E, F), (E, G), (E, H)} (Fig. 21.2(b)).
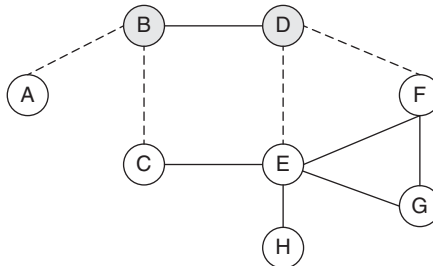
Figure 21.2 (b)   Step 2,
approximation algorithm for VCP

Step 3 In the next step, one of the edges from the set S is selected and the procedure stated in Steps 1 and 2 are repeated. Now, the set S becomes {(G, F)} (Fig. 21.2(c)).
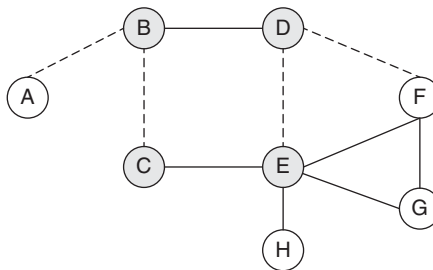
Figure 21.2 (c)   Step 3, approximation
algorithm for VCP

Step 4 In the next step, the edge GF is selected (Fig. 21.2(d)). It may be observed that if the vertices contained in the selected edges are selected, the vertex cover of the given graph would be obtained.
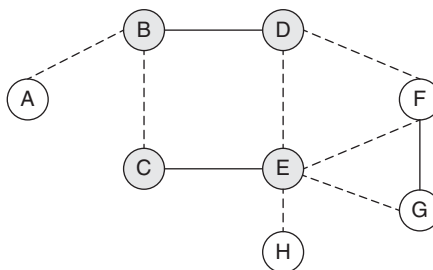
Figure 21.2 (d)   Step 4,
approximation algorithm for VCP

After Step 4, the set R becomes $\{(B, D), (C, E), (F, G)\}$. The set F, therefore, becomes, $\{B, D, C, E, F, G\}$ (Fig. 21.2(e)). The goal, in this case, was to minimize the number of elements in the set R. The length of the vertex cover obtained by this algorithm is 6. However, it can be seen that the optimal vertex cover would have value 3. So the value of $\dfrac{\tilde{c}}{c^*}$ becomes $\dfrac{1}{2}$.
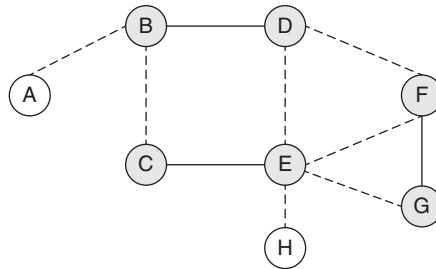
Figure 21.2 (e) Final answer

The other approach explained in this section is a blend of approximation and greedy approach. During an extensive literature review, it was found that clubbing the greedy paradigm with the approximation approach is an acceptable approach. Cormen has followed this approach in the set cover problem and in many other problems.

The approach arranges the vertices in order of the vertices. The set S is initialized to E. The vertex with the highest order is selected first. This is followed by removing all its adjacent edges from the set S. Then the vertex with the highest order from the set S (the set containing remaining vertices) is selected. The procedure has been presented in the following algorithm.

21.4.3 Modified Vertex Cover

Input: $G = (V, E)$, G is the graph having vertices stored in the set V and edges stored in the vertices E.

- Arrange the vertices V such that if V_i precedes V_j, then the order of V_i is more than that of V_j.
- Initialize S to E.
- Select V_0. Remove all the edges adjacent to V_0 from set S.
- From the remaining vertices (considering the set S), select the first element of the set V and repeat the procedure stated in the previous step.

The application of the above algorithm on graph 1, shown in Fig. 21.1, has been depicted in Figs 21.3(a)–(d).
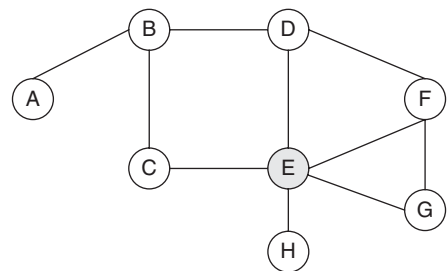
Figure 21.3 (a) Step 1, modified approximation algorithm for VCP
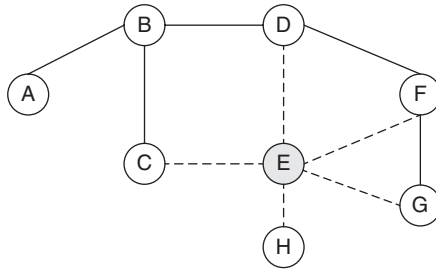
Figure 21.3 (b) Step 2, modified approximation algorithm for VCP, *E* is the first element of the set *V*
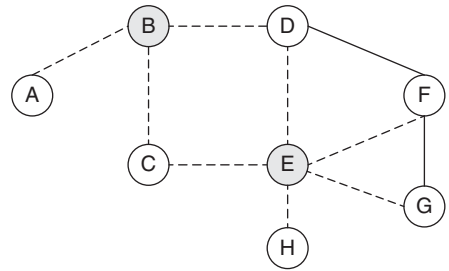
Figure 21.3 (c) Step 3, modified approximation algorithm for VCP, B, D, and F are the elements in the set *V* having highest order, *B* is selected
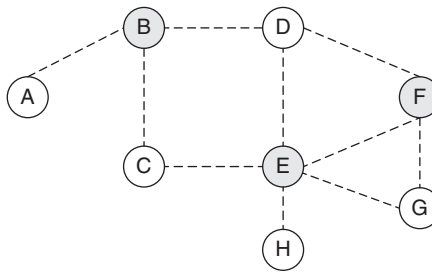
Figure 21.3 (d) Step 4, vertex cover obtained from the modified vertex cover algorithm

In the last step, the vertex D is selected. The vertex cover of the set, therefore, becomes {B, E, D}.

Hence, the modified vertex cover performs better than its previous counterpart. However, it might not be the case every time. What can be stated, though, is that the former will in no case be better than the modified one, provided that the given graph is not a bipartate. It can be safely stated that the above algorithm finds a near-optimal vertex cover for a graph.

The running time of the former is $O(E + V)$ and that of latter is $O(E \ln E)$. The former appears better than the latter, but the cost-effectiveness of the latter is better than the former in many cases.

21.5 SET COVER PROBLEM

Similar to the previous problem, the set cover problem is an NP-hard problem. The optimal solution to the problem though can be found but it takes an exponential time. In this case also, we relax the condition of finding out the optimal solution and instead settle for a near-optimal solution.

The approach used by most of the researchers is a blend of the approximation and the greedy paradigm. The approach is as follows. In the discussion that follows, U is the universal set.

21.5.1 Greedy Approach for Approximate Set Cover

Initially, the result set would be empty. At any point we select a set, from amongst the remaining sets, whose elements when taken out, minimizes the remaining U that is the universal set. The elements of the selected set are then taken out from the universal set and included in the set C, which was initially empty. The process is continued till the set C becomes equal to the initial universal set. Finally, the set C is returned. It is easy to see that the complexity of the algorithm is a quadratic one ($O(n^2)$). However, a linear algorithm for the above is also possible.

Illustration 21.1 Apply approximate set cover to the following problem:

Solution

$$U = \{2,3,5,7,9,12,14,18,20,21,25,27\}$$

$$S1 = \{2,3,5,7\}$$

$$S2 = \{9,12\}$$

$$S3 = \{2,3,9,12,18,20,14\}$$

$$S4 = \{27\}$$

$$S5 = \{7\}$$

$$S6 = \{3,5,7,18,20,27,25\}$$

$$S7 = \{14,21,25\}$$

Table 21.1 Cardinality of a set

Set	Cardinality
S1	4
S2	2
S3	7
S4	1
S5	1
S6	7
S7	3

The cardinality of various sets is as follows (Table 21.1).

According to the above approach, the set $S3$ or $S6$ should be taken out first, as they have maximum number of elements. Let us take $S3$. The set U, after taking out the elements of the set $S3$, becomes {5, 7, 27, 21, 25}. Now selecting $S6$ would make the remaining U minimal. The set F, set of selected sets, becomes {$S3$, $S6$} and the set U after taking out the elements of the set $S6$ becomes {21}. In the next step, the set $S7$ would be selected, as it is the only set containing 21. The set cover of the above problem is, therefore, {$S3$, $S6$, $S7$}.

Another version of the problem has been discussed as follows. The version has weights associated with each set.

21.5.2 Subset Cover (Sets with Weights Associated with Them)

Consider a set of sets $\{S_1, S_2, ..., S_3\}$. The number of elements in a set S_i is denoted as $|S_i|$. The weight associated with a set S_i is given by w_i. The aim is to select a set of sets S_i, which has maximum $|S_i|$ and minimum weight.

That is, $\dfrac{|\cup S_i|}{\sum w_i}$ is minimum.

The above problem is an NP-hard problem. In order to obtain the correct answer, all possible combination of sets will have to be crafted. This would be followed by the calculation of the ratio of the total cost and the number of elements. The combination that has the least value would then be selected.

Since the above procedure would be computationally very expensive, let us pick the sets which have a small value of $\dfrac{|S_i|}{w_i}$. This approach, though gives better results in some cases, but would fail to produce correct answers all the times. The algorithm to calculate the set cover, using approximation approach is as follows.

Algorithm 21.2 Approximation algorithm for subset cover

Input: Set of sets S, consisting of $\{S_1, S_2, \ldots, S_3\}$ their weights w_i and the number of elements in each set.
Output: The set of sets which will have the minimum total weight and maximum number of elements.

```
{
        S = φ;
        While ()
        {
        Select the set, S_j, which has minimum |S_i|/∑w_i;

        S = S ∪ S_j;

        }
return S_j.
}
```

Complexity: The complexity of the above algorithm is $O(n) \times O$ of the algorithm which finds the set having minimum value of $\dfrac{|S_i|}{\sum w_i}$ from the remaining sets.

The algorithm produces good results. However, for some of the cases, the algorithm fails to find the optimal result.

21.6 ρ -APPROXIMATION ALGORITHMS

One of the major differences between the approximation algorithms and heuristics is that the approximation algorithms guarantee that the algorithm is at least ρ times the optimal solution. An obvious question that arises is whether there is any way of knowing the value of ρ, when we do not know the optimal solution? The answer is yes.

Though heuristic algorithms may give us a better solution, there is no way of knowing so as to how good the solution is. This is not the case in approximation algorithms. The approximation algorithms find solution which is near to the optimal solution and the factor to which the solution is near to can be found. The idea is reflected in the above discussion. Usually, this constant factor is very low. For example, if it takes 10^{32} seconds to find the exact solution of a certain travelling salesman problem, whereas it takes only 10^{15} seconds to find a near-optimal solution (constant factor being 5%) using an approximation algorithm, then one should go for an approximate algorithm (if there is no need for exact solution). For the approximation algorithm that solves the VCP, this constant factor is 2. This section exemplifies so as to how the upper bound of the solution can be found.

It was already stated that there are two types of optimization problems: maximization and minimization. Let us revisit the definition of both of them, with respect to approximation algorithms.

> **Tip:** The heuristic algorithms find a reasonable solution in a reasonable time, the approximation algorithms find solution up to a constant factor.

Maximization problems A maximization problem is one in which we need to maximize the solution. Example of such an algorithm is maximum clique problem (MCP), in which we try to find a complete sub-graph of a given graph. In such algorithms, the value of $\rho < 1$.
Minimization problems A minimization problem is one in which we need to minimize the solution. Example of such an algorithm is travelling salesman problem, in which we find the minimum cost Hamiltonian cycle is to be found. In such algorithms, the value of $\rho > 1$.

An algorithm for which one can find out the value of ρ is referred to as a ρ approximation algorithm. For instance, the load balancing problem, explained in the following discussion, is a 2-approximation algorithm.

The various classes of problems have been shown in Fig. 21.4.

21.6.1 Load Balancing Problem Using 2-Approximation Algorithm

There are n jobs to be executed on m machines. The time of execution of a job is given, say t_i. Each machine is assigned some jobs and the total execution time on each machine is, say T_j. The problem is to distribute the jobs on the given machines so that the makespan, that is, the time by which all the jobs are finished, is minimum. It may be stated here that a job can be executed on a single machine.
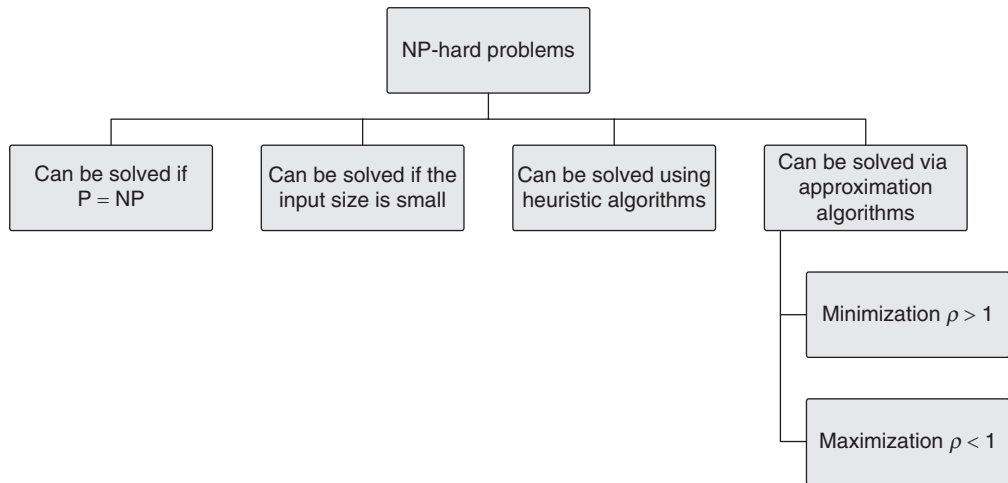
Figure 21.4 Solution of NP-hard problems

One of the algorithms that have been proposed to accomplish the above task requires a particular job to be given to a machine that has minimum load. Since the number of jobs are n and each time a job has to be assigned, the machine with the minimum load needs to be found; therefore, the running time of the algorithms becomes $O(n \log n)$.

Theorem 21.3 Load balancing is a 2-approximation algorithm.

Proof The problem is similar to that discussed in Section 21.3. Suppose at any instance M_i is a machine with the minimum load. Now the machine is assigned a job having execution time t_k before which the load of the machine was T_i. Let LB be the lower bound.

$$T_{\text{after assigning job}} = t_k + T_i$$

$$\leq t_k + \text{LB}$$

$$\leq \text{LB} + \text{LB}$$

$$\leq 2\text{LB}$$

Therefore, the problem is a 2-approximation problem.

There is another approach to solve the load balancing problem. The approach described above is based on greedy paradigm. The other approach can be found in TU Eindhoven, *Advanced Algorithms* (2IL 45 Course Notes). It is left to the reader to prove that the algorithm is a (1/3) approximation algorithm.

21.6.2 Travelling Salesman Problem

The travelling salesman problem (TSP) can be stated as follows.

If a graph $G = (V, E)$ is given, wherein the weight of an edge from v_i to v_j (v_i and $v_j \in V$) is given by w_{ij}. The aim is to find a path $v_1 v_{i1} v_{i2} \ldots v_1$ where $v_{ij} \neq v_1$, such that the net cost of the path is minimum and all the v_{ij}'s are distinct.

The problem has been discussed earlier in many chapters. The problem can be handled using the approximation approach by using the triangles inequality. The reader is advised to design an approximation algorithm for TSP and prove that the algorithm is a 2-approximation algorithm. The solution to this problem has been included as a project in the web resources of this book. The reader is advised to go through the resources.

The TSP can be handled by minimizing the cost of most costly edge cycle.

21.7 USE OF LINEAR PROGRAMMING IN APPROXIMATION ALGORITHMS

The section discusses the use of linear programming in approximation algorithms. The example of vertex cover has been taken to explain the concept. The following discussion briefly revises the VCP, introduces its weighed version, and applies linear programming to solve the problem.

VCP Using Linear Programming

The VCP selects a set of vertices V', from a given graph $G = (V, E)$, such that if $x \in V'$, then for all $e \in E$, ($e = (a, b)$), $x = a$ or $x = b$.

The minimum VCP is a variant of the VCP, wherein the vertices of the given graph have weights associated with them. The goal of the problem is to find a subset of the set of vertices such that each edge in the set E has at least one of the vertices in the set V'. In addition, the sum of the weights of the vertices in the set V' should be as low as possible. Note that the word minimum is deliberately not used in the previous sentence as it would make the problem NP-hard. Moreover, the phrase 'as low as possible' suits the idea of approximation algorithms.

The solution discussed in this chapter uses linear programming discussed in Appendix A4. The solution is, in fact, one of the most promising ways of solving problems using approximation algorithms.

The weighted VCP can be described as follows (henceforth would be referred to as integer linear programming for weighted VCP, ILPVCP).

Minimize $\sum f(x) \times w_i$, subject to

$$x + y \geq 1, \text{ for all } e \in E, e = (x, y)$$

$$x = \{0,1\}, \text{ for all } x \in V'$$

The function f can be defined as follows:

$$f = \begin{cases} 1, \text{if } x \in V' \\ 0, \text{if } x \notin V' \end{cases}$$

The above is an NP-hard problem. The problem can be relaxed by relaxing the condition that x is either 0 or 1. The following version of the problem, henceforth referred to as relaxed liner programming for minimum weight vertex cover problem (RLPVCP), has the condition (i.e., x either 0 or 1).

Minimize $\sum f(x) \times w_i$, subject to

$$x + y \geq 1, \text{ for all } e \in E, e = (x, y)$$
$$x = [0,1], \text{ for all } x \in V'$$

However, any solution of ILPVCP would always satisfy RLPVCP. As a matter of fact the lower bound of RLPVCP would be the solution of the ILPVCP. The procedure for solution can be summarized in the following algorithm.

Algorithm 21.3 Relaxed linear programming for VCP

Input: Graph $G = (V, E)$ with W, having weights of the vertices
Output: Set V′, $V' \subseteq V$

```
{
//Initially the set V′ is null
V′ = φ ;
Solve for V′ using Linear Programming, wherein
One intends to minimize ∑f(x) × w_i, subject to
x + y ≥ 1, for all e ∈ E, e = (x, y);
x = [0, 1], for all x ∈ V′;
for all, x ∈ V′
{
if (x ≥ 1/2)
        {
        V′ = V′ ∪ {x};
        }
}
return V′;
}
```

Complexity: Although there is a loop (for all x belongs to V'), the complexity of the algorithm cannot be considered as $O(n)$. The reason is the inclusion of linear programming

solution in the algorithm. The implementation of the linear programming part would decide the complexity of the algorithm.

> **Tip:** The vertex cover problem using relaxed linear programming is a 2-approximation algorithm.

21.8 CONCLUSION

The chapter introduces the concept of approximation algorithms. The algorithms are used to find the approximate solution of the NP-hard problems. The algorithms are better than heuristics as they guarantee at least ρ times the optimal solution. The concept has been explained by taking examples of vertex cover and subset sum problem. The fact that it is not difficult to prove that a given approximation algorithm is a ρ approximation has been exemplified by the load balancing problem. The chapter introduces the topic and explains the concept. Though researchers are trying hard to come up with good approximation algorithms, for almost all the problems given in the chapter, it is not possible to include each and every approach in the text. The readers are requested to visit the requisite journals for an in-depth coverage.

Points to Remember

- Heuristic algorithms do not tell us anything regarding the goodness of the solution.
- The approximation algorithms help us in judging the nearness of the solution obtained.
- The value of ρ for minimization problem is greater than 1.
- The value of ρ for maximization problem is less than 1.

KEY TERMS

Approximation algorithms These are the algorithms that find approximate solution to computationally hard problems.

Load balancing The problem assigns m processors to n processes, in a way that none of the processors is either overburdened or idle. If T_i is the time taken by the machine i to complete the tasks, the value of timespan which is the maximum of T_i's is to be minimized.

Maximization problems A maximization problem is one in which we need to maximize the value of the desired variable.

Minimization problems A minimization problem is one in which we need to minimize the value of the desired variable.

Minimum vertex cover problem The minimum vertex cover problem is a variant of the vertex cover problem, wherein the vertices of the given graph have weights associated with them. The goal of the problem is to find a subset of the set of vertices such that each edge in the set E has at least one of the vertices in the set V'. In addition, the sum of the weights of the vertices in the set V' should be as low as possible.

Subset sum with weights associated with each set Consider a set of sets $\{S_1, S_2, ..., S_3\}$. The number of elements in a set S_i is denoted as $|S_i|$. The weight associated with a set S_i is given by w_i. The aim is to select a set of sets which has maximum number of elements and minimum weight.

EXERCISES

I. Multiple Choice Questions

1. Which of the following can give a 'good solution' to an NP-hard problem?
   - (a) Heuristic
   - (b) Approximation
   - (c) Both of the above
   - (d) None of the above

2. Which of the following can guarantee that the solution is at least ρ times as good as the optimal solution?
   - (a) Heuristic
   - (b) Approximation
   - (c) Both of the above
   - (d) None of the above

3. Which of the following is a 2-approximation algorithm?
   - (a) Greedy load balancing
   - (b) Travelling salesman with triangular inequality
   - (c) Both
   - (d) None of the above

4. The greedy (approximation) vertex cover problem is a
   - (a) 2-approximation problem
   - (b) 3-approximation problem
   - (c) Both
   - (d) None of the above

5. Which of the following are the types of optimization algorithms?
   - (a) Minimization
   - (b) Maximization
   - (c) Both
   - (d) None of the above

6. Which type of approximation algorithm is the greedy set cover, given in the chapter?
   - (a) 2-approximation
   - (b) 3-approximation
   - (c) 5-approximation
   - (d) None of the above

7. For which type of problems are approximation algorithms used?
   - (a) NP-hard
   - (b) P
   - (c) Both
   - (d) None of the above

8. Which type of approximation algorithm is the max-3CNF (approximation)?
   - (a) 2-approximation
   - (b) 3-approximation
   - (c) 5-approximation
   - (d) None of the above

9. Which of the following approaches can be used to solve TSP?
   - (a) Approximation
   - (b) Randomization
   - (c) Heuristics
   - (d) All of the above

10. In which of the following approximation algorithms is not needed?
    - (a) Searching
    - (b) Subset sum
    - (c) TSP
    - (d) Maximum clique

II. Review Questions

1. Prove that MAX-3-CNF is a 3-approximation algorithm.
2. Give an approximation algorithm for maximum clique.
3. Prove that approximate vertex cover (using greedy) is a polynomial time 2-approximation algorithm.
4. Prove that if the set E and V of a graph G are given, then the complexity of the modified vertex cover is $O(E \ln E)$.
5. Prove that the modified vertex cover will be at least as good as approximate vertex cover. Otherwise, give an example to contradict the statement.
6. Prove that greedy set cover is a $O(\ln |U| + 1)$ time algorithm.
7. Design a greedy set cover algorithm that is better than that given in Section 21.4.
8. Use problem reduction approach to design a greedy algorithm that solves subset sum problem using set cover problem.
9. Prove that approximate vertex cover using linear programming is 2-approximation algorithm.
10. Prove that approximate vertex cover is 3-approximate algorithm.
11. Differentiate between the weighted vertex cover and vertex cover algorithms.
12. Explain the algorithm for weighted vertex cover using approximation.
13. Explain three approaches for approximation algorithm.
14. Explain the algorithm for weighted subset problem.
15. Explain load balancing using approximation algorithm.

Answers to MCQs

1. (c)	3. (b)	5. (c)	7. (a)	9. (d)
2. (b)	4. (a)	6. (a)	8. (a)	10. (a)

CHAPTER 22

Parallel Algorithms

OBJECTIVES

After studying this chapter, the reader will be able to

- Learn about the various generations of computers
- Understand the concept of parallel processing
- Define Moore's law
- Classify the types of parallel processing
- Explain the PRAM model
- Understand broadcast algorithm
- Learn the concept, algorithm, and implementation of prefix and pointer jumping
- Find the maximum value and minimum value from a given list.

22.1 INTRODUCTION

The development of computing can be attributed to both the advancement in hardware and that in software. This development is aptly depicted by the five generations of computers. The computing devices belonging to a newer generation of computer are generally more powerful, consume less power, and take less space than the previous one.

This chapter discusses the concepts of parallel processing. The concept of parallel processing was introduced in the third generation computers. The chapter has been organized as follows. Section 22.2 discusses the various generations of computers. Section 22.3 deals with parallel computers. The basics have been discussed in Section 22.4. Section 22.5 discusses the PRAM model. Section 22.6 deals with finding the maximum element. Section 22.7 deals with prefix computation. Section 22.8 discusses the merge procedure. Section 22.9 introduces hypercube model and their corresponding algorithms. The last section concludes.

22.2 GENERATIONS OF COMPUTERS

The first computer, ENIAC (Electronic Numerical Integrator and Calculator), was a huge, most inefficient (as per von Neumann), and did not have a good architecture to boast

about. The first computer based on the stored program concept was IAS. (It was built by engineers at the Institute of Advanced Studies and hence the name.) The above developments started, what is now referred to as the first generation computers. The first generation computers used vacuum tubes. The flip side of using vacuum tubes was the fragility and the inapt size associated with them. Machine-level language was being used at that point in time. The computers were single user. The fact that they performed the I/O using CPU made them all the more inefficient.

The second generation computers were based on transistors. The computers also supported floating-point arithmetic. The period of second generation computers also saw the rise of languages such as FORTRAN and COBOL. The examples of computers of second generation include UNIVAC and IBM 7030.

The third generation used integrated circuits (ICs). This generation used microprogramming pipelining. The introduction of cache has also been attributed to this generation. The concept of virtual memory was also introduced in this generation. Examples of this generation of computers include IBM 360, etc.

The fourth generation computers are generally associated with semiconductor memory. The computers of this generation used LSI and MSI technology. The advent of the fourth generation computers saw the rise of parallel computing, in which many calculations could be carried out in parallel. Various techniques were invented and developed to make use of parallelism. The examples of computers of this generation include IBM 3090. This generation used very large integrated circuits (VLSI) technology.

The fifth generation computers were associated with the concept of superscalar processors and cluster computing. Computers of this generation could perform at the speed of teraflops. (Teraflops is a measure of computing speed equal to one trillion floating-point operations per second.)

Firstly, we would discuss the concept of multiprocessors. If we have more than one processor at our disposal, then the task at hand can be divided and given to different processors for computations. In such cases, the cost of computation is the product of the execution time and the number of processors. The concept is similar to the calculation of man-hours in a software project. Suppose one has employed two employees to complete a project and they promise to finish the task in a month's time. By employing 10 employees he can complete the same task in a shorter time. Though mathematics leads us to believe that the same task would require 6 days if 10 people work on it, this may not be completely true. This is because the division of a task in smaller parts, solving each part and clubbing the result, requires time. So the approach does not only require 'divide' but also 'conquer'.

If it takes t_1 time to complete a task by sequential processing, and time t_2 is required to complete the task via parallel processing, the total time in the case of parallel computation is proportional to t_2 and n, where n is the number of processors.

A type of parallel algorithm in which the ratio of the costs of solving a problem on a multiprocessor to that on a single processor is constant is referred to as *cost-optimal*.

During an extensive literature review, it was found that this concept has been widely used to compare the algorithms.

If the costs of combining the results of the tasks performed by the different processors are negligible and so is the cost of splitting the algorithm into multiple parallel tasks, then the complexity can simply be found by dividing the number of steps in the worst case by the number of processors.

22.3 PARALLEL COMPUTERS

The need for faster computing compelled the conception and development of parallel computers. The implementation of this concept has not only helped in the increase in productivity but also lower costs and effective solutions of complex problems. There are numerous ways of implementing parallelism. These are multiprogramming, multithreading, multitasking, etc. These techniques are implemented at the OS and the programming level.

The speed of a processor is generally very high. If just one process executes at a time, then the time during which the process is doing its input–output tasks, the processor would be idle. In order to overcome this problem, many processes are executed simultaneously. The processor executes a particular process for some time and then move on to the next. The order in which these processes would be executed by the processor and the time for which they are executed is decided by the processor scheduling technique employed by the operating system. The above is referred to as multiprocessing.

A process may further be divided into many independent sub-processes. These sub-processes referred to as *threads* do not share resources and hence, can be executed in parallel. The threads of a single process do not get separate memory space. This concept of multithreading is the common feature of languages such as JAVA and C#. Multitasking generally refers to doing many tasks in parallel by a system. The concept of pipelining discussed in the following sections would help to understand the concept of pipelining.

The above methods of implementing parallelism is not the only way of implementation of parallel processing. It can also be implemented via hardware-oriented techniques. The following discussion throws light on the physical architecture of parallel computers. The chapter presents the models such as parallel random access machines.

Parallelism, as stated earlier, can be implemented in many ways. The functional parallelism is implemented either by having more than one functional unit in a system or by a concept called *pipelining*.

Pipelining is, again, of many types. However, the core idea behind it can be understood by the following example. Imagine a simple computer, which fetches the instruction, decodes it, and then executes it. If there are 10 such instructions and each task takes one unit of time (though fetch, decode, and execute would take different amounts of time, for the sake of simplicity the times are assumed to be equal), the total time required would be $3 \times 10 = 30$ units.

On the other hand, if a computer is developed which while decoding the first instruction, fetches the second one and while executing the first one fetches the third and decodes the second (Fig. 22.1). The whole process would take only 12 units of time in order to execute 10 instructions.

Fetch	Decode	Execute									
	Fetch	Decode	Execute								
		Fetch	Decode	Execute							
			Fetch	Decode	Execute						
				Fetch	Decode	Execute					
					Fetch	Decode	Execute				
						Fetch	Decode	Execute			
							Fetch	Decode	Execute		
								Fetch	Decode	Execute	
									Fetch	Decode	Execute
1	2	3	4	5	6	7	8	9	10	11	12
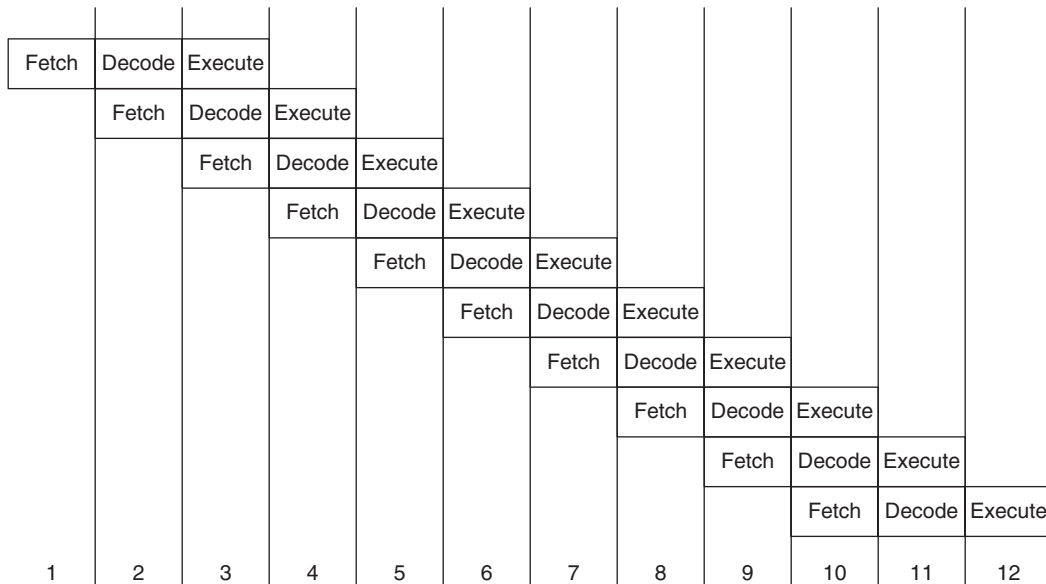
Figure 22.1 Pipelining

The following discussion throws some light on the classification of parallel processors. Computer architecture comprises both hardware organization including hardware components (such as central processor units, databases, and caches) and requisite software requirements (opcodes, registers, etc., in the different instruction sets) that are adequate for an assembly/language programmer. We initiated our discussion with von Neumann architectural model which was built as a sequential machine for executing subsequent scalar data. Although over a period of time, operating phenomenon certainly improvised from bit serial to word parallel operations. Similarly, it improves from fixed-point to floating-point operations. But apparently this type of architectural model was strategically less efficient since it facilitated sequential execution of instructions in a program.

Execution of scalar data could be achieved

(a) By sequential method
(b) By look-ahead method

It was initially introduced to prefetch instruction overlapped *I/E* (instruction fetch/decode and execute) operations and to perform functional parallelism. Functional parallelism can be implemented by using multiple functional units simultaneously or by

practising pipelining at various levels (instruction execution, arithmetic computation, and memory access operation). Pipelining performs identical operations repeatedly over vector data strings. Vector operations were carried out implicitly by software control looping using scalar pipeline processors.

Explicit vector instructions originated with the introduction of vector processors. The two families of pipeline vector processors are

(a) **Memory to memory architecture**: Supports the pipelined flow of vector operands directly from the memory to pipelines and then back to the memory.

(b) **Register to register architecture**: Supports vector register to interface between the memory and functional pipelines. Register to register architecture is classified under Flynn's classification as SIMD and MIMD.

SIMD: Vector computers equipped with scalar and vector hardware appear as single instruction multiple data machines.

MIMD: Intrinsic parallel computers execute programs in multiple instructions multiple data mode. Sharing memory multiprocessor and message passing multicomputers are two major classes of parallel computers.

Multiprocessors and multicomputers are distinguished on the basis of memory sharing and interprocessor communication mechanism. Multiprocessors communicate through shared variable in a common memory, whereas multicomputer communicates through message passing among the nodes.

22.4 BASICS

Gordon E. Moore, a co-founder of Intel, in one of his papers predicted that the processing speed of computers would roughly double in every 2 years. Interestingly, the duration was reduced to 18 months in 1975. However, the basic idea remains the same. On the face of it, the law does not suffice. The average speed of a computer was around 8 MHz in 1970s. The average speed in 2009 was in the range of 1.3–2.8 GHz. The following figure shows the variation of speed of a processor and the time had Moore's law been true, in terms of computation power. However, the law still holds if we consider the number of transistors instead of speed. According to 'www.mooreslaw.org' 'in the year 2000 the number of transistors in the CPU numbered 37.5 million, while in 2009, the number went up to an outstanding 904 million'. So the law is more or less valid if we consider the number of transistors and not the speed of a processor. The concept has been depicted in Fig. 22.2.

The statement of Moore's law is as follows.

Moore's Law Moore's law states that the number of transistors in a dense integrated circuit doubles every 2 years.
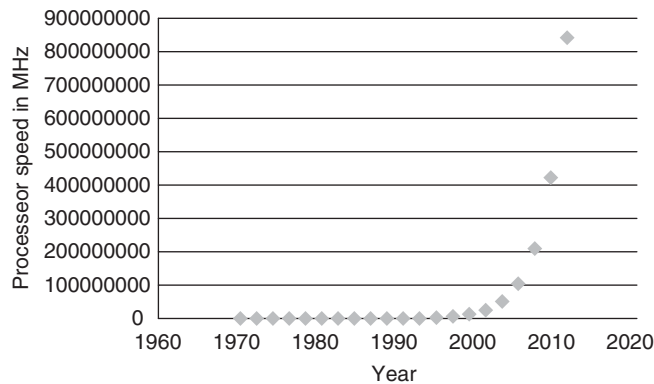
Figure 22.2   The variation of speed of a processor and time had Moore's law been true

However, there is a flip side. There comes a time when saturation occurs. So instead of focussing on doubling the speed of a processor every 2 years, the focus has now shifted to getting an equivalent amount of work, as discussed in this chapter. Even in the present context, the manufacturers sell dual core, quad core, etc. So there is a tendency to fill in more and more cores. This would be beneficial if we are able to use these cores as well. Since the advance at the hardware level is commendable, the algorithm developers should be able to make full use of the above. Here comes the need of parallel computing. This chapter discusses the models and applications of parallel computing. Interestingly, these computers are used not only because of their exceptional computational power, but also because of their ability to fetch data at a faster pace. This is because if there is more than one processor, they would be able to access memory in a parallel fashion.

22.5 PARALLEL RANDOM ACCESS MACHINE

The standard random access machine model (SRAM) has one processor, where various operations such as load and store takes unit time. In contrast, the parallel RAM (PRAM) model contains many processors and each processor can have its own memory. But there is a shared memory. The processors can access the shared memory. The input and the output are stored in a common shared memory. Moreover, the model is synchronous, owing to a common clock. The working of this model can be depicted by Fig. 22.3.
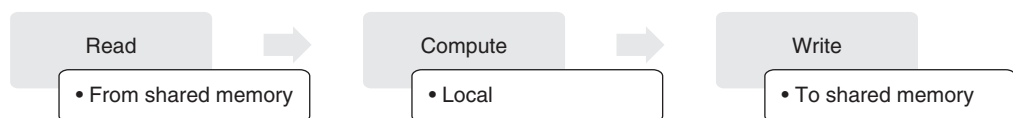
Figure 22.3   Working of PRAM

PRAM is just one of the many parallel architecture models. Generally, when memory is associated with each processor, then NUMA model is referred to. The shared memory models are, as a matter of fact, more expensive but according to many researchers they are good from the programmers' point of view.

The model works as follows. The processors do not, per se, communicate directly with each other. The communication, if it has to occur, happens owing to a common memory. One processor writes in the common memory and the other reads. The model is depicted in Fig. 22.4.

Nodes with private memories
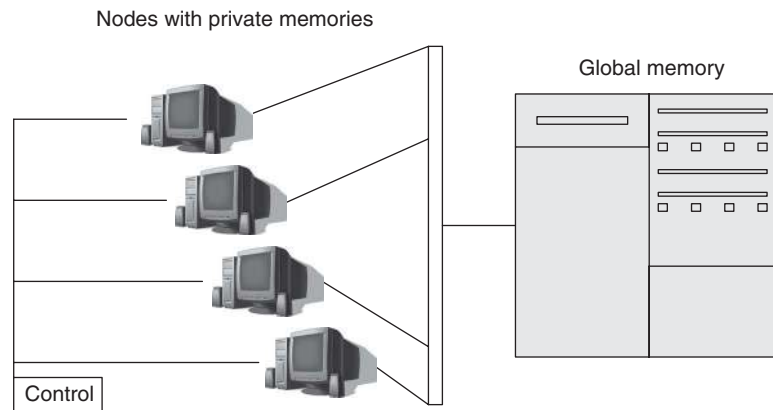
Figure 22.4 The PRAM model

The read and write can occur in one of the following ways.

- Exclusive read exclusive write (EREW)
- Exclusive read concurrent write (ERCW)
- Concurrent read exclusive write (CREW)
- Concurrent read concurrent write (CRCW)

The names of the above techniques are self-explanatory, more so if we are familiar with basics of operating systems or with database management system. The last model is the ideal one; however, it is most difficult to implement. The first is closest to implementation.

The contentious point is the concurrent write. Here, the system may employ one of the three policies. The first is priority CRCW, wherein if more than one process intends to write, then the processor with a higher priority is allowed to write. In the other variation which is arbitrary CRCW, the processor that is allowed to write is chosen randomly. If, however, each processor intends to write the same value to the common shared memory, then they are allowed to do so. This is called common CRCW.

In order to make the PRAM algorithms more effective, the number of processors selected should be $\dfrac{n}{\log n}$. This is followed by a local phase. The local phase precedes the global phase. The time for local phase would be $O(\log n)$. As a matter of fact, the time for global phase would also be same. However, the computations done by ($n/\log n$) processors would be input to some other algorithms, say, tournament algorithms.

If the value of *n* is 128, then (*n*/log *n*) becomes (128/8), which is 32. So 32 processors would be used by dividing the 128 values into 32 parts each having 8 items. The local phase would deal with their 8 elements and the global phase would take the 32 results produced by different processors.

22.6 FINDING MAXIMUM NUMBER FROM A GIVEN SET

Finding maximum or minimum from a list of elements is a $O(n)$ task, if the list is not sorted. However, this is true for a system having a single processor. For a multiprocessor system, the complexity would be much less. The following discussion throws some light on the algorithms for finding maximum in such systems.

22.6.1 Using CRCW

In order to find the maximum from a series of *n* numbers, the following procedure can be employed. The procedure requires n^2 processors. Each processor checks the *i*th (*i*th row) and the *j*th number (*j*th column). If the element at the *j*th column is greater than that at the *i*th row, a '1' is inserted at the cell; otherwise a '0' is inserted. This step requires $O(1)$ time. In the next step, the values in the cells would be seen. The last column via which result would be interpreted would be initialized to 1. If a '1' is found in the row, then the corresponding value in the last column becomes '0'. The cell having a '1' in the last column would be the maximum value. The procedure has been depicted in Algorithm 22.1.

Algorithm 22.1 Parallel_Maximum (num[], n)

Input: A 1-dimensional array of numbers num, the number of items in the list n.

```
Parallel_Maximum(num [], n)
    {
    create a 2D array having elements of a[] both at rows and columns
    for (i=1; i<=n; i++)
        {
        for(j=1; j<=n; j++)
            {
            if(a[0, i] > a[i, 0])
                {
                a[i, j]=1;
                }
            }
        }
    for(i=1; i<=n ; i++)
        {
        a[i, n+1]=0;
        }
for (i=1; i<=n; i++)
        {
            flag=0
        for(j=1; j<=n; j++)
```

```
                {
                if(a[i, j] ==1)
                    {
                    flag=0
                    }
                }
                if(flag==1)
                    {
                    a[i, n+1]=1;
                    }
            }
    for(i=1; i<=n ;i++)
            {
            if(a[i, n+1]==1)
                {
                print "maximum";
                }
            }
    }
```

Illustration 22.1 Find the maximum number using Algorithm 22.1.

Solution Example: num[ ] = {2, 5, 9, 4, 3}

                n = 5;

             a[ ]=

	2	5	9	4	3	
2	0	1	1	1	1	0
5	0	0	1	0	0	0
9	0	0	0	0	0	1
4	0	1	1	0	0	0
3	0	1	1	1	0	0

Maximum: 9.

22.6.2 Using EREW

The maximum from a list num [ ] having n elements can also be found using the following procedure that does not require n^2 processors. However, in this case, the processor needs to know whether it is the left or the right child of a node. In addition, the processor must be aware of its left and the right child. The procedure is simple. The processors are arranged in a tree structure as shown in Fig. 22.5. One element is given to each processor present as leaves. For each processor at the last but one level (denoted by index i), if the element with the processor $(2i +1)$ is greater than that at $(2i + 2)$, then the element in the variable temp is same as that at $(2i + 1)$; otherwise it is same as that at $(2i + 2)$. Finally, the value at temp is compared with that at the processor i. Out of temp and the element with the ith processor, whichever is greater is stored with the ith processor.
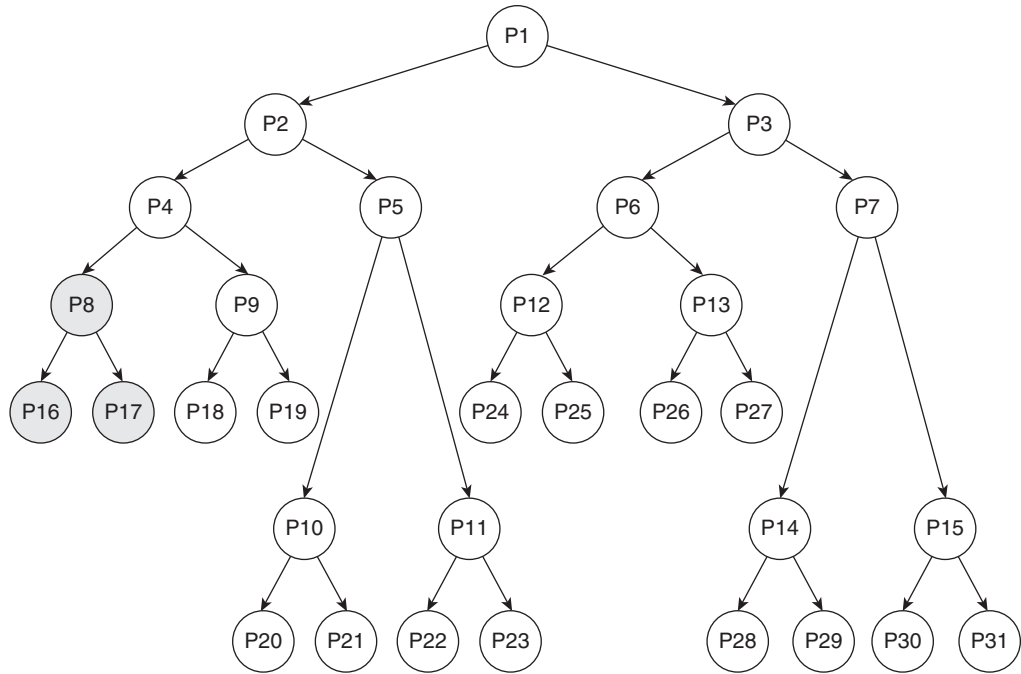
If the element with P16 is greater than that at P17, then the element at P16 would be stored in temp, otherwise that at P17 would be stored in the variable temp. The value of temp is then compared with the value at the ith processor; whichever is greater is stored with the ith processor.

Same is the case with P4 and P1

Figure 22.5 Finding the maximum from a list

The number of processors required in this case is n. However, the complexity would not be $O(1)$.

22.7 PREFIX COMPUTATION

If a set of elements A in a set R is given, the set A is $\{x_1, x_2, x_3, ..., x_n\}$. The computation of the set prefix involves the evaluation of $\{x_1, x_1 op x_2, x_1 op x_3 op x_2, ..., x_1 op x_2 op...x_n\}$, where op is an associate operator, that is, $\{x_1 op(x_2 op x_3) = (x_1 op x_2)op x_3\}$. Moreover, it has been assumed that op is closed under R, that is, $x_1 op x_2 \in R$. The elements of the resultant set are called prefixes. For the sake of simplicity, it has been assumed that $x_1 op x_2$ requires a single unit of time.

A brute force algorithm that uses a single processor would take $\sum_{i=1}^{n} 1 \times i = O(n^2)$ time. A better algorithm would take $O(n)$ time. However, the use of parallel algorithm would make the things much better. Let us see how.

Divide the input set into log n sets. For example, a set having 16 elements can be divided into 4 sets of 4 elements each. A set of 256 elements would be divided into 8 sets of 32 elements each and so on.

This is followed by the calculation of prefix of each set. The last element of the results can be stored in the global memory. This last element is operated as per the given operator with each element of the next set.

22.8 MERGE

The merge takes two sorted lists as input and produces a sorted list. The PRAM-based algorithm for merge works as follows. The given array is divided into two parts. For example, in Fig. 22.6, the array X has been divided into two parts X1 and X2.
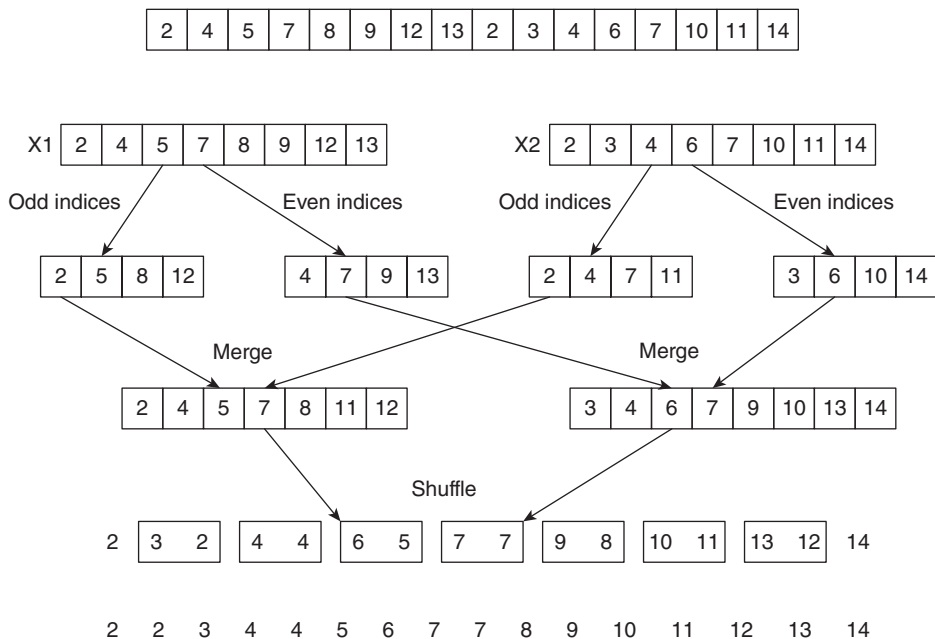
Figure 22.6 Merge using PRAM

This is followed by the following procedure:

Step 1 Separate the elements at odd places from the elements at the even places, for both X1 and X2.

Step 2 Merge the two arrays containing elements placed at odd indices.

Step 3 Do the same for arrays containing elements placed at even indices.

Step 4 Merge the 2 arrays obtained in Steps 2 and 3.

Step 5 Leaving the first and last element, make pairs of all the remaining elements.

Step 6 Write the first elements as it is, then check whether the first element of the pair is less than the second element, if true, write it to the final answer, if no, then swap the two elements and continue this to obtain the final answer.

22.9 HYPERCUBE ALGORITHMS

In a multiprocessor system, there is more than one processor. The interconnection of these processors can be directed by a hypercube. That is, a hypercube refers to the arrangement of processors. The dimension of a hypercube determines the maximum number of links required to send a message via the processors. A hypercube of dimension n would have 2^n processors.

The node of a hypercube would have processors. The numbering of vertices of a hypercube would be as follows. The vertices of the hypercube would be 0 and 1 in the case of a hypercube of dimension 1. The vertices of the hypercube, having dimension 2 would be 00, 01, 10, and 11. In the case of a hypercube with dimension three, the vertices would be 000, 001, 010, 011, 100, 101, 110, and 111. A hypercube of dimension n would have an n bit binary number associated with each processor.

The length of the path from one processor to another is given by the hamming distance of the binary numbers associated with the two processors. For example, the length of the path between two processors having code 101 and 110 would be 2. Hence, the hamming distance can easily be found by counting the number of 1's in the 'XOR'ing of two numbers.

As stated earlier, a hypercube of dimension 1 would have two processors, which can be identified by 0 and 1. The bits indicating the processors of a hypercube of order two can be easily created by prefixing the above numbers with 0 and then with 1. This would result in 00, 01, 10, and 11. Similarly, the hypercube of dimension 3 would have processors, which can be identified by the numbers. Figure 22.7 depicts hypercubes of various dimensions.
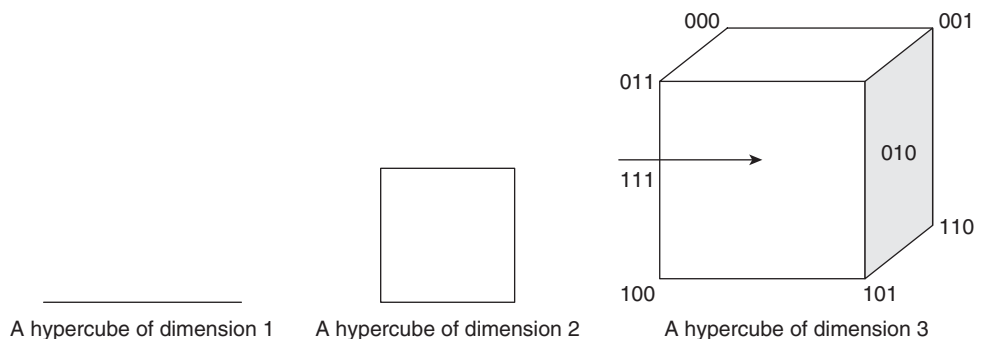
Figure 22.7 Hypercube of order n

In the discussion that follows, we assume that any processor in a hypercube has everything required for a basic computation, a memory, a CPU, and main memory. The depiction of this autonomous system is given in Fig. 22.8. Since a processor in a hypercube is connected to only two other processors, there might arise cases wherein the message is intended for a processor which is not an immediate neighbour of the sender. In such cases, the time in communication would be same as the length of the path. The length of the path is maximum n. Therefore, the delay in communication can be at maximum n.
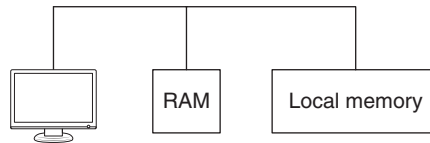
Figure 22.8 A processor in a hypercube

The hypercube structure can be used to implement parallel processing in the following ways.

22.9.1 Broadcasting

A hypercube of dimension 'd' takes $O(d)$ time to accomplish basic operations such as broadcasting and prefix computation. The algorithms, therefore, become optimal as compared to others.

At times it becomes essential to send a message from one processor to a group of processors. This group is a subset of the set of processors in the given hypercube. The broadcasting is accomplished as follows.

In order to broadcast a message, the following procedure is adopted.

Algorithm 22.2 Broadcast

```
BROADCAST (Message M)
    PTR is a Pointer to a node
    1. SET PTR = ROOT.
    2. if (PTR has children)
        {
        a. Make two copies of M received and send it to the children;
        b. Each child on receiving the copy of M becomes PTR and repeats step 2.
        }
```

Complexity: $O(d)$. Since it is a hypercube algorithm, the complexity would be $O(d)$. The process is depicted in Fig. 22.9.
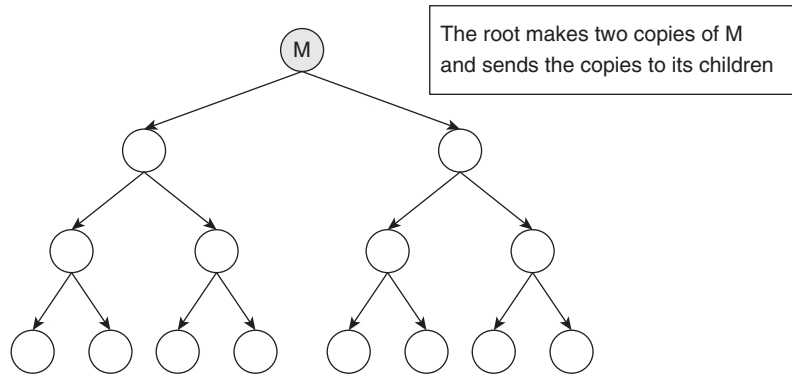
Figure 22.9(a) The root receives M
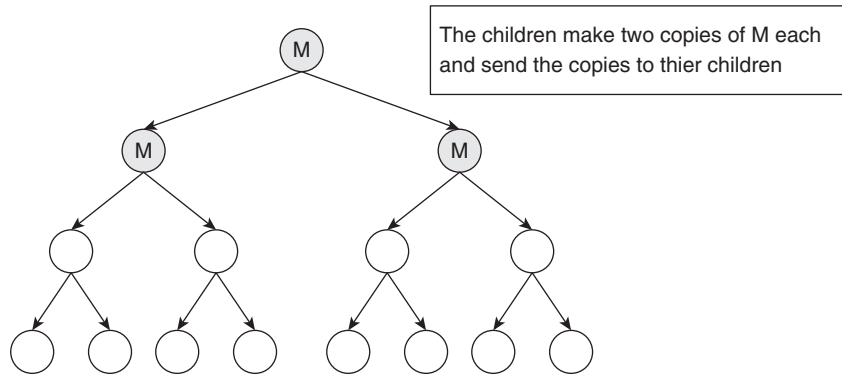
Figure 22.9(b) The root sends two copies of M to its children
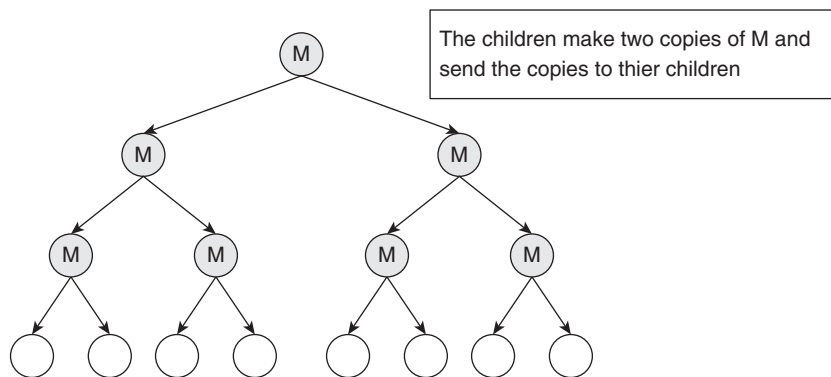
Figure 22.9(c) The children of nodes make two copies each of M and
send them to their children

Finally, M reaches all the leaves and the process stops.
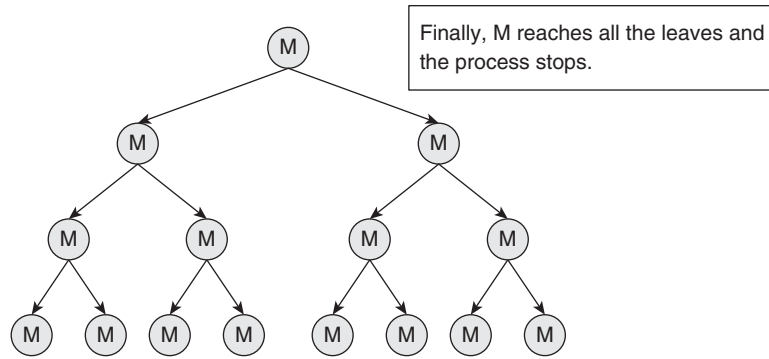
Figure 22.9(d) M reaches the leaves and the algorithm stops

22.9.2 Prefix Computation Using Hypercube Algorithm

The prefix computation can also be accomplished using hypercube algorithms. The elements of the given list are stored in the leaves of the corresponding tree. The prefix is computed as follows.

There are two phases in prefix computation.

In the first phase, a non-terminal node at the last but one level receives values from its children. The values received from the left and the right child are, say, 'a' and 'b'. The node, after receiving the values from its children, calculates their sum and retains the value of the left child. The process continues till the node. Figures 22.10(a)–(d) depict the steps of the first phase. The second phase of the procedure is as follows. The root sends 'a' to the left child and 'b' to the right child. The child, when receives data (say 'c') from its parent sends the data and the sum of the 'b' and 'c' to the right child. Figure 22.10(e) depicts the resultant tree.

When a node receives 'c' from its parent, then it calculates the sum of 'a' and 'c' and stores it as the result.
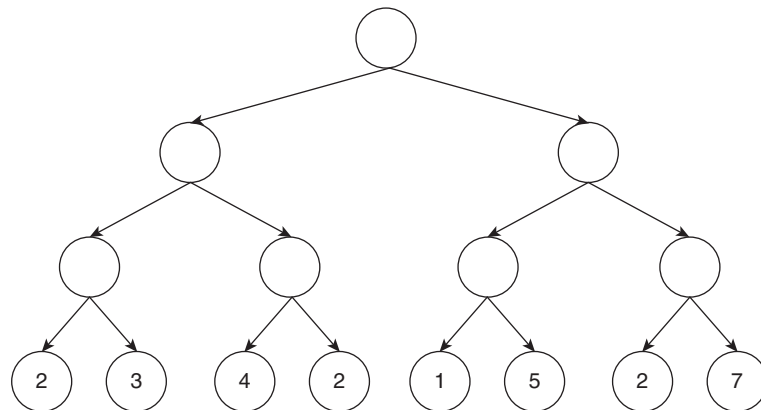
Figure 22.10(a) The process starts from the leaves
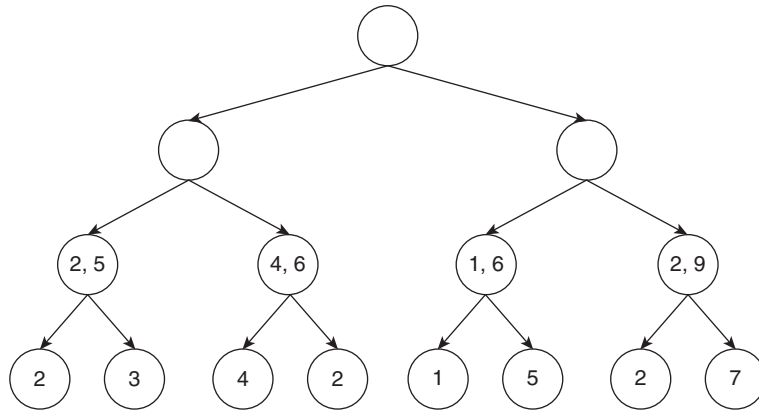
Figure 22.10(b) The element at the left along with the sum of the elements at the left and the right are written at the node (one level up)
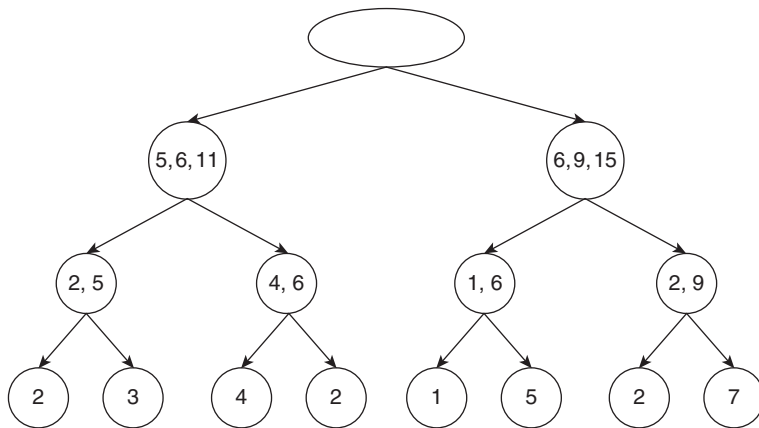
Figure 22.10(c) The process continues for the next higher level
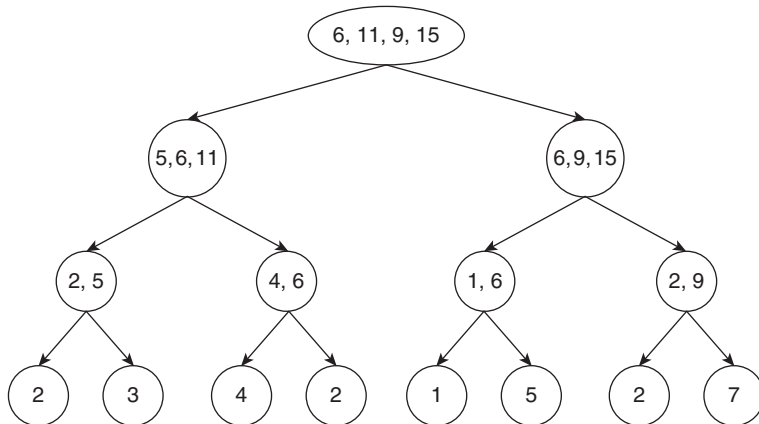
Figure 22.10(d) The process continues for the next higher level
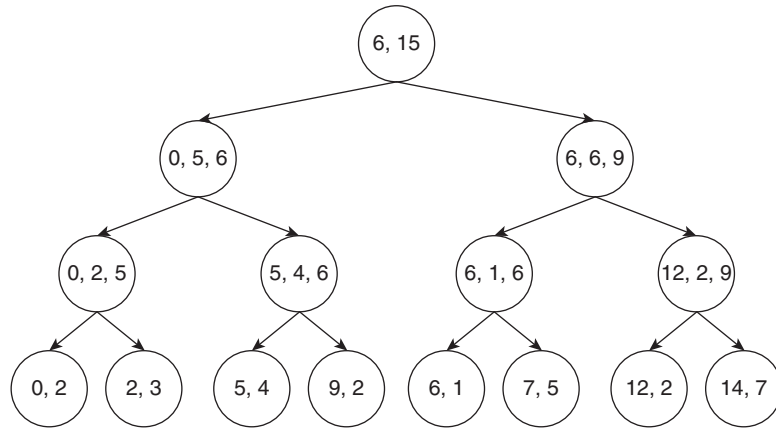
Figure 22.10(e) The final values are substituted at the root

22.10 CONCLUSION

The chapter deals with the concepts, algorithms, and implementations of parallel algorithms. The basics of parallel processing, the evolution of computers, etc., have been discussed in the chapter. The idea of PRAM and the hypercube are the important topics discussed in the chapter. Finding maximum element from a given list, calculating prefixes, etc., have already been discussed in the chapter. However, the model can also be used to calculate ranks (for hint refer to Fig. 22.11). Note that, initially, the data part in all the nodes, except for the last is 1. This indicates that the nodes are connected to a node immediately after them. This is followed by jumping the pointers by one. However, this is not possible for the last and the last but one node. So their data part remains the same. The distance between the nodes in the next step becomes four, and finally the data part of the nodes in the last but one step is added to the nodes to which they are connected. Finally, the answer is obtained.
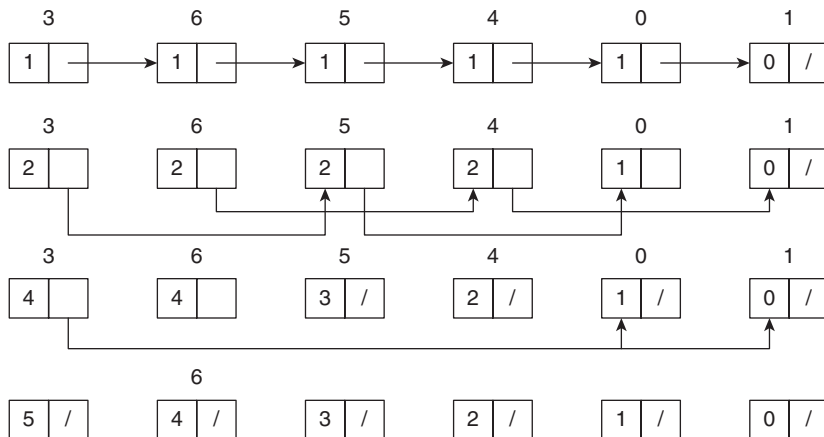
Figure 22.11 Pointer jumping

It is left for the reader to work out the algorithm for pointer jumping and list ranking based on hypercube.

Points to Remember

- The complexity of an algorithm can be found by dividing the number of steps in the worst case of the algorithm by the number of processors.
- The numerous ways of implementing parallelism are multiprogramming, multithreading, multitasking, etc.
- The threads of a single process do not get separate memory space.
- The evolution of computer can be divided into five generations.
- The function parallelism is implemented either by having more than one functional unit in a system or by a concept called pipelining.
- The standard ram model (SRAM) has one processor.
- The read and write in a parallel random access machine (PRAM) can occur in one of the following ways:
  - Exclusive read exclusive write (EREW)
  - Exclusive read concurrent write (ERCW)
  - Concurrent read exclusive write (CREW)
  - Concurrent read concurrent write (CRCW)
- In order to make the PRAM algorithms more effective, the number of processors selected should be $\dfrac{n}{\log n}$.

KEY TERMS

Arbitrary CRCW In arbitrary CRCW, the processor that is allowed to write is chosen randomly.

Common CRCW In common CRCW, each processor intends to write the same value to the common shared value.

Cost-optimal A type of parallel algorithm in which the ratio of the costs of solving a problem on a multiprocessor to that on a single processor is constant is referred to as cost-optimal.

Moore's law Moore's law states that the number of transistors in a dense integrated circuit (IC) doubles every 2 years.

Priority CRCW In priority CRCW, if more than one process intends to write, then the processor with a higher priority is allowed to write.

Threads A process may further be divided into many sub-processes. These sub-processes are referred to as threads that do not share resources and hence can be executed in parallel.

EXERCISES

I. Multiple Choice Questions

1. Which of the following is commonly attributed to the first generation computers?
   - (a) Vacuum tubes
   - (b) Transistors
   - (c) Integrated circuits
   - (d) None of the above

2. Which of the following is commonly attributed to the second generation computers?
   - (a) Vacuum tubes
   - (b) Transistors
   - (c) Integrated circuits
   - (d) None of the above

3. Which of the following is commonly attributed to the third generation computers?
   - (a) Vacuum tubes
   - (b) Transistors
   - (c) Integrated circuits
   - (d) None of the above

4. Which of the following was the first computer?
   - (a) ENIAC
   - (b) IAS
   - (c) IBM 360
   - (d) None of the above

5. Which of the following was the first computer based on the stored program concept?
   - (a) ENIAC
   - (b) IAS
   - (c) IBM 360
   - (d) None of the above

6. Which generation of computers is credited with bringing forth the concept of virtual memory?
   - (a) First
   - (b) Second
   - (c) Third
   - (d) Fourth

7. The superscalar processors and cluster computing is generally associated with which of the following generations?
   - (a) Second
   - (b) Third
   - (c) Fourth
   - (d) Fifth

8. In a PRAM model, which of the following is essential?
   - (a) A common memory
   - (b) Private memory
   - (c) Both
   - (d) None of the above

9. What is SRAM in context of parallel computing?
   - (a) Standard RAM
   - (b) Static RAM
   - (c) Sober RAM
   - (d) None of the above

10. What is PRAM in context of parallel computing?
    - (a) Parallel RAM
    - (b) Processing RAM
    - (c) PRE RAM
    - (d) None of the above

II. Review Questions

1. Write a brief about the various generations of computers. The note should throw some light on the software technology used in those generations. In addition, give at least one example of each.

2. Differentiate between multiprocessing and multiprocessor.

3. Differentiate between multiprocessing and multithreading.
4. Explain the various classifications of parallel computers.
5. What is pipelining? Explain with the help of an example.
6. What are the various types of pipelining?
7. State and explain Moore's law.
8. Discuss the PRAM model and a problem that can be solved using PRAM.

III. Application-based Questions

1. Write an algorithm to find the maximum number from a given list using the PRAM model.
2. Write an algorithm to find the minimum number from a given list using a PRAM processor.
3. Write an algorithm to compute prefixes using PRAM model.
4. Write an algorithm explaining pointer jumping using the PRAM model.
5. Write an algorithm to compute prefix using the hypercube.
6. Write an algorithm to compute maximum using hypercube.

Answers to MCQs

1. (a)	3. (c)	5. (b)	7. (d)	9. (a)
2. (b)	4. (a)	6. (d)	8. (a)	10. (a)

CHAPTER 23

Introduction to Machine Learning Approaches

OBJECTIVES

After studying this chapter, the reader will be able to

- Understand the importance of artificial intelligence
- Give the concept and types of machine learning
- Understand the basics of neural networks
- Explain the process of genetic algorithms
- Understand various operators such as crossover, mutation, and selection
- Apply genetic algorithm to subset sum problem
- Explore various methods of solving TSP using genetic algorithms
- Apply genetic algorithm to vertex cover and maximum clique

23.1 INTRODUCTION

Solving any problem (as introduced in Chapters 12 and 13) is basically a search for the correct solution from amongst the various possible permutations. For instance, there are four cities and travelling salesman problem is to be applied to find out the shortest Hamiltonian cycle. In order to find out the solution, first of all, we will generate all possible permutations of the set $\{1, 2, 3, 4\}$. There would be $4 \times 3 \times 2 \times 1 = 24$ such permutations. These permutations will then be checked for feasibility. The inability to form paths owing to absence of edge between two vertices will make the generated path infeasible. This step would be followed by the evaluation of net cost of each of the feasible paths. Thereafter, the solution will be reached.

The chapter has been organized as follows. Section 23.2 introduces the concept of artificial intelligence while Section 23.3 explains the concept of machine learning. Section 23.4 introduces neural networks and Section 23.5 throws light on genetic algorithms. Section 23.6 explains the application of genetic algorithms on the knapsack problem and subset sum problem.

23.2 ARTIFICIAL INTELLIGENCE

Imagine what happens if we are making a software for a GPS company and the number of cities to be covered is 100. The number of permutations in that case would be $O(100!)$.

The feasibility of the path and the evaluation of cost of each path will still require more resources. The process of finding out the correct solution in such cases can be a tedious task and at times even impossible.

In such cases, the artificial intelligence (AI)-based techniques such as genetic algorithms come to our rescue. Hence, first of all let us define artificial intelligence.

> **Artificial Intelligence (AI)** It may be defined as the capability of a computer to perform those activities that are usually done via manual intelligence.

Now, in order to understand the definition, it would be essential to figure out what all can be done via human intelligence. A person can be considered intelligent if
- *He understands his environment*: This means that a person should possess scientific temperament, that is, the understanding of nature and natural phenomenon.
- *He understands himself*: The individual should be able to comprehend and control his actions. In order to do so, he needs to understand the factors affecting his actions and in that way can be resourceful to the society.
- *He understands that every fact or theory can be nullified and works with that understanding*: This, according to some, is the highest order of intelligence. This implies accepting that everything including varied theories are inferences drawn to satisfy the human curiosity.

AI intends to replicate the above behaviour. Some of the tasks that can be performed using AI are
- Humdrum tasks
- Theorem proofing

The most difficult to implement from amongst the above are the mundane tasks. The tasks that are very easy for us become very difficult for a machine. However, those that are difficult for us sometimes are easy for a computer. For example, it is easy for a computer to perform very large calculations but it is difficult for a machine to communicate with a person. Conventional computing has helped us in the computational tasks. However, until the development of AI, it was considered implausible to perform the mundane tasks via machine.

Hence, AI intends to make computers as good as human beings. Now let us consider one more reason as to why a particular person is considered as intelligent. One of the most common traits in so-called intelligent people is their ability to remember something. Now remembering something can be considered as the ability to search what is already stored in the memory. This searching is much faster than the algorithmic searching techniques such as linear or binary search.

In order to understand the reason for faster memory in human beings, we must appreciate that there are millions of neurons in the human body. These neurons are connected to each other via trillions of synapses. So technically, there can be trillions of events

stored in the memory of a human being. Recapping an event from amongst these events in a few seconds requires fabulous retrieving mechanisms.

AI helps us to imitate the search process via genetic algorithms, which may be defined as follows.

Genetic algorithms These are heuristic search processes based on the theory of survival of the fittest.

Genetic algorithm is one of the many techniques used in machine learning. The concept of machine learning has been dealt with in the next section. The chapter discusses the concept of genetic algorithms from Section 23.5 onwards.

23.3 MACHINE LEARNING

Creativity and learning are two of the many important human skills which are difficult to computerize. Though artificial creativity still eludes the scientific fraternity, machine learning (ML) has made significant advances in the recent past. As a matter of fact many AI researchers put forth the example of machine learning, when asked about the development of the subject. Like a sequence of small acts make something big, in the same way little variations in common patterns may indicate something radically different. ML techniques precisely intend to analyse such patterns.

There are many approaches that help to make machines learn. This chapter throws light on the three most significant ones. These are as follows:
- Symbol-based
- Connectionist
- Evolutionary

The classification has been depicted in Fig. 23.1.
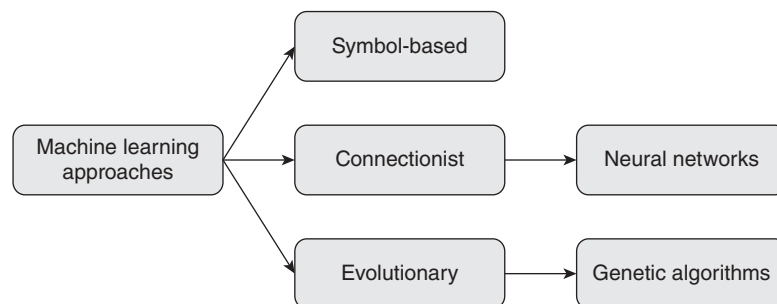
Figure 23.1 Machine learning approaches

Before introducing the concept of machine learning, let us understand the idea of learning. In fact, the last sentence is recursive, learning helps us to develop concepts.

Concept, as a matter of fact, is the development of the features of a class by generalizing the domain.

Learning has been defined in the Oxford dictionary as the process of gaining knowledge through studying, being taught, or experience. The definition itself tells us how to implement learning in machines. In fact, experience is one of the important ways of learning.

Suppose we intend to accomplish a task and we are able to do so. Our experience would help us to do the same task or perhaps a similar task, in a better way, the next time. Let us see why:

- The experience helps us to identify the problems and possible solutions in doing a given task.
- We generally develop heuristics which helps us to attain goals.
- Learning inculcates adaptability.

The following example would help us to understand the concept in a better way.

Suppose a professor of great intellect has taught artificial intelligence to you. He is extremely particular about the words you use, the impact of research, and so on. You work under him, publish a few papers, and then decide on teaching and writing books. Then you are appointed in a college that might not have the same ethos as that of the college in which you studied. So, what do you do? Leave teaching job in the college and wait for a better opportunity or try adapting yourself to the needs of the students? The second option is better. One of the prime goals of teaching is to get a better insight of the subject. That does not always come from a student studying in a high ranking college but that might as well come from other students as well. If the aim of teaching is learning, then your adaptability would help you to achieve the task. This can also be stated as follows. The urge to learn has made you adaptable. This adaptability in the structure is one of the ways of learning. This technique, of making changes in the structure, in order to learn from a given data, is referred to as the connectionist model.

The third approach is that of the evolutionary learning. This is one of the most powerful approaches of machine learning. In searching from a large domain wherein each item has a fitness associated with it, genetic algorithms are used. As a matter of fact, the evolutionary approach not just comprises genetic algorithms, but also comprises genetic programming and artificial life to a little extent. However, the latter two are beyond the scope of this book. The first, that is, genetic algorithms has been discussed in Sections 23.5.

Tip: Evolutionary approaches of learning mainly consists of
- Genetic algorithm
- Genetic programming
- Artificial life

The reader is expected to go through the research papers given in Bedau (2003) in order to get an insight of artificial life.

23.3.1 Learning

One of the main goals of learning is to infer the definition of the general class using the given data. This can be done in two ways namely the data intensive approach and the explanation-based approach. In the data intensive approach, the data are given to the learner. These data can be in the form of positive examples, for instance. It is not necessary that these examples are from the same domain as the problem. During the literature review, one of the best examples was found in Luger. The example was that of the analogy between current and water. When a teacher says that the current is like water he means that the electrons flow in the wire due to potential difference in the same way as water flows in a pipe due to the pressure difference between the source and the destination. It does, in no way mean that we can wash our hand with current.

The task of learning, though, can also be achieved by interpreting the set of rules and then deriving the definition of the general class which is referred to as the explanation-based approach.

The above rules or data should have a predefined organization which would help us to extract and interpret the data. The issues in representation are dealt in Knight (1991).

The learned knowledge is then operated upon by operators like those explained in Section 23.5 of this chapter. These operators refine the available data and even help us to derive new inferences.

The reason why this chapter focuses on genetic algorithms is because the above operations are followed by what is referred to as heuristic search, which is the use of heuristics to search in a concept space. However, the search can also be a version-based search which is based on the idea of generalization leading to ordering.

The bias in inductive learning can be used to curtail certain outcomes. This would produce a better concept in a lesser time. The next section discusses an important paradigm in learning, that is, neural networks.

23.4 NEURAL NETWORKS

When a computer recognizes our retina, or when a person's attendance is recorded using a biometric system, or when the government of the country decides to use biometric data to uniquely identify a person, the advancement of AI-pattern recognition amazes us.

There are a large number of ways in which the above tasks can be accomplished. One of the most important is neural networks. Neural networks are inspired by the working of neurons. The pattern-matching technique used in the above software is similar to the pattern-matching technique used by our brain. The brain organizes the basic structural unit of our nervous system called a *neuron*. There are millions of neurons in our body. These neurons connect with each other via synapse. The structure of a neuron has been

depicted in Fig. 23.2. The neuron contains a cell, which has a nucleus and branches called dendrites and axon.
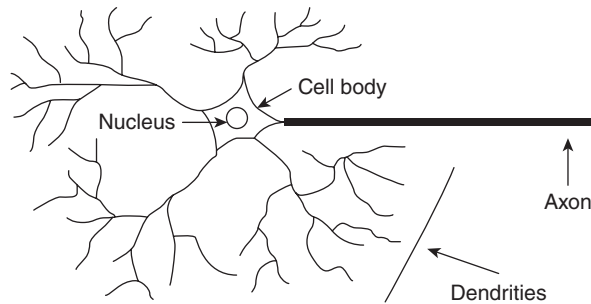
Figure 23.2 A neuron

The nervous system helps the brain to carry out parallel computation, which gives human abilities such as pattern-matching. The scientific fraternity has been trying to mimic the human brain in order to incorporate these abilities in a machine. Here it may be stated that learning is one of the abilities which the fraternity intends to replicate in a machine. The artificial neural network is a step towards making machines learnable.

When we recognize another person by seeing, it is called *perceptual recognition*. This is not just the work of our brain, but because of what is referred to as mind. The brain works like a parallel computer and aids us to identify a person, or a place, or say for that matter an event, in approximately 200–300 ms. It is amazing that a normal desktop computer would take a much longer time to accomplish the same task as the search space is massive.

As stated earlier, learning develops with experience. Our experience helps us to classify the patterns. The experience helps us to classify, regulate, and process information. This quintessence is captured by what is referred to as a neural network.

These neural networks, like brain, learn by adjusting the synaptic connections. These models are therefore referred to as connectionist models.

Definition An artificial neural network (ANN) is an information processing paradigm that is inspired by the way biological nervous systems, such as the brain process information.

As stated earlier, the basic unit of a neural network is a neuron. The type of neural network is determined by its connections. The various components of a neural network are as follows:

1. *Neurons*: The basic building block of a neural network.
2. *Synapse*: The connection of neurons through links is referred to as synapse. In an ANN, each synapse is associated with a weight.
3. The inputs which are multiplied by the respective weights and then summed using an adder. The output is fed to the next component.

4. *Activation function*: The output of a neural network is determined using an activation function.

The conventions used in the following discussion are as below:

- The inputs are denoted by x_i's, for example, x_1 is the first input.
- The weight of a synapse from neuron 1 to hidden layer 2 would be denoted by w_{12} and so on.
- The bias factor is denoted by b_k.
- The activation function is denoted by φ.
- The output is denoted by y_k

The inputs to the neural network are $x_1, x_2,..., x_n$. The weights are w_{k1}, etc. Then, the output at the summation point u_k would be as follows:

$$u_k = \sum_{j=1}^{m} w_{kj} x_j$$

The bias is added to the above output. The sum of the output and the bias has been denoted by v_k:

$$v_k = (u_k + b_k)$$

v_k is given as an argument to the activation function, which decides the output:

$$y_k = \varphi(v_k)$$

The model of the neural network has been depicted in Fig. 23.3.
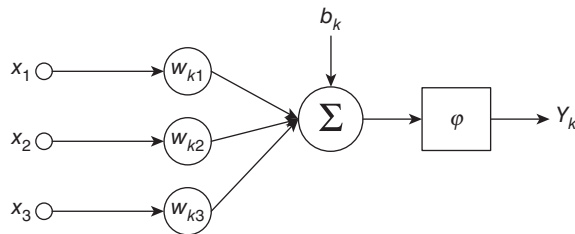
Figure 23.3   Model of a neural network

If the bias is considered as an input with weight 1, the model is depicted in Fig. 23.4.
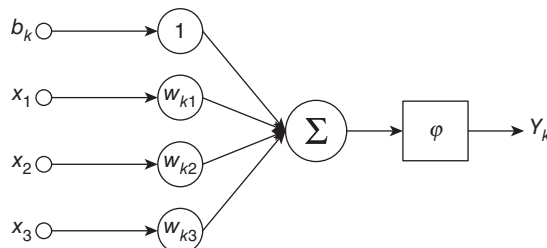
Figure 23.4   Model of a neural network taking
bias as an input

As stated earlier, there are many types of activation functions. Some of them are as follows:

Threshold Function

The output of this function is 1, if the value of v described above is greater than 1, otherwise the output is 0. The function can be described as follows:

$$y_k = \begin{cases} 1, \text{if } v_k \geq 1 \\ 0, \text{otherwise} \end{cases}$$

Figure 23.5 depicts the above activation function.
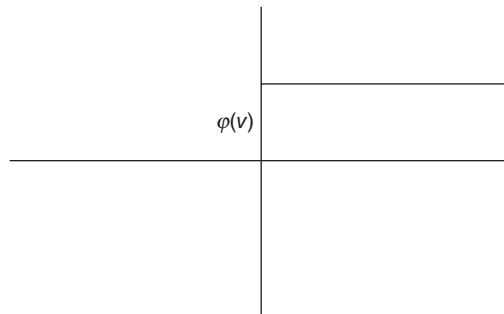
Figure 23.5   The threshold function

Sigmoid Function

If the value of v, described above, is finite then the function can be described as follows:

$$\varphi(v) = \frac{1}{1 + e^{(-av)}}$$

Figure 23.6 depicts the above activation function.
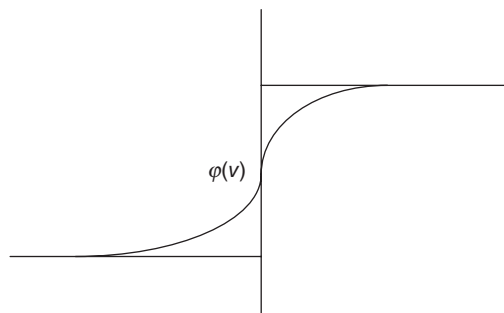
Figure 23.6   The sigmoid function

Although neural networks help to carry out many important tasks, like recognizing patterns, genetic algorithms help in optimized searches. The next section briefly discusses genetic algorithms.

23.5 GENETIC ALGORITHMS

Genetic algorithm (GA) is a heuristic search process based on the theory of survival of the fittest (Cormen, 1990). The concept is generally used in optimization problems. However, it has been observed that the solutions obtained via the process are not robust. In order to incorporate robustness, diploid genetic algorithms (DGAs) are used. It may be noted at this point that genetic algorithm is not just a tool as perceived by many software engineering theorists. It may be used as a black box that optimizes the data by those who are unaware of the problem reduction approaches and have little knowledge of AI. It is not just a box in which we give a set of data, vary the parameters, and obtain the expected results. It is an intricate process that has an aura of its own. People like David E. Goldberg spent his whole life in order to explain the advantages of the process to the mankind. Genetic algorithms are being used for solving NP class of problems, in generating test data, in the generation of pseudorandom number generation, and even in bioinformatics. However, GAs are not just a machine learning approach but the details of GAs needs to be understood; problem reduction approach should be used in order to accomplish the task. The task accomplished by using GAs would most of the times be an optimized search.

A GA starts with creating a population of chromosomes. Each chromosome has cells. The cells of a chromosome can be binary or decimal or even hexadecimal. This choice depends on the problem. For example, in subset sum problem, '1' at a certain position may indicate the inclusion of that element in the requisite sumset. The number of chromosomes in a population also depends on the problem at hand. Figure 23.7 depicts an instance of the population and the corresponding chromosomes and cells.
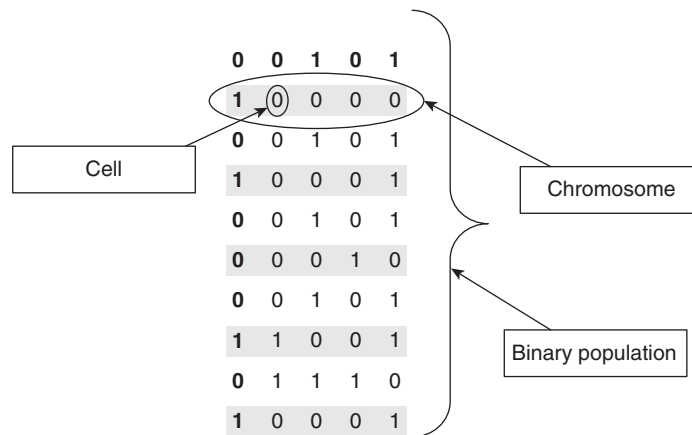
Figure 23.7   A binary population

The above population may be subjected to the genetic operators in order to optimize the results. Some of the genetic operators discussed in the chapter are crossover, mutation, and roulette wheel selection.

INTRODUCTION TO MACHINE LEARNING APPROACHES ■ 493

23.5.1 Crossover

Crossover helps to produce new chromosomes having the features of both the parents. The crossover population amalgamates two chromosomes in order to produce a new chromosome. There are many types of crossovers such as

- One-point crossover
- Two-point crossover
- Multipoint crossover
- Uniform crossover

One-point Crossover

One-point crossover is a crossover wherein a new chromosome is formed by taking the left part of the first chromosome, with respect to a randomly selected point and the right side of the second chromosome from that point. In order to carry out one-point crossover, the following steps need to be performed:

1. Select any two chromosomes from the population.
2. Generate a random number up to the number of cells in a chromosome.
3. From the first chromosome selected, extract the cells up to the cell number generated in the previous step.
4. From the next index up to the end, extract cells from the second chromosome.
5. Another chromosome can be created in the same way but taking cells from the second chromosome first and then from the first chromosome. The process has been depicted in Fig. 23.8.
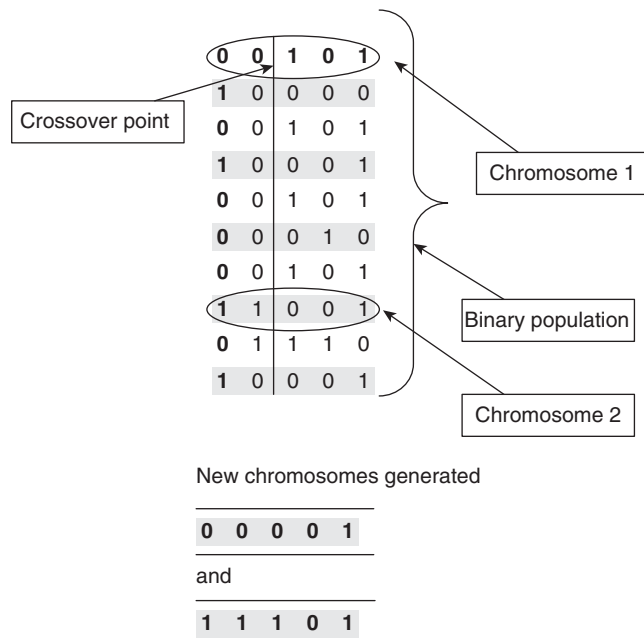
Figure 23.8 One-point crossover

Two-point Crossover

Two-point crossover is a crossover wherein two random positions govern the formation of a new chromosome. In order to carry out the two-point crossover, the following steps need to be performed:

1. Select any two chromosomes from the population.
2. Generate two random numbers up to the number of cells in a chromosome.
3. From the first chromosome selected, extract the cells up to the first cell number generated in the previous step.
4. From the next index up to the second index generated, extract cells from the second chromosome.
5. Then from the second index up to the end, extract cells from the first cell.
6. Another chromosome can be created in the same way but taking cells from the second chromosome first and then from the first chromosome.

The process has been depicted in Fig. 23.9.
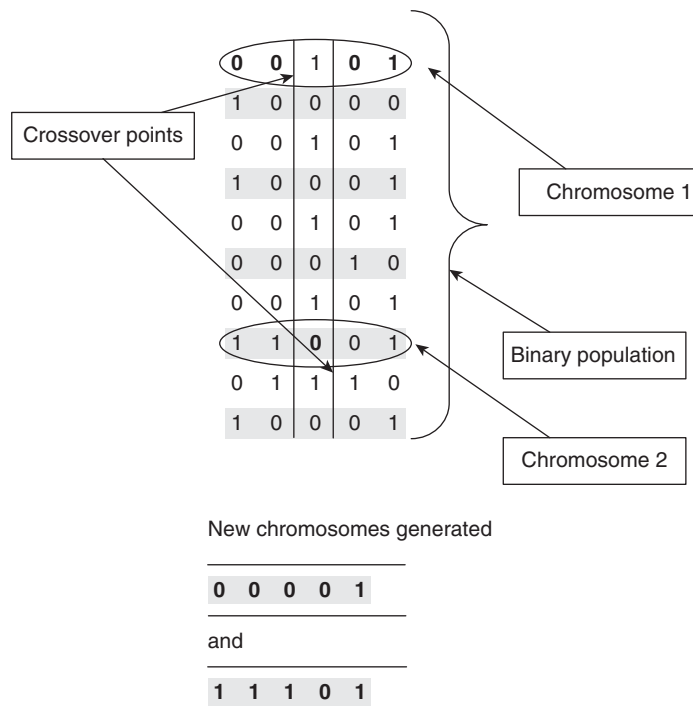
Figure 23.9 Two-point crossover

Multipoint Crossover

In multipoint crossover, more than two random points govern the formation of a new chromosome.

In order to carry out the multipoint crossover, the following steps need to be performed:

1. Select any two chromosomes from the population.
2. Generate random numbers up to the number of cells in a chromosome.
3. From the first chromosome selected, extract the cells up to the first number generated in the previous step.
4. From the next index up to the second index generated, extract cells from the second chromosome.
5. Then from the second index up to the next index, extract cells from the first cell.
6. Continue this process till a new chromosome is generated having equal number of cells as either of the chromosomes.
7. Another chromosome can be created in the same way but taking cells from the second chromosome first and then from the first chromosome.

The process has been depicted in Fig. 23.10.
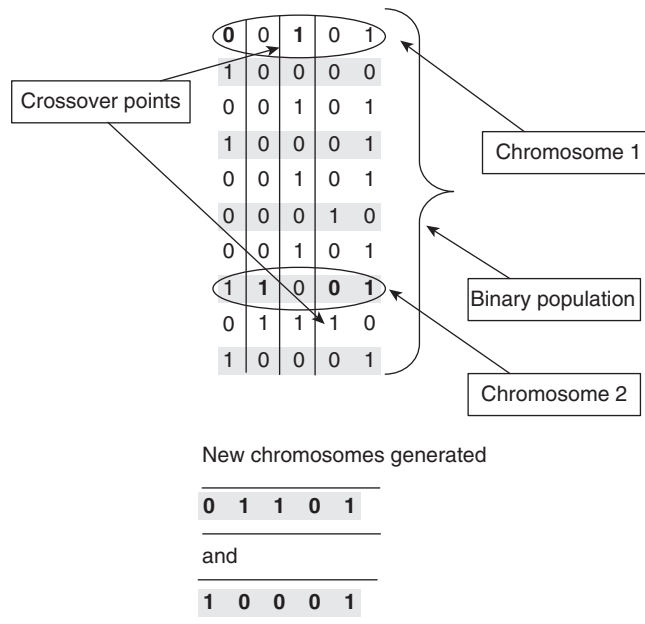
Figure 23.10   Multipoint crossover

Uniform Crossover

In this alternate cells from the 1st and 2nd chromosome are selected to generate a new chromosome.

23.5.2 Mutation

Mutation is carried out in order to break the local maxima. In the mutation operation, a chromosome is randomly selected and one of its bit is flipped. If it is '1' then it is made '0', else it is made 1.

In order to carry out mutation, the following steps need to be performed:

1. Select any chromosomes from the population.
2. Generate a random number up to the number of cells in a chromosome.
3. From the chromosome selected, flip cells having index generated in Step 2.

The process has been depicted in Fig. 23.11.
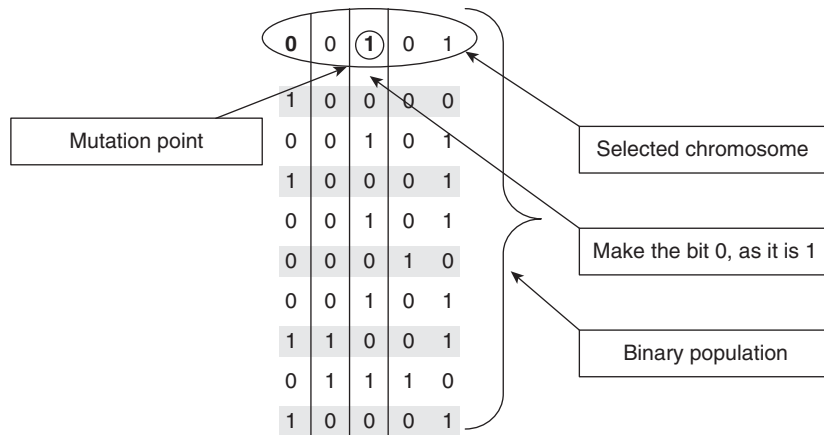
Figure 23.11 Mutation

It may be noted, though, that the mutation rate is generally kept low. The inspiration comes from the nature. Whereas the crossover helps to amalgamate features of a male and a female, mutation helps making the child different from the existing population. However, this difference can make him better than others or may make him worthless. The nature takes such risks in order to preserve the charm of uncertainty.

23.5.3 Selection

The selection process selects the chromosomes having high fitness values from amongst the population and replicates those chromosomes. In the process, the more fit chromosomes become the majority community and the probability of their selection and the final solution increases. It may be stated here that the fitness of a chromosome is evaluated by the fitness function. The fitness function is decided keeping in mind the goal. For example, if the cost of the path is to be minimized, then the fitness function would be such that more the path of the cost, lesser will be the fitness function. The fitness function in this case can be

$$\frac{1}{1+e^{-\lambda}}$$

where λ is proportional to the length of the path.

Coming back to the selection techniques, there are many selection techniques described in literature. However, roulette wheel selection is one of the most common selection techniques. The technique replicates chromosomes having high fitness value with a greater probability. For example, software is to be developed to find the root

of an algebraic equation. In order to accomplish the task, a value is generated and the value of the left-hand side of the equation is evaluated. The deviation of the answer obtained is calculated with the desired value, which in this case is 0. The fitness function is designed in such a way that if the value of deviation is low, the fitness of the solution obtained (and hence the chromosome generating the solution) is higher. Suppose the various chromosomes have fitness values shown in Table 23.1.

In order to apply roulette wheel selection, the commutative frequency (of fitness) is calculated. Table 23.2 shows the commutative frequency table. The corresponding pie chart is depicted in Fig. 23.12.

Table 23.1 Fitness of the chromosome

Chromosome	Fitness
1	10
2	7
3	5
4	4
5	3
6	1

Table 23.2 Commutative frequency table

Chromosome	Fitness	Commutative frequency
1	10	10
2	7	17
3	5	22
4	4	26
5	3	29
6	1	30
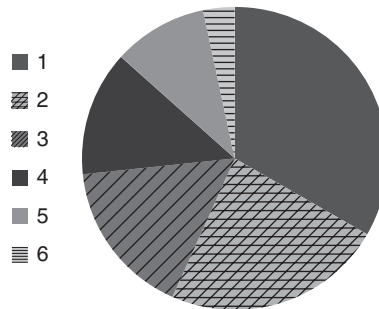
Figure 23.12 Pie chart depicting probability of selection of each chromosome is proportional to their fitness

Now a random number is generated mod 30. Say it is 21. The next step would be to select the chromosome which depicts this commutative frequency in the above table. In this case it is chromosome number 3. So the third chromosome would be replicated and the final population would contain two copies of the chromosome. However, the process is repeated many times.

23.5.4 Process

The process of genetic algorithm is given in Algorithm 23.1. The process consists of population generation, crossover, mutation, and selection until a desirable solution is reached. The process may be repeated many a times, each time a new generation is

formed. If the initial number of chromosomes is high, then the answer would be found in lesser number of generations.

Algorithm 23.1 Simple genetic algorithm

Step 1. Generate initial population
Step 2. Evaluate fitness of each chromosome
Step 3. Perform crossover
Step 4. Perform mutation
Step 5. Evaluate fitness and perform roulette wheel selection
Step 6. If solution is not found or the number of generators cross a particular value then go to Step 3 else print the result
Step 7. End

The application of the above process has been explained in the sections that follow.

23.6 KNAPSACK PROBLEM

There is a thief who has to select a few items from amongst the given items. Let the capacity of the bag be 'c'. Let there be n things. The problem can be stated as follows:
- Set of items is given by the set $\{y_1, y_2, y_3, ..., y_n\}$.
- The weights of the above items are given by the set: $\{w_1, w_2, w_3, ..., w_n\}$ and
- The profits obtained from each of the above items are $\{p_1, p_2, p_3, ..., p_n\}$.

Let $x_n = 1$ denotes that the item has been picked and $x_n = 0$ means that the item has not been picked. The problem is to select items in such a way that the total weight of the selected items is less than or equal to the weight of the bag. The constraint is depicted in Eq. (23.1),

$$x_1 \times w_1 + x_2 \times w_2 + x_3 \times w_3 ... \leq m \tag{23.1}$$

The items are to be picked in such a way that the profit earned is maximum,

$$x_1 \times p_1 + x_1 \times p_2 + x_3 \times p_3 ... \text{ is maximum}$$

The above problem is referred to as 0/1 knapsack problem.

Knapsack problem The assortment of objects from a given set in such a way that the total weight is less than or equal to the given weight, and the profit earned by picking up the elements is maximum.

Brute Force Algorithm

The problem can be solved by brute force algorithm. However, the complexity of the process is too high. Algorithm 23.2 shows the brute force approach to solve the subset sum problem.

> **Algorithm 23.2** Brute force approach for subset sum problem
>
> **Step 1.** Find out the profit per unit weight of all the items.
> **Step 2.** Arrange the array obtained in the decreasing order.
> **Step 3.** Now, pick the items from the array one by one till there is a space in the bag.

The problem, however, crops up when the number of items is too large. In that case, the brute force approach does not work. In such cases, a method based on GA can be used, as it is an optimization problem.

The most important thing while applying GAs is to design the fitness function. In this case, the fitness function should have a high value if the profit earned is high and weight of the selected subset is low. In order to solve knapsack problem via GAs, the cells of the chromosomes may represent the inclusion or non-inclusion of the item. Algorithm 23.3 depicts the solution of the problem by using GAs.

> **Algorithm 23.3** GAs approach for knapsack problem
>
> **Step 1.** Generate a binary population having number of cells equal to the cardinality of the set.
> **Step 2.** Each cell of a chromosome depicts the inclusion or non-inclusion of a cell.
> **Step 3.** The chromosomes which have the total weight of the elements greater than the target can be straightaway rejected.
> **Step 4.** Else the fitness of the chromosome can be calculated. The fitness of a chromosome may be defined as $\text{fitness} = \dfrac{1}{1+e^{\lambda}}$, where λ is the ratio of profit of the chromosome and the weight.
> **Step 5.** The chromosomes are then arranged in order of their fitness.
> **Step 6.** Then crossover is carried out.
> **Step 7.** The mutation operator is then applied in order to break the local maxima.
> **Step 8.** If the solution is obtained then the process terminates else the next generation is generated.

23.7 SUBSET SUM USING GA

The subset sum problem is a significant problem in algorithm analysis and design. We have already introduced the problem in Section 12.3 of Chapter 12. There are several ways to solve the subset sum problem. The problem can be solved via greedy approach, dynamic approach, backtracking, and branch and bound. However, the most basic way would be to form all the subsets and then cycle through all of them. Now for every subset, we must check if the subset sums to the required number. Since there are 2^n subsets of a set having n elements and the sum of each subset is to be taken, therefore, the running time of the above procedure would be of order $O(2^N N)$. Algorithm 23.4 elucidates the steps of the procedure.

Algorithm 23.4   Brute force algorithm for subset sum problem

Given:
- A set having n elements.
- A value m

Required
To find a subset of the given set such that the elements of the subset have sum = m.

Brute force algorithm
1. Find all the subsets of the given set.
2. For each subset
    a. Find the sum of elements of the subset.
    b. If the sum of the subset is equal to the required sum then
        i. Print the answer and quit.
    c. else
        i. Continue
3. End

There is another algorithm which runs in time $O(2^{N/2})$. The algorithm is based on the strategy of divide and conquer. The set containing N elements is divided into two sets of $N/2$ elements each. Each of these two sets is then sorted. The complexity of this algorithm is lower than the above set. This section explains a GA-based approach to solve the subset sum problem.

Illustration 23.1 Find the subset of the set $\{2,3,6,7,10\}$ having sum as 10.

Solution The given set is {2, 3, 6, 7, 10} and the value of the sum is 10. The various subsets of the given set are as follows:

$$\varphi,\{2\},\{3\},\{6\},\{7\},\{10\},\{2,3\},\{2,6\},\{2,7\},\{2,10\},\{3,6\},\{3,7\},\{3,10\},\{6,7\},\{7,10\}$$
$$\{2,3,6\},\{2,3,7\},\{2,3,10\},\{2,6,7\},\{2,6,10\},\{2,7,10\},\{3,6,7\},\{3,6,10\},\{6,7,10\}$$
$$\{2,3,6,7\},\{2,3,6,10\},\{3,6,7,10\},\{3,6,7,10\},\{2,6,7,10\}$$
$$\{2,3,6,7,10\}$$

The sum of each of the above set is

0	9	11	20
2	12	12	23
3	9	15	18
6	10	15	20
7	13	18	26
10	13	19	25
5	16	16	28
8	17	19	

The solution is therefore {10} and {3, 7}.

23.7.1 Solution Using GA

The above method works for small sets. However, for large sets, the method will not work. The number of calculations and the complexity of the algorithm will be too large. The time taken by a normal PC to implement the above algorithm might be in months so the applications based on the above system will not be realizable. So there is a need to use some AI technique in order to implement the algorithm.

Since the above problem is an optimization problem and finding the solution is basically the search for correct subset, GAs are best suited for solving subset sum problem.

The below steps must be followed in order to solve the problem via GA.

Step 1 Generate initial population of the GA. Each chromosome has the same number of cells as are there in the given set; 1 in a set indicates the inclusion of the corresponding element of the set, whereas 0 indicates the non-inclusion of that particular element.

For example, if set $A = \{2,3,6,7,10\}$ and value = 10.

After the generation of the population one of the chromosomes is 11001.

Step 2 Then it indicates the inclusion of the first, second, and the fifth element of the set. In this case, the subset generated by the above mapping is {2, 3, 10}.

Step 3 Then the sum of the elements of the subset is evaluated. The sum of the above subset becomes 15. This is followed by the calculation of deviation of the sum obtained from the required value. The deviation in this case is 5.

Step 4 The chromosomes are then arranged in the order of deviation. If the deviation is zero, then the chromosome represents the solution.

Step 5 If the solution is not obtained, then the crossover and mutation operations are applied in order to reach to the solution. The selection procedure explained in the introduction depends on the fitness of a chromosome. In this case, less the deviation, more is the fitness; therefore, more fit chromosomes will be generated as the process moves forward.

Example: The subset sum problem is not just one of the problems of ADA, it has many applications. It is used in cryptography. The very popular knapsack cipher uses the concept of subset sum for the decryption part. The problem also finds applications in web crawlers. The importance of subset sum problem makes the efficiency of its solution all the more important.

The most important thing while applying GAs is to design the fitness function. In this case, the fitness function should have high value if the difference between the desired sum and the sum of the subset selected is low. In order to solve the subset sum problem via GAs, the cells of the chromosomes may represent the inclusion or non-inclusion of the item. Algorithm 23.5 depicts the solution of the problem by using GAs.

Algorithm 23.5   GAs approach for subset sum problem

Step 1. Generate a binary population having number of cells equal to the cardinality of the set.

Step 2. Each cell of a chromosome depicts the inclusion or non-inclusion of a cell.

Step 3. The chromosomes that have the sum of the elements greater than the target can be straightaway rejected.

Step 4. Else the fitness of the chromosome can be calculated. The fitness of a chromosome may be defined as fitness $= \dfrac{1}{1+e^{\lambda}}$, where λ is the modulus of difference of the sum of the selected subset and the target subset.

Step 5. The chromosomes are then arranged in order of their fitness.

Step 6. Then crossover is carried out.

Step 7. The mutation operator is then applied in order to break the local maxima.

Step 8. If the solution is obtained, then the process terminates else the next generation is generated.

The process is shown as follows:

Set A = {2, 1, 4, 3, 6, 8}

Value = 15

No of chromosomes: 15

The population is

001100
000100
101000
010000
000001
100100
100000
101010
010000
010001
100000
010000
100001
000000
110000

The sums of the subsets corresponding to the above population are {7, 3, 6, 1, 8, 5, 2, 12, 1, 9, 2, 1, 10, 0, 3}. Deviation of the above sum from the required value is {8, 12, 9, 14, 7, 10, 13, 3, 14, 6, 13, 14, 5, 15, 12}. The fitness of the above chromosomes calculated by taking the inverse of the above value and multiplying the solution with 100 are {12.5, 8.333334, 11.111112, 7.1428576, 14.285715, 10.0, 7.692308, 33.333336, 7.1428576, 16.666668, 7.692308, 7.1428576, 20.0, 6.666667, 8.333334}. The fitness

is then arranged in order and the corresponding chromosomes are also arranged. The arranged population is as follows:

000000
010000
010000
010000
100000
100000
000100
110000
100100
101000
001100
000001
010001
100001
101010

This is followed by the application of roulette wheel selection. The rest of the procedure is same as the general procedure of simple GA explained in the first section of the chapter.

23.8 TRAVELLING SALESMAN PROBLEM

Definition The travelling salesman problem (TSP) is an NP-hard problem. It is applied in operations research and theoretical computer science. Given: A list of cities V and their pairwise distances D such that D is a set where $x_i \in D$ is the distance between (l, m), $l, m \in V$.

The aim of TSP is to find the shortest possible tour that visits each city exactly once and keeping the new cost minimum.

The TSP can be used in many disciplines such as planning, logistics, and the manufacture of microchips. In the theory of computational complexity, the decision version of the TSP belongs to the class of NP-complete problems. Thus, it is likely that the worst case running time for any algorithm for the TSP increases exponentially with the number of cities.

The sub-section presents a brief overview of the techniques proposed by various researchers to solve TSP. Since TSP has always been a topic of interest, many researchers have suggested the various solutions of the TSP using different AI approaches. Some of the solutions are as follows:

- In the solution proposed by Aybars Uğur, all points depicting the cities have been taken on the surface of a sphere. The method follows rigorous mathematical analysis in order to generate the solution. The complexity and the mathematical calculations of the method make it less useful.

- Gokturk Ucoluk suggested a GAs-based solution of TSP problem. According to the work, mutation and crossover operators are not applicable to the problem. The author has introduced a new operator for generating random population. However, it may be noted that many researchers have applied crossover and mutation operators in the TSP and got better results as compared to the above work. So there is little reason to exclude crossover and mutation from the GA process in order to solve TSP. It may be stated though that crossover and mutation can be applied on binary population after which encoding can be done. It would be pointless to apply crossover to the decimal population.
- Many researchers have used minimum spanning tree (MST)-based graph pyramid for finding out optimal solutions for computationally hard pattern recognition problems. Though the time complexity using this procedure has been calculated as $O(|E2|)$, the complexity has been given only for restricted conditions and many assumptions. So, we cannot really compare the method on the basis of complexity with other methods.
- An improved immune genetic algorithm was proposed by Jingui Lu to solve the TSP. A new selection strategy was included into the conventional GA to improve the performance of GA.
- A hybrid GA that incorporates the generalized partition crossover (GPX) operator to produce an algorithm that is competitive with the state of the art for the TSP was proposed by Darrell Whitley. The approach gives satisfactory results as per the experiments carried out by the author.
- Bhasin & Singla (2012) proposed a problem reduction approach to convert the binary population generated by GAs into the sequence of cities that can give a solution of the TSP. The fitness function of the city has been crafted in accordance with the cost of the path. The present section explains this approach and exemplifies it.

However, no paper was found that compared the above approaches with each other. So it is difficult to state which approach is the best. In addition, different researchers have taken different assumptions and different environments in order to carry out the experiments.

23.8.1 GA Approach to Solve Travelling Salesman Problem

As stated earlier, there are many methods of solving TSP via GAs. The following section, however, uses the approach proposed by Bhasin et al. According to the technique, in order to solve TSP with the help of GAs, the steps given in Algorithm 23.6 may be followed.

Algorithm 23.6 GAs approach for travelling salesman problem

Step 1. First of all a binary population is generated with the help of pseudorandom number generator of the language.

Step 2. Each chromosome of the population is divided into parts having m bits. Each set of m bits depicts a valid city.

Step 3. Each set is mapped with a valid city.

Step 4. The path generation
  a. From the matrix obtained, path is generated.
Step 5. Each path is tested for feasibility.
Step 6. Repeat the above process for each row and generate a different path for each row.
Step 7. Path selection
  a. Calculate the cost of traversing each path using the cost matrix of the cities and select the path with lowest cost.
Step 8. Re-analysis
  a. The solution can be enhanced by crossover and mutation operations. The population formed after applying the above operators may generate better results.

Figure 23.13 depicts the above process.
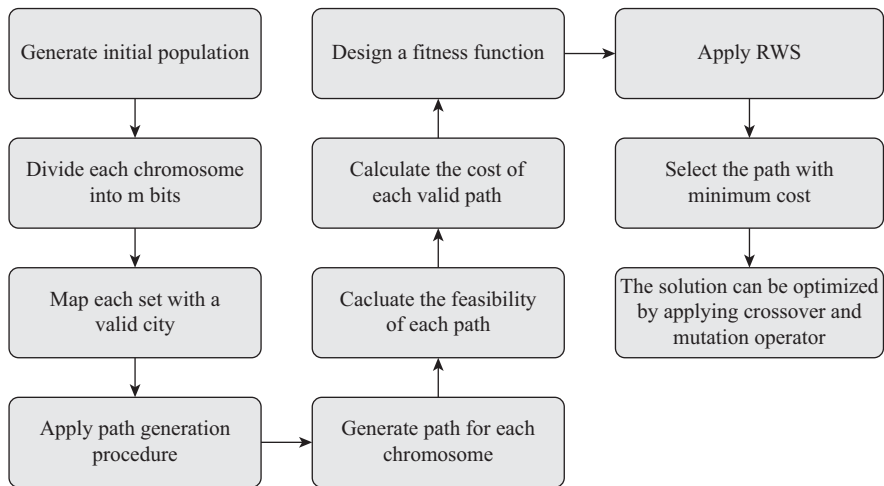
Figure 23.13 TSP via GAs

Illustration 23.2 Suppose there are 5 cities $\{A, B, C, D, E\}$ and the distances between the cities are given by the following matrix:

$$\begin{pmatrix} 0 & 2 & 3 & 2 & 4 \\ 2 & 0 & 1 & 2 & 1 \\ 3 & 1 & 0 & 3 & 5 \\ 2 & 2 & 3 & 0 & 7 \\ 4 & 1 & 5 & 6 & 0 \end{pmatrix}$$

Solve the above travelling salesman problem using GAs.

Solution The above matrix depicts the distance between any two cities. Since the above represents the cost of the path between any two cities, it has to be a symmetric matrix.

In order to solve the problem, an initial population is generated. The following population has 10 chromosomes each having 5 × m cells, since there are 5 cities in the problem. The value of m will be three as explained in the following discussion.

Since there are 5 cities, 3 cells are needed in order to map the bits with the cities. Table 23.3 depicts the mapping of cities with the binary bits.

Table 23.3   Mapping of chromosomes

Bits	Binary equivalent	Number % 5
000	0	0
001	1	1
010	2	2
011	3	3
100	4	4
101	5	0
110	6	1
111	7	2

Now a population of 10 chromosomes having 15 cells each is generated.

Each chromosome will now be divided in the groups of three and a sequence of numbers would be generated. For example, the first chromosome depicts the sequence 14420. The repeated value will now be replaced by the missing numbers in order. In this case, the missing number is 3. The sequence now becomes 14320. Now all the paths that are generated by the above population would be checked for feasibility. Those paths that are feasible will now go to the next step (Table 23.4).

Table 23.4   Chromosomes

0	0	1	1	0	0	1	0	0	0	1	0	1	0	1
0	1	0	0	1	0	1	1	1	1	0	1	1	1	1
0	1	1	1	1	1	1	1	0	0	0	0	1	1	1
1	0	1	1	0	0	1	1	1	0	1	0	0	0	0
0	0	0	1	1	0	0	0	0	1	1	1	1	1	1
0	0	0	0	0	0	0	1	1	0	0	0	1	0	1
0	1	1	0	0	1	1	0	0	1	0	0	1	1	1
1	0	1	1	0	1	0	1	0	0	0	1	1	0	0
1	1	1	1	0	0	1	0	1	1	0	1	0	0	0
1	1	1	1	1	1	1	1	0	0	1	0	1	0	0

In the next step, the fitness of each path will be evaluated. In this case, the fitness may be given by $\text{fitness} = \dfrac{1}{1 + e^{-1*\lambda}}$, where λ is the cost of each path.

The rest of the process is same as that applied earlier. However, it may be noted that crossover and mutation should be carried out on the binary population, not the sequence.

23.9 VERTEX COVER PROBLEM

The vertex cover problem calls for selecting the set of vertices of a graph (V, E), such that the edges adjacent to the selected vertices give the set V of the original graph.

Given a graph $G = (V, E)$, the vertex cover problem calls for the crafting of the set V' such that $V' \subseteq V$ such that all the edges of the set E are covered by the vertices of the set V'.
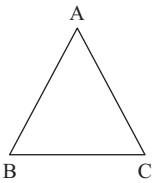
Figure 23.14 Graph 1
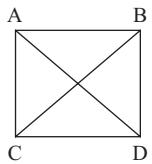
For example, for the graph shown in Fig. 23.14, the vertex cover is {A, B}. This is because the edges having A as one of their vertices are {(A, B), (A, C)} and the edges having B as one of their vertices are {(B, C), (A, C)}. The union of these two sets gives {(A, B), (A, C), (B, C)}, which is same as the set depicting the edges of the given graph. However, it may be noted at this point that there can be more than one answer to the above problem. The set {A, C} or the set {B, C} represents an equally good solution.

Figure 23.15 Graph 2

However, there can be graphs in which the vertex cover has more vertices, even equal to the number of vertices in the original set. For example, the vertex cover of the graph shown in Fig. 23.15 is the set {A, B, C}.

The problem has already been covered in chapter on approximation algorithms; however, this section contains a brief description of the algorithm.

The approximation algorithm of the above problem selects a vertex and removes the edges which have that vertex as one of its end points from the set of vertices. The selected vertex is added to the set A and the reduced set of edges becomes the set of edges for the next iteration. The following two algorithms depict the same procedure written in different forms (Algorithms 23.7 and 23.8).

23.9.1 Approximation Algorithm

Vertex Cover Algorithm 1

Algorithm 23.7 Vertex cover algorithm 1

Select $x \in V$ from the set of vertices V.
Remove x from the set V to create a reduced set V'.
Find all the edges which have x as one of their end points. For all such edges carry out the following steps:
Remove all such edges from the set of edges. Continue till the set of edges reduces to a NULL set.

Algorithm 23.8 Vertex cover algorithm 2

```
A←∅
    while (E != ∅)
            pick any {x, y} ∈ E
            A ← A∪ {x, y}
            delete all edges incident to either x or y
    return A
```

However, the solution of the problem via the above algorithm can be tedious. For example, it is very difficult to find the vertex cover of the graph shown in Fig. 23.16 by approximation algorithms.
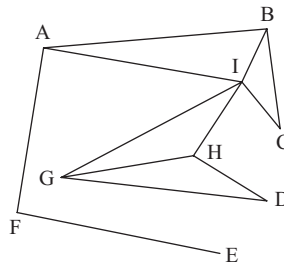
Figure 23.16 Graph

The brute force algorithm will not work for a graph having many vertices. In such cases, GAs come to our rescue. The solution of the problem via GAs would be on the same lines as the previous problems. In order to apply GAs to vertex cover, the following steps may be followed (Algorithm 23.9).

23.9.2 Solution of Vertex Cover via GAs

Algorithm 23.9 GAs approach for vertex cover problem

Step 1. Generate an initial population having *n* chromosomes.

Step 2. The number of vertices in each chromosome should be same as the number of vertices in the graph.

Step 3. Since the population is binary, a mapping is needed to map each chromosome with the set of vertices. In order to handle the above problem, each 1 depicts the inclusion of the vertex, whereas a 0 depicts the non-inclusion. For example, if the graph consists of 6 cells, then chromosome 100101 implies that the first fourth and last vertices are to be taken.

Step 4. First of all, the sequence selected will be checked for feasibility. Each chromosome is assigned a fitness value based on the above step. The value will be more if the number of vertices selected is less.

Step 5. The above step is followed by the ordering of the sequences formed on the basis of their fitness value and then applying roulette wheel selection.

Step 6. The final population generated is subjected to crossover and mutation operations in order to further optimize the results.

Step 7. Many generations can be created and the results of the previous with the next one be compared.

However, it may be stated at this point that there can be some graphs in which optimization of the result is not possible, for example, a graph in which the vertex cover is the set of vertices itself.

23.10 MAXIMUM CLIQUE PROBLEM

The problem has been discussed in the previous chapters. However, a brief overview of the problem has been given in the section. The overview is followed by an overview of the technique for applying GA to the problem.

A clique of a graph G is a set of vertices MC (maximal clique) in which $\{u,v\} \in$ MC implies $\{u,v\} \in E$. A maximum clique of a graph G is a clique whose size is as large as that of any other clique in the graph G. A maximal clique (MC) of a graph G is a clique for which it is not possible to add an additional vertex to MC and MC remains a clique (Kleinberg, 2011).

In a graph, vertices may represent individual genes. In such cases, the edges between vertices depict the molecules that are interrelated. Such molecules, as a matter of fact, have high tendency of co-occurrence. By this analogy, a clique should indicate the set of modules which are correlated to each other. The maximum clique problem, therefore, is also important in computational biology.

The brute force algorithm for finding out the maximum clique will not work for a graph having many vertices, in this case also. In such cases, we can use GAs. The solution of the problem via GAs would be on the same lines as the previous problems. In order to apply GAs to maximum clique problem, the following steps may be followed (Algorithm 23.10).

23.10.1 Solution of Maximum Clique via GAs

Algorithm 23.10 GAs approach for maximum clique problem

Step 1. Generate an initial population having n chromosomes.

Step 2. The number of vertices in each chromosome should be same as the number of vertices in the graph.

Step 3. The population is binary; therefore, a mapping is needed to map each chromosome with the set of vertices. A '1' in a chromosome depicts the inclusion of the vertex, whereas a 0 depicts the non-inclusion.

Step 4. Now, the sequence selected will be checked for feasibility. Each chromosome is assigned a fitness value based on the above step. The value will be more if the number of vertices selected is more.

Step 5. The above step is followed by the ordering of the sequences formed on the basis of their fitness value and then applying roulette wheel selection.

Step 6. The final population generated is subjected to crossover and mutation operations in order to further optimize the results.

Step 7. Many generations can be created and the results of the previous with the next one should be compared.

However, it may be stated at this point that like in the case of vertex cover, there can be some graphs for which optimization of the result is not possible.

23.11 CONCLUSION

This chapter throws light on the applicability of GAs to those problems which cannot be solved by standard algorithmic approaches. However, it may be stated that first of all a theoretically sound problem reduction approach is needed in order to apply GA. This should be followed by the design of a good fitness function. The application of crossover and mutation should also not be mindless. The mutation rate should always be low. It may also be stated that our approach can be tested and verified by changing the crossover rate and finding the optimal rate. Same can be done for the mutation rate also. The type of crossover can also be changed to see which type best suits the problem at hand (both in terms of nature and data). Lastly, we should be clear about the goals. If we want to achieve good optimization GAs but if the goal is robustness, then approaches such as diploid genetic algorithms can be used.

Points to Remember

- Genetic algorithms are heuristic search processes.
- The fundamental operators of GAs are crossover, mutation, and replication.
- Crossover can be of many types like one-point crossover, two-point crossover, and uniform crossover.
- The crossover rate should not be too high.
- Mutation is generally achieved by flipping a bit of the binary population.
- The mutation rate is generally much lower than the crossover rate.
- Genetic algorithms do not always provide the correct solution.

KEY TERMS

Artificial intelligence It may be defined as the capability of a computer to perform those activities that are usually done via manual intelligence.

Crossover The crossover population amalgamates two chromosomes in order to produce a new chromosome.

Genetic algorithms They are heuristic search processes based on the theory of survival of the fittest.

Mutation in a binary chromosomes In the mutation operation, a chromosome is randomly selected and one of its bits is flipped. If it is '1' then it is made '0', else it is made 1.

Selection The selection process selects the chromosomes having high fitness values from amongst the population and replicates those chromosomes.

EXERCISES

I. Multiple Choice Questions

1. Which of the following statements is correct?
   (a) Genetic algorithms are heuristic search processes based on the theory of survival of the fittest
   (b) Genetic algorithms are like a black box which optimize results
   (c) Genetic algorithms can be used mindlessly without any problem reduction approach for publishing papers in software engineering.
   (d) All of the above

2. Which of the following is not an operator in genetic algorithms?
   (a) Crossover   (c) Reproduction
   (b) Mutation   (d) Reincarnation

3. For which type of problems GAs are most appropriate?
   (a) Optimization   (c) Pattern machining
   (b) Solving food problems   (d) All of the above

4. A simple GA population consists of which of the following?
   (a) Chromosomes   (c) Genotypes
   (b) Ketones   (d) None of the above

5. What is the purpose of applying mutation operator?
   (a) Breaking global maxima   (c) Amalgamation of features
   (b) Breaking local maxima   (d) None of the above

6. What is the purpose of applying crossover operator?
   (a) Breaking global maxima   (c) Amalgamation of features
   (b) Breaking local maxima   (d) None of the above

7. Which of the following is a type of crossover?
   (a) Single-point crossover   (c) Uniform crossover
   (b) Multipoint crossover   (d) All of the above

8. Which of the following should be low?
   (a) Crossover rate   (c) Both of them should be equal
   (b) Mutation rate   (d) Any of the above can be higher

9. What can be used to search a solution in a huge space?
   (a) Genetic algorithms   (c) Both
   (b) Fuzzy logic   (d) None

10. Which of the following is better?
    - (a) Genetic algorithms
    - (b) Randomized algorithms
    - (c) Both are equally good
    - (d) None of the above

II. Review Questions

1. What is artificial intelligence?
2. What is meant by machine learning?
3. Classify the techniques of machine learning.
4. Explain symbol-based learning.
5. Explain the concept of learning. Can it be implemented in computers?
6. Explain the model of neural network.
7. What are genetic algorithms? For what type of problems are they used?
8. Explain the various steps involved in GAs.
9. What are the different types of crossovers?
10. What is the reason for carrying out mutation?
11. Explain the process of reproduction in genetic algorithms.
12. Examine the steps in order to solve travelling salesman problem using GAs.
13. Explain the steps to solve subset sum problem using GAs.
14. Explain the steps in order to solve vertex cover problem using GAs.
15. Explain the steps to solve maximum clique problem via GAs.

III. Numerical Problems

1. Design appropriate fitness function for the following problems (different from that given in the text)
    - (a) Subset sum
    - (b) Vertex cover
    - (c) Maximum clique
    - (d) Travelling salesman problem
    - (e) Knapsack problem
2. Implement GAs taking crossover rate = 2% and number of chromosomes in the initial population = 100. Plot the graph of fitness value obtained in various generations n in the following problems.
    - (a) Subset sum problem: Input set $\{1, 2, 5, 9, 10, 15, 17, 23, 25\}$. Desired sum =32
    - (b) Vertex cover problem

Input graph
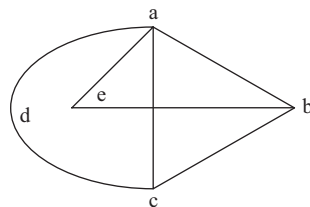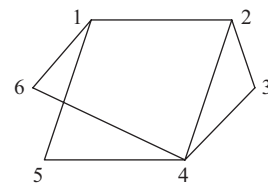
Graph-1

Graph-2

(c) Maximum clique problem

Input graph
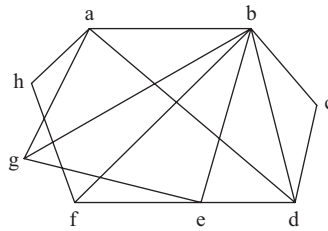
Graph-1

3. Repeat the above problem taking crossover rate as 80%. It may be stated at this point that some of the research papers have taken crossover rate as 60–80%. Analyse the results obtained and state the problems in taking such high crossover rates.

4. Repeat problem number 2 by taking mutation rate as 0.2%. Plot a graph of the fitness value and the generation number.

Answers to MCQs

1. (a)	3. (a)	5. (b)	7. (d)	9. (a)
2. (d)	4. (a)	6. (c)	8. (b)	10. (a)

CHAPTER 24

Introduction to Computational Biology and Bioinformatics

OBJECTIVES

After studying this chapter, the reader will be able to

• Understand the concept of bioinformatics
• Differentiate between bioinformatics and computational biology and between DNA computing and bioinformatics
• Explain the applications of bioinformatics
• Explain the basic terms used in bioinformatics literature
• Use algorithms to solve problems in biology

24.1 INTRODUCTION

Biology and computer science are like Jensen and Winchester of series Supernatural. The two protagonists in the series go around the country to hunt friends. A viewer in America cannot think of the series with just one character. The two disciplines computers and biology also have come together to find the demon of 'unsolved problems' in biology through computer science. It is a win-win situation for both. Biology will have the answers for many problems, which are dependent on empirical analysis. Computer science, on the other hand, will find some real-life problems to solve.

As a matter of fact, biology is not just dependent on computer science, but also on statistics. This is because of the huge data that is constantly generated by numerous sources. These data require quantization tools, and the use of these tools has made the people who work on them and create them immensely important for biology. Let us just go through a few examples so as to why statistics is important in biology. The statistical and mathematical techniques help manipulate biological databases, which are used to store information. The tools help us to answer biological questions, count bacteria colonies, etc. The importance of counting can be gauged from the fact that just by counting the variations, Gregor Mendel and Thomas Morgan derived the laws of genetic inheritance. The discipline also helps in another immensely important process, that is, drug designing. Although this topic is beyond the scope of this book, it is the one of the most important application of bioinformatics.

This chapter has been designed for a computer science student. The goal is to introduce him/her to the fascinating discipline of bioinformatics and computational biology; so that he should be able to apply the skills he has developed during the course to solve some real-life problems. The chapter is not meant to be taken as a 'life science' text.

24.2 BASICS OF COMPUTATIONAL BIOLOGY AND BIOINFORMATICS

Broadly, both computational biology and bioinformatics have same goals. An extensive literature review was carried out to write this chapter. Many contradicting views were reported as regards the difference between bioinformatics and computational biology. One of the most convincing distinctions was given by Achuthsankar S. Nair. According to him, bioinformatics is for the life science people who master the use of computational techniques, whereas computational biology is for the computer science people who learn biology. The goals of the two are seemingly same (Luscombe, 2001).

According to many researchers, bioinformatics is not just the analysis of biological data but much more than that. It is used in drug designing to locate genes via DNA sequencing, predicting the structure of RNAs through the sequences, and so on. Analysis of data, nevertheless, remains one of the most important goals of this discipline.

> **Definition** Bioinformatics is a field that applies computer science to solve the problems of biologists and is concerned with the management and analysis of biological data.

The biological data that bioinformatics intends to work upon can be stored using databases and worked upon using ontologies. Each domain of disclosure has a set of entities that are fundamental to it. The formal name, properties, and interrelationship of these types are called ontology. The analysis of these data requires the knowledge of algorithms, which we have studied till now. This chapter also introduces some of the important algorithms to accomplish specific tasks in the discipline.

The term 'bioinformatics' was coined by a Dutch biologist, Paulien Hogeweg. Her dedication towards the discipline can be judged from the fact that she founded the 'Theoretical Biology and Bioinformatics Research group' in 1977. The most important events related to the growth of the discipline are listed in Table 24.1.

The concept central to the subject is sequencing, and the problems involved therein. For example, the huge amount of data makes the case of conventional algorithms too weak.

This discipline has no commonality with DNA computing. DNA computing is concerned with the creation of biocomputers using DNA, etc. The latter uses biology to make computers and bioinformatics uses computer science to solve problems in biology.

> **Definition** DNA computing is concerned with the creation of biocomputers using DNA and enzymes.

Table 24.1   Important events in bioinformatics

Year	Event
1956	The first protein sequence of Bovine insulin reported
A decade later	First nucleic acid sequence reported
1972	Creation of Protein Data Bank
1987	The SWISSPROT protein sequence database began
1988	The Human Genome organization (HUGO) founded
1989	The first complete genome map was published
1990	The Human Genome Project started
1993	A physical map of the human genome was produced by Genethon, Human Genome Research Centre in France
1996	The final version of the Human Genetic Map was produced by Genethon

Bioinformatics is also used in many disciplines, some of which are as follows.
- Signal processing
- Genomics, which deals with finding the function and structure of a genome using sequencing, etc.
- Biophysics, which uses the theory of physics to study biological systems.
- Biochemistry, which deals with the study of chemical processes in biological systems.

Sequencing is one of the most important tasks in bioinformatics. The task involves the determination of the order of amino acids in a protein or that of nucleotides in a deoxyribonucleic acid (DNA) or a ribonucleic acid (RNA).

Strictly speaking, computational biology is not concerned with the biomedicines, rather with the evolutionary biology. However, the use of data-intensive methodologies is as important in computational biology as in bioinformatics.

24.3 BASICS OF LIFE SCIENCES

This section discusses the basics of life sciences for a student of computer science. The goal of the following discussion is to briefly describe the terms used in bioinformatics/computational biology. As stated earlier, the following text is not meant for life science people.

24.3.1 Cell

The cell, discovered by Robert Hooke in 1665, is the basic unit of a living being. It is not only the basic structural unit but also the basic functional unit, as they also contain the hereditary information. Organisms are made up of cells. On this basis, there can be two segregations of animals:
- Unicellular, those made up of a single cell
- Multicellular, those made up of many cells.

A cell contains protoplasm within a membrane. The various components of a cell are shown in Fig. 24.1. The membrane envelopes a cell and maintains the requisite potential of a cell. Almost all the cells contain DNA and RNA. These are described in the following sub-section. However, the DNA contains the heredity information and the RNA has the functionality to build proteins. The structure of a cell is maintained by cytoskeleton.
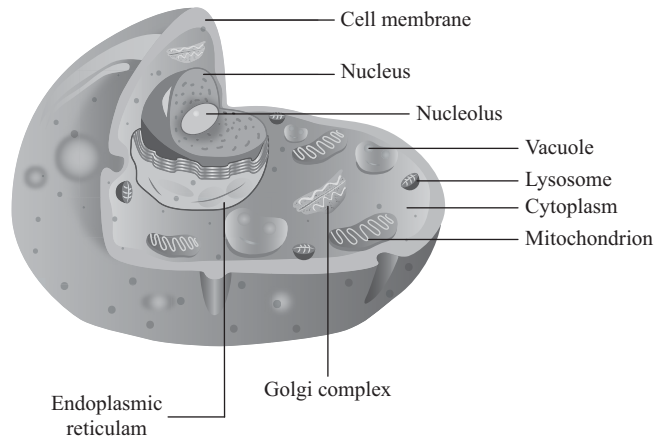
Figure 24.1   A Cell

A cell may be prokaryotic or eukaryotic. The former does not contain a nucleus, whereas the latter contains nucleus. The above classification is also valid as per organisms are concerned. The lower forms of life are referred to as prokaryotic. One of the examples of a prokaryotic is a bacterium. Eukaryotic are somewhat developed organisms. Some of the examples of eukaryotic are human beings and animals that we see around us.

24.3.2   DNA and RNA

There are two kinds of genetic material, DNA and RNA. Interestingly, the stored program concept of computer science would be the best analogy to explain this concept. Like the computer contains both the data and the program. The program would operate on the data. In the same way, the cell contains both the material for protein synthesis and the algorithms required therein.

The basic unit of a DNA or an RNA is a nucleotide. A nucleotide has a base, a phosphate group, and sugar. The bases are adenine (A), cytosine (C), guanine (G), and thymine (T) in the case of DNA and A, C, G, and U (uracil) in the case of RNA. In DNA, the sugar is deoxyribose while that of an RNA is ribose.

DNA is a nucleic acid that encodes the genetic information. A DNA has double-stranded structure. These are held together by bonds. Though a DNA contains two strands, the information can be gathered from a single strand. These bonds are formed between the basepairs. In the case of DNA, these pairs can exist between A and T, and G and C (Fig. 24.2). It may be stated here that DNA is used for storing biological information.
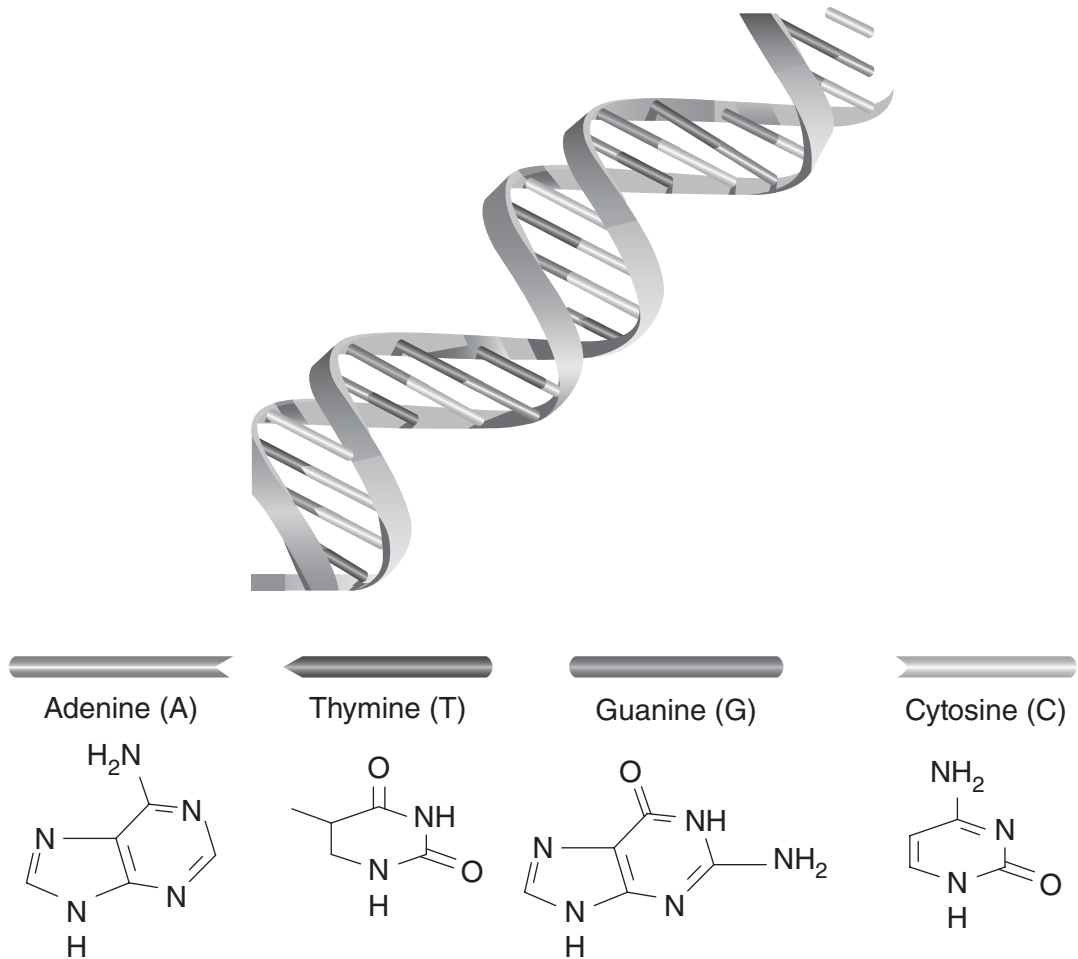
Figure 24.2 DNA

The concept of information and the hugeness of data can be gauged from the fact that a single drop of blood contains information regarding the cell, which helps to gather information regarding nucleus. This information, in turn, can be used to find details of genomes. The data that can be generated using this can be as long as 3.2 GBs of text.

RNA helps not only in encoding but also in the decoding and regulation of genes. RNA, though somewhat similar to DNA, is less stable. There are three types of RNA:

• transfer RNA (t-RNA)
• messenger RNA (m-RNA)
• and ribosomal RNA (r-RNA)

It may also be stated here that an RNA is single-stranded unlike a DNA (Fig. 24.3).
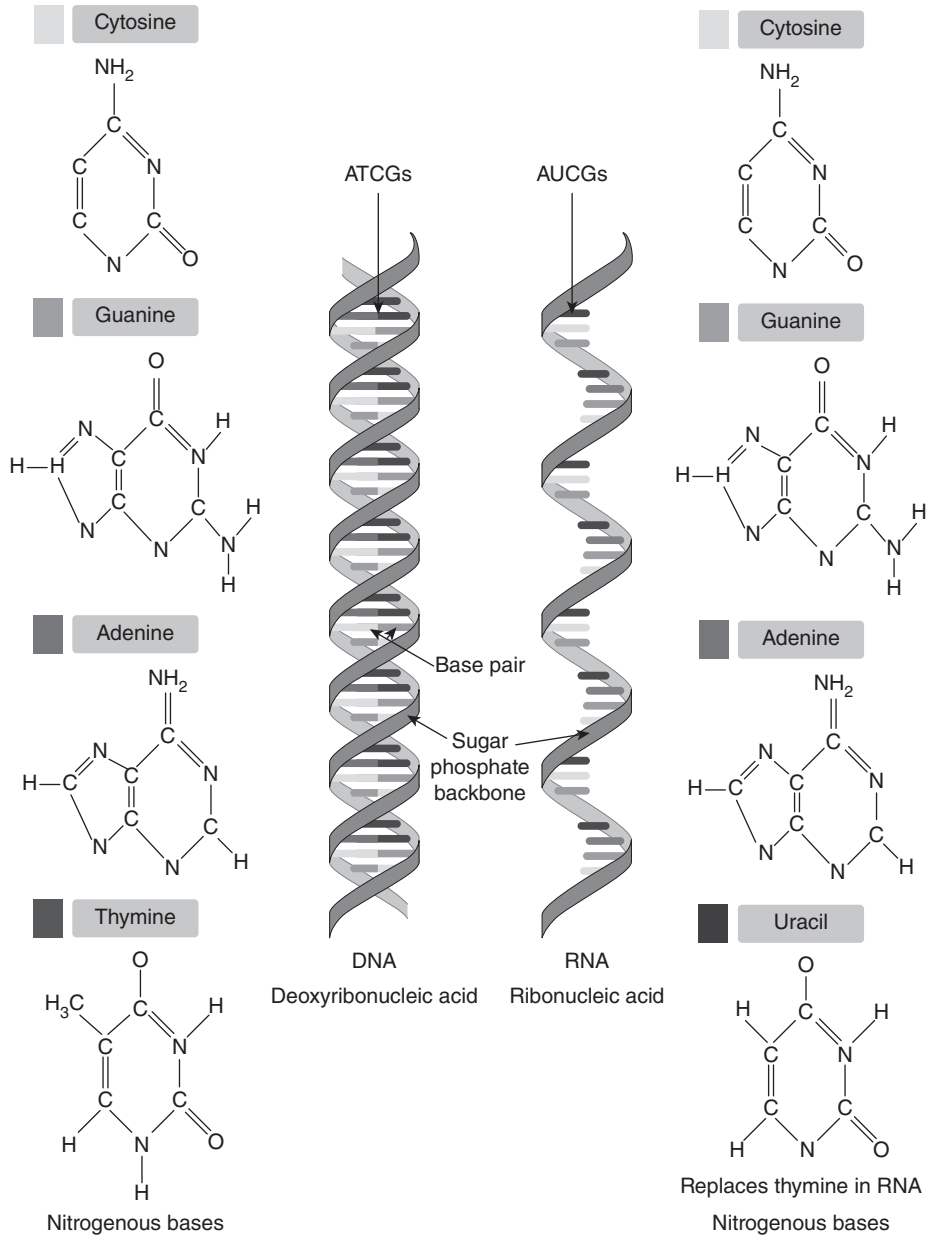
Figure 24.3 Comparison of DNA and RNA

24.3.3 Genome

The DNAs are packed into chromosomes. A chromosome has DNA, RNA, and proteins. There are 23 pairs of chromosomes in a human being. These are collectively known as genomes. The study of these genomes can lead to fascinating discoveries. This study is referred to as genomics.

The study of genomes has also revealed that the humans differ from one another only by 0.2% of their genome. The analysis of these data clearly points to the fact that we are basically similar.

24.3.4 Amino Acids

An amino acid is a basic constituent of a protein. Chemically, an amino acid contains both amino group and a carboxylic acid group. In animals, there are 20 amino acids and in plants, there are more than a hundred. Of these 20 amino acids, 12 are produced within the body of the adult human beings. The various amino acids are given in Table 24.2, and the structures of amino acids are depicted in Fig. 24.4.

A protein is a biomolecule that is produced by cells using the information contained in the DNA. The structure of a protein can be one of the following.

Table 24.2 Amino acids

Name	Abbreviation
Alanine	A
Arginine	R
Asparagines	N
Aspartic acid	D
Cysteine	C
Glutamine	Q
Glutamic acid	E
Glycine	G
Histidine	H
Isoleucine	I
Leucine	L
Lysine	K
Methionine	M
Phenylalanine	F
Proline	P
Serine	S
Threonine	T
Tryptophan	W
Tyrosine	Y
Valine	V
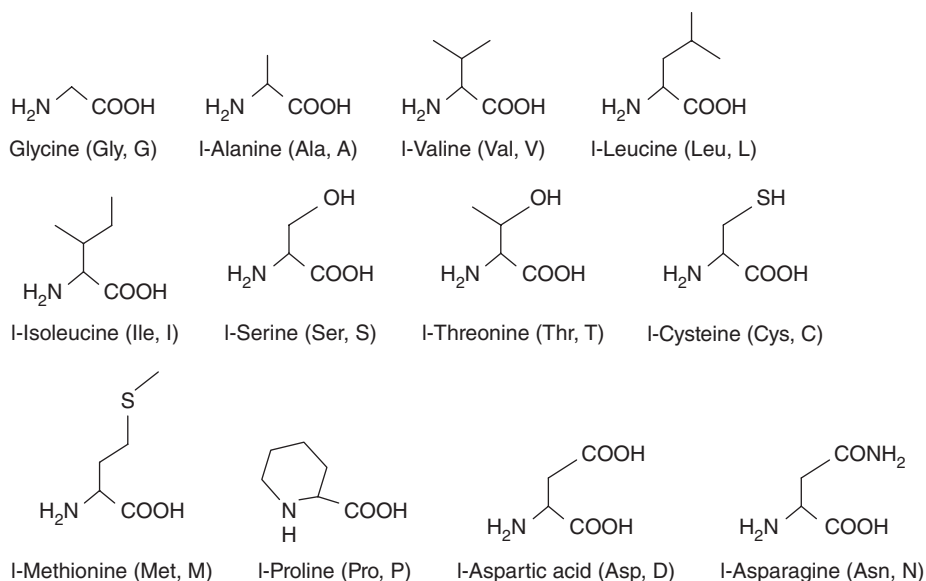
Figure 24.4 Structures of amino acids (*Contd*)
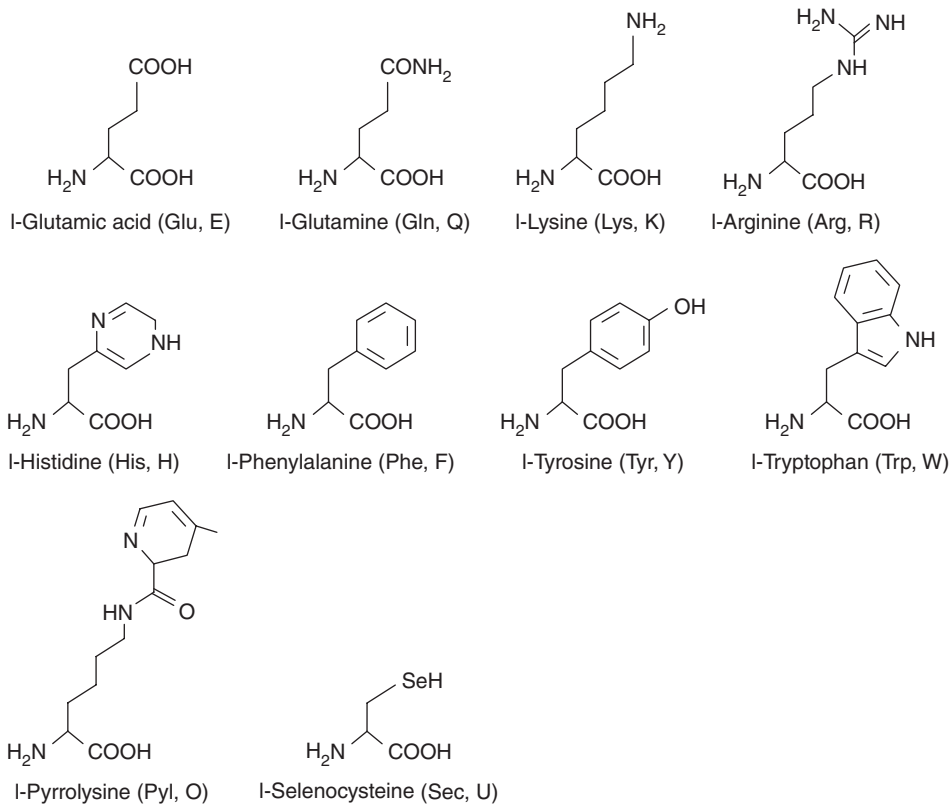
Figure 24.4 (Contd) Structures of amino acids

Structures of Protein

A protein can have the following structures:
- **Primary**: Containing the linear sequence of amino acids.
- **Secondary**: It determines the regions of local regularity, within a fold.
- **Tertiary**: The overall fold of protein sequence.
- **Quaternary**: The structure that considers the protein–protein and protein–nucleic acid interactions.

Having gone through the basics of life sciences, let us now move on to sequencing and the problems therein.

24.4 SEQUENCING AND PROBLEMS THEREIN

As stated earlier, sequencing is one of the most important tasks in bioinformatics. The protein sequencing, which earlier relied on the separation of protein and peptide followed by identification of amino acids, has come a long way. The number of sequences available today is around 30,000. This sequencing required special methods. The earlier sequencing methods were applicable to RNAs only. The reason being that they are short, containing a maximum of 95 nucleotides.

A human DNA, on the other hand, contains as many as 250×10^6 basepairs. A single experiment can sequence up to 500 basepairs. So in order to sequence a DNA, many basepairs would have to be segregated. In the recent past, many methods of DNA sequencing have been developed.

Many advanced methods of collection of biological data have been developed. This data need to be analysed and managed. This analysis would help in genome analysis, robotics, etc. These methods are collectively referred to as bioinformatics.

24.4.1 Sequence–Structure Deficit

As stated earlier, the number of sequences stored in the sequence database is doubling each year. However, the number of unique 3D structures in Protein Data Bank is less than 1500. This is called the sequence–structure deficit.

The quest to understand a human genome started as early as 1980s, when the project to determine the full nucleotide sequences and to locate some genes began in the United States. There was, therefore, an urgent need to develop new methods for the analysis. This is called the genome project.

24.4.2 Folding Problem

Scientists have been able to find many sequences. Now, the question that arises is the ability of these sequences to dictate the formation of other sequences. For example,

- Can we craft a protein structure using an amino acid sequence?
- Can the structure of proteins be predicted by their sequences?
- Can the linear sequence of amino acids find the final 3D fold?

The concept, though theoretically feasible, has kept many scientists puzzling. The research was started around half a century ago. Now, it has grown into a full-fledged research topic.

> **Tip:** One of the most important applications of finding sequence is homology and analogy. Homology is finding out whether two sequences are related by a divergence in the common ancestor. When the folds are similar but the sequences are different, then they are called analogous proteins.

24.5 ALGORITHMS

Now let us see how algorithms can help us in bioinformatics. The two most important problems in bioinformatics are

- Pattern recognition
- Prediction

In both the cases, algorithms come to our rescue. The methods that we have studied so far, such as dynamic algorithms, help us in sequencing. However, there is a problem

the things have come to such a pass that comparisons fail to detect structural similarity and even alignments are not significant. The following discussion explains some of the most basic algorithms.

The discussion that follows discusses the fundamentals of sequence comparison and sequence alignment. In order to compare two sequences, they need to be aligned. The result of alignment is comparison. As a matter of fact, more than two sequences can be aligned. There are many types of sequence alignments. Some of them have been stated in the following section. The section also discusses the scoring schema which is used in the algorithm that follows.

Sequence comparison is used not just to find the variation or similarity in two sequences but also to find whether the given sequences have emerged from some common sequence. One of the easiest methods of sequence comparison is the Dotplot. The concept of Dotplot is very simple. The two sequences are written in the headers of the row and column of a 2D matrix. The cell where the row and the column have the same value is marked with a X. The formation of the final matrix is followed by finding the longest diagonal of X's.

In order to understand the Dotplot, let us consider two sequences ATGATGCTGA and TAGCATCTGA. The corresponding Dotplot is shown in Fig. 24.5.

	A	T	G	A	T	G	C	T	G	A
T		X			X			X		
A	X			X						X
G			X			X			X	
C							X			
A	X			X						X
T		X			X			X		
C							**X**			
T		X			X			**X**		
G			X			X			**X**	
A	X			X						**X**

Figure 24.5 Dot plot

The longest sequence as per the above Dotplot is CTGA (the dark X's).

The advantage of Dotplot is that it is visual and hence easy to comprehend. The problem, though, is that if the sequence is too large then this method becomes cumbersome.

Protein sequence alignment is a bit more complex as compared to DNA sequence alignment. There are 20 amino acids. Properties such as size, polarity, charge, and hydrophobicity of these amino acids are also considered while aligning the sequence. This makes the alignment all the more difficult.

There is another method of comparison, that is, sequence alignment. In this method, the best alignment between the sequences is determined, which is used to find the similarity. The various types of sequence alignment are as follows:

- Global
- Local
- Multiple sequence

The global sequence alignment finds similarity over the entire length of the sequence. The method is suitable if the given sequences are of equal length. On the other hand, if the given sequences have short patches of similarity, then local similarity is used. When more than two sequences are to be compared, then multiple sequence method is used.

The alignment is done with the help of a scoring scheme. In a scoring scheme, higher the score, more is the similarity. In a scoring scheme, an amino acid residue is matched, the score increases, and penalty is given for a mismatch. One of the easiest ways of doing this is to increase the score by 1 when the residues match, 0 if we cannot say anything regarding the match, and −1 if a mismatch occurs. The method can also be implemented using a matrix.

As stated earlier, the best global alignment is difficult to find. Let us see why. The brute force strategy of finding the best alignment would be to enlist all the alignments and then find the one with the best score. The possible number of alignments between two sequences of length N is $\dfrac{2^N}{\sqrt{\pi N}}$.

This massive number of alignments can be reduced by dynamic programming as explained in Chapter 11. The algorithm has been discussed in Section 11.3 of Chapter 11.

24.6 CONCLUSION

The chapter introduced the discipline of bioinformatics. It has greatly helped in many fields such as drug design. It is now being used to tackle problems, which at one point in time were considered impossible to solve. For instance, the proteins produced by our body can become faulty. This can be detected by finding active sites of molecules. Computational biology helps us to tackle the above problem and many more such problems.

The reader of this book may not be well versed in biology. This is the reason why a primer of biology-related terms, which are used in the discipline, has been included in the chapter. The goal of including this chapter in the book is to motivate the reader to use the algorithms studied till now to handle real-life problems. One of the most important tasks in bioinformatics is sequence detection. The concepts studied in dynamic algorithm would help in solving the problem.

Points to Remember

- DNA computing is concerned with the creation of biocomputers using DNA, etc.
- A cell may be prokaryotic or eukaryotic. The former do not contain a nucleolus, whereas the latter contains a nucleus.
- DNA is a nucleic acid that encodes genetic information. The nucleotide in DNA contains the bases namely adenine (A), guanine (G), cytosine (C), and thymine (T).
- A DNA has double-stranded structure.
- RNA helps not only in encoding but also in decoding and regulation of genes.
- RNA, though somewhat similar to DNA, is less stable. There are three types of RNA: transfer RNA (t-RNA), messenger RNA (m-RNA), and ribosomal RNA (r-RNA).
- There are 23 pairs of chromosomes in a human being. These are collectively known as genome.
- The study of genomes has also revealed that humans differ from one another only by 0.2% of their genome.
- There are many types of sequence alignment.
- Sequence comparison is used not just to find the variation or similarity in two sequences but also to find whether the given sequences have emerged from some common sequence.
- One of the easiest methods of sequence alignment is the Dotplot.

KEY TERMS

Bioinformatics It is a field that applies computer science to solve the problems of biologists and is concerned with the management and analysis of biological data.
DNA computing It is concerned with the creation of biocomputers using DNA and enzymes.

EXERCISES

I. Multiple Choice Questions

1. Which of the following uses computer science to solve problems of biology?
   - (a) Computational biology
   - (b) Bioinformatics
   - (c) Both
   - (d) None of the above
2. Which of the following statements is true?
   - (a) DNA computing is same as bioinformatics
   - (b) DNA computing is same as computational biology
   - (c) Both
   - (d) None of the above

3. Which of the following can be used to store biological data?
   (a) Ontology
   (b) Databases
   (c) Both
   (d) None of the above

4. Bioinformatics is used in which of the following?
   (a) Signal processing
   (b) Genomics
   (c) Biophysics
   (d) All of the above

5. Sequencing is a task that can be accomplished by which of the following?
   (a) Biochemistry
   (b) Bioinformatics
   (c) Biophysics
   (d) None of the above

6. Which of the following envelopes a cell and maintains the required potential difference?
   (a) Cell membrane
   (b) Cytoskeleton
   (c) Both
   (d) None of the above

7. Which of the following maintains the structure of a cell?
   (a) Cell membrane
   (b) Cytoskeleton
   (c) Both
   (d) None of the above

8. A cell that contains a nucleus is
   (a) Prokaryotic
   (b) Eukaryotic
   (c) Both
   (d) None of the above

9. A cell that does not contain a nucleus is
   (a) Prokaryotic
   (b) Eukaryotic
   (c) Both
   (d) None of the above

10. Which of the following bases are contained in a DNA?
    (a) Adenine
    (b) Guanine
    (c) Cytosine
    (d) Thymine
    (e) All of the above

11. A DNA has which type of structure?
    (a) Double-stranded structure
    (b) Single-stranded structure
    (c) Both
    (d) None of the above

12. Which of the following is not a type of RNA?
    (a) m-RNA (b) t-RNA (c) r-RNA (d) x-RNA

13. Which of the following bases is not present in DNA?
    (a) A (b) C (c) U (d) G

14. How many pairs of chromosomes are present in a human being?
    (a) 23
    (b) 46
    (c) 13
    (d) None of the above

15. An amino acid contains
    (a) An amino group
    (b) A carboxylic acid group
    (c) Both
    (d) None of the above

16. How many amino acids are present in an animal?
    (a) 20
    (b) 40
    (c) 10
    (d) None of the above

17. How many base pairs are there in a human DNA?
    (a) 3×10^9
    (b) 3×10^8
    (c) 2×10^{10}
    (d) None of the above
18. Which of the following is the folding problem?
    (a) To craft a protein structure using an amino acid sequence
    (b) To predict the structure of proteins from their sequences
    (c) To find the final 3D fold from the linear sequence of amino acids
    (d) None of the above
19. Which of the following properties of amino acids are also considered while aligning the sequence
    (a) Size
    (b) Polarity
    (c) Charge
    (d) Hydrophobicity
    (e) All of the above
20. One of the easiest methods of sequence alignment is
    (a) Dotplot
    (b) Dotdot
    (c) Plotplot
    (d) Plotdot

II. Review Questions

1. Explain folding problem.
2. What is meant by sequencing and what are the problems in carrying out sequencing?
3. What is bioinformatics? How is it different from DNA computing?
4. Explain the scope and importance of bioinformatics.
5. Briefly explain the history of bioinformatics.
6. What is computational biology? Is it different from bioinformatics?
7. Explain how analysis and design of algorithms help us to solve problems in biology?

Answers to MCQs			
1. (c)	6. (a)	11. (a)	16. (a)
2. (d)	7. (b)	12. (d)	17. (a)
3. (c)	8. (b)	13. (c)	18. (d)
4. (d)	9. (a)	14. (a)	19. (e)
5. (b)	10. (e)	15. (c)	20. (a)

FEATURES OF

CHAPTER 20

An Introduction to PSpace

CHAPTER 22

Parallel Algorithms

CHAPTER 23

Introduction to Machine Learning Approaches
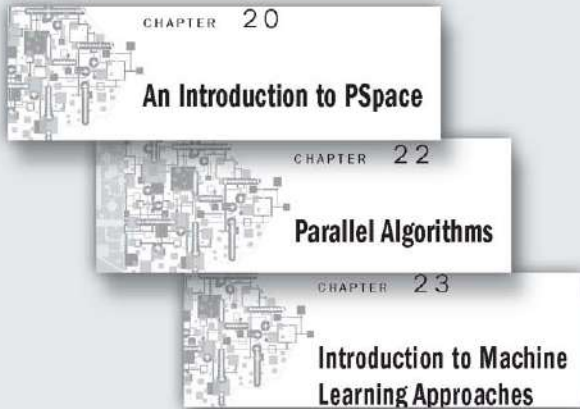
Span of Coverage

The book covers fundamental concepts and complexity analysis of algorithms in Section I, data structures in Section II, and various design techniques in Section III. Finally, in Section IV, it deals with the advanced topics such as decrease and conquer, transform and conquer, number theoretics, PSpace, parallel algorithms, and applications of algorithms in machine learning and computational biology.

Treatment of Concepts

The explanations and problems related to design paradigms are supported with mathematical expressions, various examples, and algorithms.

Algorithm 9. 5 Selection

Input: Array a[], and the first and the last index
Output: The requisite element

SELECTION (a[], int low, int high, int k) returns val

Note:
1. The above algorithm assumes that $a[]$ is unsorted. The u only if $a[]$ is unsorted. Had $a[]$ been sorted we would h straight away.

Complexity: The above algorithm runs $O(n)$ in the worst ca case.
   The recursive algorithm of SELECT uses the same pre correct element is not found in the first iteration, the left or is explored as per the case.

Tip: The dynamic implementation of a queue can be begin() algorithms of linked lists (refer to Section 5.4).

Tips

The text is interspersed with Tips that highlights important points in each chapter.

Points to Remember

Points to Remember section at the end of each chapter enables quick recapitulation of the important concepts discussed in the chapter.

Points to Remember

· Backtracking can be easily implemented using recursion
· There are techniques such as branch and bound backtracking.

THE BOOK

Figures and Tables

Each chapter is interspersed with numerous simple figures and tables that complement the discussions in the text.
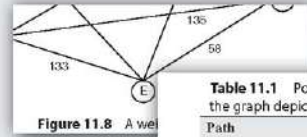
Figure 11.8 A we[...]

Table 11.1 Possible paths in the graph depicted in Fig. 11.8

Path	Cost
AECBDA	610
AECDBA	516
AEDBCA	588

Illustration 10.2 Trace the steps of Kruskal's algorithm Fig. 10.13.

Solution

Solved Examples

Numerous simple and relevant solved examples are given as *Illustrations* in each chapter to augment the understanding of concepts.

Chapter-end Exercises

All chapters end with an Exercises section that include multiple choice questions with answers, review questions, and application-based questions/numerical problems.

EXERCISES

I. Multiple Choice Questions

1. Who coined the term backtracking?

II. Review Questions

1. Explain the process of backtracking. What are the adv against brute force algorithms?

III. Application-Based Questions

1. Implement *N*-queens problem via backtracking.
2. Implement graph colouring problem via backtracking [...] lowing maps.

Answers to MCQs

| 1. (a) | 3. (d) | 5. (a) | 7. (a) |
| 2. (a) | 4. (d) | 6. (d) | 8. (a) |

APPENDIX A2

2-3-4 and Red–Black Trees

APPENDIX A4

Linear Programming

APPENDIX A6

Probability

APPENDIX A10

Problems

Appendices

Appendices (A1 to A9) deal with topics on probability, matrix operations, Red-black tress, linear programming, DFT, scheduling, and a reprise of sorting, searching, and amortized algorithms. The last Appendix A10 includes some interesting problems based on almost all the topics discussed in the book for practice and better understanding.

Companion Online Resources for Faculty and Students

To aid teachers and students, the book is accompanied with online resources which are available at http://oupinheonline.com/book/bhasin-algorithms/9780199456666. The contents of online resources include:

For Faculty
- Chapter-wise PowerPoint Slides
- Solution manual for select chapter-end problems
- Assignment questions with answers

For Students
- Additional MCQs for test generator (with answers) for each chapter
- C language implementation of algorithms
- Interview questions with answers

Steps to Register

Step 1: Getting Started
- Go to www.oupinheonline.com

Step 2: Browse quickly by:
- BASIC SEARCH
  - → Author
  - → Title
  - → ISBN
- ADVANCED SEARCH
  - ▷ Subjects
  - ▷ Recent titles

Step 3: Select Title
Select title for which you are looking for resources.

Step 4: Search Results with Resources available

Click here

Step 5: To download Results
Download All

Step 6: Login to download
The page you wish to access is password-protected.
Please login to access the resources.
User Name :
Password :
Login
Forgot Password? | New Student? Register | New Instructor? Register

New Instructor? Register

Step 7: Registration Form
Please fill correct details and *marked fields are mandatory

Instructor Registration

Personal Details
User Name *
(Enter a valid email address as your User Name) e.g. support@gmail.com
Password *
Confirm Password *
Security Question *   --Select--
Security Answer *
Salutation *   --Select--
Name *
Designation *
Department *
Mobile *
Alternative Email Id

Username should be email ID

Please fill correct Mobile no. to get SMS after verification

Institute Details
Institute Name *   (Please enter full Institute Name.)
University *   --Select--
Address *
(Please enter complete address of the institute.)
Country *   --Select--
State
City *

Course Taught

S.No.	Course	Sem/Year	#.enrolment
1		--Select--	
2		--Select--	
3		--Select--	
4		--Select--	
5		--Select--	

Submit   SUBMIT

Step 8: Message after completing the registration form
Thank you for registering with us. We shall revert to you within 48 hours after verifying the details provided by you. Once validated please login using your username and the password and access the resources.

Step 9: Verification
You will receive a confirmation on your mobile & email ID.

For any further query please write to us at **HEMarketing.in@oup.com** with your mobile number

Step 10: Visit us again after validation
- Go to www.oupinheonline.com
- Login from Member Login

Member Login

User Name :

Password :

Login
Forgot Password?

Step 11: My Subscriptions

My Subscriptions

Valid Subscriptions

Operations Research, 1/e
Yadav & Malik
Valid Till :
16 Oct 2015

OPERATIONS RESEARCH

You can view Subscriptions in your account
- Click on the title
- Select Chapter or "Select All"
- Click on "Download All"
- Click on "I Accept"
- A zip file will be downloaded on your system. You may use this along with the textbook.

APPENDICES

Pure mathematics is, in its way, the poetry of logical ideas.

— Albert Einstein

Amortized Analysis–Revisited

OBJECTIVES

After studying this appendix, the reader will be able to
- Explain the concept of amortized analysis
- Learn the concept of `multi_pop` operation in stack
- Understand the idea of dynamic tables
- Apply aggregate, accounting, and potential methods

A1.1 INTRODUCTION

Many data structures have been discussed in the book so far. A data structure comes with its own set of operations, some of which are more expensive than others. For example, a stack has two operations: push and pop. The push operation inserts an item in the given stack, if the stack is not already full. The pop operation takes out an element from the given stack, if the stack is not empty. The push operation is more expensive than the pop operation. If the expensive operations are carried out lesser number of times, then the average cost of an operation can be reduced. It may be stated that the discussion that follows talks neither about probability nor randomization. In amortized analysis, the average of sequence of *n* worst case operations is considered. The amortized analysis presents the average cost per operation in the worst case. It was introduced by Tarjan in 1985. However, the aggregate method which is now considered a part of amortized analysis was introduced way back in 1972. The amortized analysis is of three types: aggregate method, accounting method, and potential method. The topic has already been introduced in Section 4.5 of Chapter 4. The following discussion explains the above three types by taking appropriate examples.

A1.2 AGGREGATE ANALYSIS

In aggregate analysis method, the different operations are not differentiated on the basis of cost. In order to understand the concept, the following example may be considered. One of the most common examples of aggregate analysis is that of `multi_pop` operation in a stack.

As stated earlier, a single pop takes out an item from the given stack. The multi-pop operation takes out k elements from the given stack. The algorithm for `multi_pop` is as follows.

Algorithm A1.1   multi_pop

Input: Stack S; the value of the index TOP; k the number of times pop is to be carried out
Output: The algorithm either removes the data from the stack or displays an error

```
mupti_pop (S, k, TOP)
    {
    if (TOP ==-1)
        {
        Print "Underflow";
        exit();
        }
    else
        {
        multi_pop(S, k-1, TOP--);
        }
    }
```

Complexity: In the worst case, the value of k would be n, thus making the complexity of algorithm $O(n)$. The average complexity would be $O(n)/n$, that is, $O(1)$. However, if one analyse the algorithm, he/she will find that actually this would not be the case. Theoretically, the above discussion suggests that if the `multi-pop` operation is carried out n times, then the complexity would be $O(n^2)$, which would never be the case because k pops would be followed by k pushes. As stated earlier, `push` is less expensive as compared to `pop`. So the total complexity of n `multi-pop` operations would be less than $O(n^2)$. Nevertheless, there can be no case wherein the average complexity exceeds the complexity carried out by the aggregation method.

Conclusion: As stated earlier, in any data structure, not all the operations are of same costs, some are more expensive than others. A task will have a mix of these operations. The aggregate analysis would always find the total time taking into consideration the operation which would be of maximum cost, and then calculates the average. The average found by this method would, therefore, help us to analyse the algorithm in a better way.

A1.3 DYNAMIC TABLES: AGGREGATION, ACCOUNTING, AND POTENTIAL AMORTIZED ANALYSIS ■

Another common example of amortized analysis is that of dynamic tables. Consider a situation wherein we need to allocate space for a table. However, we do not know what

would be the size of the table in the future, that is, the size of the table might change dynamically. In that case, a simple rule can be followed, the size of the table initially, is that of a single row. If a new data arrives and the table is filled, the size of the table becomes double.

- For the first insertion, the size of the table is one unit and at that space the item is inserted.
- For the second insertion, a new table having double the size of previous table, in this case 2 units, is created. Data are copied from the previous table to the new table and then a new item is inserted. Note that the new table is completely filled as one row has been copied from the old table and a new row has been inserted.
- For the third insertion, a new table is created. The size of the new table would be four units. Two rows from the previous table would be copied to the new table and then a new row would be inserted.
- For the fourth insertion, no new table needs to be created.
- When the fifth item is inserted in the table, a new table of size eight units needs to be created. Four items from the previous table would be copied to the new table and then the fifth item would be inserted.
- For the sixth, seventh, and eighth insertions, no new table needs to be created.
- For the ninth insertion, a new table of size 16 would have to be created. Eight items from the previous table would be copied to the new table and then a new item would be inserted in the new table.
- For the 10th, 11th, 12th, 13th, 14th, 15th, and 16th insertions, there would not be any need of creating a new table.

Table A1.1 shows the variation of the table size with the number of items if the above approach is used and Fig. A1.1 depicts the pictorial representation of the scenario. From the table, it can be inferred that the size of the table never exceeds thrice the number of items in the table.
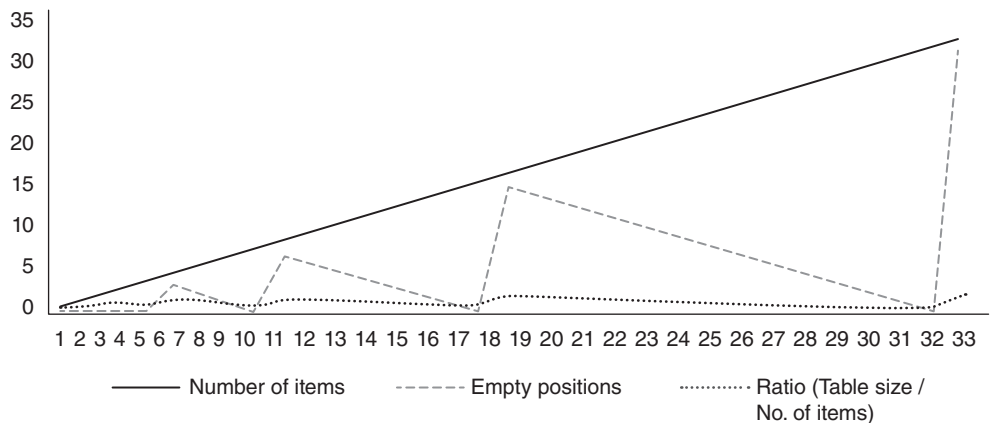
Figure A1.1   Relationship between the number of items and the table size in a dynamic table.

Table A1.1   Size of a dynamic table and the number of items in it

Number of items	New table created	Size of the table	Filled positions	Empty positions	Ratio (table size/ number of items)
1	Yes	1	1	0	1
2	Yes	2	2	0	1
3	Yes	4	3	1	1.333333
4	No	4	4	0	1
5	Yes	8	5	3	1.6
6	No	8	6	2	1.333333
7	No	8	7	1	1.142857
8	No	8	8	0	1
9	Yes	16	9	7	1.777778
10	No	16	10	6	1.6
11	No	16	11	5	1.454545
12	No	16	12	4	1.333333
13	No	16	13	3	1.230769
14	No	16	14	2	1.142857
15	No	16	15	1	1.066667
16	No	16	16	0	1
17	Yes	32	17	15	1.882353
18	No	32	18	14	1.777778
19	No	32	19	13	1.684211
20	No	32	20	12	1.6
21	No	32	21	11	1.52381
22	No	32	22	10	1.454545
23	No	32	23	9	1.391304
24	No	32	24	8	1.333333
25	No	32	25	7	1.28
26	No	32	26	6	1.230769
27	No	32	27	5	1.185185
28	No	32	28	4	1.142857
29	No	32	29	3	1.103448
30	No	32	30	2	1.066667
31	No	32	31	1	1.032258
32	No	32	32	0	1
33	Yes	64	33	31	1.939394

Had we considered just the maximum size of the table, it would have been i in the ith iteration, thus making the total equal to $\sum_{i=1}^{n} i = \dfrac{n(n+1)}{2} = O(n^2)$. However, as per the above analysis, the table size is $O(3n) = O(n)$.

In the accounting method, we credit some number when an insertion is made. If the credit amount is not used at that point in time, it is used at a later stage. The idea is simple. In order to understand the concept, consider that based on the number of electronic devices in your house, you figure out that in no case can the electricity bill of your house can be greater than ₹3000 per month. Now, you allocate ₹3000 each month to the electricity bill in your budget. It is quite possible that the bill is less than ₹3000, in which case you keep the remaining amount for the next bill. This will lead to a situation wherein your credit would never become negative. That is, $\sum_{n}^{i=1} C_j \leq \sum_{n}^{i=1} C_j^*$ where C_i^* is the amortized cost of the ith iteration and C_i is the actual cost.

In the case of dynamic tables, if we assign ₹3 as the amortized cost of insertion, then 1 rupee would be used for immediate insertion and 2 would be stored for the future (in the case of a new table creation). Table A1.2 shows the amortized cost and the actual cost in the case of insertion in a dynamic table.

Table A1.2 Accounting method for dynamic table

Number of items	New table created	Size of the table	Filled positions	Amortized cost	Actual cost	Used	Left
1	Yes	1	1	3	1	1	2
2	Yes	2	2	3	2	2	3
3	Yes	4	3	3	3	3	3
4	No	4	4	3	1	1	5
5	Yes	8	5	3	5	5	3
6	No	8	6	3	1	1	5
7	No	8	7	3	1	1	7
8	No	8	8	3	1	1	9
9	Yes	16	9	3	9	9	3
10	No	16	10	3	1	1	5
11	No	16	11	3	1	1	7
12	No	16	12	3	1	1	9
13	No	16	13	3	1	1	11
14	No	16	14	3	1	1	13
15	No	16	15	3	1	1	15
16	No	16	16	3	1	1	17
17	Yes	32	17	3	17	17	3

(Contd)

Table A1.2 (*Contd*)

Number of items	New table created	Size of the table	Filled positions	Amortized cost	Actual cost	Used	Left
18	No	32	18	3	1	1	5
19	No	32	19	3	1	1	7
20	No	32	20	3	1	1	9
21	No	32	21	3	1	1	11
22	No	32	22	3	1	1	13
23	No	32	23	3	1	1	15
24	No	32	24	3	1	1	17
25	No	32	25	3	1	1	19
26	No	32	26	3	1	1	21
27	No	32	27	3	1	1	23
28	No	32	28	3	1	1	25
29	No	32	29	3	1	1	27
30	No	32	30	3	1	1	29
31	No	32	31	3	1	1	31
32	No	32	32	3	1	1	33
33	Yes	64	33	3	33	33	3

It is evident from the table that the amount left never becomes negative. The reader is expected to figure out from where did the figure of 3 (on insertion) come. What if the figure would have been 4 or 2 instead of 3? On analysis, the reader will find out that had we used figure '2' on insertion, the amount left would have become negative. In the case of '4', this would not be the case but we will end up spending more with no added advantage. The difference between the amortized and the actual cost is also, therefore, important if we want to keep the things well within our budget. This essence is captured in the potential method.

As per Cormen (1990), the total amortized cost of *n* operations in the case of potential method is

$$\sum_{i=1}^{n} C_i + \varphi(D_i) - \varphi(D_{i-1})$$

The quantity $\varphi(D_i) - \varphi(D_{i-1})$ is referred to as potential difference. *D* is the data structure and φ is a function which is 0 for the initial value of the data structure and is not negative for any value of the data structure. If the amortized cost is high then the value of the potential difference is positive, otherwise it is negative. The value of φ for dynamic table is taken as $\varphi = 2i - 2^{\lceil \ln i \rceil}$. The reader is expected to analyse the relation between the potential difference and the value of *i* for Table A1.2.

A1.4 CONCLUSION

The performance of a data structure can be easily judged using amortized analysis. The analysis is of three types. The aggregate method, though easy, is less precise. The other two methods require a thorough analysis of the data structure performance before deciding on the amortized cost or the potential function. It may be stated here that one can come up with a different strategy for assigning cost to an operation or deciding the value of the potential function. The analysis helps to arrive at the bounds in complex problems. The method can also be applied to red–black trees, Fibonacci heaps, hash tables, etc. The mandate of this appendix was to accustom the reader with the three methods. The reader is expected to apply amortized analysis in the above cases as well.

Points to Remember

- For a particular data structure, different operations have different costs.
- The amortized analysis considers the average cost of a sequence of n worst case operations.
- The analysis does not depend on probability or randomization.
- The amortized analysis is of three types: aggregate, accounting, and potential.
- In aggregate analysis method, the different operations are not differentiated on the basis of cost.
- The aggregate method is the easiest amongst the three methods.
- The accounting and potential methods yield better results as compared to the aggregate method.

KEY TERM

Amortized analysis In an amortized analysis, the time essential to carry out a chain of data structure operations is averaged over all the operations carried out.

EXERCISES

I. Multiple Choice Questions

1. Which of the following is the basis of amortized analysis?
   - (a) Probability
   - (b) Randomization
   - (c) Both of the above
   - (d) None of the above
2. Which of the following are not the types of amortized analysis?
   - (a) Aggregate
   - (b) Potential
   - (c) Accounting
   - (d) Integration

3. Which of the following methods is least suitable for the analysis of dynamic tables?
   (a) Aggregate   (c) Potential
   (b) Accounting   (d) All of the above are equally good
4. Which of the following correctly represents the potential difference with reference to the potential method of the amortized analysis?
   (a) $\varphi(D_i) - \varphi(D_{i-1})$ (c) $\varphi(D_{i-1})$
   (b) $\varphi(D_i)$ (d) All of the above
5. Which of the following is the maximum ratio of the table size to the number of elements in the case of dynamic tables which doubles its size when an overflow occurs?
   (a) 2   (c) 4
   (b) 3   (d) None of the above
6. In the case of potential method or amortized analysis, the value of φ cannot be
   (a) Negative   (c) Zero
   (b) Positive   (d) None of the above
7. In which of the following problems should amortized analysis be applied?
   (a) Insertion in an array   (c) Both
   (b) Insertion in a dynamic table   (d) None of the above
8. For which of the following, amortized analysis must be carried out?
   (a) Single pop from a stack   (c) Both of the above
   (b) Single push in a stack   (d) None of the above
9. In which of the following, there is hardly any need to carry out the amortized analysis?
   (a) Red–black trees   (c) A 1-dimensional array
   (b) Splay trees   (d) None of the above
10. The average worst case analysis of a hash table can be carried out using
   (a) Amortized analysis   (c) Mortal analysis
   (b) Asymptotic analysis   (d) None of the above

II. Review Questions

1. What is amortized analysis?
2. What are the types of amortized analysis?
3. Explain aggregate method of amortized analysis by taking an example of `multi_pop` from a stack.
4. Explain accounting method of amortized analysis by taking an example of dynamic tables.
5. Explain potential method of amortized analysis by taking an example of dynamic tables.

III. Application-based Questions

1. In the dynamic table had the size been tripled instead of being doubled, what would have been the ratio of table size and the number of elements?

2. In the above question suggest a model for accounting analysis, for example, what should be credited in the case of insertion. Analyse your model by carrying out an empirical analysis for $n = 1$ to 129.

3. Suggest a formula for the potential function in the above case.

4. Consider a binary heap, having two operations associated with it: find_min(), which finds the minimum element of the heap and insert, which inserts an element in the heap. Design a potential function which makes the average cost of insert as $O(\lg n)$ and that of find_min() as $O(1)$.

5. Design a queue using two stacks and design a potential function for the so formed data structure, which makes the average complexity of insert and delete as $O(1)$.

Answers to MCQs

1. (d)	3. (a)	5. (a)	7. (b)	9. (c)
2. (d)	4. (a)	6. (a)	8. (d)	10. (a)

APPENDIX A2

2-3-4 and Red–Black Trees

OBJECTIVES

After studying this appendix, the reader will be able to

- Define the concept of 2-3-4 and red–black trees
- Learn the creation of a 2-3-4 tree
- Explain the insertion in a 2-3-4 tree
- Understand the algorithm for searching an element in a 2-3-4 tree
- Define the concept of a red–black tree
- Insertion and searching in a red–black tree

A2.1 INTRODUCTION

One of the major problems in binary search trees is that, at times, they become skewed. The problem can be handled by variants of binary search trees. This appendix discusses two such variants of binary search trees: 2-3-4 trees (or 2-4 trees) and red–black trees. The topics have not been included in the main chapters as they generally form a part of an advanced course in algorithms. The concept 'insertion and searching' has been included in this appendix. The reader is expected to go through the concept of binary search trees before beginning to go through the following discussion.

The appendix has been divided into the following sections. The second section discusses the concept of insertion and searching in a 2-3-4 tree. The third section discusses the concept of insertion in a red–black tree. The last section concludes.

A2.2 2-3-4 TREE

A 2-3-4 tree is a tree wherein all the leaves are at the same level and there is

A single value in a node and the node has two children

or

Two values in a node and the node has three children

or

Three values in a node and the node has four children.

A node in a 2-3-4 tree can either be a 2-node, a 3-node, or a 4-node tree depending upon the number of children it has (Fig. A2.1).
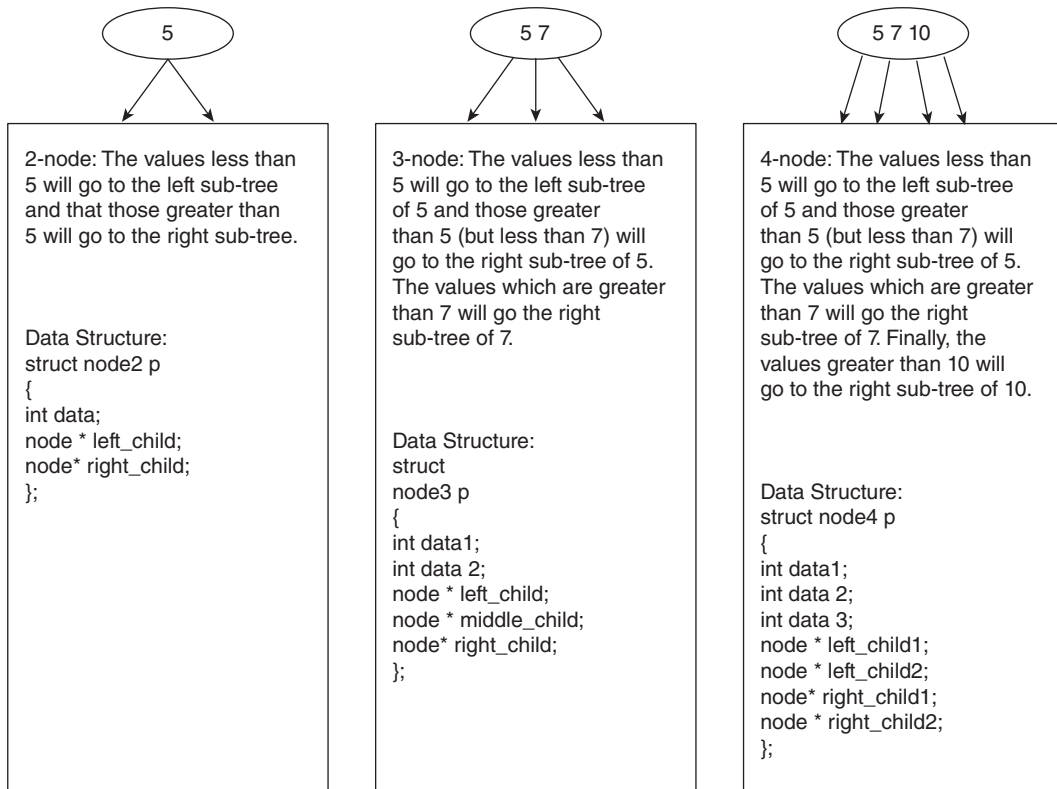
Figure A2.1 Legal nodes in a 2-3-4 tree

The advantage of having all the nodes at the same level is that while searching a node in this tree the upper bound would always be $O(\log n)$. As a matter of fact a 4-node tree requires more pointers as compared to a 2-node. So a tree having more than 2 nodes would require lesser memory as that having 4 nodes. In order to insert an element in a 2-3-4 tree, the following steps need to be pursued (Algorithm A2.1).

Algorithm A2.1   INSERT 2-3-4(int key)

//Inserts the 'key' at its appropriate position with due consideration to the properties of the
//trees stated earlier
Input: 'key', the value to be inserted and T, the tree
{

Step 1. Find the position of the key (value to be inserted);
Step 2. Place the key at its appropriate position;
Step 3. If the number of values in that node becomes greater than three, then the node is split in two parts the median element is promoted to the parent and the rest of the elements are put into the left and the right nodes (two nodes created);

}

Illustration A2.1 Create a 2-3-4 tree out of the following values:

20, 11, 26, 7, 90, 67

Solution Initially, the tree is empty.

Step 1 Key = 20 (Fig. A2.2)

20
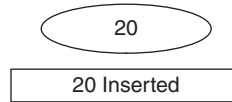
20 Inserted

Figure A2.2 20 inserted in the 2-3-4 tree.

Step 2 Key = 11 (Fig. A2.3)

11, 20

Since a node can have 3 values, 11 would be inserted in the same node
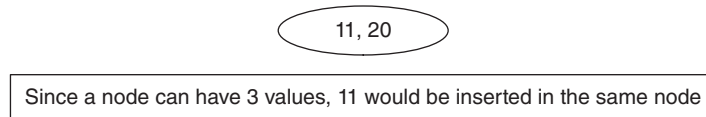
Figure A2.3 Insertion of '11'

Step 3 Key = 26 (Fig. A2.4)

11, 20. 26

Since a node can have 3 values, 26 would also be inserted in the same node
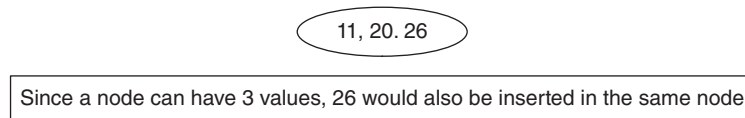
Figure A2.4 26 inserted in the 2-3-4 tree.

Step 4 Key = 7 (Fig. A2.5)

Take the second value to the parent and split the node

7, **11**, 20, 26

11

7   20, 26

Now the number of values in the node becomes 4, so the node needs to be split.
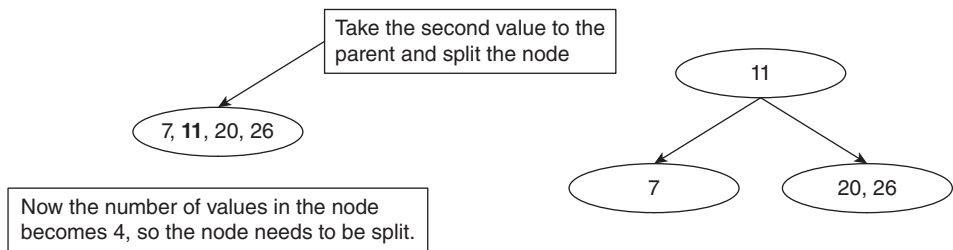
Figure A2.5 Insertion of '7'.

Step 5 Key = 90 (Fig. A2.6)

11

90 will be in the right of 11.

7   20, 26, 90
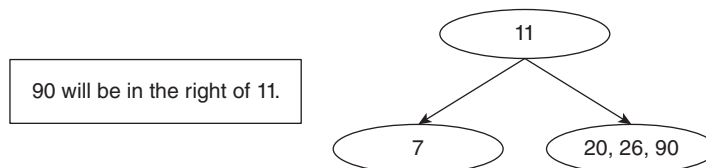
Figure A2.6 Insertion of '90'.

Step 6 Key = 67 (Fig. A2.7)

The right node of the root would split into two
parts. The value '26' would go to the root
and '67' will come along with '90'
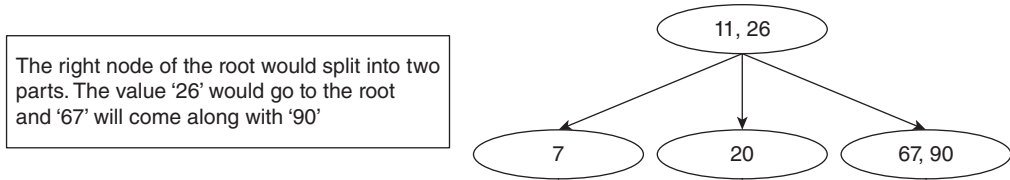
Figure A2.7 Inserting '67'.

Searching in a 2-3-4 tree

A 2-3-4 tree is essentially a binary search tree, so searching for a key in this tree
would be same as in a binary search tree (Fig. A2.8). Algorithm A2.2 shows the search-
ing 'key' in a 2-3-4 tree, T.

2-node: The left tree of the
root would henceforth be
referred to as 'left sub-tree'
and the right tree of the root
would henceforth be referred
to as the 'right sub-tree'.

3-node: The left tree of the
root would henceforth be
referred to as 'left sub-tree'
and the right tree of the
root would henceforth be
referred to as the 'right
sub-tree', the sub-tree initiating
from the middle pointer would
be referred to as the 'middle
sub-tree'

4-node:

Data Structure:

struct node4 p
{
int data1;
int data 2;
int data 3;
node * left_child1;
node * left_child2;
node* right_child1;
node * right_child2;
};

The sub-tree along with
left_child1 would be
referred to as 'left sub-tree1'.

The sub-tree along with
left_child2 would be
referred to as 'left sub-tree2'.

The sub-tree along with
right_child2 would be
referred to as 'right sub-tree1'.

The sub-tree along with
right_child2 would be
referred to as 'right sub-tree2'.

Figure A2.8 Terminology of 2-3-4 node

Algorithm A2.2   Search (T, Key)

Input: Tree, T and the value to be inserted (key)

```
{
Ptr= root;
If root is a 2-node
{
    if (Key < root-> data)
        {
        Search (Left sub-tree of root, Key);
        }
    else if(Key > root->data)
        {
        Search (Right sub-tree of root, Key);
        }
}
else if root is a 3-node
{
    if (Key < root-> data1)
        {
        Search (Left sub-tree of root, Key);
        }
    else if((Key > (root->data1))&&(key (<root->data2))
        {
        Search (Middle sub-tree of root, Key);
        }
    if (Key < root-> data2)
        {
        Search (Right sub-tree of root, Key);
        }
}
else if root is a 4-node
{
    if (Key < root-> data1)
        {
        Search (Left1 sub-tree of root, Key);
        }
    else if((Key > (root->data1))&&(key (<root->data2))
        {
        Search (Left2 sub-tree of root, Key);
        }
    else if((Key > (root->data2)&&(key <root->data3))
        {
        Search (Right1 sub-tree of root, Key);
        }
    If (Key > root-> data3)
```

```
        {
        Search (Right2 sub-tree of root, Key);
        }
    }
}
```

Having discussed 2-3-4 trees, let us now move to the red–black trees. As a matter of fact there is a one-to-one correspondence between the 2-3-4 trees and the red–black tree. A 2-3-4 tree can be converted into a red–black tree and vice versa.

A2.3 RED–BLACK TREES

Making searc hing effective and efficient has always been and will remain one of the supreme goals of the computing fraternity. Binary search trees are an effective method of accomplishing the task. Binary search trees (BSTs) are the binary trees, in which the value of the data stored at the left child of a node is less than that of the node and that stored on the right child of the node is greater than the node. The red–black trees are a type of BSTs which follow the following properties.

The nodes of a red–black tree are either red (denoted by ◯ in the following discussion) or black (denoted by ◯ in the following discussion).

- The root of the tree is always black.
- A black node can have a black or a red child.
- A red node cannot have a red child. It can only have a black child.
- The black depth of a terminal node is the number of black nodes encountered while travelling from the terminal node to the root.
- The black depth of a terminal node is always same.

> **Definition Black Depth** The number of black nodes from the terminal to the root is called the black depth of the node.

The leaves of a red–black tree would always be a NULL node (denoted by ◌ in the following discussion). Each black or red node (last in the hierarchy) will have NULL nodes as children.

Based on the above discussion, let us try to segregate the non-red–black trees from the red–black trees.

Figure A2.9 is a red–black tree as

- The root of this tree is black
- The red nodes have black children
- The black depth of each terminal node is same.
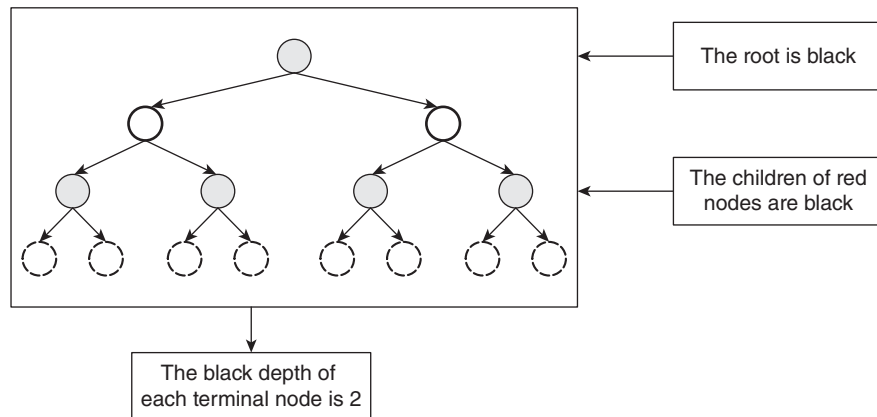
Figure A2.9 An example of a red–black tree

Figure A2.10 is not a red–black tree as
- The root of this tree is red
- A red node cannot have a red child, but in Fig. A2.10, this is not the case
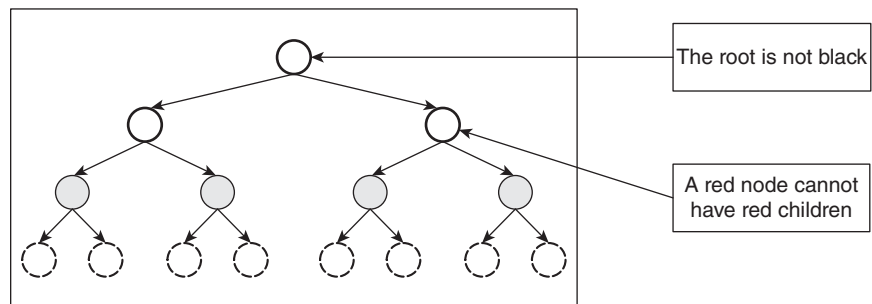
Figure A2.10 An example of a tree which is not red–black tree

Figure A2.11 is not a red–black tree as
- The black depth of the terminals are not same
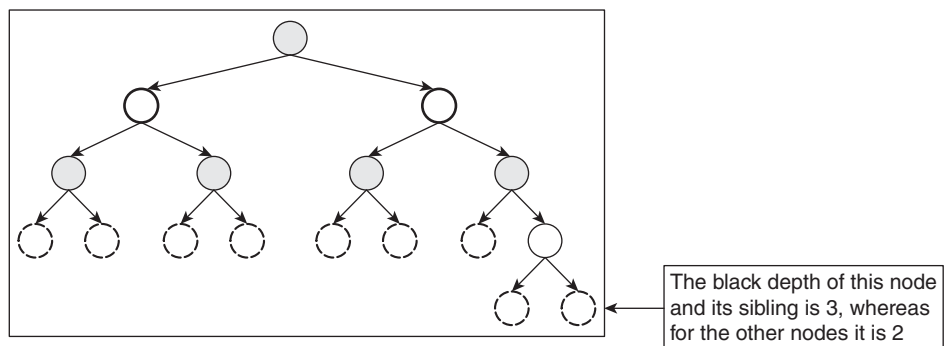
Figure A2.11 An example of a tree which is not red–black tree

Having been able to segregate trees in red–black and those which are not, let us move our discussion to the problems in red–black trees. The following discussion throws light on two such problems namely double red problem.

Double Red Problem

The case wherein the child of a red node is a red node is called the double red problem. Figure A2.12 depicts the double red problem. The problem will arise in insertion.
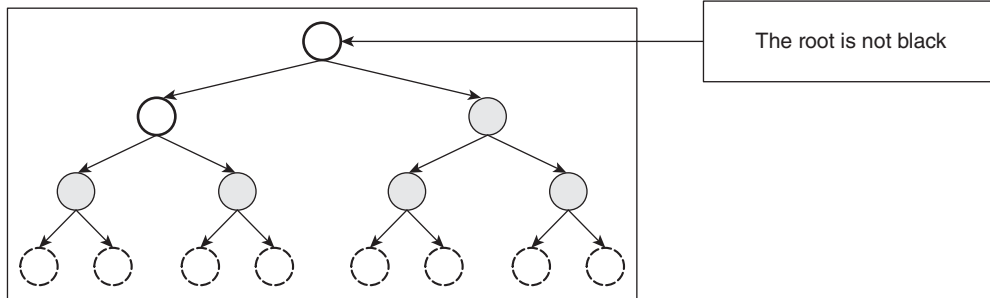
Figure A2.12   An example of a tree which is not red–black tree due to double red problem

The red–black tree has maximum height if there are red and black nodes at the alternate levels. If all the nodes of the tree are black, then it will have minimum length. The height of a red–black tree, h, satisfies the following constraint:

$$\log_4 n < h < \log_2 n$$

where n is the number of nodes.

The red–black trees are fundamentally similar to the 2-4 trees. As a matter of fact, any red–black tree can be converted to a 2-4 tree and any 2-4 tree can be converted into a red–black tree. In order to convert a red–black tree to a 2-4 tree, the following procedure may be used.

Select a black node and its red children. The node can have either two red nodes, in which case the corresponding node of the 2-4 tree will have 3 keys. Fig. A2.13(a) depicts the case.
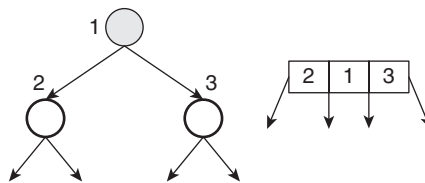
Figure A2.13(a)   Conversion of a red–black tree into a 2-4 tree I

The selected node can have a red node, in which case the corresponding node of the 2-4 tree will have 2 keys. Fig. A2.13(b) depicts the case.
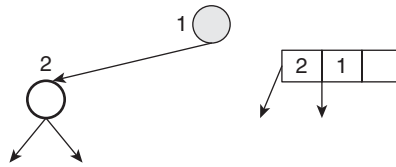
Figure A2.13(b) Conversion of a
red–black tree into a 2-4 tree II

There can also be a case wherein the selected node will not have any red child. In this case the corresponding node of the 2-4 tree will have 1 key. Fig. A2.13(c) depicts the case.

Figure A2.13(c) Conversion
of a red–black tree into
a 2-4 tree III

Fig. A2.14 shows the conversion of a complete red–black tree into a 2-4 tree.
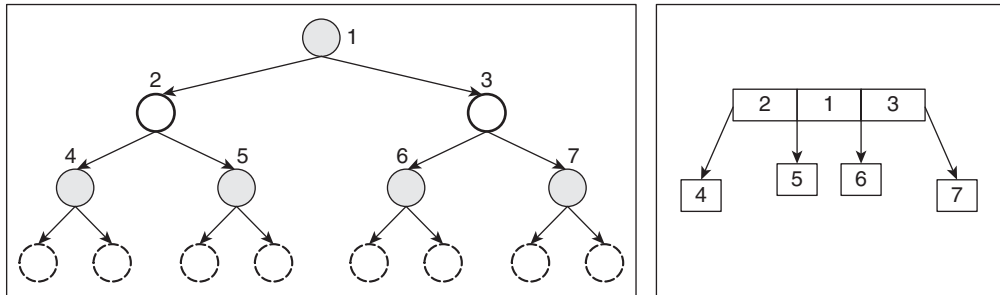
Figure A2.14 An example of conversion of red–black tree into a 2-4 tree

Insertion of a Node in a Red–Black Tree

The red–black trees, as stated earlier, are BSTs, so the first step of insertion is the same as that in the case of a BST. The value to be inserted is searched in the given tree; if it already exists, then the insertion returns an error. In the other case, the appropriate position of the value to be inserted is searched. If it is the leaf node, then the insertion is easy. However, still the colour of the node needs to be decided (it has to be either red or black).

We begin by colouring the terminal node red. However, this may lead to the double red problem, defined earlier in the discussion. The following cases explain the various possibilities that may arise:

Case 1 When the parent is black, the double red problem will not crop up nor does the black depth of the leaf node change (Fig. A2.15).
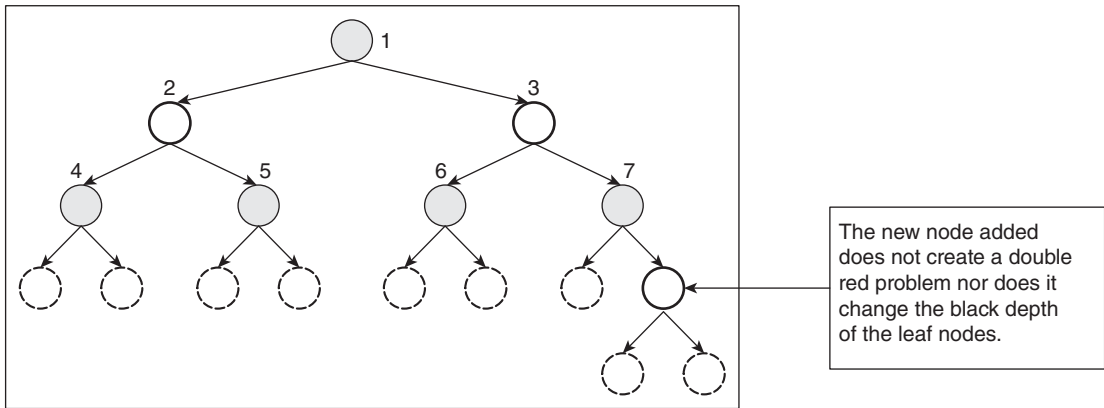
Figure A2.15   Addition of node in a red–black tree

Case 2 If the parent, however, is red, then a double red problem crops up. The following figures explains the steps to be taken in order to handle the problem, the following rotation is carried out (Fig. A2.16(a)):
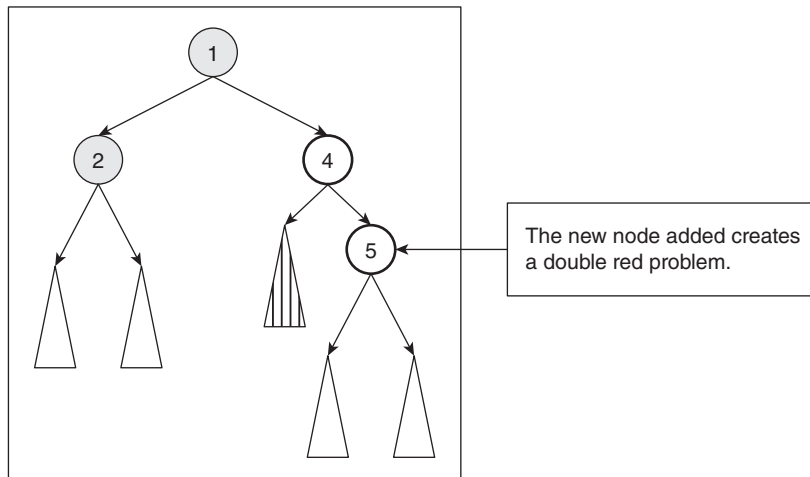
Figure A2.16(a)   Insertion of a node may lead to double red problem

Case 3 The sibling of the parent is red.

In the case where the sibling of the parent is black, the double red problem can be handled by changing the colour of the node starting from the bottom. This may lead to the shifting of the double red problem to one level up. At this level one of the two cases (the uncle of the node to be inserted is red or black) will have to be handled.
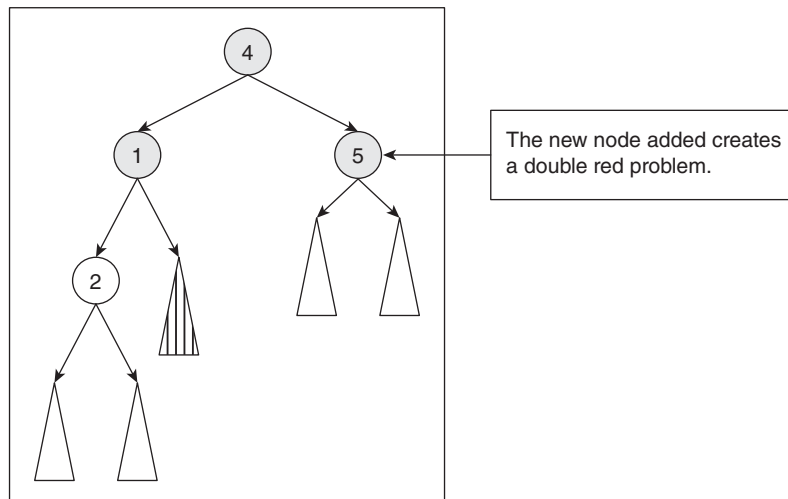
Figure A2.16(b)   Handling double red problem
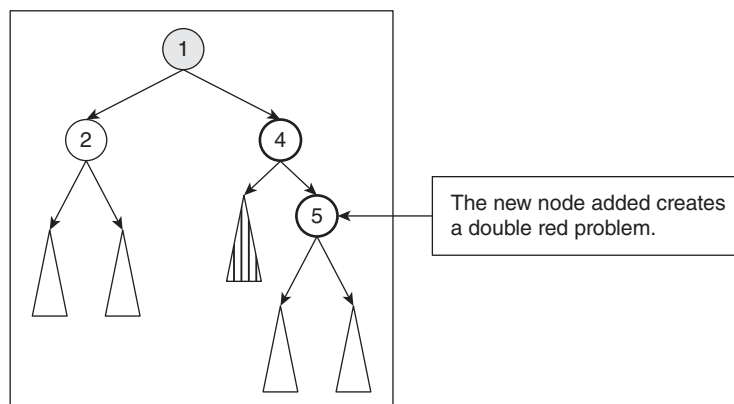
Figure A2.16(c)   Handling double red problem

A2.4 CONCLUSION

The deletion of a node from a red–black tree and a 2-3-4 tree has been included in the web resources of this book. The reader is expected to go through the material for a comprehensive coverage of the topic. The trees discussed in the appendix are variants of binary search trees but are more efficient than the BSTs. Appendix A8 contains the insertion and deletion of a node in a binary search tree. One must understand that those algorithms to be able to compare the efficiency of red–black trees with binary search trees.

Finally, the 2-3-4 trees are primarily used as theoretical foundation of red–black trees. The red–black trees in turn are used in K-mean clustering in data mining, databases, and searching words in a dictionary.

Points to Remember

- A node in a 2-3-4 tree can either be a 2-node, a 3-node, or a 4-node tree depending upon the number of children it has.
- In a red–black tree, the root of the tree is always black.
- In a red–black tree, a black node can have a black or a red child but a red node can only have a black child.
- The black depth of a terminal node is always same.

KEY TERMS

2-3-4 Tree A 2-3-4 tree is a tree wherein all the leaves are at the same level and there is
  a single value in a node and the node has two children
  or
  two values in a node and the node has three children
  or
  three values in a node and the node has four children.
Black depth The black depth of a terminal node is the number of black nodes encountered while travelling from the terminal node to the root.
Red–black tree A red–black tree is a binary search tree in which the nodes are coloured as red or black. The root of the tree is always black and the black depth of all the terminal nodes is same.

EXERCISES

I. Multiple Choice Questions

1. The root of a red–black tree can be
   - (a) Red
   - (b) Black
   - (c) Both
   - (d) None of the above

2. The black depth of leaf nodes in a red–black tree
   - (a) Is same for all leaves
   - (b) May differ by one for the leaf nodes
   - (c) May be different for different nodes
   - (d) None of the above

3. Which of the following never crops up while deleting a node from a red–black tree?
   - (a) Double red problem
   - (b) Double black problem
   - (c) Difference in the black depth of the leaves
   - (d) None of the above

4. A red–black tree can be converted into which of the following?
   - (a) 2-4 trees
   - (b) Binary search trees
   - (c) Both
   - (d) None of the above

5. A red–black tree is a variant of which of the following?
   (a) Plex
   (b) Binary search tree
   (c) Both
   (d) None of the above
6. In a 2-3-4 tree which of the nodes are legal?
   (a) 2-node
   (b) 3-node
   (c) 4-node
   (d) All of the above
7. In a 2-3-4 tree which of the following is true for a 3-node?
   (a) It has three children
   (b) It can have two data items
   (c) Both
   (d) None of the above
8. The depth of leaf nodes in a 2-3-4 tree
   (a) Are same for all leaves
   (b) May differ by 1
   (c) May be different for different nodes
   (d) None of the above
9. Which of the following is more efficient?
   (a) A 2-3-4 tree with more 4-nodes
   (b) A 2-3-4 tree with more 2-nodes
   (c) Both are equally good
   (d) None of the above
10. Which of the following is not a type of binary search tree?
    (a) 2-3-4 tree
    (b) Red–black tree
    (c) Heap
    (d) None of the above

II. Review Questions

1. What is a 2-3-4 tree?
2. Write an algorithm to insert a value in a 2-3-4 tree.
3. Write an algorithm to search a value in a 2-3-4 tree.
4. What is the complexity of searching an element in a 2-3-4 tree?
5. Which of the two is more efficient: a tree having more 4-nodes or a tree having more 2-nodes?
6. What is a red–black tree?
7. How do you insert an element in a red–black tree?
8. How do you search an element in a red–black tree? Write the algorithm.
9. Explain the double red problem. When does it arise? How is it handled?
10. Explain so as to how to convert a red–black tree into a 2-4 tree.
11. Can all the 2-4 trees be converted into red–black trees?

III. Application-based Questions

1. Create a 2-3-4 tree from the following values:
   (a) 21, 20, 34, 67, 6, 5, 9, 10
   (b) 1, 2, 3, 4, 5, 6, 7, 8
   (c) 8, 7, 6, 5, 4, 3, 2, 1
   (d) 100, 1000, 10,000, 100,000, 1, 10
2. In the above question find the number of pointers required in the tree formed.
3. If a pointer takes two bytes of memory and an integer also takes two bytes of memory, what would be the memory requirement of the trees formed in question number 1?
4. Create binary search trees out of the values given in question number 1.

5. What would be the memory requirement in each case, if the above binary search trees are stored in arrays?
6. Explain the linked list representation of the binary search trees formed above.
7. What is the number of pointers in each tree; compare the number with that obtained in question number 3.
8. On the basis of the above question, can you state which of the two, binary search trees or 2-3-4 trees, are more efficient.
9. Create red–black trees from the data given in question number 1.
10. From the first tree (formed in the above question), perform the following operations.

(a) Insert 27	(e) Delete 20	(i) Now insert 5 (It
(b) Insert 2	(f) Delete 5	was deleted in (f))
(c) Insert 100	(g) Delete 10	(j) Insert 10
(d) Insert 69	(h) Delete 34	

Answers to MCQs

1. (b)	3. (a)	5. (b)	7. (c)	9. (b)
2. (a)	4. (a)	6. (d)	8. (a)	10. (c)

APPENDIX A3

Matrix Operations

OBJECTIVES

After studying this appendix, the reader will be able to
- Explain the concept and applications of matrices
- Define the types of matrices
- Understand the operations on matrices
- Solve linear system of equations using Cramer's rule and inverse of matrices

A3.1 BASICS

A matrix is a two-dimensional array of elements. An element of a matrix is represented by two subscripts. The first subscript depicts the row and the second depicts the column number. The number of rows and the number of columns decide what is referred to as, the order of a matrix. For example, the following matrix has three rows and four columns:

$$\begin{bmatrix} 2 & 23 & 1 & 4 \\ 9 & 5 & 3 & 12 \\ 7 & 6 & 21 & 2 \end{bmatrix}$$

The elements of a matrix of order 3×4 are depicted as follows. The element a_{ij}, means ith row and jth column.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \end{bmatrix}$$

For example, a_{32} means the element in 3rd row and the 2nd column.

Row matrix A matrix that has just one row is referred to as a row matrix. The order of a row matrix is $1 \times n$. The following matrix A is a row matrix. The order of A is 1×3.

$$A = \begin{bmatrix} 2 & 3 & 5 \end{bmatrix}$$

Column matrix A matrix that has just one column is referred to as a column matrix. The order of a column matrix is $n \times 1$. The following matrix B is a column matrix. The order of B is 3×1.

$$B = \begin{bmatrix} 3 \\ 5 \\ 7 \end{bmatrix}$$

Diagonal matrix A diagonal matrix is one that has elements only at the main diagonal. That is, $a_{ij} = 0, i \neq j$. The following matrix C of order 3×3 is a diagonal matrix.

$$C = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 8 & 0 \\ 0 & 0 & 9 \end{bmatrix}$$

The above matrix can also be represented as $\{2, 8, 9\}$.

Scalar matrix A scalar matrix is a diagonal matrix in which all the elements at the main diagonal are same. That is, $a_{ij} = \begin{cases} 0, & i \neq j \\ k, & i = j \end{cases}$. The following matrix D depicts a scalar matrix, wherein the value of k is 2.

$$D = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{bmatrix}$$

Identity matrix An identity matrix is a scalar matrix in which the value of k is 1. That is, $a_{ij} = \begin{cases} 0, & i \neq j \\ 1, & i = j \end{cases}$. The following matrix E depicts a scalar matrix, wherein the value of k is 1.

$$E = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The matrices F and G represent identity matrices of order 2 and 4.

$$F = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$G = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Upper triangular matrix An upper triangular matrix is one in which the elements below the main diagonal are 0. That is, $a_{ij} \neq 0$ if $i \leq j$. The following matrix H is an upper triangular matrix.

$$H = \begin{bmatrix} 2 & 4 & 5 \\ 0 & 8 & 5 \\ 0 & 0 & 9 \end{bmatrix}$$

Lower triangular matrix A lower triangular matrix is the one in which the elements above the main diagonal are 0. That is, $a_{ij} \neq 0$ if $i \geq j$.

The following matrix I is a lower triangular matrix.

$$I = \begin{bmatrix} 2 & 0 & 0 \\ 3 & 8 & 0 \\ 7 & 9 & 9 \end{bmatrix}$$

A3.2 OPERATIONS ON MATRICES

The following discussion explores some of the basic operations on matrices. This would help the reader to implement transformations in graphics and manipulate signals. The following topics intend to introduce the concepts and all are not inclusive.

A3.2.1 Equality of Matrices

Two matrices are equal if their order is same and the corresponding elements are equal.

For example, If $A = \begin{bmatrix} 2 & 3 \\ 3 & 5 \end{bmatrix}$ and $A = \begin{bmatrix} x & 3 \\ z & x+y \end{bmatrix}$, then $A = B$, if $x = 2$, $x + y = 5$, and $z = 3$.

This implies that $y = 3$. The algorithm for checking the equality of matrices is as follows (Algorithm A3.1).

Algorithm A3.1   Bool Equal (A, B)

Input: Two matrices A and B having order n × m
Output: The algorithm returns a True if the matrices are equal, otherwise it returns a False

```
{
//flag is initially zero. If any corresponding element is not equal, then it
//becomes 1.
flag=0;
for (i=0; i<n; i++)
    {
    for (j=0; j<m; j++)
    {
    if(A[i,j] != B [i, j])
        {
        flag=1;
        }
```

```
        }
    }
    if(flag==1)
        {
        return False;
        }
    return True;
    }
```

Complexity: Owing to the nesting of loops, the complexity of the above algorithm is $O(mn)$.

A3.2.2 Addition of Matrices

Two matrices can be added only if they have the same order. The addition of two matrices A and B gives C, where the (i, j)th element of A when added to the (i, j)th element of B, gives the (i, j)th element of C. That is,

$$c_{ij} = a_{ij} + b_{ij} \quad \forall i, j$$

For example, if $A = \begin{bmatrix} 2 & 3 \\ 3 & 5 \end{bmatrix}$ and $B = \begin{bmatrix} 1 & 3 \\ 3 & 2 \end{bmatrix}$, then $A + B = \begin{bmatrix} 3 & 6 \\ 6 & 7 \end{bmatrix}$.

The algorithm of addition of two 2-dimensional matrices has been given in Section 5.3.4 (Chapter 5). The complexity of addition is $O(mn)$, if the matrix has order $n \times m$.

A3.2.3 Subtraction of Matrices

Two matrices can be subtracted only if they have same order. The subtraction of two matrices A and B gives C, where the (i, j)th element of A when subtracted to the (i, j)th element of B gives the (i, j)th element of C. That is,

$$c_{ij} = a_{ij} - b_{ij} \quad \forall i, j$$

For example, if $A = \begin{bmatrix} 2 & 3 \\ 3 & 4 \end{bmatrix}$ and $B = \begin{bmatrix} 1 & 3 \\ 3 & 2 \end{bmatrix}$, then $A - B = \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix}$

The algorithm of subtraction of two 2-dimensional matrices is similar to that of addition which is given in Section A3.2.2. The complexity of subtraction is $O(mn)$, if the matrix has order $n \times m$.

A3.2.4 Scalar Multiplication

The scalar multiplication means multiplying each element of a matrix with a scalar, say k. The scalar multiplication of a matrix A with k, gives C, where the (i, j)th element of A when multiplied with k, gives the (i, j)th element of C. That is,

$$c_{ij} = k \times a_{ij} \quad \forall i, j$$

For example, if $A = \begin{bmatrix} 2 & 3 \\ 3 & 5 \end{bmatrix}$ and $k = 3$, then $k \times A = \begin{bmatrix} 6 & 9 \\ 9 & 15 \end{bmatrix}$

The algorithm of multiplication of a matrix *A* with a scalar *k* is as follows. The complexity of the algorithm is $O(mn)$, if the matrix has order $n \times m$ (Algorithm A3.2).

Algorithm A3.2   Scalar_Multiplication (A, k) returns C

Input: A matrix A and a scalar K
Output: The algorithm returns a matrix C

```
{
for (i=0; i<n; i++)
    {
    for (j=0; j<m; j++)
    {
    C [i, j] = k × A[i, j] }
    }
returnC;
}
```

Complexity: Owing to the nesting of loops, the complexity of the above algorithm is $O(mn)$.

A3.2.5 Transpose of a Matrix

The transpose of a matrix A is obtained by converting the rows of the matrix into columns or columns into rows. For example, the transpose of the matrix

$$A = \begin{bmatrix} 2 & 23 & 1 & 4 \\ 9 & 5 & 3 & 12 \\ 7 & 6 & 21 & 2 \end{bmatrix}$$

is given by

$$A^T = \begin{bmatrix} 2 & 9 & 7 \\ 23 & 5 & 6 \\ 1 & 3 & 21 \\ 4 & 12 & 2 \end{bmatrix}$$

A3.2.6 Symmetric Matrix

A matrix is deemed as symmetric, if it is equal to its transpose. That is,

$$A = A^T$$

For example, the matrix $A = \begin{bmatrix} 1 & 3 & 5 \\ 3 & 7 & 9 \\ 5 & 9 & 11 \end{bmatrix}$ is a symmetric matrix if it is equal to its transpose.

> **Tip:** A matrix of order $n \times m$ is symmetric, if $m = n$ & $a_{ij} = a_{ji}$

The algorithm to determine whether a matrix A is symmetric or not is as follows (Algorithm A3.3). The complexity of the algorithm is $O(n^2)$, if the matrix has order $n \times n$.

Algorithm A3.3   Boolean is symmetric (A, n)

Input: A matrix A and the order of the matrix n.
Output: The algorithm returns true if the given matrix is symmetric otherwise it returns a false

```
flag=0;
for (i=0; i<n; i++)
    {
    for (j=0; j<n-i; j++)
        {
        if (a[i, j]! = a[j, i]
            return false
        }
    }
return true;// note that the control reaches here only if all a[i, j]'s are
//same as a[j, i]'s.
}
```

Complexity: Owing to the nesting of loops, the complexity of the above algorithm is $O(n^2)$.

A3.2.7 Skew-symmetric Matrix

A matrix is deemed as skew-symmetric, if it is equal to negative of its transpose. That is,

$$A = -1 \times A^T$$

For example, the matrix $A = \begin{bmatrix} 0 & 3 & 5 \\ -3 & 0 & 9 \\ -5 & -9 & 0 \end{bmatrix}$ is a skew-symmetric matrix as it is equal to the negative of its transpose.

> **Tip:** If any of the elements at the diagonal of a matrix is non-zero, then it cannot be skew-symmetric.

The algorithm to determine whether a matrix A is skew symmetric or not is as follows (Algorithm A3.4). The complexity of the algorithm is $O(n^2)$, if the matrix has order $n \times n$.

Algorithm A3.4   Boolean is skew-symmetric (A, n)

Input: A matrix A and the order of the matrix n
Output: The algorithm returns true if the given matrix is symmetric otherwise it returns a false

```
flag=0;
for (i=0; i<n; i++)
    {
    for (j=0; j<n-i; j++)
        {
        if (a[i, j]! = -1 × a[j, i]
            return false
        }
    }
return true;// note that the control reaches here only if all a[i, j]'s are same
as the negatives of a[j, i]'s.
}
```

Complexity: Owing to the nesting of loops, the complexity of the above algorithm is $O(n^2)$.

A3.2.8 Multiplication of Matrices

Two matrices can be multiplied only if the number of rows of the first matrix is equal to the number of columns of the second. That is, if the matrix A has order $n \times m$ and matrix B has order $q \times p$, then they can be multiplied only if $q = m$.

The multiplication is carried out in the following way. The order of the resultant matrix on multiplying two matrices of order $n \times m$ and $m \times p$ would be $n \times p$. In order to find the (i, j)th element of the product matrix, the elements of the ith row of the first matrix are multiplied with the corresponding elements of the jth column of the second matrix and the products are summed. The algorithm for the multiplication of matrices is given in Chapter 10. The complexity of multiplication is $O(n^3)$, if the two matrices are of order $n \times n$. The process is depicted in Fig. A3.1.

Multiply the elements of the cells having same colour and add the results.
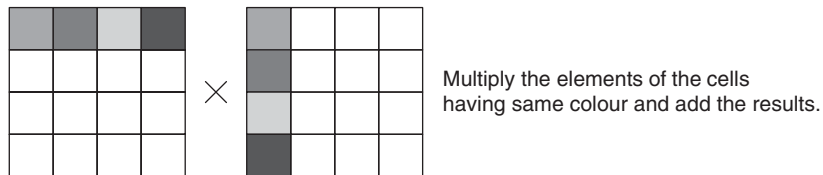
Figure A3.1   Multiplication of matrices

A more efficient algorithm for matrix multiplication has been discussed in Section 9.8 (Chapter 9).

A3.2.9 Determinant of a Matrix

The determinant of a 2×2 matrix $\begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{vmatrix} a & b \\ c & d \end{vmatrix} = a \times d - b \times c$. This helps in find-

ing the determinant of higher order matrices. The determinant of a 3×3 matrix can be found by that of order 2 which is stated as follows:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = a_{11} \begin{vmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{vmatrix} - a_{12} \begin{vmatrix} a_{21} & a_{23} \\ a_{31} & a_{33} \end{vmatrix} + a_{13} \begin{vmatrix} a_{21} & a_{22} \\ a_{31} & a_{32} \end{vmatrix}$$

$$= a_{11}(a_{22} \times a_{33} - a_{23} \times a_{32}) - a_{12}(a_{21} \times a_{33} - a_{23} \times a_{31}) + a_{13}(a_{21} \times a_{32} - a_{22} \times a_{31})$$

For example, $\begin{vmatrix} 4 & 1 \\ 6 & 2 \end{vmatrix} = 4 \times 2 - 6 \times 1 = 2$

$$\begin{bmatrix} 2 & 3 & 1 \\ 6 & 7 & 5 \\ 4 & 8 & 9 \end{bmatrix} = 2 \begin{vmatrix} 7 & 5 \\ 8 & 9 \end{vmatrix} - 3 \begin{vmatrix} 6 & 5 \\ 4 & 9 \end{vmatrix} + 1 \begin{vmatrix} 6 & 7 \\ 4 & 8 \end{vmatrix} = 2 \times (63 - 40) - 3 \times (54 - 20) + 1 \times (48 - 28) = -36$$

$$a_{11}(a_{22} \times a_{31} - a_{23} \times a_{31}) - a_{12}(a_{21} \times a_{31} - a_{23} \times a_{31}) + a_{13}(a_{21} \times a_{31} - a_{22} \times a_{31})$$

A3.2.10 Minor and Cofactor of an Element

The minor of an element $a[i, j]$ can be obtained by finding the determinant of the matrix which can be obtained by hiding the ith row and the jth column of the matrix (Fig. A3.2).

The minor of the element $a[1, 2]$ can be obtained by finding the determinant of the matrix obtained by hiding 1st row and the 2nd column.
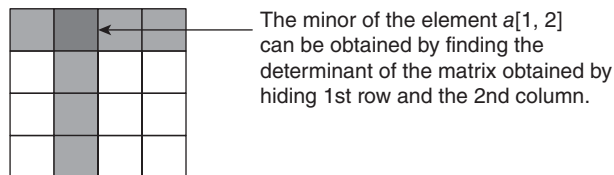
Figure A3.2 Finding minor

For example, the minor of $a[1, 1]$ can be found as follows.

$$\text{minor of } a[1,1] \text{ in} \begin{bmatrix} 2 & 3 & 1 \\ 6 & 7 & 5 \\ 4 & 8 & 9 \end{bmatrix} = M_{1,1} = \begin{vmatrix} 7 & 5 \\ 8 & 9 \end{vmatrix} = 23$$

The cofactor of an element can be calculated from its minor using the following formula:

$$C_{i,j} = (-1)^{i+j} M_{i,j}$$

A3.2.11 Inverse of a Matrix

The inverse of a matrix, A is B such that $A \times B = B \times A = I$, where I is an identity matrix of order n. The order of A is $n \times n$. The inverse of A is generally denoted by A^{-1}.

The inverse of a matrix can be found as follows:
- For each element $a[i, j]$, find the cofactor.
- Replace each element with its cofactor.
- Find the transpose of the matrix obtained in the previous step.

For example, the cofactor of $a[1, 1]$ is

$$\text{cofactor of } a[1,1] \text{ in} \begin{bmatrix} 2 & 3 & 1 \\ 6 & 7 & 5 \\ 4 & 8 & 9 \end{bmatrix} = \begin{vmatrix} 7 & 5 \\ 8 & 9 \end{vmatrix} = 23$$

$$\text{cofactor of } a[1,2] \text{ in} \begin{bmatrix} 2 & 3 & 1 \\ 6 & 7 & 5 \\ 4 & 8 & 9 \end{bmatrix} = \begin{vmatrix} 6 & 5 \\ 4 & 9 \end{vmatrix} = 34 \times -1 = -34$$

$$\text{cofactor of } a[1,3] \text{ in} \begin{bmatrix} 2 & 3 & 1 \\ 6 & 7 & 5 \\ 4 & 8 & 9 \end{bmatrix} = \begin{vmatrix} 6 & 7 \\ 4 & 8 \end{vmatrix} = 20$$

$$\text{cofactor of } a[2,1] \text{ in} \begin{bmatrix} 2 & 3 & 1 \\ 6 & 7 & 5 \\ 4 & 8 & 9 \end{bmatrix} = \begin{vmatrix} 3 & 1 \\ 8 & 9 \end{vmatrix} = 19 \times -1 = -19$$

$$\text{cofactor of } a[2,2] \text{ in} \begin{bmatrix} 2 & 3 & 1 \\ 6 & 7 & 5 \\ 4 & 8 & 9 \end{bmatrix} = \begin{vmatrix} 2 & 1 \\ 4 & 9 \end{vmatrix} = 14$$

$$\text{cofactor of } a[2,3] \text{ in} \begin{bmatrix} 2 & 3 & 1 \\ 6 & 7 & 5 \\ 4 & 8 & 9 \end{bmatrix} = \begin{vmatrix} 2 & 3 \\ 4 & 8 \end{vmatrix} = 4 \times -1 = -4$$

$$\text{cofactor of } a[3,1] \text{ in} \begin{bmatrix} 2 & 3 & 1 \\ 6 & 7 & 5 \\ 4 & 8 & 9 \end{bmatrix} = \begin{vmatrix} 3 & 1 \\ 7 & 5 \end{vmatrix} = 8$$

$$\text{cofactor of } a[3,2] \text{ in} \begin{bmatrix} 2 & 3 & 1 \\ 6 & 7 & 5 \\ 4 & 8 & 9 \end{bmatrix} = \begin{vmatrix} 2 & 1 \\ 6 & 5 \end{vmatrix} = 4 \times -1 = -4$$

$$\text{cofactor of } a[3,3] \text{ in} \begin{bmatrix} 2 & 3 & 1 \\ 6 & 7 & 5 \\ 4 & 8 & 9 \end{bmatrix} = \begin{vmatrix} 2 & 3 \\ 6 & 7 \end{vmatrix} = -4$$

Replace each element of the matrix by its cofactor

$$\begin{bmatrix} 23 & -34 & 20 \\ -19 & 14 & -4 \\ 8 & -4 & -4 \end{bmatrix}$$

Now, take the transpose of the matrix obtained in the previous step:

$$\begin{bmatrix} 23 & -19 & 8 \\ -34 & 14 & -4 \\ 20 & -4 & -4 \end{bmatrix}$$

A3.3 SOLVING SYSTEM OF LINEAR EQUATIONS: CRAMER'S RULE

For a system of simultaneous linear equations

$$a_1 x + b_1 y = c_1$$
$$a_2 x + b_2 y = c_2$$

The value of the variables can be found by using Cramer's rule. In order to find the equations, the following determinants are required to be evaluated:

$$D = \begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix}$$

$$D_1 = \begin{vmatrix} c_1 & b_1 \\ c_2 & b_2 \end{vmatrix}$$

$$D_2 = \begin{vmatrix} a_1 & c_1 \\ a_2 & c_2 \end{vmatrix}$$

The procedure of solution is as follows (Algorithm A3.4).

Algorithm A3.4   Cramer's Rule

```
//For two equations in two variables
Input: System of linear equations
Output: Values of the unknown variables

{
Find D, D₁ and D₂
If (D ≠ 0)
```

```
    {
    x=D₁/D
    y=D₂/D
    }
else if (D==0)
    {
    if ((D₁==D₂) &&(D₂==D)&&(D==0))
        {
        Put y=k
        get x in terms of k
    }
}
else
    {
    Print "No solution";
    }
}
```

Illustration A3.1 Find the values of variables by using Cramer's rule.

$$2x + 3y = 5$$

$$4x - y = 3$$

Solution Using Cramer's rule, we get

$$D = \begin{vmatrix} 2 & 3 \\ 4 & -1 \end{vmatrix} = -2 - 12 = -14$$

$$D_1 = \begin{vmatrix} 5 & 3 \\ 3 & -1 \end{vmatrix} = -5 - 9 = -14$$

$$D_2 = \begin{vmatrix} 2 & 5 \\ 4 & 3 \end{vmatrix} = 6 - 20 = -14$$

So, $x = D_1/D = 1$, $y = D_2/D = 1$

Illustration A3.2 Find the values of variables by using Cramer's rule.

$$2x + 3y = 5$$

$$4x + 6y = 10$$

Solution Using Cramer's rule, we get

$$D = \begin{vmatrix} 2 & 3 \\ 4 & 6 \end{vmatrix} = 12 - 12 = 0$$

$$D_1 = \begin{vmatrix} 5 & 3 \\ 10 & 6 \end{vmatrix} = 30 - 30 = 0$$

$$D_2 = \begin{vmatrix} 2 & 5 \\ 4 & 10 \end{vmatrix} = 20 - 20 = 0$$

So Put $y = k$

$$2x + 3k = 5$$

$$x = (5 - 3k)/2$$

$((5 - 3k)/2, k)$

Illustration A3.3 Find the solution of the following equations:

$$2x + 3y = 5$$

$$2x + 3y = 7$$

Solution

$$D = \begin{vmatrix} 2 & 3 \\ 2 & 3 \end{vmatrix} = 6 - 6 = 0$$

$$D_1 = \begin{vmatrix} 5 & 3 \\ 7 & 3 \end{vmatrix} = 15 - 21 = -6$$

$$D_2 = \begin{vmatrix} 2 & 5 \\ 2 & 7 \end{vmatrix} = 14 - 10 = 4$$

So as $D = 0$ and $D_1 \ne D_2$
No solution
For system of linear equations:

$$a_1 x + b_1 y + c_1 z = d_1$$
$$a_2 x + b_2 y + c_2 z = d_2$$
$$a_3 x + b_3 y + c_3 z = d_3$$

$$D = \begin{vmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{vmatrix}, D_1 = \begin{vmatrix} d_1 & b_1 & c_1 \\ d_2 & b_2 & c_2 \\ d_3 & b_3 & c_3 \end{vmatrix}, D_2 = \begin{vmatrix} a_1 & d_1 & c_1 \\ a_2 & d_2 & c_2 \\ a_3 & d_3 & c_3 \end{vmatrix}, D_3 = \begin{vmatrix} a_1 & b_1 & d_1 \\ a_2 & b_2 & d_2 \\ a_3 & b_3 & d_3 \end{vmatrix}$$

Algorithm A3.5   Cramer's rule for three equations

//For three equations in three variables
Input: System of linear equations
Output: Values of the unknown variables

```
{
Find D, D₁, and D₂
If (D≠0)
    {
    x=D₁/D
    y=D₂/D
    z=D₃/D
    }
else if (D=0)
    {
    If(D₁=D₂=D₃=0)
        {
        Print "Many solutions";
        }
    }
else
    {
    Print "No solution";
    }
}
```

Illustration A3.4 Find the values of variables by using Cramer's rule.

$$x + y + z = 3$$
$$x + 2y + 2z = 5$$
$$x + 2y + 3z = 6$$

Solution

$$D = \begin{vmatrix} 1 & 1 & 1 \\ 1 & 2 & 2 \\ 1 & 2 & 3 \end{vmatrix} = 1 \times (6-4) - 1 \times (3-2) + 1 \times (2-2) = 1$$

$$D_1 = \begin{vmatrix} 3 & 1 & 1 \\ 5 & 2 & 2 \\ 6 & 2 & 3 \end{vmatrix} = 3 \times (6-4) - 1 \times (15-12) + 1 \times (10-12) = 1$$

$$D_2 = \begin{vmatrix} 1 & 3 & 1 \\ 1 & 5 & 2 \\ 1 & 6 & 3 \end{vmatrix} = 1 \times (15-12) - 3 \times (3-2) + 1 \times (6-5) = 1$$

$$D_3 = \begin{vmatrix} 1 & 1 & 1 \\ 1 & 2 & 5 \\ 1 & 2 & 6 \end{vmatrix} = 1 \times (12 - 10) - 1 \times (6 - 5) + 1 \times (2 - 2) = 1$$

As $D \neq 0$

$$x = D_1/D = 1$$
$$y = D_2/D = 1$$
$$z = D_3/D = 1$$

Illustration A3.5 Find the value of variables by using Cramer's rule.

$$x + y + z = 1$$
$$x + 2y + 3z = 4$$
$$x + 3y + 5z = 7$$

Solution

$$D = \begin{vmatrix} 1 & 1 & 1 \\ 1 & 2 & 3 \\ 1 & 3 & 5 \end{vmatrix} = 1 \times (10 - 9) - 1 \times (5 - 3) + 1 \times (3 - 2) = 0$$

$$D_1 = \begin{vmatrix} 1 & 1 & 1 \\ 4 & 2 & 3 \\ 7 & 3 & 5 \end{vmatrix} = 1 \times (10 - 9) - 1 \times (20 - 21) + 1 \times (12 - 14) = 0$$

$$D_2 = \begin{vmatrix} 1 & 1 & 1 \\ 1 & 4 & 3 \\ 1 & 7 & 5 \end{vmatrix} = 1 \times (20 - 21) - 1 \times (5 - 3) + 1 \times (7 - 4) = 0$$

$$D_3 = \begin{vmatrix} 1 & 1 & 1 \\ 1 & 2 & 4 \\ 1 & 3 & 7 \end{vmatrix} = 1 \times (14 - 12) - 1 \times (7 - 4) + 1 \times (3 - 2) = 0$$

As $D = D_1 = D_2 = D_3 = 0$
So given set of equations has infinite solutions.

Illustration A3.6 Find the value of variables by using Cramer's rule.

$$x + y + z = 3$$
$$x + 2y + 3z = 5$$
$$2x + 3y + 4z = 6$$

Solution

$$D = \begin{vmatrix} 1 & 1 & 1 \\ 1 & 2 & 3 \\ 2 & 3 & 4 \end{vmatrix} = 1 \times (8-9) - 1 \times (4-6) + 1 \times (3-4) = 0$$

$$D_1 = \begin{vmatrix} 3 & 1 & 1 \\ 5 & 2 & 3 \\ 6 & 3 & 4 \end{vmatrix} = 3 \times (8-9) - 1 \times (20-18) + 1 \times (15-12) = -2$$

$$D_2 = \begin{vmatrix} 1 & 3 & 1 \\ 1 & 5 & 3 \\ 2 & 6 & 4 \end{vmatrix} = 1 \times (20-18) - 3 \times (4-6) + 1 \times (6-10) = 4$$

$$D_3 = \begin{vmatrix} 1 & 1 & 3 \\ 1 & 2 & 5 \\ 2 & 3 & 6 \end{vmatrix} = 1 \times (12-15) - 1 \times (6-10) + 3 \times (3-4) = -2$$

As $D = 0$ but D_1, D_2, D_3 are different, therefore, the given system is inconsistent.

A3.4 SOLVING SYSTEM OF LINEAR EQUATIONS: INVERSE METHOD

The system of equations can also be solved by using the inverse of matrices. The coefficients of the variables in the given system of equations are written in a matrix, say A. The constants are written in another matrix, say B. This is followed by finding the inverse of A. If the inverse exists, then $A^{-1} \times B$ is found. The result of the above calculation gives the solution.

If the inverse of A cannot be found, then the following cases arise. In the first case, both adjoint of A and determinant are 0, hence the system is dependent. In this case, the value of variables can be found in terms of one of the variables.

In the second case, the determinant is 0; however, the adjoint is not zero. In this case, the system is inconsistent. The following examples will help in understanding the above technique.

Illustration A3.7 Solve the following system of equations by finding the inverse of the coefficient matrix:

$$2x + y + z = 4$$
$$x - y + 3z = 3$$
$$x + 2y + 3z = 6$$

Solution The coefficient matrix, in the above case is

$$\begin{bmatrix} 2 & 1 & 1 \\ 1 & -1 & 3 \\ 1 & 2 & 3 \end{bmatrix}$$

The adjoint of the above matrix is

$$\begin{bmatrix} -9 & -1 & 4 \\ 0 & 5 & -5 \\ 3 & -3 & -3 \end{bmatrix}$$

The determinant is −15, therefore, the inverse is

$$\frac{1}{-15} \begin{bmatrix} -9 & -1 & 4 \\ 0 & 5 & -5 \\ 3 & -3 & -3 \end{bmatrix}$$

The above equation when multiplied by B gives

$$\begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

Therefore, the values of x, y, and z are same, 1.

Illustration A3.8 Can the following system of equations be solved by using the inverse method of the co-efficient matrix?

$$x + y + z = 4$$
$$2x + 2y + 2z = 6$$
$$x - y + z = 4$$

Solution The coefficient matrix, in the above case is

$$\begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 1 & -1 & 1 \end{bmatrix}$$

The adjoint of the above matrix is

$$\begin{bmatrix} 4 & -2 & 0 \\ 0 & 0 & 0 \\ -4 & 2 & 0 \end{bmatrix}$$

The determinant is 0; therefore, the inverse cannot be found. However, in this case adj $(A \times B)$ is 0, therefore, the given system of equations does not have any solutions.

A3.5 ELEMENTARY ROW OPERATIONS

The row operations help us to solve equations, to find inverse, and so on. The concept has already been used in Chapter 15. The row operations of matrices are as follows.

Interchanging rows of a matrix is an elementary row operation. For example, the following transformation interchanges the second and the third row.

$$
\begin{bmatrix} 2 & 1 & 1 \\ 1 & -1 & 3 \\ 1 & 2 & 3 \end{bmatrix} \sim \begin{bmatrix} 2 & 1 & 1 \\ 1 & 2 & 3 \\ 1 & -1 & 3 \end{bmatrix}
$$

Multiplying the elements of a row by a non-zero scalar is also an elementary row operation. For example, the following transformation multiplies the second row by 2.

$$
\begin{bmatrix} 2 & 1 & 1 \\ 1 & -1 & 3 \\ 1 & 2 & 3 \end{bmatrix} \sim \begin{bmatrix} 2 & 1 & 1 \\ 2 & -2 & 6 \\ 1 & 2 & 3 \end{bmatrix}
$$

Adding or subtracting a scalar multiple of a row with any other row is an elementary row transformation.

A3.6 CONCLUSION

The chapter introduces the concept of matrices. Though it is a topic of mathematics, it is widely used in a variety of sub-disciplines in computer science. For example, in computer graphics, the transformations are carried out with the help of matrices. Moreover, the knowledge of matrices is also essential in implementing some techniques in applied cryptography and cellular automata. The readers are expected to find the complexity of the algorithm developed while solving the exercises. The web resource of this book contains all the programs of the algorithms dealt with in the chapter. The readers are encouraged to implement the programs in languages like C# and JAVA in order to get a better insight of the topic.

Points to Remember

For two matrices of order $n \times n$

- The complexity of the algorithm for addition of matrices is $O(n^2)$.
- The complexity of subtraction is $O(n^2)$.
- The complexity of multiplication is $O(n^3)$.
- The complexity of multiplication by divide and conquer using Strassen's matrix multiplication is $O(n^{2.7})$.
- The Cramer's rule for solving the system is an equation is one of the most inefficient methods.

KEY TERMS

Column matrix A matrix that has just one column is referred to as a column matrix. The order of a column matrix is $n \times 1$.

Diagonal matrix A diagonal matrix is one which has elements only at the main diagonal. That is, $a_{ij} = 0, i \neq j$.

Identity matrix An identity matrix is a scalar matrix in which the value of k is 1. That is,

$$a_{ij} = \begin{cases} 0, & i \neq j \\ 1, & i = j \end{cases}$$

Lower triangular matrix A lower triangular matrix is one in which the elements above the main diagonal are 0. That is, $a_{ij} = 0$ if $i \geq j$

Matrix A matrix is a two dimensional array of elements. An element of a matrix is represented by two subscripts. The first subscript depicts the row and the second depicts the column number.

Row matrix A matrix which has just one row is referred to as a row matrix. The order of a row matrix is $1 \times n$.

Scalar matrix A scalar matrix is a diagonal matrix in which all the elements are same. That is,

$$a_{ij} = \begin{cases} 0, & i \neq j \\ 1, & i = j \end{cases}$$

Skew-symmetric matrix A matrix is deemed as skew-symmetric, if it is equal to negative of its transpose. That is, $A = -1 \times A^T$

Symmetric matrix A matrix is deemed as symmetric, if it is equal to its transpose. That is, $A = A^T$.

Upper triangular matrix An upper triangular matrix is one in which the elements below the main diagonal are 0. That is, $a_{ij} = 0$ if $i \leq j$

EXERCISES

1. Solve the following equations by Cramer's rule:

   (a) $2x + y + z = 4$
   $4x - y + 3z = 6$
   $9x + 2y + 3z = 13$

   (b) $x + 2y + 8z = 11$
   $x - 3y + 3z = 1$
   $x + 2y + 3z = 6$

   (c) $x + 2y + 2z = 5$
   $5x - y + z = 5$
   $7x - 2y - 3z = 2$

   (d) $2x + y + z = 4$
   $4x + 2y + 2z = 8$
   $x + 2y + 3z = 6$

   (e) $2x + y + z = 4$
   $2x + y + z = 7$
   $x + 2y + 3z = 6$

2. Solve equation numbers a–e by using matrix inverse method.
3. Define the following:
   (a) Row matrix
   (b) Column matrix
   (c) Diagonal matrix
   (d) Scalar matrix
   (e) Identity matrix
   (f) Upper triangular matrix
   (g) Lower triangular matrix

APPENDIX A4

Linear Programming

OBJECTIVES

After studying this appendix, the reader will be able to
- Explain the importance of linear programming
- Understand the graphical method of solving a linear programming problem
- Learn the simplex method to solve a linear programming problem
- Understand the concept of dual and get an idea of the dual simplex method

A4.1 INTRODUCTION

In the second half of 1940s, the American mathematicians were faced with practical problems of the industry. The same probl ems persisted in the army owing to the war that was going on. The problems had few constraints and a function to be maximized or minimized. In order to handle these problems, they developed a method called linear programming. A linear programming problem has constraints represented by equations or inequalities and a function that needs to be maximized or minimized.

A linear programming problem uses graphical, simplex, or a dual simplex method. The problems having a few constraints and two variables can easily be solved using the graphical method explained in Section A4.2. However, most of the real-time problems have a large number of variables and constraints. Methods like simplex can be used to tackle such problems. Linear programming has helped in solving many optimization problems and hence saving money of many companies. This is the reason why the Noble Prize was awarded to Kantorovich and Koopmans in 1975 for their contribution in linear programming.

One of the most important applications of linear programming has been in solving the food problems. A farm animal, for example, needs to be given a balanced diet which has at least the minimum quantity of nutrients, at the same time keeping the total cost as low as possible. The amounts of different foods are depicted by various variables. The minimum amount of each variable is represented by various constraints and the objective function can be obtained by adding the products of quantity of a food with the respective prices. An example of food problem is presented in Illustration A4.4.

Linear programming is not just a mathematical tool; it is also an important part of operational research. The chapter briefly describes the various techniques and exemplifies them. The appendix has been organized as follows. The second section presents the graphical method of solving linear programming. The third section introduces the simplex method, fourth section presents the duality principle and introduces the dual simplex method. The last section gives the conclusion.

A4.2 GRAPHICAL METHOD

A linear programming problem can be represented as follows:

$$\text{Maximize (or minimize) } Z = c_1 x_1 + c_2 x_2 + \cdots + c_n x_n$$

Subject to

$$a_{11} x_1 + a_{12} x_2 + \cdots + a_{1n} x_n (\leq \text{or} \geq) b_1$$

$$a_{21} x_1 + a_{22} x_2 + \cdots + a_{2n} x_n (\leq \text{or} \geq) b_2$$

$$a_{m1} x_1 + a_{m2} x_2 + \cdots + a_{mn} x_n (\leq \text{or} \geq) b_m$$

In order to solve the problem, the following steps need to be followed:
Step 1 Plot the inequalities.
Step 2 Find the common region (if any)
Step 3 Find the vertices of the common region.
Step 4 The value of the objective function is obtained at the values of variables obtained in Step 3. From amongst these values, the maxima or minima is selected.

The maxima or minima will exist at one of the points obtained in Step 3. The maximum or the minimum value is obtained in Step 4. In order to understand the formulation of a linear programming problem, let us consider the following example.

Illustration A4.1 Formulation of a Linear Programming Problem: Hari's Dilemma
Hari wants to start a business. He has 200,000 rupees. He intends to start a C# training institute. He rents a basement in the central market of the city he lives in. The monthly rent of the shop is ₹11,000. The owner wants 1 month's rent in advance. The total amount that needs to be given to the owner, therefore, is ₹22,000. The rest of the money is to be spent judiciously. He intends to buy a signboard for the institute, 10 chairs for students, 2 rolling chairs, a table, and a board. A chair for a student ranges between ₹500 and ₹1500, rolling chairs between ₹3000 and ₹6000, table between ₹2500 and ₹7500, and a board between ₹1000 and ₹1500. The total amount spent should be minimum. As per his advisors, he must spend at least ₹150,000 to start the intended institute. State the above problem as a linear programing problem.

Solution If the amount spend on a chair is 'x', on table is 'y', on a rolling chair is 'z' and board is 't', then the following inequalities must hold.

$$500 \le x \le 1500$$

$$3000 \le z \le 6000$$

$$2500 \le y \le 7500$$

$$1000 \le t \le 1500$$

and

$$10x + y + 2z + t \le 178{,}000$$

The above inequalities are referred to as *constraints*. In general, the constraints depict the conditions that must be satisfied while solving a given problem. Any value that satisfies all the constraints is called a *feasible value*. Moreover, in the above example,

$$x, y, z, t \ge 0$$

as none of the costs can be negative. These are referred to as the *non-zero constraints*. The aim is to minimize the cost, i.e. minimize

$$Z = 10x + y + 2z + t$$

The function that needs to be minimized or maximized is referred to as an *objective function*. The set of values of variables (x, y, z, and t in this case) which satisfy the objective function and maximize (or minimize, as the case may be) the objective function is called *optimal values*.

The above problem is an *optimization problem*.

Let us try to solve a special case of the above problem. In the problem discussed above, Hari gets a white board from his cousin for ₹400 and 2 rolling chairs for ₹1000. The variables z and t are no longer needed. In addition, ₹2400 has been reduced from the estimated amount. The above inequalities now reduce to

$$500 \le x \le 1500$$

$$2500 \le y \le 7500$$

and

$$10x + y \le 175{,}600$$

$$x, y \ge 0$$

The aim is to minimize the cost, i.e. minimize

$$Z = 10x + y$$

There are many techniques to reach the optimal solution. The easiest is the use of graphs in linear programming. This graphical method begins with plotting the inequalities on a graph and then, if the common region is a closed curve, finding the vertices of the polygon depicting the common region. This method is called the *corner point method*. For example, in the ongoing illustration, the common region is depicted in Fig. A4.1.
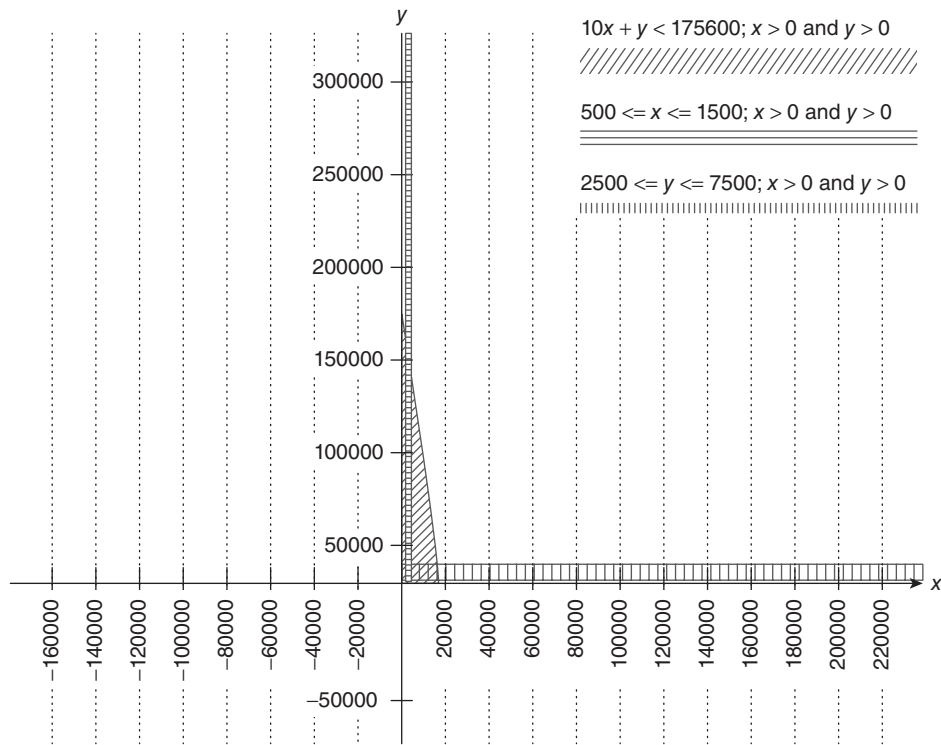
Figure A4.1 Solution of optimization problem via graphical method

The corner points of the common region are (500, 2500), (1500, 2500), (500, 7500), and (1500, 7500). Clearly, the value of the objective function is minimum at $x = 500$ and $y = 2500$.

A more involved example of the graphical method is as follows.

Illustration A4.2 Maximize $Z = 3x + 4y + 1$ subject to the following constraints:

$$2x + y \leq 6$$
$$x + 2y \leq 6$$
$$x \geq 0, y \geq 0$$

Solution First of all, the inequalities need to be plotted. This is followed by finding the common region. The following graph (Fig. A4.2) depicts the common region of the inequalities.

The corner points of the common region are (0, 0), (3, 0), (0, 3), and (2, 2). The values of the objective function at these points are 1, 10, 14, and 15, respectively. Clearly, the value of the objective function is maximum for $x = 2$ and $y = 2$.

For some of the problems, however, it is not possible to find the maximum or the minimum value of the objective function. For example, had the inequalities in the above question been as follows, it would not be possible to find the optimal solution.
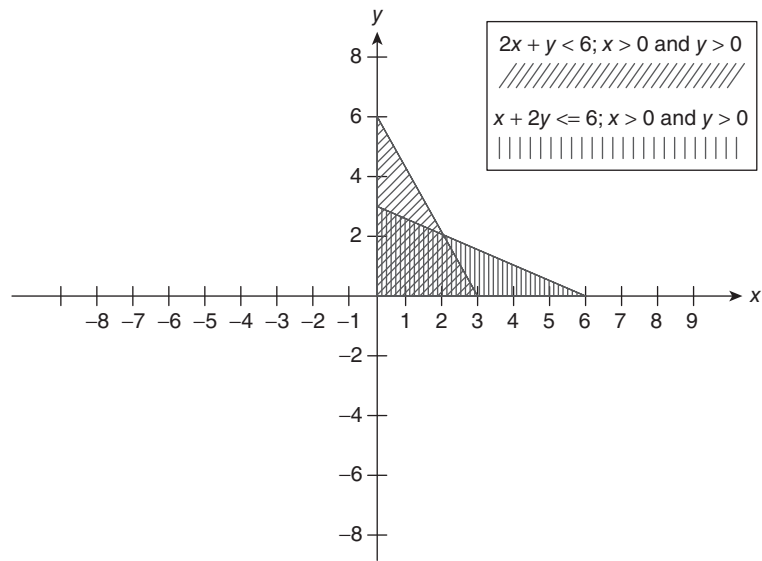
Figure A4.2 Graph for Illustration A4.2

Illustration A4.3 Maximize $Z = 3x + 4y + 1$ subject to the following constraints:

$$2x + y \leq 6$$
$$x + 2y \leq 6$$
$$x \geq 0, y \geq 0$$
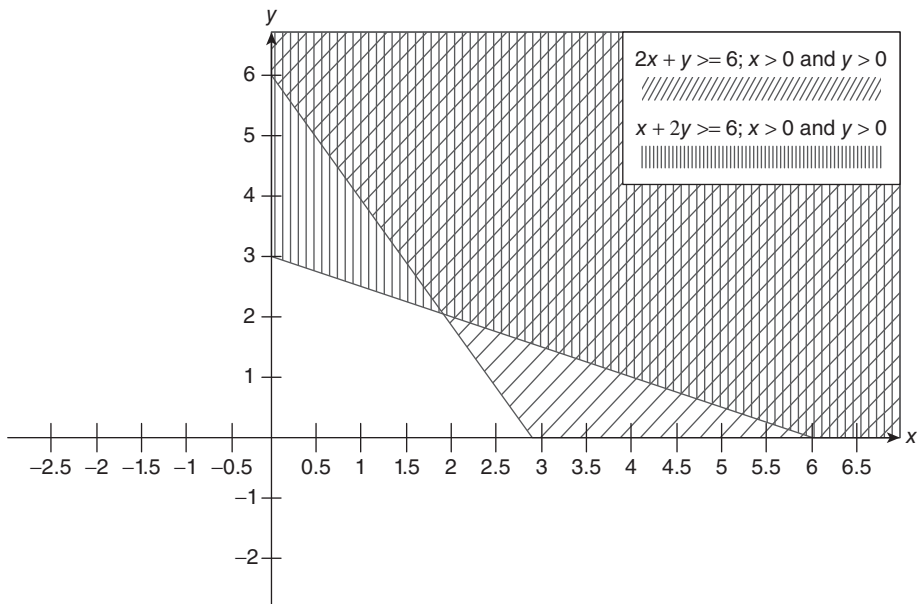
Figure A4.3 Graph for Illustration A4.3

Solution First of all, the inequalities need to be plotted. This is followed by finding the common region. The following graph (Fig. A4.3) depicts the common region of the inequalities.

The corner points of the common region are (0, 6), (6, 0), and (2, 2). The values of the objective function at these points are 1, 10, and 14, respectively. However, in this case, we cannot say that the maximum value of the objective function is 14 at $x = 0$ and $y = 3$. The reason is that the region is an open-ended one.

The graphical method for finding the solution of an optimization problem works well if the number of inequalities is less and the number of variables is two. In other cases, the method either does not work or becomes cumbersome. Another method of solving an optimization problem using linear programming is the simplex method.

The diet problem: The aim of this problem is to select the quantity of n food items so that the net cost of the food is minimum and at the same time, fulfils the minimum dietary requirements of different nutrients. The following illustration (Illustration A4.4) explains the concept.

Illustration A4.4 The Vitamin A content of a food X is 3 units and that of food Y is 2 units. The Vitamin B content of X is 2 units and that of food Y is 3 units. The minimum requirement in a diet of Vitamin A is 10 units and so is that of Vitamin B. The cost of food X is 5 units and that of food Y is 7 units. What amount of X and Y should be included in the diet in order to fulfil the minimum dietary requirements and keep the cost as low as possible?

Solution The above problem can be formulated as follows. Let the quantity of the food X be x and that of Y be y. The constraints are as follows:

$$3x + 2y \geq 10$$
$$2x + 3y \geq 10$$

Moreover, the quantity of X or that of Y cannot be negative,

$$x \geq 0$$
$$y \geq 0$$

The objective function, in this case, is the sum of the products of the quantities (in this case x and y) and the corresponding costs (in this case, 5 and 7).

$$z = 5x + 7y$$

The graph of the above problem is as follows (Fig. A4.4).

The corner points of the above graph are (0, 5), (5, 0), and (2, 2). The values of the objective function at these points are 35, 25, and 24, respectively. Clearly, the value of the objective function at (2, 2) is minimum. Therefore, the quantity of food X should be 2 and that of Y should be 2.

A4.3 SIMPLEX METHOD

As stated in the introduction, a problem that has just two variables can be solved easily using the graphical method. However, if the number of variables is more than two, the
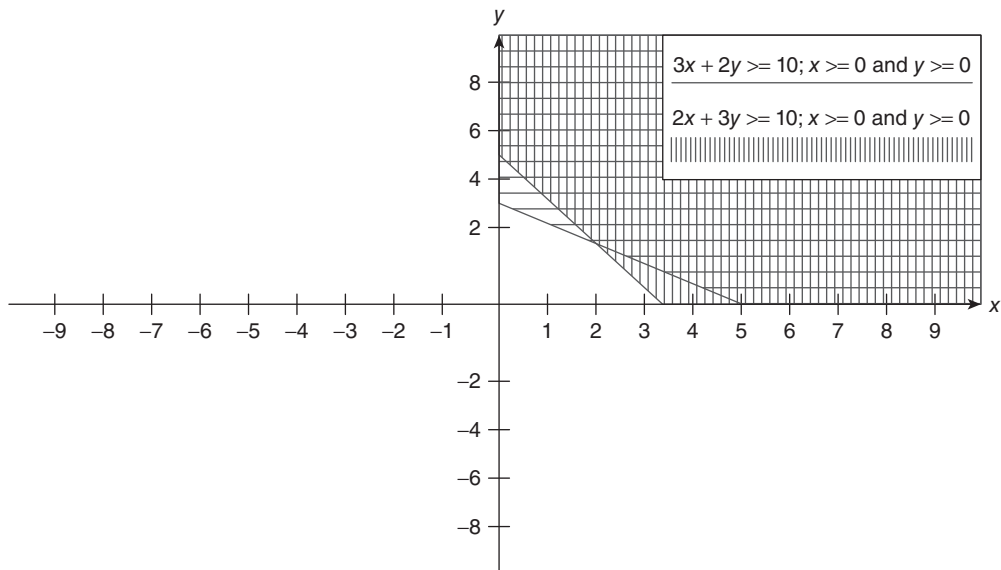
Figure A4.4 Food problem

simplex method helps in solving the given problem. The method was developed in 1947, by George B. Dantzig. The constraints in this method are converted into equations. Using the simplex method, there can be any number of constraints and those constraints can have any number of variables. Theoretically, the solution of the given problem lies at the corners of the n-dimensional polyhedron formed by the given equations. The method involves crafting of new tables by modifying the old ones. The method has been explained and identified in the following example. In order to understand the method, the following terms must be understood:

Slack variables These are the variables added to the left side of the $\leq$ type inequality to convert it into an equation.

For example, the inequality $ax + by \leq c$ can be converted into an equation $ax + by + s = c$, by adding a slack variable s.

Surplus variables These are the variables subtracted from the left side of the $\geq$ type inequality to convert it into an equation.

For example, the inequality $ax + by \leq c$ can be converted into an equation $ax + by - s = c$, by subtracting a surplus variable s.

Basic solution If there are n variables and m constraint equations/inequalities, then the basic solutions can be obtained by setting $(n - m)$ variables equal to 0.

Degenerate solutions If one or more variables in the basic feasible solution are zero(s), then the solutions are called degenerate solutions.

The Standard form The following conditions must be satisfied by the standard form in a linear programming problem (LPP) in order to apply the simplex method.

• The objective function should be maximization type.

- Each constraint should be converted into an equation.
- The right-hand side of the equations should be negative.

The standard form of an LPP can be stated as follows:

$$\text{Maximize } Z = c_1 x_1 + c_2 x_2 + \cdots + c_n x_n$$

Subject to

$$a_{11} x_1 + a_{12} x_2 + \cdots + a_{1n} x_n + s_1 = b_1$$
$$a_{21} x_1 + a_{22} x_2 + \cdots + a_{2n} x_n + s_2 = b_2$$
$$a_{m1} x_1 + a_{m2} x_2 + \cdots + a_{mn} x_n + s_1 = b_m$$

In order to solve the problem, an initial table is created as follows. The first row contains the coefficients of the variables in the objective function, starting from the third column. The second row of the table contains the headers of the columns. The first column would contain the coefficients of the basics in the objective functions, the second column would contain the basics, the third would contain the solution of the basics, and next columns would contain the coefficients of the corresponding variables. As per the last two rows are concerned, the last row is obtained by subtracting the elements of the previous row from the coefficients of the variables in the objective function (in the first row).

The variable corresponding to the highest valued element in the last row is then seen. This becomes the incoming variable in the next table. The elements corresponding to the column having header 'Solution' are then divided by the elements in the selected column. The ratio is written in the last column. From this column, the row having the least value is selected. The corresponding variable becomes the outgoing variable. In the second table, the outgoing variable would be replaced by the incoming variable. The intersection of the selected row and selected column is then selected. The row of which this element is a member is then divided by this element. The element at the intersection of the selected row and column now becomes 1. With the help of this 1, the other elements in the selected column (except for the elements in the last two rows) should be converted to 0. In order to accomplish this task, in this case, we multiply the row by the corresponding element in the second row and subtract it from the succeeding rows.

The process continues till a positive number is obtained in the last row. When the process stops, the solutions corresponding to the basics give the answers.

In order to understand the process, let us go through the following illustration.

Illustration A4.5 Solve the following problem using simplex method:

$$\text{Maximize: } z = 4x + 3y$$

Subject to the following constraints:

$$2x + y \leq 6$$
$$x + 2y \leq 9$$
$$x \geq 0$$
$$y \geq 0$$

Solution In the first step, the inequalities need to be converted to equations

$$2x + y + s_1 = 6$$
$$x + 2y + s_2 = 9$$
$$x \geq 0$$
$$y \geq 0$$
$$s_1 \geq 0$$
$$s_2 \geq 0$$

and the objective function becomes $z = 4x + 3y + 0s_1 + 0s_2$

Number of equations $= 2$

Number of variables $= 4$

Number of basics $= 4 - 2 = 2$

So, we substitute $x = 0$ and $y = 0$ in the above equations and hence the value of s_1 becomes 6 and that of s_2 becomes 9.

The problem is solved by making some tables, described as follows. As stated earlier, the second row of the table contains the headers of the columns. The first column would contain the coefficients of the basics in the objective functions, the second column would contain the basics, the third would contain the solution of the basics, and next columns would contain the coefficients of the corresponding variables. Consider the last two rows; the last row is obtained by subtracting the elements of the previous row from the coefficients of the variables in the objective function (in the first row).

The initial table, therefore, is as follows (Table A4.1)

Table A4.1 Table 1 of simplex method

The variable corresponding to the highest valued element in the last row is then seen (In this case x for 4). This becomes the incoming variable in the next table. The elements corresponding to the column having header "Solution" are then divided by the elements in the selected column (2 and 1 in this case). The ratio is written in the last column (an extra column added in Table A4.2, having 3 and 9, in this case). From this column, the row having the least value is selected (3 in this case, corresponding to s_1). The corresponding variable becomes the outgoing variable. In the second table, the outgoing variable would be replaced by the incoming variable. The intersection of the selected row and selected column is then selected. The row of which this element is a member is then divided by this element. In this case, the elements of the row would become (3, 1, ½, ½, 0). The element at the intersection of the selected row and column now becomes 1. With the help of this 1, the other elements in the selected column (except for the elements in the last two rows) are to become 0. In order to accomplish this task, in this case, we multiply the row by the corresponding element in the second row (1 in this case) and subtract it from the second row. The elements of the second row now become (8, 0, 3/2, –½, 1).

The second table is therefore as follows (Table A4.2).

Table A4.2 Table 2 of simplex method

The coefficients of basics in the objective function

The coefficients of variables in the objective function

	Basics	Solution	X	Y	S_1	S_2	
			4	3	0	0	
	Basics	Solution	X	Y	S_1	S_2	
4	X	3	1	1/2	1/2	0	6
0	S_2	6	0	3/2	–1/2	1	4
			4	2	2	0	
			0	1	–2	0	

Variables

Selected Row (minimum value)

Numbers obtained by multiplying constants of basics with the elements of this column.

The highest value in this case is 1. This corresponds to the column having y. The incoming variable is therefore y. The elements in the solution column are then divided by the elements of the selected column.

The process continues as the last row has some positive entries. The steps explained above are repeated. A new table (Table A4.3) is then crafted. Note that, at every step, the values of *x* and *y* obtained are feasible, though the optimal values are obtained in the last step. The reader is expected to complete the last row of the table to see whether there is any requirement of making a new table.

Table A4.3 Table 3 of simplex method

| The coefficients of basics in the objective function | | | | | | |

	Basics	Solution	X	Y	S_1	S_2
			4	3	0	0
4	X	1	1	0	2/3	−1/3
3	Y	4	0	1	−1/3	2/3

The coefficients of variables in the objective function

Variables

A4.4 FINDING DUAL AND AN INTRODUCTION TO THE DUAL SIMPLEX METHOD

The dual of a given problem can be found as follows.

Check: All the inequalities are of ≥ type or ≤ type. If not, then they can be made so by multiplying the inequalities by −1.

Step 1 Write the matrix corresponding to the coefficients of the variables in the given inequalities and a matrix corresponding to the constants of the inequalities.

Step 2 The number of variables in the dual would be equal to the number of rows in the above matrices. Call these variables y_1, y_2, y_3, and so on. Create a row matrix consisting of these variables.

Step 3 Multiply the above matrix with the matrix of coefficients obtained in the first step. The result would be the left side of the new constraints.

Step 4 The right-hand side of the constraints would be the coefficients of the given objective functions. If the given inequalities are of ≤ type convert to ≥ type and vice versa.

Step 5 If the objective function of the given problem is to be maximized, the objective function of the dual would have to be minimized and if the objective function is to be minimized, then the objective function of the dual would have to be maximized.

Illustration A4.6 exemplifies the above steps.

Illustration A4.6 Find the dual of the following:

$$\text{Maximize: } z = 4x + 3y$$

Subject to the following constraints:

$$2y + y \leq 6$$
$$x + 2y \leq 9$$
$$x \geq 0$$
$$y \geq 0$$

Solution The matrix corresponding to the coefficients of the variables in the inequalities

is $\begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}$, the matrix corresponding to the constants is $\begin{pmatrix} 6 \\ 9 \end{pmatrix}$ and that corresponding to

the coefficients of the variables in the objective function is $\begin{pmatrix} 4 \\ 3 \end{pmatrix}$.

The dual of the given problem therefore is as follows:

$$2y_1 + y_2 \geq 4$$
$$y_1 + 2y_2 \geq 3$$
$$y_1 \geq 0$$
$$y_2 \geq 0$$

The objective function of the dual can be obtained by multiplying the constants of the constraints (of the given problem) with the new variables. Moreover, in the given problem, the objective function needs to be maximized; in the dual, the new objective should be minimized. That is, Minimize $z^* = 6y_1 + 9y_2$.

Tip: The optimal solution of a problem is same as that of its dual.

Dual Simplex Method

The dual simplex method is similar to the simplex method discussed in the Section A4.4. In the dual simplex method, we first select the outgoing variable and then the incoming variables. The method starts with a basic solution which is infeasible, as against the simplex method which starts with a feasible solution. The goal therefore is to convert the infeasible solutions to the feasible ones. The solution is obtained by the following steps.

- In order to obtain the solution using this method, first of all, the inequalities are converted into the ≤ type, if they are ≥ type.

- The inequalities are converted into equations by adding the slack variables.
- The initial basic solution is obtained by setting $(n - m)$ variables equal to zero.
- The initial table is formed in the same way as explained in the previous illustration.
- The next step is to select the key row and the outgoing variable. This is followed by the selection of the key column and the incoming variable.
- The process is continued till a feasible solution is obtained or it becomes clear that no feasible solution is possible.

A4.5 CONCLUSION

Linear programming is one of the greatest inventions of the 20th century. It is a practical model for solving optimization problems. As a matter of fact, many problems can be reduced to this technique. Moreover, it is one of the methods which help to solve one of the most important classes of NP problems, that is, optimization problems. It is used in compiler design for register allocation, in sports for scheduling basketball, in bioinformatics for constraint-based metabolic models, in marketing for advertising, in finance for portfolio management, in medicine for radioactive cancer treatment, and so on. This appendix explains the graphical method and simplex method of linear programming, the dual of a problem, and introduces the concept of dual simplex. The topic though not a part of the core algorithm syllabus, it is necessary to solve many problems discussed in the text.

Points to Remember

- A linear programming problem has constraints represented by equations or inequalities and a function that needs to be maximized or minimized.
- A linear programming problem uses graphical, simplex, or a dual simplex method.
- A graphical method is used when the number of variables is not greater than two.
- The solution of an LPP lies at the corners of a feasible region.
- The solution of the dual of a problem is same as that of the problem.

KEY TERMS

Basic solution If there are n variables and m constraint equations/inequalities, then the basic solutions can be obtained by setting $(n - m)$ variables equal to 0.

Constraints The inequalities in an LPP are referred to as constraints. The constraints depict the conditions that must be satisfied while solving a given problem.

Degenerate solutions If one or more variables in the basic feasible solution are zero(s), then the solutions are called degenerate solutions.

Feasible solution Any value that satisfies all the constraints is called a feasible value.

Objective function The function that needs to be minimized or maximized is referred to as an objective function.

Optimal solution The set of values of variables (x, y, z, and t in this case) that satisfy the objective function and maximize (or minimize, as the case may be) the objective function is called optimal values.

Slack variables These are the variables added to the left side of the $\leq$ type inequality to convert it into an equation.

Standard form of LPP Maximize (or minimize) $Z = c_1 x_1 + c_2 x_2 + \cdots + c_n x_n$

Subject to

$$a_{11} x_1 + a_{12} x_2 + \cdots + a_{1n} x_n (\leq \text{ or } \geq) b_1$$

$$a_{21} x_1 + a_{22} x_2 + \cdots + a_{2n} x_n (\leq \text{ or } \geq) b_2$$

$$a_{m1} x_1 + a_{m2} x_2 + \cdots + a_{mn} x_n (\leq \text{ or } \geq) b_m$$

Surplus variables These are the variables subtracted from the left side of the $\geq$ type inequality to convert it into an equation.

EXERCISES

I. Review Questions

1. Explain the importance and applications of linear programming.
2. Explain the graphical method of solving a linear programming problem.
3. Explain the simplex method of solving the linear programming problem.
4. What are surplus and slack variables?
5. Explain the steps to obtain the dual of a given problem.
6. What is the difference between simplex and dual simplex method?
7. Explain the difference between optimal and feasible solution?
8. Can graphical method be used to solve an LPP having 3 variables? If not, which method would you suggest to solve such LPP?
9. An LPP has some $\leq$ type inequalities and some $\geq$ type inequalities; can simplex method be applied to solve such problem?
10. Explain one use of finding dual of a given LPP.

II. Applications

1. A meal consists of two foods X and Y. X costs ₹9 per unit and Y costs ₹12 per unit. X contains 4 units of vitamin A and 2 units of vitamin B. The food Y, on the other hand, contains 2 units of vitamin A and 4 units of vitamin B. It is required to have at least 12 units of vitamin A in the diet and at least 10 units of vitamin B. Express the problem as a linear programming problem and solve it using graphical method.
2. In the above problem, the maximum requirement of A is 20 and 15 units of vitamin B. What would be the answer in this case?

3. In a factory, the manufacturing of a chair requires 3 hours on the machine A and 4 hours on machine B. That of a table requires 4 hours on machine A and 4 hours on machine B. The machine A is available for 8 hours and machine B for 10 hours per day. If the profit earned by selling a table to a vendor is ₹100 and that by selling a table is ₹200, how many chairs and how many tables should be manufactured by the factory in order to maximize the profit?

4. In the above problem, if the management decides to install another machine C, in the factory, the machine would be used for finishing purposes. A chair would require 1 hour on C and a table requires 1.5 hours on C. Formulate the above problem as a linear programming problem and solve it using the graphical method.

5. In the tri-state area, there are two depots of gasoline. The first depot, X has 7000 L of gasoline and the second depot, Y, has 4000 L of gasoline. There are three stations A, B, and C. The requirements of A, B, and C are, 2000, 2000, and 3000, respectively. The cost of transportation of gasoline from X to A is 2 (per unit per km), that from X to B is 3 (per unit per km), from X to C is 4 (per unit per km), from Y to A is 4 per unit per km, from Y to B is 3 per unit per km, and from Y to C is 4 per unit per km. The distances are given in the following table.

.	A	B	C
X	10	12	23
Y	21	20	13

Find the amount of gasoline that should be transported from the depots to the stations in order to minimize the cost of transportation.

6. Solve the above problems (Question 1–5) using simplex method.

7. Find the dual of the following:

(a) Maximize $Z = 9x + 11y$ subject to

$$7x + 8y \leq 30$$
$$2x + 5y \leq 21$$
$$x \geq 0$$
$$y \geq 0$$

(b) Maximize $Z = 9x + 9y$ subject to

$$x + y \leq 3$$
$$2x + 5y \leq 81$$
$$x \geq 0$$
$$y \geq 0$$

(c) Maximize $Z = 11x + 3y$ Subject to

$$x - y \leq 13$$
$$2x + 7y \leq 213$$
$$x \geq 0$$
$$y \geq 0$$

(d) Maximize $Z = 2x + y$ Subject to

$$3x + 4y \leq 32$$
$$9x + 5y \leq 11$$
$$x \geq 0$$
$$y \geq 0$$

(e) Maximize $Z = x + 6y$, Subject to

$$17x + 81y \leq 113$$
$$2x + 2y \leq 10$$
$$x \geq 0$$
$$y \geq 0$$

8. Solve the above linear programming problems (Question 7) using graphical method (if possible).
9. Write a program to find the dual of a standard LPP.
10. Write a program to solve a standard LPP using simplex method.

APPENDIX   A5

Complex Numbers and Introduction to DFT

OBJECTIVES

After studying the appendix, the reader will be able to

- Explain the concept of complex numbers
- Learn the polar form of complex numbers
- Understand the procedure to find the power and roots of a complex number
- Apply algorithm for finding the discrete Fourier transform of a signal

A5.1 INTRODUCTION

So far we have studied the various techniques to develop and analyse algorithms. These techniques are also widely used in electronics to carry out the spectrum analysis. One of the ways of doing so is to find the Fourier transform of the signal. The following discussion gives a brief introduction of spectrum and the need of Fourier transform. The second section discusses the basics of complex roots of unity. This concept is used to calculate the DFT of a signal, which has been discussed in the last section.

The word spectrum was used in a paper presented by Newton in 1672. In the paper, he showed that white light can be split into seven colours and these colours can recombine to form white light again. Since all the colours have a particular frequency, the output from a prism can be considered as a spectrum. The spectrum of a signal is obtained by a mathematical tool like Fourier transform.

The frequency spectrum of a signal represents its frequency components. The process of splitting a signal into its frequency components, with the help of some mathematical tool, is referred to as the spectrum analysis of that signal. The spectrum analysis of a signal is also called frequency analysis. Spectrum estimation, on the other hand, is the process of estimating the frequency components of a signal by measurements. The latter is important as all the signals cannot be represented in mathematical form. If mathematical form of a signal is given, then we carry out the spectrum analysis. On the other hand, if we have a real-time signal and the requisite instruments, then spectrum estimation can be carried out. It is important to understand the difference between the two.

The signals that are generally dealt with, in signal processing, are of four types. These are continuous periodic, continuous non-periodic, discrete periodic, and discrete non-periodic. A periodic signal is one which repeats itself after a fixed interval of time. For such signal $f(t) = f(T + t)$, that is, the signal $f(t)$ has same value after every T, which is the fundamental period. For example, a sine function is periodic as it repeats itself after an interval of 2π. The fundamental period of sin is therefore 2π. The fundamental period of sin ωt is $2\pi/\omega$.

A continuous signal is one in which the left-hand limit at every point is same as the right-hand limit and both of them are equal to the value of the function at that point. A sine function, for example, is both continuous and periodic at every point.

A deterministic signal is generally represented by a mathematical function. The spectrum of a signal can be found by mathematical tools such as Fourier series and Fourier transform. In order to find the Fourier transform of a signal in an effective and efficient way, the know how of complex numbers is essential. This appendix gives a brief introduction of complex numbers and presents the related algorithms. The appendix then goes on to introduce the concept of discrete Fourier transform (DFT) and presents an efficient way to calculate the DFT.

A5.2 COMPLEX NUMBERS

Complex numbers are to computer science and electronics as water to mankind. These numbers are used in converting the time-domain signals to frequency domain and vice versa. Various algorithms in digital signal processing also use complex numbers. There are numerous other applications of these numbers. Because of this importance, it becomes essential to understand the basics of complex numbers to be able to use them in various applications. The following discussion throws light on the basics of complex numbers, their power, and roots. The complex roots of unity and their properties have also been discussed in this section. This is required in order to understand DFT.

A5.2.1 Complex Number: Cartesian and Polar Form

A complex number has a real and an imaginary part. The imaginary part is written along with 'i' which is $\sqrt{-1}$. In a complex number $z = x + iy$, the real part is 'x' and the imaginary part is 'y'. The above form is referred to as the Cartesian form. A complex number that does not have an imaginary part is purely real and the one that does not have a real part is purely imaginary. The complex number $3i$ is purely imaginary. The complex number 7 is purely real. A complex number can also be written in the polar form, which is $r(\cos\theta + i\sin\theta)$, where $r = \sqrt{x^2 + y^2}$ and $\theta = \tan^{-1}\dfrac{y}{x}$. This is because of the following:

$$x = r\cos\theta \qquad\qquad (A5.1)$$

$$y = r\sin\theta \qquad\qquad (A5.2)$$

Squaring and adding the above two equations (A5.1 and A5.2)

$r^2 = x^2 + y^2$, therefore $\qquad\qquad r = \sqrt{x^2 + y^2}$

Dividing Eqns (A5.1) and (A5.2)

$$\tan\theta = \frac{y}{x}$$

$$\theta = \tan^{-1}\frac{y}{x}$$

The following illustrations would help in understanding the conversion of Cartesian form to polar form.

A5.2.2 Conversion of a Complex Number into Polar Form

(i) The complex number $1 + i$ has 1 as its real part and 1 as its imaginary part. The value of r becomes $\sqrt{1^2 + 1^2} = \sqrt{2}$ and $\theta = \tan^{-1}1 = \pi/4$.

Therefore, $1 + i = \sqrt{2}\left(\cos\frac{\pi}{4} + i\sin\frac{\pi}{4}\right)$

(ii) The complex number $1 - i$ has 1 as its real part and -1 as its imaginary part. The value of r becomes $\sqrt{1^2 + (-1)^2} = \sqrt{2}$ and $\theta = \tan^{-1}-1 = -\pi/4$ (as cos is positive and sin is negative in the fourth quadrant).

Therefore, $1 - i = \sqrt{2}\left(\cos\left(-\frac{\pi}{4}\right) + i\sin\left(-\frac{\pi}{4}\right)\right) = \sqrt{2}\left(\cos\left(\frac{\pi}{4}\right) - i\sin\left(\frac{\pi}{4}\right)\right)$

(iii) The complex number $-1 + i$ has -1 as its real part and 1 as its imaginary part. The value of r becomes $\sqrt{(-1)^2 + 1^2} = \sqrt{2}$ and $\theta = \tan^{-1}-1 = 3\pi/4$ (as cos is negative and sin is positive in the second quadrant).

Therefore, $1 + i = \sqrt{2}\left(\cos\left(\frac{3\pi}{4}\right) + i\sin\left(\frac{3\pi}{4}\right)\right) = \sqrt{2}\left(\cos\left(\frac{3\pi}{4}\right) + i\sin\left(\frac{3\pi}{4}\right)\right)$

(iv) The complex number $-1 - i$ has -1 as its real part and -1 as its imaginary part. The value of r becomes $\sqrt{(-1)^2 + (-1)^2} = \sqrt{2}$ and $\theta = \tan^{-1}1 = 5\pi/4$ (as both cos and sin are negative in the third quadrant).

Therefore, $1 - i = \sqrt{2}\left(\cos\left(\frac{5\pi}{4}\right) + i\sin\left(\frac{5\pi}{4}\right)\right)$

A5.2.3 Power and Root of a Complex Number

Although the nth power of a complex number can be found by multiplying the complex number n times, as per the following procedure (Algorithm A5.1), the procedure is inefficient.

Algorithm A5.1 Power of a complex number (z, n) returns a complex number

Input: The complex number, Z; the exponent, n.
Output: p, which is, z^n

```
{
//p.real indicates the real part of p and p.imaginary indicates the imaginary
part of p
p.real = 1;
p.imaginary =0;
for( i= 1; i< =n; i++)
    {
    p.real = p.real × z.real – p.imaginary × z.imaginary;
    p.imaginary = p.real × z.imaginary + p.imaginary × z.real;
    }
return p;
}
```

Complexity: A single loops makes the complexity of the above algorithm $O(n)$. The number of multiplications would be $4n$. This complexity can be improved using the method explained in the following discussion.

A5.2.4 Finding Powers and Roots of a Complex Number Using the Polar Form

As stated earlier, the polar form is another way of expressing a complex number. The real part of the complex number is in terms of cosine of some angle and the imaginary part in terms of sine. The polar form helps to find the root and power of a number easily. The power of a complex number can be found by converting the number into polar form and then applying the following formula:

$$(r(\cos\theta + i\sin\theta))^n = (r^n(\cos n\theta + i\sin n\theta)) = r^n e^{n\theta}$$

The algorithm to find the power of complex number using polar form is as follows (Algorithm A5.2).

Algorithm A5.2 Power of a complex number (z, n) returns a complex number

Input: The complex number, z; the exponent, n.
Output: p which is, z^n

```
{
```
$$r = \sqrt{z.real^2 + z.imaginary^2};$$
$$\theta = \tan^{-1}\frac{|z.imaginary|}{|z.real|};$$

```
//to find correct quadrant
    if( (z.real <0) &&(z.imaginary >0))
        {
        θ = π - θ;
        }
    else if ( (z.real <0) &&(z.imaginary <0))
        {
        θ = π + θ;
        }
    else if ( (z.real >0) &&(z.imaginary <0))
        {
        θ = -θ;
        }
θ = nθ;
r = rⁿ
p.real = r cos θ
p.imaginary = r sin θ;
return p;
}
```

Complexity: There is no loop and a recursive function call in the algorithm. This makes the complexity of this algorithm as $O(1)$. The number of multiplications would be $O(1)$.

Illustration A5.1 Calculate $\left(1+i\sqrt{3}\right)^6$.

Solution The complex number $\left(1+i\sqrt{3}\right)$ needs to be converted to polar form. The real part of the complex number is 1 and the imaginary part is $\left(\sqrt{3}\right)$. Therefore,

$$r\cos\theta = 1$$

and
$$r\sin\theta = \sqrt{3}$$

The value of r is therefore,

$$r = 2,\ \tan\theta = \sqrt{3},\ \text{and}\ \theta = \pi/3$$

Since, $(r(\cos\theta + i\sin\theta))^n = (r^n(\cos n\theta + i\sin n\theta))$.

Therefore, $(2(\cos\pi/3 + i\sin\pi/3))^6 = (2^6(\cos 2\pi + i\sin 2\pi)) = 64$.

A5.2.5 Roots of a Complex Number

The nth root of unity can be found by the following formula:

$$(r(\cos\theta + i\sin\theta))^{1/n} = \left(r^{1/n}\left(\cos\frac{\theta + 2\pi k}{n} + i\sin\frac{\theta + 2\pi k}{n}\right)\right),\ k \text{ varies from 0 to } (n-1).$$

The procedure for calculating the n, nth roots is as follows (Algorithm A5.3).

Algorithm A5.3 Root of a complex number (z, n) prints the roots of the complex number

Input: The complex number, z; the value of n.

Output: p, which is, $z^{\frac{1}{n}}$

```
{
r = √z.real² + z.imaginary²;

        |z.imaginary|
  θ = tan⁻¹ ─────────── ;
           |z.real|
//to find correct quadrant
    if( (z.real <0) &&(z.imaginary >0))
        {
        θ = π − θ;
        }
    else if ( (z.real <0) &&(z.imaginary <0))
        {
        θ = π + θ;
        }
    else if ( (z.real >0) &&(z.imaginary <0))
        {
        θ = −θ;
        }
for(k=0; k< n ;k++)
        {
            θ = (θ + 2πk)/n ;
        r = r¹ᐟⁿ;
        p.real = r cos θ;
        p.imaginary = r sin θ;;
        print: p;
        }
}
```

Complexity: There is a loop in the algorithm, which makes the complexity of this algorithm as $O(n)$.

A5.2.6 Cube Roots of Unity

Since 1 can be written as $(\cos 0 + i \sin 0)$, the nth root of unity can also be found by using the above formula. For example, the cube roots of unity can be found by

$$\left(1\left(\cos 0 + i \sin 0\right)\right)^{1/3} = \left(1^{1/3}\left(\cos\frac{0 + 2\pi k}{3} + i \sin\frac{0 + 2\pi k}{3}\right)\right), \ k \text{ varies from 0 to 2.}$$

That is, the first root is $\left(\cos \dfrac{0+2\pi 0}{3} + i\sin \dfrac{0+2\pi 0}{3} \right) = 1$

The second root is $\left(\cos \dfrac{0+2\pi \times 1}{3} + i\sin \dfrac{0+2\pi \times 1}{3} \right) = \left(\cos \dfrac{2\pi}{3} + i\sin \dfrac{2\pi}{3} \right) = -\dfrac{1}{2} + \dfrac{i\sqrt{3}}{2}$,

generally denoted by ω.

The third root is $\left(\cos \dfrac{0+2\pi \times 2}{3} + i\sin \dfrac{0+2\pi \times 2}{3} \right) = \left(\cos \dfrac{4\pi}{3} + i\sin \dfrac{4\pi}{3} \right) = -\dfrac{1}{2} - \dfrac{i\sqrt{3}}{2}$,

generally denoted by ω^2.

The following points may be observed as regards the cube roots of unity:
- The sum of the cube roots of unity is 0.
- The product is one.
- Each complex root is the conjugate of other.
- Each is the square root of the other.
- Each is the square of the other.
- Each is the reciprocal of the other.

A5.2.7 nth Roots of Unity

The nth roots of unity are the solutions of $x^n = 1$. They are denoted by $\omega_0^n, \omega_1^n, \omega_2^n, ..., \omega_{n-1}^n$, where $\omega_i^n = e^{2\pi i/n}$.

Product of Roots of Unity

The product of these roots can be calculated as follows:

$$\omega_0^n \times \omega_1^n \times \omega_2^n ... \times \omega_{n-1}^n$$

$$= e^{\frac{2\pi 0}{n}} \times e^{\frac{2\pi 1}{n}} \times ... \times e^{\frac{2\pi (n-1)}{n}}$$

$$= e^{\frac{2\pi (0+1+2...+(n-1))}{n}}$$

$$= e^{\frac{2\pi n(n-1)}{2n}}$$

$$= e^{2\pi (n-1)}$$

$$= 1$$

Sum of Roots of Unity

The sum of these roots of unity can be calculated as follows:

$$\omega_0^n + \omega_1^n + \omega_2^n ... + \omega_{n-1}^n$$

$$= 1 + e^{\frac{2\pi 1}{n}} + \cdots, + e^{\frac{2\pi (n-1)}{n}}$$

which is a geometric progression (GP) (Section 2.2.3 Chapter 2) having $a = 1$ and $r = e^{\frac{2\pi 1}{n}}$. The sum of this GP is therefore,

$$1\left(e^{\frac{n \times 2\pi}{n}} - 1\right)$$
$$\overline{e^{\frac{2\pi}{n-1}}}$$

$$= 0$$

Cancellation Law

The $\omega_{in}^{ik} = \omega_n^k$

The value of ω_{in}^{ik}

$$e^{\frac{2 \neq ik}{in}}$$

$$e^{\frac{2 \neq k}{n}}$$

which is same as ω_n^k.

A5.3 DISCRETE FOURIER TRANSFORM

For a given vector $\{x_1, x_2, x_3, \ldots x_n\}$, the discrete Fourier transform (DFT) is defined as $y_j = \sum_{i=0}^{n-1} a_j \times \omega^{ij}, j = 0, 1, \ldots, (n-1)$, where ω is the complex cube roots of unity.

The algorithm for DFT is as follows (Algorithm A5.4).

Algorithm A5.4   DFT(x) returns y

```
{
    for (j=0; j<n; j++)
        {
            y_j = 0;
            for(i=0; i<n; i++)
            {
            y_j = y_j + x_i × ω_n^{ij}
            }
        }
    return y;
}
```

Complexity: The complexity of the above algorithm is $O(n^2)$ owing to nested loops.

A5.4 USE OF DIVIDE AND CONQUER IN DFT

The DFT can be made efficient by dividing the problem into independent sub-problems, solving them, and then combining the results. This calls for the application of divide and conquer approach. The approach requires the division of the input into two vectors, the first containing the values at the odd index and the second containing the values at

the even index. The two arrays are then given as input to the algorithm and the results contribute to the final answer.

Concept: Any polynomial $x = a_0 \times x^0 + a_1 \times x^1 + \cdots + a_n \times x^n$, can be written as

$$x = x_1 + x \times x_2$$

where $x_1 = a_0 \times x^0 + a_2 \times x^2 + \cdots$ and $x_2 = a_1 \times x^0 + a_3 \times x^2 + \cdots$
On substituting $x^2 = t$, we get

$$x_1 = a_0 \times x^0 + a_2 \times x^2 + \cdots = a_0 \times t^0 + a_2 \times t^2 + \cdots$$

and $x_2 = a_1 \times x^0 + a_3 \times x^2 + \cdots = a_0 \times t^0 + a_2 \times t^2 + \cdots$

The above substitution reduces the n degree bound FFT into $(n/2)$ degree bound FFT. The algorithm is given as follows (Algorithm A5.5).

Algorithm A5.5   FFT (x, n)

```
//x is the given vector, n is the number of elements in the array
    {
    if (n==1)
        {
        return x;
        }
    else
        {
              2πk
        ωₙ = e  ⁿ ;
        x₁ = {a₀, a₂, ...};
        x₂ = {a₁, a₃, ...};
        y₁ = FFT (x₁);
        y₂ = FFT (x₂);
        ω = 1;
        for (i=0; i<n/2 -1; i++)
            {
            yᵢ = y¹ₖ + ωy²ₖ;
            yᵢ₊ₙ/₂ = y¹ₖ + ωy²ₖ;
            ω = ωωₙ;
            }
        return y;
}
```

Complexity: The complexity of the above algorithm can be found by applying the Master theorem.

The recursive equation for the above is $T(n) = 2T\left(\dfrac{n}{2}\right) + O(n)$, as the FFT of order n is converted into that of order $(n/2)$ on the recursive call. Moreover, the complexity of the above is $n \log n$, which is better than that of Algorithm A5.4.

A5.5 CONCLUSION

This appendix presents the basics of complex numbers. The procedure of the conversion of a complex number to its polar form has been discussed in the appendix. This polar form helps us to find the power and the roots of a complex number. This has also been exemplified in the above discussion. The properties of complex cube roots of unity have also been discussed. These cube roots of unity help us to find the DFT of a signal. The reader is expected to implement the algorithms present in the chapter and analyse the efficiency. It is also desirable from the reader to explore the applicability of divide and conquer in finding the Fourier transform of a discrete signal.

Points to Remember

- The word spectrum was used in a paper presented by Newton in 1672.
- The spectrum of a signal is obtained by a mathematical tool like Fourier transform.
- The process of splitting a signal into its frequency components, with the help of some mathematical tool, is referred to as the spectrum analysis of that signal.
- Spectrum estimation is the process of estimating the frequency components of a signal by measurements.
- The signals that are generally dealt with, in signal processing, are of four types. These are continuous periodic, continuous non-periodic, discrete periodic, and discrete non-periodic.
- A periodic signal is one which repeats itself after a fixed interval of time.
- A continuous signal is one in which the left-hand limit at every point is same as the right-hand limit and both of them are equal to the value of the function at that point.

KEY TERMS

Complex number A complex number may have a real and an imaginary part. For $z = a + ib$, a is the real part and b is the imaginary part.

DFT The discrete Fourier transform of a given vector $\{x_1, x_2, x_3, ..., x_n\}$, the Discrete Fourier transform (DFT) is defined as

$y_j = \sum_{i=0}^{n-1} a_j \times \omega^{ij}$, $j = 0, 1, ..., (n-1)$, where ω is the complex cube roots of unity.

Polar form A complex number can be represented in polar form. The polar form of a complex number $z = a + ib$ is $r (\cos \theta + I \sin \theta)$, where $r = \sqrt{a^2 + b^2}$ and $\theta = \tan^{-1}\left(\dfrac{b}{a}\right)$.

EXERCISES

I. Review Questions

1. Write an algorithm to find the power of a given complex number.
2. Write an algorithm to convert a complex number into its polar form.

3. Write an algorithm to find the nth root of a complex number.
4. Write an algorithm to find the DFT of a given signal using divide and conquer.

II. Applications-based Questions

1. Write an algorithm to find the (m/n)th power of a complex number.
2. Write an algorithm to find the DFT of a given signal by dividing the signal in four parts on each call of the algorithm.
3. Can dynamic programming be used to find the nth root of a complex number?
4. Can dynamic programming be used to find the DFT of a given signal?
5. Design a calculator of complex numbers (in the language of your choice) which can perform the following tasks:
   (a) Addition   (e) Power
   (b) Subtraction   (f) Root
   (c) Multiplication   (g) Conjugate
   (d) Division

APPENDIX A6

Probability

OBJECTIVES

After studying this appendix, the reader will be able to

- Explain the concept of probability
- Understand pigeonhole principle, independent events, and Bay's theorem
- Apply probability distribution
- Learn the various probability distributions such as binomial, Poisson's, and normal distribution

A6.1 INTRODUCTION

There are many methods for analysing an algorithm. One of them is probabilistic analysis. The introduction to this was given in Section 4.6 of Chapter 4. The topic requires the basic know-how of the concept of probability. Moreover, the knowledge of probability is also required in designing randomized algorithms as well. This chapter introduces the concept of probability and intends to introduce the reader to the concept by using problems.

The chapter has been organized as follows. Section A6.2 of this chapter deals with the basic concept, Section A6.3 deals with independent events, and Section A6.4 explores the concept of probability distribution. Some of the most important distributions such as 'binomial distribution', 'Poisson's distribution,' and 'normal distribution' have been dealt with in Sections A6.5–A6.7, respectively. This chapter has been designed for a reader having basic knowledge of combinations and permutations. So the reader is advised to go through the principles of counting before starting off with this appendix.

A6.2 BASICS

This section deals with the basics of probability including the taxonomy of probability theory such as event, sample space, probability of success, sure event and impossible event, and pigeonhole principle.

A6.2.1 Taxonomy

An event is the possible outcome of an experiment. The set of these events constitutes what is called the sample space. Some examples of sample space are as follows. Table A6.1 enlists some of the events and the corresponding sample spaces.

Table A6.1 Examples of sample space

Event	Sample space
A dice is thrown	{1, 2, 3, 4, 5, 6}
Two dice are thrown	{(1,1), (1,2), (1,3), (1,4), (1,5), (1,6), (2,1), (2,2), (2,3), (2,4), (2,5), (2,6), (3,1), (3,2), (3,3), (3,4), (3,5), (3,6), (4,1), (4,2), (4,3), (4,4), (4,5), (4,6), (5,1), (5,2), (5,3), (5,4), (5,5), (5,6), (6,1), (6,2), (6,3), (6,4), (6,5), (6,6)}
A coin is tossed	{H, T}, H stands for a head turning up and T for a tail turning up.
Two coins are tossed	{HH, HT, TH, TT}
A coin is tossed and a dice is thrown	{(H, 1), (H, 2), (H, 3), (H, 4), (H, 5), (H, 6), (T, 1), (T, 2), (T, 3), (T, 4), (T, 5), (T, 6)}

The probability of happening of an event may be defined as the ratio of the number of favourable cases to the total number of cases. If the number of favourable cases is n and that of unfavourable cases is m, then the

Probability of success is $p = \dfrac{n}{n+m}$

probability of failure is $q = \dfrac{m}{m+n}$

$\therefore\ p+q = \dfrac{n+m}{n+m} = 1.$

The probability of an event lies between 0 and 1.

A *certain event* is one in which each element of a sample space is a favourable event. In this case, the probability of success is 1.

Tip: The probability of an impossible event is 0 that of a sure event is 1 and in all other cases, it lies between 0 and 1.

An *impossible event* is one in which the sample space is an empty set. In this case, the probability of success, p is 0.

If there are two events A and B having probability $P(A)$ and $P(B)$, then

$$P(A \cup B) = P(A) + P(B) - P(A \cap B)$$

where $P(A \cup B)$ is the probability of A union B and $P(A \cap B)$ is the probability of A intersection B.

In the case of mutually exclusive events, $P(A \cap B) = 0$, so

$$P(A \cup B) = P(A) + P(B)$$

The following illustrations exemplify the above concepts.

Illustration A6.1 Two cards are drawn from a well-shuffled pack of 52 cards. What is the probability that the two cards belong to the same suite?

Solution In a well-shuffled pack of 52 cards, there are four suites namely spade, diamond, club, and heart. There are 13 cards in each suite. Now, as per the question, the two cards can either be drawn from the first or the second or the third or the fourth suite.

Thus, $^{13}C_2$ combinations can crop us in all the four cases. This makes the total number of favourable cases as $4 \times {}^{13}C_2$. Here, the total number of cases is $^{52}C_2$. The probability of success of the said event is, therefore, $4 \times {}^{13}C_2 / {}^{52}C_2$.

Illustration A6.2 There are 5 boys and 7 girls in a group. Find the probability of a subgroup of four be formed with at least one girl.

Solution The subgroup must have at least one girl. The number of girls can, therefore, be 1, 2, 3, or 4. The corresponding number of cases are as follows:

Number of ways in which a group can be formed having a single girl $= {}^5C_3 \times {}^7C_1$ (3 boys and 1 girl)

Number of ways in which a group can be formed having two girls $= {}^5C_2 \times {}^7C_2$ (2 boys and 2 girls)

Number of ways in which a group can be formed having three girls $= {}^5C_1 \times {}^7C_3$ (1 boy and 3 girls)

Number of ways in which a group can be formed having four girls $= {}^5C_0 \times {}^7C_4$ (0 boy and 4 girls)

The total number of favourable cases $= {}^5C_3 \times {}^7C_1 + {}^5C_2 \times {}^7C_2 + {}^5C_1 \times {}^7C_3 + {}^5C_0 \times {}^7C_4$

The total number of cases $= {}^{12}C_4.$

The required probability $= ({}^5C_3 \times {}^7C_1 + {}^5C_2 \times {}^7C_2 + {}^5C_1 \times {}^7C_3 + {}^5C_0 \times {}^7C_4) / {}^{12}C_4$.

Illustration A6.3 A pair of dice is thrown. Find the probability that the sum of numbers appearing is a multiple of three.

Solution The total number of cases is $6 \times 6 = 36$.
The favourable cases are
(1, 2), (2, 1), (1, 5), (5, 1), (2, 4), (4, 2), (3, 3), (5, 4), (4, 5), (3, 6), (6, 3), (6, 6).
The number of favourable cases is 12.
Therefore, the required probability is $= 12/36 = 1/3$.

Illustration A6.4 A bag contains 5 red and 3 black balls. Two balls are drawn at random. What is the probability of both being red?

Solution Two red balls can be drawn in $^5C_2$ ways.

Two balls can be drawn in $^8C_2$ ways.

Therefore, the probability of success is $^5C_2/^8C_2$.

Illustration A6.5 In the above illustration, what is the probability of drawing a red and a black ball?

Solution A red and a black ball can be drawn in $^5C_1 \times ^3C_1$ ways.

Two balls can be drawn in $^8C_2$ ways.

Therefore, the probability of success is $^5C_1 \times ^3C_1/^8C_2$.

Illustration A6.6 What is the probability of having 52, 53, and 54 Sundays in a leap year?

Solution A leap year has 366 days. The number of weeks is 52. So, there would be 52 Sundays. The remaining two days can be {Sunday Monday, Monday Tuesday, Tuesday Wednesday, Wednesday Thursday, Thursday Friday, Friday Saturday, Saturday Sunday}.

So, out of seven possibilities two have Sundays. The probability of having 52 Sundays in a leap year is 1, as it is a sure event.

The probability of having 53 Sundays is 2/7 (as explained above).

And finally, the probability of having 54 Sundays in a leap year is 0 (as it is an impossible event).

A6.2.2 Pigeonhole Principle

The pigeonhole principle is one of the most important principles of counting. The principle has been stated, proved, and exemplified in the following discussion.

Statement

If n pigeons fly into K pigeonholes, $K < n$ as there are more than one pigeon in at least one of the pigeonhole.

Proof We can prove the above theorem by contradiction. If the given statement is false, then we can account for K pigeonholes (there are n pigeons) but $n > K$ so rest of the pigeons cannot be accounted for. Therefore, there are more than one pigeon in at least one pigeonhole.

The principle is helpful in solving many problems and is also used in proving the pumping lemma in the theory of automata. However, the above principle does not determine which pigeonhole has more than one pigeon. Figures A6.1(a) and (b) depicts the above with the help of an example.

Figure A6.1(a) Shown are 5 pigeons, 4 pigeonholes, so there is at least 1 pigeonhole with more than one pigeon
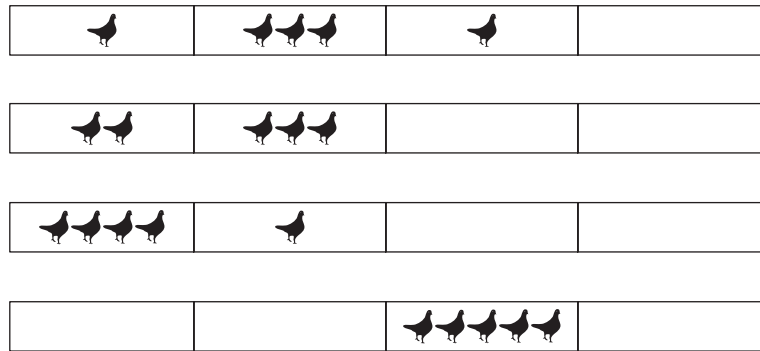
Figure A6.1(b)   Shown are some of the possible cases so as to how 5 pigeons can fit into 4 pigeonholes. Note that there is at least one pigeonhole in each case where number of pigeons > 1

The applications of the above principle also include proving in functions and relations. For instance, let there be a function *f* from X to Y and number of elements in Y is less than that in X, then there are at least two elements in X that map to some value in Y. Figure A6.2 depicts the concept.
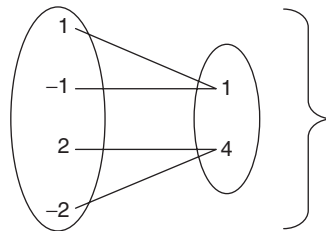

Figure A6.2   Many to one mapping in the case number of elements in Y < number of elements in X

Illustration A6.7   Show that if we select 201 distinct CS courses numbered between 1 and 200 (both inclusive) at least two are consecutively numbered.

Solution Let the selected courses be,

$$C_1, C_2, \ldots, C_{201}$$

Now the first 200 courses can be numbered distinctly but the 201th course will have number from 1 to 200 only (there are no other numbers). Same is the case for the rest of the courses. Problems 1–8 in the Exercises section can be solved using the pigeonhole principle.

A6.3 INDEPENDENT EVENTS

Two events are said to be independent if the occurrence or non-occurrence of one does not affect the probability of the occurrence of the other. Mathematically, in the case of independent events, the probability of intersection of two events is the product of

probability of the two. In the case of independent events, the conditional probability of A, provided that B has occurred becomes $P(A)$. Similarly, the conditional probability of B provided that A has occurred becomes $P(B)$,

$$P(A \cap B) = P(A)\, P(B)$$

i.e.,
$$P(A/B) = P(A)$$

and
$$P(B/A) = P(B)$$

Since $P(A/B) = \dfrac{P(A \cap B)}{P(B)} = \dfrac{P(A)P(B)}{P(B)} = P(A)$

and, $P(B/A) = \dfrac{P(B \cap A)}{P(A)} = \dfrac{P(B)P(A)}{P(A)} = P(B)$

If $A_1, A_2, \ldots, A_n$ are independent events, then

$$P(A_1 \cap A_2 \cap, \ldots, \cap A_n = P(A_1)P(A_2), \ldots, P(A_n)$$

Illustrations A6.8 through A6.12 exemplify the above concept.

Illustration A6.8 Events A and B are such that $P(A) = 1/2$ and $P(B) = 7/12$. In addition, given that $P\left(\overline{A} \text{ or } \overline{B}\right) = 1/4$. State whether A and B are independent or not.

Solution

$$\begin{aligned}
P\left(\overline{A} \text{ or } \overline{B}\right) &= P\left(\overline{A \cap B}\right)\ \left(\text{since } \overline{A} \cup \overline{B} = \overline{A \cap B}\right)\\
&= 1 - P\left(A \cap B\right)\\
&= 1 - P(A)\,P(B)\ \text{(since A and B are independent)}\\
&= 1 - (1/2 \times 7/12)\\
&= 1 - 7/24\\
&= 17/24 \text{ but } P\left(\overline{A} \text{ or } \overline{B}\right) \text{ is } 1/4.
\end{aligned}$$

Therefore, A and B are not independent events.

Illustration A6.9 If A and B are independent events and $P(A) = 1/4$, $P(B) = 1/2$, $P(A \cap B) = 1/8$, find $P(\text{not A and not B})$.

Solution

$$\begin{aligned}
P\left(\overline{A} \text{ and } \overline{B}\right) &= P\left(\overline{A} \cap \overline{B}\right)\\
&= P\left(\overline{A \cup B}\right)\ \text{(by De Morgan's Law)}\\
&= 1 - P(A \cup B)\\
&= 1 - [P(A) + P(B) - P(A \cap B)]\\
&= 1 - [P(A) + P(B) - P(A)\,P(B)]
\end{aligned}$$

$$=1-(1/4+1/2-1/8)$$
$$=1-(3/4-1/8)$$
$$=1-5/8$$
$$=3/8$$

Illustration A6.10 A bag contains 5 white, 7 red, and 4 black balls. If 4 balls are drawn at random, one by one, with replacement, what is the probability that none is white?

Solution

P(not getting white ball in first trial) $= 11/16$

P(not getting white ball in second trial) $= 11/16$

Similarly,

P(not getting white ball in third and fourth trials) $= 11/16$

Therefore, $P(\text{I}) = P(\text{II}) = P(\text{III}) = P(\text{IV})$

Since the events are independent

$$\therefore\ P = \left(\frac{11}{16}\right)^4$$

Illustration A6.11 A can solve 90% of problems and B can solve 70%. What is the probability that at least one of them will solve the problem, selected at random?

Solution There are three possible cases:
(a) Either both solve the problem

$$P = P(A)\,P(B)$$
$$= \left(\frac{90}{100} \times \frac{70}{100}\right) = \frac{63}{100}$$

(b) A solves the problem that B does not

$$P' = P(A)\,P(\overline{B})$$
$$= \left(\frac{90}{100} \times \frac{30}{100}\right) = \frac{27}{100}$$

(c) B solves the problem but A does not

$$P' = P(\overline{A})\,P(B)$$
$$= \left(\frac{10}{100} \times \frac{70}{100}\right) = \frac{7}{100}$$

Therefore, the required probability $= \dfrac{63+27+7}{100} = \dfrac{97}{100} = 0.97$

Illustration A6.12 In Illustration A6.11, what is the probability that the problem would be solved?

Solution The answer can be found by subtracting $P(\overline{A}) \times P(\overline{B})$ from 1, i.e., $[1 - P(A) P(B)]$

$$= 1 - P\left(\overline{A}\right) P\left(\overline{B}\right)$$

$$= 1 - \left(1 - \frac{90}{100}\right)\left(1 - \frac{70}{100}\right)$$

$$= 1 - \left(\frac{10}{100} \times \frac{30}{100}\right)$$

$$= 1 - 3/100$$

$$= \frac{97}{100}$$

A6.3.1 Bay's theorem

It is perhaps one of the most important theorems in probability that helps us to find the conditional probability in special cases. Let us understand the meaning of conditional probability before starting off with the topic.

$P(A/B)$ is the probability of occurrence of A provided that B has already occurred. This is called conditional probability. In order to understand the concept, consider the following illustration. Suppose there are two bags A and B. A contains 5 red and 3 black balls and B contains 3 red and 5 black balls, then the probability of drawing a red ball provided the first bag is selected is $P(R/I)$ which is 5/8. Similarly, the probability of drawing a red ball provided the second bag is selected is $P(R/II)$ which is 3/8. So what happens if a ball has been drawn from some bag and its colour is noticed and we have been asked to find the probability from which the ball is drawn being the first one.

In such situations, Bay's theorem comes to our rescue. The statement of Bay's theorem is as follows.

Bay's Theorem

Let S be the sample space and E_1, E_2, ..., E_n be n mutually exclusive events associated with random experiment. If A is the event that occurs with E_1 or E_2 or E_n, then

$$P(E_i/A) = \frac{P(E_i)P(A/E_i)}{\sum_{j=1}^{n} P\left(E_j\right) P\left(A/E_j\right)}$$

The illustration that follows exemplifies the concept.

Illustration A6.13 In a bolt manufacturing factory, machines A, B, and C produce 25%, 35%, and 40%, respectively of total bolts. Of them 5%, 4%, and 2%, respectively, are defective. A bolt is drawn at random from the product. If bolt drawn is found to be defective, what is the probability that it was produced by machine B?

Solution

P(Bolt produced by machine A) = P(A) = 25/100

P(Bolt produced by machine B) = P(B) = 35/100

P(Bolt produced by machine C) = P(C) = 40/100

P(Bolt is defective given that it was produced by A) = P(defective/A) = 5/100

P(Bolt is defective given that it was produced by B) = 4/100

P(Bolt is defective given that it was produced by C) = 2/100

We are required to find out,

P(Bolt is produced by A, given that it is defective) P(A/defective)

Applying Bay's theorem, we get

$$P(\text{A/defective}) = \frac{P(\text{defective/A})P(\text{A})}{P(\text{defective/A})P(\text{A}) + P(\text{defective/B})P(\text{B}) + P(\text{defective/C})P(\text{C})}$$

$$= \frac{\dfrac{1}{3} \times \dfrac{6}{10}}{\left(\dfrac{1}{3} \times \dfrac{6}{10}\right) + \left(\dfrac{1}{3} \times \dfrac{4}{10}\right) + \left(\dfrac{1}{3} \times \dfrac{5}{10}\right)}$$

$$= 2/5$$

Illustration A6.14 An urn A contains 2 white, 1 black, and 3 red balls. The urn B contains 3 white, 2 black, and 4 red balls; C contains 4 white, 3 black, and 2 red balls. An urn is chosen at random, and ball is drawn if it is red, find probability that it was drawn from B.

Solution

P(red ball from A) = 3/6 = P(R/A)

P(red ball from B) = 4/9 = P(R/B)

P(red ball from C) = 2/9 = P(R/C)

Now, since nothing is given about the probabilities of selection of bags A, B, and C.

Let, P(A) = P(B) = P(C) = 1/3

We have to find P(bag A given that we selected a red ball)

$$P(\text{A/R}) = \frac{P(\text{R/A})\,P(\text{A})}{P(\text{R/A})\,P(\text{A}) + P(\text{R/B})\,P(\text{B}) + P(\text{R/C})\,P(\text{C})}$$

$$= \frac{\dfrac{3}{6} \times \dfrac{1}{3}}{\left(\dfrac{3}{6} \times \dfrac{1}{3}\right) + \left(\dfrac{4}{9} \times \dfrac{1}{3}\right) + \left(\dfrac{2}{9} \times \dfrac{1}{3}\right)}$$

$$= \frac{\dfrac{3}{6}}{\dfrac{3}{6} + \dfrac{4}{9} + \dfrac{2}{9}}$$

$$= \frac{\dfrac{3}{6}}{\dfrac{7}{6}} = 3/7$$

Illustration A6.15 In the above question, what will be the probability (if two balls are drawn instead of 1) of selecting a red and a black ball.

Solution

$P(\text{red or black}/A) = (^3C_1 \times {}^1C_1)/{}^6C_2$

i.e., $P(\text{red or black}/A) = 1/5$

$P(\text{red or black}/B) = 2/9$

$P(\text{red or black}/C) = 1/6$

Therefore, $P(A/\text{red or black ball is drawn})$

$$= \frac{\dfrac{1}{3} \times \dfrac{1}{5}}{\left(\dfrac{1}{3} \times \dfrac{1}{5}\right) + \left(\dfrac{1}{3} \times \dfrac{2}{9}\right) + \left(\dfrac{1}{3} \times \dfrac{1}{6}\right)}$$

$$= \frac{\dfrac{1}{15}}{\dfrac{1}{15} + \dfrac{2}{27} + \dfrac{1}{18}}$$

$$= \frac{6 \times 3}{35} = \frac{18}{35}$$

Illustration A6.16 A man is known to speak truth in 3 out of 4 times. He throws a dice and reports that it is a six. Find probability that it is actually a six.

Solution

$$P(\text{reports six/six occurred}) = 3/4$$

$$P(\text{reports six}/\overline{\text{six occurred}}) = 1/4$$

$$P(\text{six occurred}) = 1/6$$

$$P(\overline{\text{six occurred}}) = 1 - 1/6 = 5/6$$

$$P(\text{six occurred/reports six}) = \cfrac{\begin{array}{c}P(\text{six occurred})\\P(\text{reports six/six occurred})\end{array}}{\begin{array}{c}P(\text{six occurred})P(\text{reports six/six occurred}) +\\P(\text{six does not occurred})P(\text{six occurred})\\P(\text{reports six/six does not occurred})\end{array}}$$

$$= \cfrac{\dfrac{1}{6} \times \dfrac{3}{4}}{\left(\dfrac{1}{6} \times \dfrac{3}{4}\right) + \left(\dfrac{5}{6} \times \dfrac{1}{4}\right)}$$

$$= 3/8$$

Illustration A6.17 In the previous question, find the probability that if person has not reported six, then six occurred.

Solution Please try yourself.

Illustration A6.18 An insurance company insured 3000 scooters, 4000 cars, and 5000 trucks. The probability that scooter meets an accident is 0.002, for car it is 0.003, and for truck it is 0.004. If one of the insured vehicles meets with an accident, what is the probability that it is a scooter?

Solution
Let the scooter is insured = S.
So, $P(\text{Scooter}) = P(S) = 3000/12000 = 1/4$
$P(\text{Car}) = P(C) = 4000/12000 = 1/3$
$P(\text{Truck}) = P(T) = 5000/12000 = 5/12$
$P(\text{accident}/S) = 0.002$
$P(\text{accident}/C) = 0.003$
$P(\text{accident}/T) = 0.004$
Now, we have to find $P(\text{scooter/accident})$

$$\text{i.e., } P(\text{S/accident}) = \frac{P(\text{accident/S}) \, P(S)}{P(\text{accident/S}) + P(\text{accident/C}) + P(\text{accident/T})P(T)}$$

By putting the values, we get 3/9.

Illustration A6.19 In the previous illustration, find the probability that the vehicle is a car.

Solution Please try yourself.

Illustration A6.20 In a question set, there are 40 questions of Algorithms (A), 30 of theory of computation (TOC) and 30 of artificial intelligence (AI). The probability that an average student solves a problem of algorithms is 0.1, that of TOC is 0.2 and that of

AI is 0.2. A student is able to solve a problem, find the probability that the question is of algorithms.

Solution The probability of solving a question of algorithms is

$P(\text{Sol/A}) = 0.1$

$P(\text{Sol/AI}) = 0.2$

$P(\text{Sol/TOC}) = 0.2$

$P(A) = 0.4$

$P(\text{AI}) = 0.3$

$P(\text{TOC}) = 0.3$

We have to find the probability of a question of algorithm provided that the person is able to solve it, that is $P(\text{A/Sol})$.

Applying, Bay's theorem

$P(\text{A/Sol}) = (P(\text{Sol/A}) \times P(A))/(P(\text{Sol/A}) \times P(A) + P(\text{Sol/TOC}) \times P(\text{TOC}) + P(\text{Sol/AI}) \times P(\text{AI}))$

$\qquad = 0.25$

A6.4 PROBABILITY DISTRIBUTION

If a random variable X takes values $x_1, x_2, \ldots, x_n$ with probabilities $P_1, P_2, \ldots, P_n$ then

X:　　$x_1, x_2, x_3, \ldots, x_n$

$P(X)$: $P_1, P_2, P_3, \ldots, P_n$

is known as probability distribution of X.

　　Thus, a tabular description of random variables along with corresponding probabilities is called probability distribution. The probability distribution of random variable X is defined only when we have probabilities P_i^s satisfying

$$\sum P_i = 1$$

i.e.,　　　　　　　　$P_1 + P_2 + P_3 +, \ldots, + P_n = 1$

The following illustrations (21–30) exemplify the above concept.

Illustration A6.21　Is the following distribution a probability distribution?

X:	0	1	2	3
$P(x)$:	0.3	0.1	0.1	0.2

Solution Since in the case of a probability distribution, the sum of the probabilities is unity, it is imperative to check the sum of probabilities. If the sum comes out to be one, then the distribution is a probability distribution, otherwise it is not.

$$\text{The sum of probabilities} = P(0) + P(1) + P(2) + P(3)$$

$$= 0.3 + 0.1 + 0.2 + 0.2$$

$$= 0.8 \neq 1$$

Therefore, it is not a probability distribution.

Illustration A6.22 A random variable X has the following probability distribution:

X:	0	1	2	3	4	5	6	7
$P(x)$:	0	K	2K	2K	3K	K2	2K2	7K2 + K

Find (a) the value of K
   (b) $P(X \geq 6)$
   (c) $P(X < 6)$
   (d) $P(0 < X < 5)$

Solution

(a) Since the sum of probabilities of P_i's must be 1.

   Therefore, $\quad 0 + K + 2K + 2K + 3K + K^2 + 2K^2 + 7K^2 + K$

   i.e., $\quad P(0) + P(1) + P(2) + P(3) + P(4) + P(5) + P(6) + P(7) = 1$

   i.e., $\qquad\qquad\qquad 10K^2 - 9K - 1 = 0$

   $$K = -1 \text{ or } 1/10$$

   Since $P(1) = K$, and the probability cannot be negative.

   Therefore, $K = 1/10$.

(b) $P(X \geq 6) = P(6) + P(7)$

   $\qquad\qquad = 2K^2 + 7K^2 + K$

   $\qquad\qquad = 9K^2 + K$

   $\qquad\qquad = 19/100$

(c) $P(X < 6) = 1 - P(X \geq 6)$

   $\qquad\qquad = 1 - 19/100$

   $\qquad\qquad = 81/100$

(d) $P(0 < X < 5) = P(1) + P(2) + P(3) + P(4)$

   $\qquad\qquad\qquad = K + 2K + 2K + 3K$

   $\qquad\qquad\qquad = 8K$

   $\qquad\qquad\qquad = 8/10 = 4/5$

Illustration A6.23 From a lot of 7 containing 3 defective items, a sample of 4 is drawn at random. Let X denotes the number of defective items in the sample. If sample is drawn without replacement, then find
(a) Probability distribution of X
(b) $P(X \leq 1)$
(c) $P(X < 1)$
(d) $P(0 < X < 2)$

Solution

(a) Probability of no defective item

$$= {}^7C_4/{}^{10}C_4 = 1/6$$

   $P(X = 1)$, i.e., probability of one defective item $= ({}^3C_1 \times {}^7C_3)/{}^{10}C_4 = 1/2$

X:	0	1	2	3
P(x):	$\dfrac{4324}{5525}$	$\dfrac{1128}{5525}$	$\dfrac{72}{5525}$	$\dfrac{1}{5525}$

Illustration A6.27 An urn contains 4 white and 3 red balls. Find the probability distribution of the number of red balls in a random draw of 3 balls.

Solution The number of red balls can be 0, 1, 2, or 3. The corresponding probabilities are as follows:

$P(X = 0) = {}^4C_3/{}^7C_3 = 4/35$

$P(X = 1) = ({}^7C_1 \times {}^4C_2)/{}^7C_3 = 18/35$

$P(X = 2) = ({}^3C_2 \times {}^4C_1)/{}^7C_3 = 12/35$

$P(X = 3) = {}^3C_3/{}^7C_3 = 1/35$

Therefore, the probability distribution is as follows:

X:	0	1	2	3
P(X):	$\dfrac{4}{35}$	$\dfrac{18}{35}$	$\dfrac{12}{35}$	$\dfrac{1}{35}$

A6.4.1 Mean and Variance of a Probability Distribution

If X is a discrete random variable that has values x_1, x_2, ..., x_n with probabilities P_1, P_2, ..., P_n, then the mean of X is defined as follows:

Mean $= (P_1x_1 + P_2x_2 +, ..., + P_nx_n)/(P_1 + P_2 +, ..., + P_n)$

In the case of a probability distribution, $\sum P_i = 1$

Therefore, $\overline{X} = \sum P_i X_i$.

The variance of probability distribution

x	x_1	x_2	...	x_n
P	P_1	P_2	...	P_n

is $\sum P_i x_i^2 - \left(\sum P_i x_i \right)^2$.

Therefore, the root of variance is referred to as the standard deviation.

The following illustrations explore the concept of mean and variance of a probability distribution.

Illustration A6.28 A dealer of TV from his past experience estimates the probability of selling of his TV in a day. The probabilities are

x_i:	0	1	2	3	4	5	6
P_i:	0.03	0.2	0.23	0.25	0.12	0.10	0.07

Find the mean

(where $x = 0$, is a Monday and so on)

Solution

$$\text{Mean} = \sum P_i x_i$$
$$= (0 \times 0.3) + (1 \times 0.2) + (2 \times 0.23) + (3 \times 0.25) + (4 \times 0.12) + (5 \times 0.10) + (6 \times 0.07)$$
$$= 2.75$$

Illustration A6.29 Find variance of number of heads in two tosses of a coin.

Solution We made a table for different probabilities when we get different number of heads

x	$P(x)$
0	TT = 1/4
1	TH + HT = 1/2
2	HH = 1/4,

Now we extend the table and find $P_i x_i$ and $P_i x_i^2$.

x_i	P_i	$P_i x_i$	$P_i x_i^2$
0	1/4	0	0
1	1/2	1/2	1/2
2	1/4	1/2	1
		$\sum P_i x_i = 1$	$\sum P_i x_i^2 = 3/2$

Since variance $= \sum P_i x_i^2 - \left(\sum P_i x_i\right)^2$
$$= 3/2 - (1)^2$$
$$= 3/2 - 1$$
$$= 1/2$$

Illustration A6.30 Two dice are thrown. If X denotes the number of sixes, then find the expectation of X.

Solution Clearly, X can take values 0, 1, 2.

$P(0) = q \times q = 5/6 \times 5/6 = 25/36$

$P(1) = Pq + qP = 10/36$

$P(2) = PP = 1/36$

Therefore, distribution is

X:	0	1	2
P(X):	$\dfrac{25}{36}$	$\dfrac{10}{36}$	$\dfrac{1}{36}$

x_i	P_i	$P_i x_i$	$P_i x_i^2$
0	25/36	0	0
1	10/36	10/36	10/36
2	1/36	2/36	4/36

$$\sum P_i x_i = 12/36 \quad \sum P_i x_i^2 = 14/36$$

$$E(X) = \sum P_i x_i = 1/3$$

A6.5 BINOMIAL DISTRIBUTION

Trials of a random experiment are called Bernoulli's trials if they satisfy the following conditions:

(a) They are finite in number.
(b) They are independent of each other.
(c) Each trial results in either success or failure and the sum of probabilities of success and failure is 1.

The probability of rth success in a binomial distribution with number of trial $= n$, probability of success $= p$, and the probability of failure $= q$ is

$$P(X = r) = {}^nC_r P^r q^{n-r}$$

Illustration A6.31 A dice is thrown 4 times. Find the probability distribution of number of heads.

Solution If we throw a coin, then we may get a head or a tail,

Therefore, $\qquad\qquad\qquad P(\text{head}) = 1/2$

$$P(\text{tail}) = 1 - P = 1/2$$

Now if we throw it 4 times, then we may get head once, twice, thrice, 4 times, or not even once. Let X denote the number of heads then,

$$X = 0,\ qqqq = P^0 q^4$$
$$X = 1,\ Pqqq + qPqq + qqPq + qqqP = 4Pq^3$$
$$X = 2,\ PPqq + qqPP + PqqP + qPPq + PqPq + qPqP = 6P^2 q^2$$
$$X = 3,\ PPPq + PPqP + PqPP + qPPP = P^3 q$$
$$X = 4,\ PPPP = P4$$

We can generalize the above result as

$$P(X = 0) = {}^nC_0\,P^0q^n;\ n = 4$$
$$P(X = 1) = {}^nC_1\,P^1q^{n-1}$$
$$P(X = 2) = {}^nC_2\,P^2q^{n-2}$$
$$P(X = 3) = {}^nC_3\,P^3q^{n-3}$$

and, $$P(X = 4) = {}^nC_4\,P^4q^{n-4}$$

So, probability of rth success becomes

$$P(X = r) = {}^nC_r P^r q^{n-r}$$

The mean of the above distribution is nP and variance is nPq.

Hence, if an event occurs n times such that

(a) n is finite

(b) $p + q = 1$

Then, $P(X = r) = {}^nC_r P^r q^{n-r}$, mean $= nP$, variance $= nPq$.

Illustration A6.32 A coin is tossed five times. What is the probability of getting at least three heads?

Solution The value of $n = 5$, as the experiment is repeated five times. Moreover, the probability of getting a head on throwing a coin is 1/2, so is the probability of not getting a head. So, $p = 1/2$ and $q = 1/2$.

Since, $P(r\text{th success}) = {}^nC_r p^r q^{n-r}$

(P) Probability of getting a head $= 1/2$,

(q) Probability of not getting a head $= (1 - P) = 1/2 = 1/2$

P (at least three successes)

$= P(r = 3) + P(r = 4) + P(r = 5)$

$= {}^5C_3(1/2)^3(1/2)^2 + {}^5C_4(1/2)^4(1/2) + {}^5C_5(1/2)^5$

$= 1/2$

Illustration A6.33 A pair of dice is tossed six times. If getting a sum of 9 is considered as a success, what is the probability of getting the sum of 9 five times in the six throws?

Solution In this case, the value of $n = 6$, as the experiment is repeated 6 times.

Since, $P(r\text{th success}) = {}^nC_r P^r q^{n-r}$.

$P \rightarrow$ Probability of getting 9 as a sum in the throw of a pair of dice.

The number of favourable cases are 4, as the pairs (3, 6), (6, 3), (5, 4), (4, 5) give sum 9.

Total number cases $= 6 \times 6 = 36$

$P = 4/36 = 1/9$

$q = 1 - 1/9 = 8/9$

$r = 5$

$P(r = 5) = {}^6C_5\,(1/9)^5\,(8/9)^1$

$= 6 \times 8/9^6 = 48/9^6$

Illustration A6.34 Find the probability of 4 turning up at least once in 2 tosses of a fair dice.

Solution The dice has been tossed two times, therefore, $n = 2$.
$P = $ Probability of getting a 4 in the throw of a dice $= 1/6$
$q = 5/6$
$P(r\text{th success}) = {}^nC_r p^r q^{n-r}$
Therefore, probability $= P(1\text{th}) + P(2\text{th})$
$$= {}^2C_1(1/6)^1(5/6)^1 + {}^2C_2(1/6)^2(5/6)^0$$
$$= 2(1/6)(5/6) + (1/6)(1/6)$$
$$= 11/36$$

Illustration A6.35 A coin is tossed 5 times. What is the probability that head appears an even number of times?

Solution Since the coin has been tossed five times, the value of $n = 5$. Since in a throw of a coin either a head or a tail appears:
$P \rightarrow$ Probability of getting head $\rightarrow 1/2$,
$q = 1 - P = 1/2$.
We need the head appearing even number of times.
Therefore, the required probability $= P(2\text{th}) + P(4\text{th})$
$$= {}^5C_2(1/2)^2(1/2)^3 + {}^5C_4(1/2)^4(1/2)^1$$
$$= (10/2^5) + (5/2^5) = 15/32$$

Illustration A6.36 The probability of a man hitting of target is 1/4. If he fires 7 times, what is the probability of his hitting the target at least twice?

Solution The man hits the target 7 times. Therefore, $n = 7$
$P = 1/4$ (given),
$q = 1 - P = 3/4$
Since, $P(r\text{th success}) = {}^nC_r P^r q^{n-r}$
As per the question, the target must hit 7 times; therefore, the required probability
$= P(r = 2) + P(r = 3) + P(r = 4) + P(r = 5) + P(r = 6) + P(r = 7)$
$= {}^7C_2(1/4)^2(3/4)^5 + {}^7C_3(1/4)^3(3/4)^4 + {}^7C_4(1/4)^4(3/4)^3 + {}^7C_5(1/4)^5(3/4)^2$
$\qquad + {}^7C_6(1/4)^6(3/4)^1 + {}^7C_7(1/4)^7(3/4)^0$
$= (21)(3^5/4^7) + (35)(3^4/4^7) + (35)(3^3/4^7) + (21)(3^2/4^7) + (7)(3/4^7) + (1/4^7)$
$= 4547/8192$

Illustration A6.37 Eight coins are thrown simultaneously. Find the chance of obtaining at least six heads.

Solution
Since eight coins have been thrown, the value of $n = 8$.

The probability of getting a head, $P = 1/2$

$q = 1 - P = 1/2$

Since, $P(r\text{th success}) = {}^nC_rP^rq^{n-r}$; therefore,

the required probability $= P(r = 6) + P(r = 7) + P(r = 8)$

$= {}^8C_6(1/2)^6(1/2)^2 + {}^8C_7(1/2)^7(1/2) + {}^8C_8(1/2)^8(1/2)^0$

$= 1/2^8[{}^8C_6 + {}^8C_7 + {}^8C_8]$

$= [28 + 8 + 1]/2^8$

$= 37/256$

Illustration A6.38 Three cards are drawn successively with replacement from a well-shuffled pack of 52 cards. What is probability that

(i)   All the three cards are spade

(ii) Only two cards are spade

(iii) None is a spade

Solution

(i) $n = 5$, as card is drawn five times with replacement

      The probability of getting a spade, $P = 13/52 = 1/4$

(q) Probability of not getting a spade $q = 1 - P = 3/4$

      As per the question, $r = 5$

      Since, $P(r\text{th success}) = {}^nC_rP^rq^{n-r}$; therefore, the required probability $= P(r = 3)$

$$= {}^5C_3(1/4)^3(3/4)^2$$

$$= 45/512$$

(ii) $n = 5$

      $P = 1/4$

      $q = 1 - P = 3/4$

      Since, $P(r\text{th success}) = {}^nC_rP^rq^{n-r}$; therefore, the required probability $= P(r = 2)$

$$= {}^5C_3(1/4)^2(3/4)^1$$

$$= 15/32$$

(iii) $n = 5$

      $P = 1/4$

      $q = 1 - P = 3/4$

      Since, $P(r\text{th success}) = {}^nC_rP^rq^{n-r}$; in this case, $r = 0$; therefore, the required probability $= {}^3C_0(1/4)^0(3/4)^3$

$$= 27/64$$

Illustration A6.39 A bag contains 7 red, 5 white, and 8 black balls of 4 balls are drawn one by one with replacement. What is the probability that

(i)   None is white

(ii) All are white

(iii) Any two are white

Solution

(i) $n = 4$, as experiment is done 4 times

Probability of coming white ball = (number of white balls/total number of balls)

$= 5/20 = 1/4$

$P = 1/4, q = 3/4$

$P(r\text{th success}) = {}^nC_rP^rq^{n-r}$

$\qquad = {}^4C_0(1/4)^0(3/4)^4 = 81/256$

(ii) $r = 4$

$P = 1/4, q = 3/4$

$P(r\text{th success}) = {}^nC_rP^rq^{n-r}$

$\qquad = {}^4C_4\ (1/4)^4 = 1/256$

(iii) $r = 1$

Therefore, $P = {}^nC_1\ (1/4)\ (3/4)^3$

$\qquad = ({}^4C_1) \times (1/4) \times (27/64)$

$\qquad = 4 \times (27/256) = 27/128$

Illustration A6.40   The frequency and values of x are given in the following table. Fit a binomial distribution to the following:

$x \rightarrow$	0	1	2	3	4	5	6	7	8	9	10
$f \rightarrow$	6	20	28	12	8	6	0	0	0	0	0

Solution

$x \rightarrow$	0	1	2	3	4	5	6	7	8	9	10	
$f \rightarrow$	6	20	28	12	8	6	0	0	0	0	0	$\Sigma f = 80$
$f_x \rightarrow$	0	20	56	36	32	30	0	0	0	0	0	$\Sigma f_x = 174$

$$\text{Mean} = \left(\Sigma fx / \Sigma f \right) \quad = 174/80 = 20175$$

However, in a binomial distribution, mean is np,

Therefore, $np = 2.175$

n is 10

$P = 0.2175$

$q = 1 - 0.2175$

$n = 10$

The total frequency, $N = 80$

Probability of 0th success $= N\ {}^nC_rP^rq^{n-r}$

$\qquad = 80\ {}^{10}C_0\ (0.2175)^0\ q^{10}$

$\qquad = 80\ (0.2175)^0\ (0.7825)^{10}$

Similarly,

r	$P(r)$
0	$^{10}C_0\,80 \times (0.2175)^0\,(0.7825)^{10}$
1	$^{10}C_1\,80 \times (0.2175)^1\,(0.7825)^9$
2	$^{10}C_2\,80 \times (0.2175)^2\,(0.7825)^8$
3	$^{10}C_3\,80 \times (0.2175)^3\,(0.7825)^7$
4	$^{10}C_4\,80 \times (0.2175)^4\,(0.7825)^6$
5	$^{10}C_5\,80 \times (0.2175)^5\,(0.7825)^5$
6	$^{10}C_6\,80 \times (0.2175)^6\,(0.7825)^4$
7	$^{10}C_7\,80 \times (0.2175)^7\,(0.7825)^3$
8	$^{10}C_8\,80 \times (0.2175)^8\,(0.7825)^2$
9	$^{10}C_9\,80 \times (0.2175)^9\,(0.7825)^1$
10	$^{10}C_{10}\,80 \times (0.2175)^{10}$

A6.5.1 Recurrence Formula for Binomial Distribution

Since, $$P(r) = {}^nC_r P^r q^{n-r}$$

$$= \frac{n!}{r!(n-r)!}\left(q^{n-r}p^r\right)$$

And, $$P(r+1) = \frac{n!}{(n-r-1)!(r+1)!}\left(p^{r+1}q^{n-r-1}\right)$$

Therefore, $$\frac{P(r+1)}{P(r)}\frac{n!}{(n-r-1)!}\frac{r!}{(r+1)!}\frac{p}{q}$$

$$= \left(\frac{n-r}{r+1}\right)\left(\frac{p}{q}\right)$$

Therefore, $$P(r+1) = \left(\frac{n-r}{r+1}\right)\left(\frac{p}{q}\right)e(r)$$

Mean and variance of binomial distribution

$$\text{Mean} = \sum_{r=0}^{n} rP(r)$$

$$\sum_{r=0}^{n} r\,{}^nC_r\,P^r q^{n-r}$$

$$= 0 + (1.\,{}^nC_1 q^{n-1}\,P) + (2.\,{}^nC_2 q^{n-2}\,P^2) +, \ldots, + n\,{}^nC_n P^n$$

Putting value ${}^nC_r = \dfrac{n!}{(n-r)r!}$

we get,

$$= nP\left[q^{n-1} + (n-1)q^{n-2}P + \frac{(n-1)(n-2)}{2.1}q^{n-3}P^2 + \ldots + P^{n-1} \right]$$

$$= nP\left[{}^{n-1}C_0 q^{n-1} + {}^{n-1}C_1 q^{n-2}P + \ldots + {}^{n-1}C_{n-1}P^{n-1} \right]$$

$$= nP(p+q)^{n-1}$$

$$= nP\{\text{Since } p+q = 1\}$$

Variance of binomial distribution,

$$\sigma^2 = \sum_{r=0}^{n} r^2 P(r) - \mu^2$$

$$\sum_{r=0}^{n-1} (r + r(r-1))P(r) - \mu^2$$

$$\sum_{r=0}^{n} rP(r) + \sum_{r=0}^{n} r(r-1)P(r) - \mu^2$$

$$\mu - \mu^2 + \sum_{r=0}^{n} r(r-1)P(r)$$

$$= \mu + n(n-1)P^2 (q+P)^{n-2} - \mu^2 \quad \left\{ P(r) = {}^{n}C_r P^r q^{n-r} \right\}$$

$$\mu + n(n-1)P^2 - \mu^2$$

Since $\mu = nP$ and $(P + q) = 1$; therefore, variance $= nPq$

Standard deviation of binomial distribution,

Standard deviation $= \sqrt{nPq}$

Measure of skewness of binomial distribution, $Y_1 = (q-P)/\sqrt{nPq}$ or $(1-2P)/\sqrt{nPq}$

Measure of Kurtosis $= \beta_2 = 3 + \dfrac{1-6Pq}{nPq}$

If, $P < 1/2$, skewness is positive

$P > 1/2$, skewness is negative

$P = 1/2$, skewnes is zero

A6.6 POISSON'S DISTRIBUTION

The Poisson's distribution is a special case of a binomial distribution, in which the value of n approaches infinity and the value of p approaches 0,

$$P(X = r) = {}^{n}C_r p^r q^{n-r}$$

If we put $n \to \infty$, and $P \to 0$,

$$P(X = r) = {}^{n}C_r q^{n-r} P^r$$

$$= n(n-1)(n-2), \ldots, (n-r+1)(1-P)^{n-r} P^r$$

Let, $nP = m$(mean)

$$= \frac{n(n-1)(n-2)...(n-r+1)}{r!}(1-m/n)^{n-r}(m/n)^r$$

$$= (m^r/r!)\left[\{n(n-1)(n-2)...(n-r+1)\}/(n^r)\right]\left[(1-m/n)n/(1-m/n)r\right]$$

$$= (m^r/r!)\left(\frac{n}{n}\right)\left(\frac{n-1}{n}\right)\left(\frac{n-2}{n}\right)...\left(\frac{n-r+1}{n}\right)\times(1-\lambda/n)^{n-r} \text{ as, } n\to\infty \text{ each of } (r-1)$$

factors tend to 1

$$(1-1/n)(1-2/n)\ldots[1-\{(r-1)/n\}]$$

also $\lim_{x\to\infty}\left(1-\frac{1}{x}\right)^n = e$

Therefore, $P(X=r) = m^r e^{-m}/r!$, which is Poisson's probability distribution.

A6.6.1 Recurrence Formula for Poisson's Distribution

The probability of rth success is given by the following formula:

$$P(r) = m^r e^{-m}/r!$$

$$P(r+1) = m^{r+1}e^{-m}/(r+1)!$$

Therefore,
$$\frac{P(r+1)}{P(r)} = \frac{m}{r+1}$$

$$P(r+1) = \frac{m}{n+1}P(r)$$

So in a Poisson's distribution, if $n\to\infty$; (here, n stands for number of trials) and $P\to0$ (Probability of success),

nP taken as m(mean of distribution)

Probability of rth success $= P(r) = (e^{-m}m r/r!)$

Variance $= m$

Illustration A6.41 Prove that if $P(x) = e^{-m}m^x/x!$ then for, $x = 0, 1, 2$ we get a probability function, i.e., $P(x) = e^{-m}m^r/r!$ is actually a probability distribution.

Solution Since

$$\sum_{x=0}^{\infty}P(x)$$
$$= \sum_{x=0}^{\infty}e^{-m}m^x/x!$$
$$= e^{-m}\sum_{x=0}^{\infty}\frac{m^x}{x!}$$
$$= e^{-m}e^m$$
$$= 1$$

Hence, it is a Poisson's distribution.

Illustration A6.42 The probability that an individual will suffer from a bad reaction is 0.001. Find out the probability that out of 2000 individuals, exactly 3 will suffer a bad reaction.

Solution

P = Probability that an individual will suffer from a bad reaction = 0.001

$n = 2000$

Therefore, $\qquad nP = 2000 \times 0.001$
$$= 2$$

In addition, note that P is very small and $n \gg P$.

So we can apply Poisson's distribution. Therefore, $m = nP = 2$ and
$$P(3) = e^{-m}m^3/3!$$
$$= e^{-2}.2^3/3! = e^{-2}.4/3 = 4e^{-2}/3$$
$$= (4/3e^2) = 0.180$$

Illustration A6.43 In the above question, find probability that more than 2 individuals will suffer a bad reaction.

Solution In this case,

$$r = 3, 4, 5, \ldots, 2000, \text{ which is very difficult to evaluate}$$

Therefore, let us find the probability of $r = 0, 1, 2$ and subtract the sum from 1 (note that the sum of all probabilities of a distribution is 1)

i.e., $\qquad P(X > 2) = 1 - (P(X = 0) + P(X = 1) + P(X = 2))$
$$= 1 - (2^0 e^{-2}/0! + 2e^{-2}/1! + 2^2 e^{-2}/2!)$$
$$= 1 - 5e^2 = 0.323$$

Illustration A6.44 If 3% of the bulbs manufactured by a company are defective, then find the probability that a sample of 100 bulbs will not contain any defective bulb.

Solution Let P be the probability of finding defective bulb, $P = 3/100$

$$n = 100$$
$$m = nP = 3$$
$$P(r = 0) = e^{-m}m^r/r!$$
$$= e^{-3}3^0/0! = e^{-3} = 1/e^3$$
$$= 0.049$$

Illustration A6.45 In the above question, find probability of finding three defective bulbs.

Solution

$$P = 3/100$$
$$n = 100 \quad m = 3$$
$$P(r = 3) = e^{-3}3^3/3!$$
$$= 0.2241$$

Illustration A6.46 In QA6.44, find the probability that more than 5 bulbs are defective.

Solution

$$P(X \geq 5) = 1 - P(X < 5)$$
$$= 1 - \{P(X = 0) + P(X = 1) + P(X = 2) + P(X = 3) + P(X = 4)\}$$

$$P(X = 0) = e^{-m}m^0/0! = e^{-m} = e^{-3}$$

$$P(X = 1) = e^{-m}m/1! = e^{-3}.3/1! = e^{-3}$$

$$P(X = 2) = e^{-m}m^2/2! = e^{-3}.9/2 = (9/2)e^{-3}$$

$$P(X = 3) = e^{-m}m^3/3! = e^{-3}(27/3 \times 2) = (9/2)e^{-3}$$

$$P(X = 4) = e^{-m}m^4/4! = (e^{-3}.27/4 \times 2 \times 1) = (27/8)e^{-3}$$

Therefore, $1 - \{P(X = 0) + P(X = 1) + P(X = 2) + P(X = 3) + P(X = 4)\}$
$$= 1 - e^{-3}(1 + 3 + 9/2 + 9/2 + 27/8)$$
$$= 0.0838$$

Illustration A6.47 Six coins are tossed 6400 times. Find the probability of getting 6 heads r times.

Solution

Probability of getting one head = 1/2
Arrange number of six heads with 64,000 throws

$$nP$$
$$= 100$$

i.e., $$m = 100$$

So, $$P(X = r) = m^r e^{-m}/r!$$
$$= 100^r e^{-100}/100!$$

Illustration A6.48 If in a Poisson's distribution $P(X = 2) = (2/3) P(X = 1)$, then find $P(X = 3)$.

Solution

$$P(X = 2) = (2/3)P(X = 1)$$
$$e^{-m}m^2/2! = (2/3)(e^{-m}m/1!)$$
$$m = 4/3$$

Therefore,
$$P(X = 3) = e^{-4}(4/3)^3/3!$$
$$= 32/81 \times e^{-4}$$

Illustration A6.49 The probability that a man aged 35 years will die before reaching 40 years is 0.018. Out of 400 men aged 35 years, what is the probability that 2 men will die within the next 5 years?

Solution

$$P = 0.018$$
$$n = 400$$
$$m = nP = 7.2$$
$$r = 2$$
$$P(X = 2) = e^{-7.2}(7.2)^2/2!$$
$$= 0.1936$$

Illustration A6.50 Fit a Poisson's distribution to the following:

x	0	1	2	3	4
f	122	160	15	2	1

Solution

x	f	xf
0	122	0
1	60	60
2	15	30
3	2	6
4	1	4
	N = 200	100

$$\text{Mean } (m) = \sum f(x) / \sum xf(x) = 100 / 200 = 0.5$$
$$m = nP = 0.5$$
$$P = 0.5/4 = 0.125$$

Now, $P(X = 0) = Ne^{-m} = Ne^{-0.5}$
$$= 200e^{-0.5}$$

$P(X = 1) = Ne^{-m}m/1! = 200 \times e^{-0.5} \times 0.5$
$P(X = 2) = Ne^{-m}m^2/2!$

$P(X = 3) = Ne^{-m}m^3/3!$
$P(X = 4) = Ne^{-m}m^4/4!$

by taking integral approximation, we get

X	0	1	2	3	4
P(X)	121	61	15	3	0

A6.7 NORMAL DISTRIBUTION

The normal distribution is also a special case of binomial distribution, where the value of P approaches 1/2 and that of n approaches ∞ (see web resources of the book for normal distribution table),
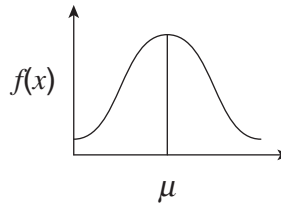
The function depicting the distribution is given by

$$f(x) = 1/(\sigma\sqrt{2\pi})e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$

x can assume values $-\infty$ to ∞.

μ is the mean and standard deviation is σ.

$f(x)$ is also written as $N(\mu, \sigma^2)$

Please note that:

(a) $f(x) \geq 0$

(b) $\int_{-\infty}^{\infty} f(x)dx = 1$.

i.e., total area of normal curves is 1 (unity).

Proof:

$$= \frac{1}{\sigma\sqrt{2\pi}}\int_{-\infty}^{\infty} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} dx = \frac{1}{\sigma\sqrt{2\pi}}\int_{-\infty}^{\infty} e^{-t^2\sigma\sqrt{2\pi}} dt$$

Putting $\left(\dfrac{x-\mu}{\sigma\sqrt{2}}\right) = t = \dfrac{1}{\sqrt{\pi}}\int_{-\infty}^{\infty} e^{-t^2 dt}$

and $dx = \sigma\sqrt{2}\ dt = \dfrac{2}{\sqrt{\pi}}\int_{-\infty}^{\infty} e^{-t^2 dt}$

Now, $\int_{0}^{\infty} e^{-t^2} = \dfrac{\sqrt{\pi}}{2}$ (γ function)

Therefore, area $= \dfrac{1}{\sqrt{\pi}}\int_{0}^{\infty} e^{-t} dt = \dfrac{1}{\sqrt{\pi}} \times \dfrac{\sqrt{\pi}}{2} = 1$

(c) Normal distribution has maximum value at $x = \mu$.

Proof: for maximum|minimum value

$$\frac{d}{dx}f(x)=0$$

So,
$$\frac{1}{\sigma\sqrt{2\pi}}\frac{d}{dx}\left\{e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}\right\}$$

$$\frac{1}{\sigma\sqrt{2\pi}}e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}\times\left\{\frac{-2}{2\sigma^2}(x-\mu)^2\right\}=0$$
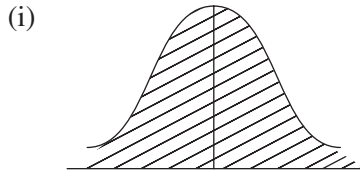
i.e., $(x-\mu)=0$

i.e., $x=\mu$.

In addition, $f''(\mu)<0$.

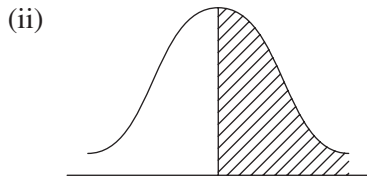So the above function has maxima at $(x=\mu)$.
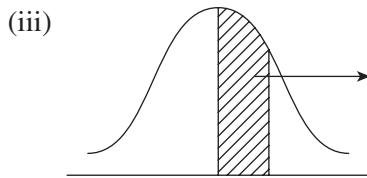
(d) To solve problems and normal distribution:

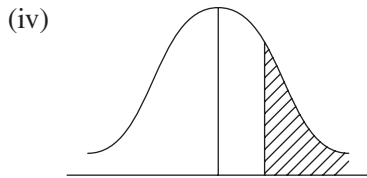Put, $z=\dfrac{x-\mu}{\sigma}$ we will get the following curve with maxima at $z=0$

(i)

Total area = 1.

(ii)

Half area = ½, $P(X<\lambda)$, and $(X>0)$.

(iii)

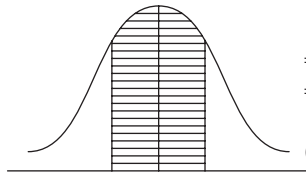This value can be found by seeing the normal distribution table.
$X=\lambda$

(iv)

$P(X>\lambda)=0.5-\underbrace{P(X<\lambda)}$

By table

e.g., $P(X>1)=0.5-P(0<X<1)$
$$=0.5-0.34143$$
$$=0.1587.$$

(v) $P(-\lambda_1 < X < \lambda_2)$

$$= P(-\lambda_1 < X < 0) + P(0 < X < \lambda_2)$$
$$= P(0 < X < \lambda_1) + P(0 < X < \lambda_2)$$

(due to symmetry)

Illustration A6.51 The mean weight of 500 female students in a college is 151 lbs and standard deviation is 151 lbs. Assuming that weights are normally distributed, find how many students weight between 120 lbs and 155 lbs.

Solution

$$X = 151 \text{ lbs} = \mu$$
$$\sigma = 151 \text{ lbs}$$

Therefore,
$$Z = \frac{X - 151}{151}$$

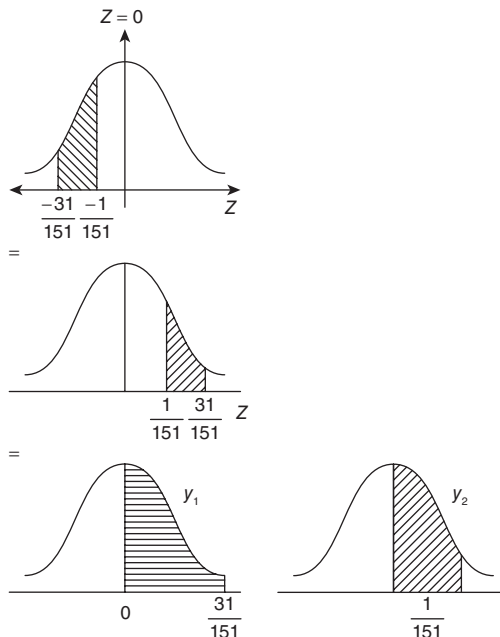Now, $P(120 < X < 155)$, when $X = 120$, $Z = \dfrac{120 - 151}{151} = \dfrac{-31}{151}$

And when
$$X = 155$$
$$Z = \frac{-1}{151}$$

So we need,
$$P\left(\frac{-31}{151} < Z < \frac{-1}{151}\right) Z = 0$$

Let,
$$x_1 = P(Z > 0) =$$
$$= 0.5$$

$$x_2 = P\left(0 < Z < \frac{1}{151}\right)$$

Now,
$$P\left(X > \frac{1}{151}\right) = x_1 - x_2 = y_1 \text{ (say)}$$

And
$$y_2 = P\left(X > \frac{1}{151}\right) = 0.5 - P\left(X < \frac{31}{151}\right)$$

So, answer is $y_1 - y_2 = 0.6$
Therefore, number of students $= 500 \times 0.6 = 300$

A6.8 CONCLUSION

The appendix explores the topic of probability. The concept of probability or chance of happening of an event is one of the most important constituents of probalistic analysis and randomized algorithms as stated earlier. Moreover, the topic lays the foundation of mathematical analysis of an algorithm. The topic also finds its applications in non-deterministic finite acceptor design. The reader is advised to go through the concepts of counting in order to fully understand the topic. The basic know-how of permutations, combinations, multisets, and sequences would help the reader to deal with the problems of probability in a better way. The topic has been dealt with via the problem-solving approach as the goal of this chapter was that the reader should be able to apply the concepts of probability in the real world. However, it may be stated here that the above discussion only provides an introduction to the topic and does not intend to present an all-inclusive version of it.

Points to Remember

- $P(A \cup B) = P(A) + P(B) - P(A \cap B)$
- In the case of mutually exclusive events, $P(A \cap B) = 0$ so $P(A \cup B) = P(A) + P(B)$
- If $A_1, A_2, \ldots A_n$ are independent events then $P(A_1 \cap A_2 \cap \ldots \cap A_n) = P(A_1) P(A_2) \ldots P(A_n)$
- Mean of a distribution is $\sum P_i x_i$ and its variance is, $\sum P_i x_i^2 - \left(\sum P_i x_i\right)^2$.
- The probability of rth success in a binomial distribution with number of trial $= n$, probability of success $= p$ and the probability of failure $= q$ is $P(X = r) = {}^nC_r P^r q^{n-r}$
- In the case of the Poisson's distribution, the value of n approaches infinity and the value of p approach 0. The probability of rth success is given by $P(X = r) = (e^{-m}m^r)/r!$.
- The function depicting the distribution is given by $f(x) = 1/(\sigma\sqrt{2\pi})e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$

KEY TERMS

Bay's Theorem Let S be the sample space and E_1, E_2, …, E_n be n mutually exclusive events associated with random experiment. If A is the event that occurs with E_1 or E_2 or E_n then

$$P(E_i/A) = \frac{P(E_i)P(A/E_i)}{\sum_{j=1}^{n} P(E_j)P\left(\dfrac{A}{E_j}\right)}$$

Certain event A certain event is one in which each element of a sample space is a favourable event. In this case, the probability of success is 1.

Event An event is the possible outcome of an experiment.

Independent events Two events are said to be independent if the occurrence or non-occurrence of one does not affect the probability of the occurrence of the other. Mathematically, in the case of independent events, the probability of intersection of two events is the product of probability of the two.

Impossible event An impossible event is one in which the sample space is an empty set. In this case, the probability of success, p is 0.

Pigeonhole principle If n pigeons fly into K pigeonholes, at $K < n$ there are more than one pigeon in at least one of the pigeonhole.

Probability It is the ratio of the number of favourable events to the number of unfavourable events.

Sample space The set of these events constitute what is called the sample space.

EXERCISES

Pigeonhole principle

1. State and prove pigeonhole principle.
2. Ten persons have first name Hari, Shiva, and Brahm and last name Sharma, Singh, and Bhasin. Show that at least two persons have the same first and last name.
3. An inventory contains 100 items, each marked 'available' or 'unavailable'. There are 60 available items. Show that there are at least two items in the list exactly four terms apart.
4. If, $a_k = \begin{cases} K, & K \le m/2 \\ 1 & K > m/2 \end{cases}$

   (a) How many elements are there in domain of a
5. In Q #4 show that range of a is n {1, …, n}.
6. In (Q #4) & (Q #5), show that $a_i = a_j$ for some $i \ne j$.
7. Show that in decimal expression of quotient of two integers, eventually some block of digits repeat.
8. Show that every set of 15 socks chosen among 14 pairs of socks contains at least one matched pair.

Basics

9. Find the probability of having 53 Mondays in a leap year.
10. Find the probability of having 50 Tuesdays in a leap year.
11. Find the probability of having 60 Mondays in a leap year.
12. A class has 10 boys and 15 girls. Find the probability of forming a group of 4 having at least 2 girls.
13. In the above question, find the probability of forming a group having at most two girls.
14. In question number 12, find the probability of forming a group having two girls.
15. Three cards are drawn from a well-shuffled pack of 52 cards. Find the probability of the entire set of selected cards belonging to the same suite.
16. In question number 15, find the probability of all the cards belonging to the same suite.
17. In question number 15, find the probability of two cards belonging to one suite and the third to a different.
18. An integer is selected from the first 200 integers. Find the probability that the selected integer is a multiple of 7 or 11.
19. In question number 18, find the probability that the number selected is neither a multiple of 7 nor 11.
20. In question number 18, find the probability that the number is not a multiple of 7 but 11.
21. Two cards are drawn from a well-shuffled pack of 52 cards. Find the probability that both are face cards.
22. In question number 21, find the probability that one is a face card and the other is not.
23. In question number 21, find the probability that none of the selected card is a face card.

Independent Events

24. A and B are independent events, $P(A) = 0.35$ and $P(A \cup B) = 0.6$. Find $P(B)$.
25. If A and B are independent events $P(A) = 0.4$ and $P(B) = p$, $P(A \cup B) = 0.6$. Then find p.
26. A bag contains 5 white, 7 red, and 4 black balls. If four balls are drawn at random one by one with replacement. What is the probability that all are white?
27. The odds against A solving a problem are 4 to 3, and that against B are 7 to 5. Find probability that problem will be solved.
28. In two successive throw of a pair of dice, find probability of getting a total of 8 each time.
29. A can hit a target 4 times in 5 shots, B 3 times in 4 shots, and C 2 times in 3 shots. Find probability that target will be hit.
30. In the previous question, find probability that the target will not hit.

Probability Distribution

31. Determine whether the following is a random probability distribution?

   (a) X: 0   1   2

   (b) P(x): 0.5   0.4   0.1

   where $P(x)$ is the probability of X_i.

32. Four cards are drawn from a pack of 52 playing cards. Find probability distribution of the number of aces.

33. Four bad oranges are mixed accidently with 16 good ones. Find the probability distribution of the number of bad oranges in a draw of two oranges.

34. Two cards are drawn successively with replacement from a well-shuffled pack of 52 cards. Find the probability distribution of the number of kings.

35. A bag contains 3 green and 5 white balls. Find the probability distribution of the number of green balls drawn when 3 balls are drawn.

36. A random variable has following probability distribution

(i)	X:	−2	−1	0	1	2	3
(ii)	P(x):	0.1	K	0.2	2K	0.3	K

   (a) Find the value of K

37. A random variable X has following probability distribution:

(i)	X:	0	1	2	3	4	5	6	7	8
(ii)	P(x):	a	3a	5a	7a	9a	11a	13a	15a	17a

   (a) Find,

      1. 'a'   2. $P(X < 3)$ 3. $P(X \geq 3)$ 4. $P(0 < X < 5)$

38. From a lot of 30 bulbs which include 6 defective bulbs, a sample of 4 bulbs are drawn with replacement. Find probability distribution of number of defective bulbs.

39. In the above question find the mean.

40. In (Q #8) find the variance.

Binomial Distribution

41. Fit a binomial distribution for the following data and compare theoretical frequencies with actual ones

   (i) x: 0   1   2   3   4   5

   (ii) f: 2   14   20   34   22   8

42. A coin is tossed 5 times. What is the probability of getting atleast 3 success?

43. A pair of dice is thrown 7 times. If getting a total of 7 is considered a success, what is the probability of

   (a) no success   (b) 6 successes

44. Two cards are drawn from a well-shuffled pack of 52 cards. Find the probability distribution of the number of aces.
45. The probability that a student entering a university will graduate is 0.4. Find the probability that out of 3 students
    (a) none will graduate   (b) only one will graduate
46. In the above question, find the probability that all will graduate.
47. The items produced by a company contain 10% defective items. Show that the probability of getting 2 defective items in a sample of items is
    i. $(28 \times 9^6)/10^8$.
48. Five dice are thrown simultaneously. If occurrence of 3, 4, or 5 in a single die is considered success. Find probability of at least 3 successes.
49. In a hurdle race, a player has to cross 10 hurdles. The probability that he crosses a hurdle is 5/6. Find probability that he will cross fewer than 2 hurdles.
50. A bag contains 7 green, white, and red balls. If 4 balls are drawn one by one with replacement, what is the probability that one is red?

Poisson's Distribution

51. The following data were collected showing the number of accidents in each of 200 sectors in the city of Faridabad (including colonies, housing enclaves, etc.)

(i) Number of accidents	0	1	2	3	4
(ii) Frequency	109	65	22	3	1

    (a) Fit a Poisson's ratio to the data.
52. Prove that in a Poisson's distribution, mean deviation about mean $= (2/e)$X standard deviation.
53. Suppose a book of 585 pages contain 43 errors distributed randomly. What is the probability that 10 pages selected at random contain no errors?
54. If 20% of the bolts selected at random are defective, then find the probability that out of 4 bolts chosen at random, none will be defective.
55. In the above question, find probability that less than 2 bolts will be defective.
56. Establish that Poisson's distribution is an approximation to binomial distribution.
57. Discuss the properties of Poisson's distribution.
58. A dice is tossed 120 times. Find probability that face 4 will turn up 18 times.
59. In the above question, find probability that the face 4 will turn up 14 or less times.
60. In (Q #58), find $P(X < 40)$.

Normal Distribution

61. Prove that in a binomial distribution, curve area (total) $= 1$.
62. Prove that normal distribution curve has maximum value at mean.
63. What is normal distribution, when it is used?
64. A sample of 100 dry battery cells tested to find the length of life produced the following results.

(a) x = 10 hours, σ = 3 hours. Assuming that data are normally distributed, what percentage of cells have life more than 12 hours?

65. In the above question, what percentages of cells have life less than 5 hours?

66. In (Q #64) what percentages of bulbs have value between 10 and 14 hours?

Answers

Selected Problems

31. Yes

32.

X:	0	1	2
P(x):	$\dfrac{12}{19}$	$\dfrac{32}{95}$	$\dfrac{3}{95}$

33.

X:	0	1	2	3
P(x):	$\dfrac{5}{28}$	$\dfrac{15}{28}$	$\dfrac{32}{95}$	$\dfrac{3}{95}$

34. (a) 1/81   (b) 1/9   (c) 8/9

41. $P = 0.432$
    $q = 0.568$
    $n = 5$
    $N = 100$.

42. $P = 1/2$, $q = 1/2$
    $P(X \geq 3) = 1/2$

43. $(5/6)^7$, $35(1/6)^7$

44.

X =	0	1	2
P(x) =	$\dfrac{144}{169}$	$\dfrac{24}{169}$	$\dfrac{1}{169}$

45. 0.216, 0.432

46. 0.064

48. 1/2

49. $5/2(5/9)^9$

50. $5/4(11/16)^3$

51.

r	0	1	2	3	4
υ	109	66	20	4	1

53. 0.4795

54. 0.4096

55. 0.8192

APPENDIX A 7

Scheduling

OBJECTIVES

After reading this appendix, the reader will be able to
- Understand the importance of scheduling
- Enlist and explicate the various job scheduling problems
- Understand as to why job scheduling are NP-complete
- Learn some of the tools which help in scheduling our tasks

A7.1 INTRODUCTION

The problem discussed in this appendix constitutes one of the most important class of problems in algorithm design: scheduling problem. It finds its applications not only in operating systems but also in the planning of manufacturing systems. The problem is important but the solution is not easy. The following discussion discusses the various versions of job scheduling and as to why the problem is NP-complete. The approach to solve the problem has also been discussed in this appendix. The prerequisites of this appendix are the chapters on NP problems, approximation algorithms, and artificial intelligence approaches. The reader is requested to go through these topics before starting with this appendix.

A7.1.1 Scheduling Problems

Scheduling problems are the problems that require allocation of resources or/and that of time slots with some constraints. These problems are generally optimization problems. One of the most common examples of the problem is that of job scheduling. The problem requires the allocation of jobs to machines as per the given constraints. A schedule is generally generated as the output, keeping in view the optimization objectives. The constraints of the job scheduling problem are as follows:
- A job is assigned to a particular machine
- A machine can do a single job at a particular time
- If a machine has been allotted a job, it would complete the job and then move to the next job.

The problem can be classified as a single machine, multi-machine, single-stage, and multistage problem. A single machine version has just one machine and many jobs; the multi-machine version has more than one machine at its disposal, the single-stage multiple machine problems generally requires parallel machines. The models that are used to solve these problems can be identical parallel machine, uniform parallel machines, and unrelated parallel machines. The concept of parallel computations has already been discussed in the chapter on PRAMS (Chapter 22).

The multistage multi-machine problems can be classified as Flow Shop, Open Shop, Job Shop, and Group Shop. The taxonomy of these problems is as follows:

- **Completion time:** The completion time is the earliest time in which a job is completed.
- **Lateness:** It is defined as the difference between the completion time and the arrival time of a job.
- **Tardiness:** It is same as lateness if lateness is greater than zero, otherwise it is zero.
- **Earliness:** It is the negative of lateness or zero, whichever is greater.
- **Makespan or the maximum completion time:** It is the maximum of the completion times of the given jobs.

The solution to the problem is one of the permutations. Permutation generation in itself is an algorithmically colossal task. The constraints checking of the generated permutations would further increase the complexity.

The appendix has been organized as follows. Section A7.2 discusses the various versions of the problem, Section A7.3 discusses the established approaches to deal with the problem, Section A7.4 gives a brief overview of the tools that can be used to tackle the problem, and the last section gives the conclusion.

A7.2 DEFINITIONS AND DISCUSSIONS

The following section introduces various job scheduling problems. The discussion throws light on why the problems are NP-complete.

A7.2.1 Job Scheduling

If there are n jobs $\{J_1, J_2, J_3, \ldots, J_n\}$. The time required to carry out these jobs is given by the set $\{t_1, t_2, \ldots, t_n\}$ and the number of processors, k. The task is to schedule the jobs in such a way that the maximum time to execute the tasks t_{max} is minimum.

Discussion: The problem is an NP-complete problem. This was proved by J. D. Ullman in one of his papers in the *Journal of System and Computer Sciences* (Vol. 10, Issue 3). In the paper, he proved that the problem is NP-complete even if each job takes one unit of time. Moreover, even if the jobs are scheduled by two processors and each job takes a unit time or two, the problem is still NP-complete. This implies that processor scheduling is an NP-complete problem. Interestingly, this 10 page paper has been cited 973 times (Google Scholar, 30th January, 2015), so it must be a very important work.

Since there are k processors, at a time k jobs can be executed. The jobs $\{J_1, J_2, J_3, \ldots, J_n\}$ are allotted to processors $\{p_1, p_2, \ldots, p_k\}$. A problem, as stated in chapter of NP problems, is an NP problem if it can be solved by a non-deterministic algorithm in polynomial time. Apparently in order to prove that a problem is NP-complete, it must be shown that there is no polynomial time algorithm for the problem. The task in itself is a tedious one. Moreover, it was also stated in Chapter 19 (NP Problems) that an NP-complete problem answers in a yes or a no; therefore, it is important to express the scheduling problem as a yes–no type problem in order to classify it as an NP-complete problem. The following definition given by J.D. Ullman expresses the problem as a yes–no problem.

A7.2.2 NP-complete Job Scheduling Problem

Given a set of n jobs, a partial order $<$ on S, a weighing function W, number of processors k and a time t, does there exist a total function f from S to $\{0, 1, \ldots, (t-1)\}$ such that
- If $J < J'$, then $f(J) + W(J) \leq f(J')$
- for each J in S, $f(J) + W(J) \leq t$
- for each i, between 0 and t, there are at least k values of J for which $f(J) \leq i < f(J) + W(J)$?
  (J. D. Ullman, NP-complete Scheduling Problems, Journal of Computer and System Sciences, 10, 384–393)

Discussion: The above representation would allow the Turing machine to able to check the solution in n symbols (and hence in polynomial time). The problem, therefore, cannot be solved in a polynomial time but can be verified in polynomial time. Hence, the problem is an NP-complete problem.

The principle of reducibility can then be applied to strengthen the argument. One may note that every NP-complete problem can be converted into the SAT problem, as stated by Karp. This principle has been used to show that a lot of problems are NP-complete. In fact in the paper Ullman also took a special case wherein $W(J) = 1$ for all J's. Another version of the problem assumed that the jobs take either one unit time or maximum of two units of time. It has been proved that both the above cases cannot have a polynomial time deterministic algorithm.

A7.2.3 Single Execution Time Scheduling with Variable Number of Processors

The definition of the problem uses a function which precedes as the argument (Job) precedes. That is,
- If $J < J'$, then $f(J) < f(J')$
- The inverse function $f^{-1}(J)$ has c_i members
Here, c_i is the sequence of integers from c_0 to c_{t-1}

Discussion: The above problem can be represented as a Turing machine that requires $O(n)$ memory. Hence, the problem can be verified in polynomial time. The scheduling problem, stated earlier, can be converted into the single execution time scheduling with

variable number of processors. Therefore, the former is also an NP-complete problem. The conversion of the former to the latter has been shown by Ullman. As a matter of fact the SAT3 problem can be converted into the single execution time scheduling with variable number of processors in polynomial time. The SAT3 problem is an NP-complete problem, so is the former.

Another important scheduling problem is the pre-emptive scheduling problem. The problem can be stated as follows.

A7.2.4 Pre-emptive Scheduling

Given a set S of n jobs, a partial order $<$ on S, a weighting function W, a number of processors k, and a time limit t, does there exist a total function f from S to subsets of $\{0, 1, …, t − 1\}$, such that

- $f(J)$ has $W(J)$ members for all J in S,
- if $J < J'$, i is in $f(J)$ and i' in $f(J')$, then $i < i'$, and
- for each i, there are at most k values of J for which $f(J)$ contains i?

Discussion: It has been proved that the pre-emptive scheduling problem is an NP-complete problem as the scheduling problem introduced at the beginning of the discussion can be converted into the pre-emptive scheduling in polynomial time.

A7.3 HOW TO HANDLE SCHEDULING PROBLEMS?   ■

The simplest way of dealing with the problem is to enumerate all the possible solutions and then select the best solution. But this is easier said than done. The enumeration of all the combinations, in itself, takes exponential time. The problem does not end here. The fitness values for all these enumerations are then calculated. This fitness value would be calculated as per the problem. This procedure would require a colossal amount of time (exponential if better algorithms are used and factorial if convention are used). The complexity would render the above approach useless.

The other method of handling such problems is as follows. In order to explain the approach, an example of airline scheduling has been taken. It is a known fact that the airlines need to produce hundreds of schedules everyday. These schedules must take care of the given constraints like the availability of routes, that of the staff, and so on. Moreover, there are some of the factors that cannot be planned in advance, for example, the weather and storm. So airlines require efficient algorithms capable of taking care of all the problems and at the same time.

The following example would take care of a very few constraints and probably in no airlines the things would be so simple. However, it is important to be able to solve small problems rather than starting with huge problems and going nowhere. It is like a Sheldon Cooper trying to put forth a theory that makes relativity look small, but effectively not being better than any of his counterparts.

Example: In the market analysis, it was found that there are n routes which the airlines must cater to, in order to make profit. Each flight would be characterized by the starting time, the time of arrival, the source airport, and the destination airport. The airlines must have the following seven flights in its roaster of a day.

Flight	Origin	Destination	Departure time	Flight time
1	New Delhi	Mumbai	22:00	23:50
2	Mumbai	Banglore	23:50	2:00
3	Banglore	Calcutta	1:00	3:40
4	Mumbai	New Delhi	08:00	10:00
5	Chandigarh	New Delhi	09:00	10:00

The indicated times are that in which the flights as well as the airports must be served. Now the problem is to find whether a single plane can be used for more than one segment. One may note that this would be possible only if the source of the first segment is same as the destination of the second and there is an ample time to let the passengers of the first flight to get off the first plane and that of the second to get into the second with the requisite time in between the two tasks.

Moreover, this is also possible if the two flights do not have anything in common but there is an ample time for the plane to reach from the destination of the first to the source of the second (probably the destination of the first is near to the source of the second).

In this case, the concept of reachability comes into picture. Kleinberg (Algorithm Design, Pearson) in his book has defined reachability as follows. The flight i is reachable from flight j if it is possible to use same plane for i that was used for the flight j.

The problem can be expressed as an NP-complete problem as follows:

Is it possible to use k planes to serve m flights as per the given schedule? The inputs to the problem are

- k, the number of planes
- n, the number of flights
- Schedule containing source airport, destination airport, starting time, and destination time.

The output of the above problem is a yes or a no. A yes is produced if it is possible to accomplish the above task using k planes, otherwise a no is produced.

The problem can be handled by converting the problem into a flow problem. The feasibility of the latter would ensure the solvability of the former. The problem can also be solved via dynamic approach. Both the methods are computationally expensive and may lead to intractability. However, availability of tools has helped tackling some versions of the scheduling problems efficiently. Some of the tools have been discussed in the next section.

A7.4 TOOLS

The intricate scheduling algorithms can be implemented using tools available for MATLAB and implementations in R. However, the purpose of the following section is

not to discuss such implementations but to introduce you to practice tools which would help you in scheduling your meetings and manage your tasks efficiently. The tools discussed in this section are Doodle, Volentter Spot, YouCanBookMe, and Wiggio. The science that goes in the development of such tools requires the concepts discussed in this appendix. The readers are requested to go to the websites of the tools and explore the features of each of them. Perhaps, one can develop such tool as a project. Such project would help in the development of designing an algorithm and programming skills of the reader.

Doodle

One of the most important scheduling problems is the scheduling of meeting between the group members. Doodle is one of the free tools for scheduling such group meetings. Doodle is fundamentally a polling podium. In order to schedule meetings, the time of the meeting and the list of persons to be invited are added as input. The persons are then required to fill their data (as in the availability of time and dates). The software schedules the meetings accordingly. The concept is simple but the science that goes in is deep.

Volunteer Spot

Volunteer Spot is a free scheduling service that teachers, coaches, and others use to coordinate volunteers. The tool helps one to post calendars and signup sheets online. The software also takes care of the point where at which the slots are filled.

YouCanBook.Me

YouCanBook.Me is a free scheduling tool that integrates with your Google Calendar. YouCanBook.Me allows people to book fixed blocks of time in your calendar. You specify the length of each block of time and the dates and times you are available. Visitors to your calendar can click a block and enter their email addresses to reserve a block of your time. When a block of time is reserved you receive an email alert.

Wiggio

Wiggio is another tool that makes the scheduling of group meetings easier. The tool also helps in resource scheduling and planning of projects. The tools has some brilliant features like

- Group calendar
- Mass messaging system
- Group polling system

A7.5 CONCLUSION

The appendix focuses on the definition of scheduling problems. The reason so as to why they are NP-complete lies in the definition itself. The reader is requested to explore some of the open source tools (like that in MATLAB) that help in tackling the scheduling problems. The reader should also go through some of the research papers on the application of soft computing techniques in scheduling problems (See the Project in the exercise).

Points to Remember

- The job scheduling problem can be classified as a single machine, multi-machine, single-stage, and multistage problem.
- The multistage multi-machine problems can be classified as Flow Shop, Open Shop, Job Shop, and Group Shop.
- Doodle, Volunteer Spot, YouCanBook.Me, and Wiggio are some of the tools that are used in scheduling.

KEY TERMS

Earliness It is the negative of lateness or zero, whichever is greater.
Job scheduling This problem requires the allocation of jobs to machines as per the given constraints. A schedule is generally generated as the output keeping in view the optimization objectives.
Lateness it is defined as the difference between the completion time and the arrival time of a job.
Makespan or the maximum completion time It is the maximum of the completion times of given jobs.
Scheduling problems These are the problems that require allocation of resources or/ and that of time slots with some constraints. These problems are generally optimization problems.
Tardiness It is same as lateness if lateness is greater than zero. Otherwise it is zero.
Completion time It is the earliest time in which a job is completed.

EXERCISES

I. Multiple Choice Questions

1. Scheduling problems are the problems which require
   - (a) Allocation of resources
   - (b) Allocation of time slots
   - (c) Both
   - (d) None of the above
2. The scheduling problems are generally
   - (a) P problems
   - (b) NP-complete
   - (c) NP-hard but not NP-complete
   - (d) None of the above
3. The scheduling problems generally fall in which of the category?
   - (a) Optimization
   - (b) Always solvable
   - (c) Cannot be verified in polynomial time
   - (d) None of the above
4. Which of the following is the definition of the completion time in job scheduling?
   - (a) The completion time is the earliest time in which a job is completed
   - (b) The completion time is the earliest time in which at least one job is completed

   (c) Both
   (d) None of the above
5. Which of the following correctly defines lateness in job scheduling?
   (a) It is defined as the difference between the completion time and the arrival time of a job
   (b) It is the maximum arrival time
   (c) It is the minimum arrival time
   (d) None of the above
6. Which of the following is same as lateness if lateness is greater than zero, otherwise it is zero?
   (a) Tardiness   (c) Tiredness
   (b) Tidiness   (d) None of the above
7. Which of the following is the maximum completion time of given jobs?
   (a) Makespan   (c) Make
   (b) Makeslow   (d) None of the above
8. Which of the following tools are used in scheduling?
   (a) Doodle   (c) YouCanBook.Me
   (b) Volunteer Spot   (d) All of the above
9. Which of the following approaches are used to solve a scheduling problem?
   (a) Dynamic programming   (c) Both
   (b) Linear programming   (d) None of the above
10. Which of the following are the multistage multi-machine problems?
   (a) Flow Shop   (c) Job Shop
   (b) Open Shop   (d) Group Shop

II. Review Questions

1. What is job shop scheduling?
2. Define job scheduling? Explicate the types of job scheduling.
3. Prove that the above scheduling problems (question number 2) are NP-complete.
4. Explain some of the tools that help you to schedule tasks.

III. Project

1. Go through some of the research papers of Job Shop scheduling.
2. Classify the techniques used to solve the problem.
3. Select a paper that solves the problem using genetic algorithms and implement the algorithm.

Answers to MCQs				
1. (c)	3. (a)	5. (a)	7. (a)	9. (c)
2. (c)	4. (a)	6. (a)	8. (d)	10. (c)

APPENDIX A8

Searching Reprise

OBJECTIVES

After studying this appendix, the reader will be able to
- Compare searching techniques
- Understand the insertion in binary search tree
- Explain the deletion of a node in a binary search tree
- Learn insertion and deletion in an AVL tree

A8.1 INTRODUCTION

Searching is one of the most important procedures in computer science. We use searching not only in Internet but also in databases and even in the text editors. The aim of an algorithm developer should be to constantly come up with effective and efficient procedures for searching. In Chapter 5, we have already discussed linear search. The complexity of linear search is $O(n)$, so it becomes inefficient when the value of n is too large. Another method of searching is using binary search. Binary search is used when the elements in the list are sorted. The complexity of binary search is $O(\log n)$, which is better as compared to linear search. The binary search divides the array into two equal halves. The Fibonacci search, on the other hand, divides the given array into two unequal parts. The position at which the array is split is decided by the Fibonacci series. In the series, the first element is 1, second is 1, and the rest of the elements are the sum of last two terms. The terms of the series are 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, and so on. The array in Fibonacci search must be ordered. The last index of the array must be a Fibonacci number. For example, if the first index of the array is 1 and the last index is 34 (one of the Fibonacci terms), then the array is split at position 21 (the term in the Fibonacci series before 34). In this process, we divide the array into two parts, the first having 21 elements and the second having 13 elements. This makes the search faster, on an average. As a matter of fact, the complexity of the algorithm is $O(\log n)$, which is same as that of binary search, but in many cases, it performs better than the binary search. The variants of binary search find the point of splitting the array via different methods. Interpolation search is one such method. Table A8.1 summarizes the above points.

Table A8.1 Techniques of searching

Technique	Complexity	When to use	Disadvantage
Linear search	$O(n)$	Elements are not sorted	Becomes inefficient when the value of n is large
Binary search	$O(\log n)$	Elements are sorted	The elements need to be sorted before applying binary search
Fibonacci search	$O(\log n)$	Elements are sorted	The elements need to be sorted before applying the search

There are many data structures that ease the search of a key. Binary search tree is one of them. The following discussion focuses on binary search tree and the insertion and deletion in them. This appendix has been organized as follows. The following section discusses the insertion in a binary search tree. Section A8.3 discusses deletion in the tree. Section A8.4 presents the problems of BST and AVL trees and the last section gives the conclusion.

A8.2 BINARY SEARCH TREE—REVISITED

A tree is a graph that has no cycle or no isolated vertex. A binary tree is one in which all the nodes have at maximum two children. A binary search tree (BST) is a binary tree in which the keys at the left are less than that at the root and that at the right are greater than the root. The topic was introduced in Section 6.7 of Chapter 6. Creation of a BST and searching for a key in it have been already discussed. Before proceeding with the insertion and deletion, the following points may be noted with reference to a BST.

The leftmost node of the left sub-tree is the smallest in a BST. The process of finding the leftmost node of the left sub-tree has been depicted in Algorithm A8.1. In the algorithm, the pointer ptr is set to the root of the given tree. Till the left pointer of the ptr is NULL, ptr is set to the left pointer of ptr.

Algorithm A8.1 Smallest_BST(T)

Input: The given BST, T
Output: The least key in the given BST
```
//ptr is a pointer to a node. It has three parts: data, ptr->left (points to
the left sub-tree of ptr),
//ptr->right (points to the right sub-tree of ptr). 'root' is the root node of
the tree,
Smallest_BST(T)
    {
    ptr = root;
//If ptr is the leftmost nnode of the left sub-tree
if( ptr->left == NULL)
        {
```

```
            return (ptr->data);
            }
    else
        {
        Smallest_BST(left sub tree of T);
        }
    }
```

Complexity: On an average, the above algorithm takes $O(\log n)$ time. In the case of a skewed tree, the complexity would be $O(n)$.

The rightmost node of the right sub-tree is the largest node in a BST. The process of finding the rightmost node of the right sub-tree has been depicted in Algorithm A8.2. In the algorithm, the pointer ptr is set to the root of the given tree. Till the right pointer of the ptr is NULL, ptr is set to the right pointer of ptr.

Algorithm A8.2   Largest_BST(T)

Input: The given BST, T
Output: The largest key in the given BST
```
//ptr is a pointer to a node. It has three parts: data, ptr->left (points to
the left sub-tree of ptr),
//ptr->right (points to the right sub-tree of ptr). 'root' is the root node of
the tree,
Largest_BST(T)
    {
    ptr= root;
//If ptr is the rightmost node of the left sub-tree
if( ptr->right == NULL)
        {
        return (ptr->data);
        }
    else
        {
        Largest_BST(right sub tree of T);
        }
    }
```

Complexity: On an average, the above algorithm takes $O(\log n)$ time. In the case of a skewed tree, the complexity would be $O(n)$.

The number of nodes in a BST can also be calculated recursively. A BST having a single node would return 1. In all other cases, the number of nodes is

the number of nodes of the left sub-tree
+ the number of nodes in the right sub-tree + 1

Algorithm A8.3 presents the procedure.

Algorithm A8.3 Number_of_nodes_BST(T)

Input: A BST, T
Output: The number of nodes in the BST
```
Number_of_nodes_BST(T)
    {
    ptr = root;
    if ((ptr-> left == NULL) && (ptr->right == NULL))
        {
        return 1;
        }
    else
        {
        return ((Number_of_nodes_BST (left sub-tree of T) + Number_of_nodes_BST
        (right sub-tree of T) + 1);
        }
    }
```

The following discussion explores the addition and deletion of a key in a BST.

The insertion of a key in a BST can be performed by Algorithm A8.4. While inserting a key in a BST, the first step is to find the appropriate position of the key. As stated earlier, if the key is less than the value at the node, we proceed to the left, else right. Once we have found the appropriate position of key, we insert it.

Algorithm A8.4 Insert(T, key)

Input: Tree T, key
Output: A binary search tree with key inserted at the appropriate position
```
Insert (T, key)
//T is a BST and key is to be inserted at the appropriate place in it, ptr is
a pointer to a node
/*ptr or node has a data part, a left pointer (ptr->left) which points to the
left sub-tree and a right pointer (ptr->right) which points to the right sub-
tree.*/
/*The Memory_allocate function allocates memory to the node, the sizeof(a)
function calculates the size in bytes of the argument, a */
ptr = root;
if (ptr! = NULL)
{
if (key > ptr->data)
    {
    Insert ( right sub-tree of T, key);
    }
else if (key < ptr->data)
```

```
    {
    Insert (left sub-tree of T, key);
    }
}
else
{
    ptr = Memory_allocate(sizeof(node));
    ptr->data = key;
    ptr->left = NULL;
    ptr->right = NULL;
    }
}
```

Complexity: The key in a BST is always inserted at the last level. Now, depending on whether the tree is balanced or skewed the insertion will take $O(\log n)$ or $O(n)$ time, where n is the number of nodes in the given tree.

Illustration A8.1 In the BST given in Fig. A8.1, insert 50.
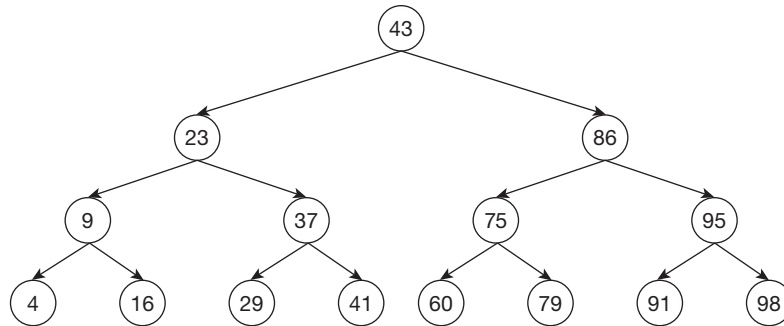
Figure A8.1 Binary search tree for Illustration A8.1

Solution The steps of insertion have been explained in the following Figs A8.2–A8.6.
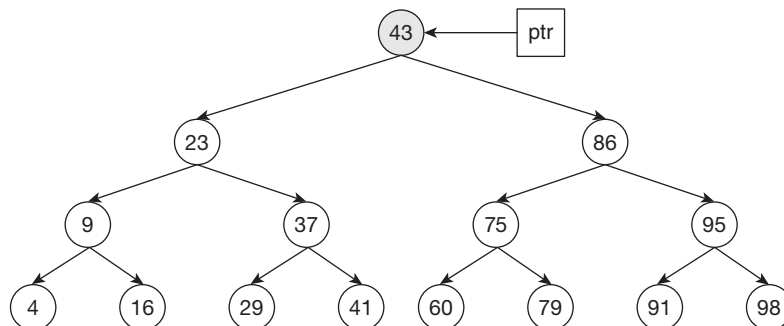
Figure A8.2 Key > Ptr-> data, therefore, the right sub-tree of ptr is processed
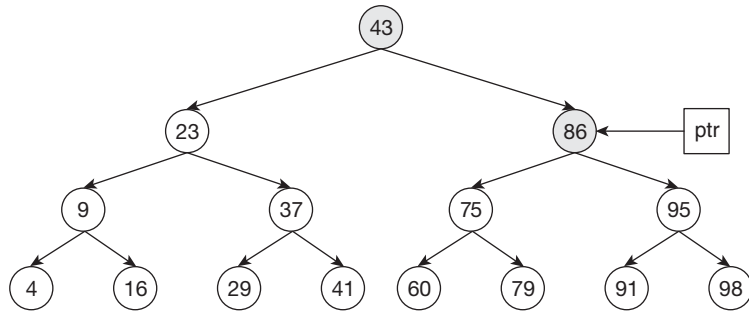
Figure A8.3 Key > Ptr-> data, therefore, the left sub-tree of ptr is processed
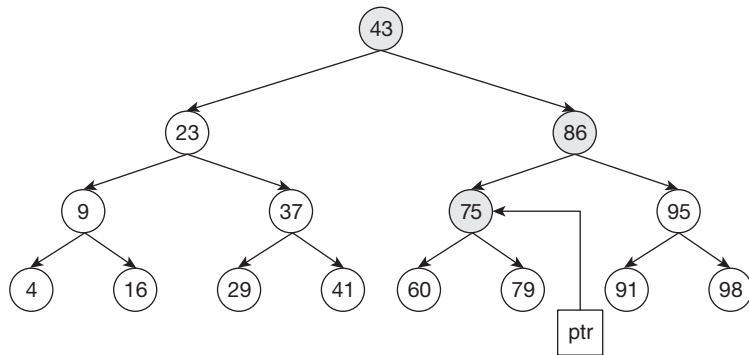
Figure A8.4 Key < Ptr-> data, therefore, the left sub-tree of ptr is processed
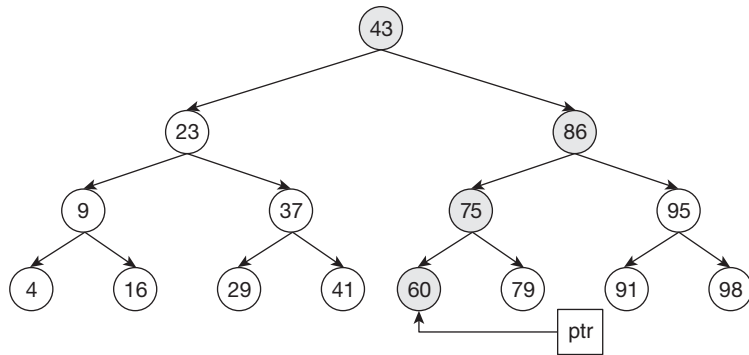
Figure A8.5 Key < Ptr-> data, therefore, the left sub-tree of ptr is processed

'ptr' in the figures refers to the pointer used in Algorithm A8.4. The reader is expected to look out for the trace of the algorithm, in order to understand the steps clearly. This illustration exemplifies a case where in the complexity of insertion is $O(\log n)$. The next illustration depicts a case wherein the complexity is $O(n)$.

The number of comparisons in the above illustration is four. Note that the number of nodes in the given tree is 15 $(= n)$, the comparisons are nearly $\log_2 n$. In fact, the average case insertion time in a BST is $O(\log n)$.
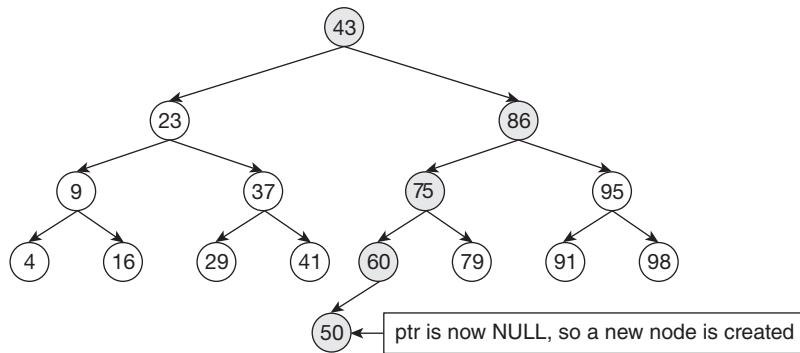
Figure A8.6   A new node is created and the data part of the node becomes the key. The left and the right pointers to the node become NULL

However, in the case of a skewed tree, the number of comparisons can be as large as $O(n)$. Illustration A8.2 depicts the case.

Illustration A8.2   Consider the tree as given in Fig. A8.7. The value 3 is to be inserted in the tree.
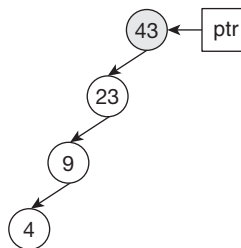
Figure A8.7   The pointer ptr is initially set to the root. The value of the key is less than that of ptr-> data; therefore, the left sub-tree is processed

Solution The ptr is first set to the root. Since the key is less than the value at the root node, the left sub-tree of the root is processed. The process is continued (Figs A8.8–A8.11).
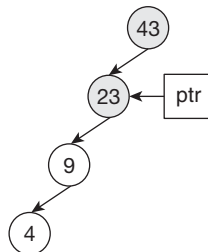
Figure A8.8   The value of the key is less than that of ptr->data (3 < 23); therefore, the left sub-tree is processed
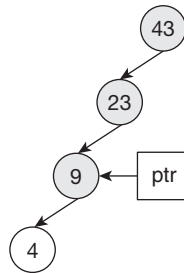
Figure A8.9   The value of the key is less than that of ptr->data (3 < 9);
therefore, the left sub-tree is processed
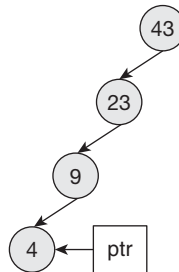
Figure A8.10   The value of the key is less than that of ptr->data (3 < 4);
therefore, the left sub-tree is processed
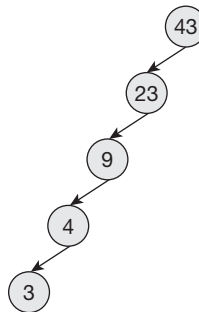
Figure A8.11   A new node is created. The data part of the node is set to 3 and
the left and the right pointers are set to NULL

Note that the number of comparisons in this case is 4, which is also the number of nodes in the given tree. Therefore, in this case (note that the given tree is a skewed tree), the number of comparisons is $O(n)$.

A8.3 DELETION IN A BST

In the case of deletion, the following procedure is followed. If the node to be deleted is the leaf node of the tree, then the node is simply deleted. Suppose if it is not the leaf node, the successor of the node to be deleted is found (Figs A8.12 and A8.13). The node is replaced with its successor and the problem now reduces to deletion of the node where this value goes (Figs A8.14–A8.16).
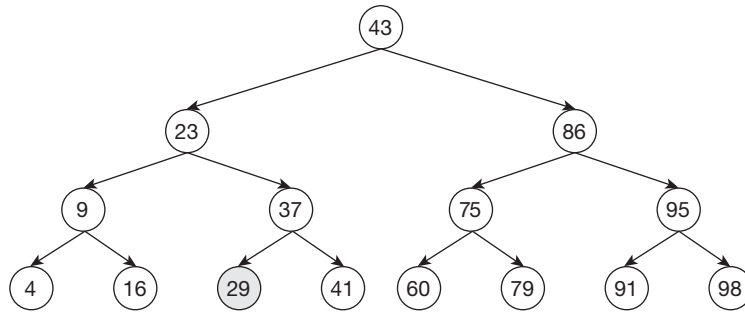
Figure A8.12   If 29 is to be deleted, the left pointer in the parent is set to NULL
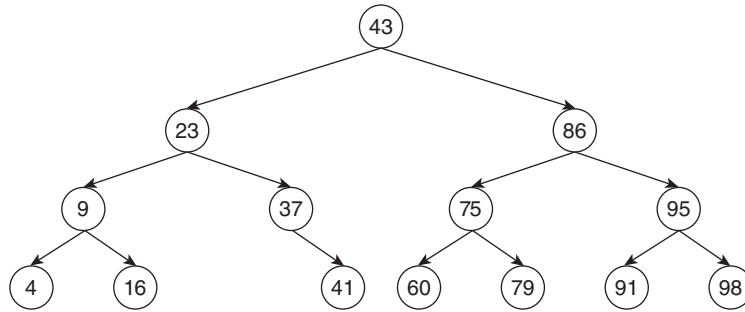
Figure A8.13   Deletion of 29
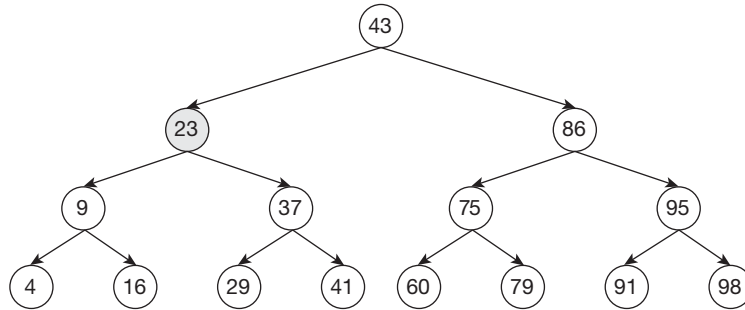
Figure A8.14   Twenty-three is to be deleted
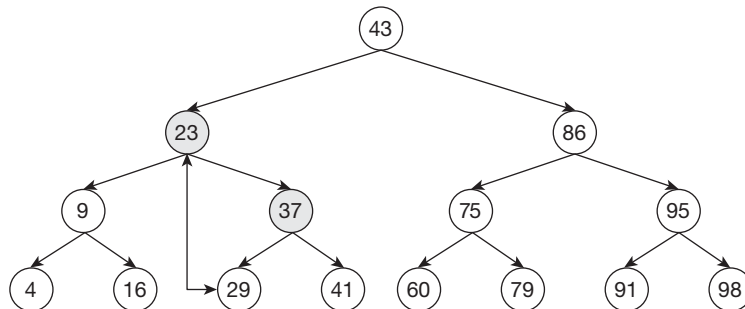
Figure A8.15   The in-order successor of 23 is 29; 29 comes in the place earlier occupied by 23 and then 29 is deleted
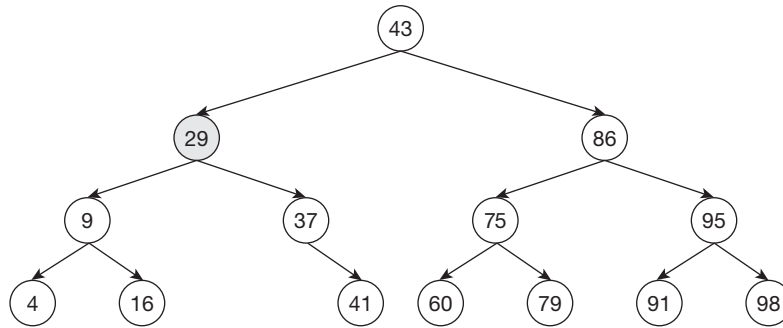
Figure A8.16   Since 29 is at the left node, it is simply deleted

The algorithm for deletion in a BST is as follows (Algorithm A8.5).

Algorithm A8.5   Deletion_BST(Key)

Input: Tree T, key
Output: A binary search tree with the node having value 'key' deleted

```
//The SUCCESSOR function find the successor of a node and the Parent(a) finds
the parent of a
Algorithm Deletion_BST(Key)
{
    ptr = root;
    while (ptr ->data != key)
        {
        if (key < ptr->data)
            {
            ptr=ptr->left
            }
        else
            {
            ptr = ptr->right;
            }
        }
    //When the node is a leaf node
    if ((ptr->left ==NULL) && (ptr->right == NULL))
        {
        x = parent (ptr);
        x->left = NULL;
        x->right=NULL;
        }
    else if ((ptr->left ==NULL) || (ptr->right ==NULL)&&!(( ptr->left ==NULL)
&& (ptr->right ==NULL))
    //When the node has just one child
    {
        y= SUCCESSOR(ptr);
```

```
        ptr->data = y-> data;
        ptr->left = NULL;
        ptr->right = NULL;
    }
    else
    {
        y = SUCESSOR (ptr);
        ptr->data = y->data;
        Deletion_BST(y);
    }
}
```

Complexity: The complexity of the first `while` loop can be $O(\log n)$ or $O(n)$ depending upon whether the tree is balanced or skewed (in the best case, it can even be $O(1)$).

The key, if terminal, would be deleted in time $O(1)$, otherwise the pointers would have to be reset.

The BSTs, therefore, are good for lookup and insertion. However, in the case of deletion, they can be expensive as deletion may lead to many swapping, as explained in the previous illustration.

A8.4 PROBLEM WITH BST AND AVL TREES

The BSTs are good for insertion and lookup wherein they show an average case complexity of $O(\log n)$. However, when the tree is skewed, the complexity increases to $O(n)$. The problem is solved using the AVL trees. The concept of AVL trees was introduced in Chapter 6 where the insertion of elements in the trees has been discussed.

The following discussion focuses on the deletion of elements in the AVL trees. The deletion of a node from a given AVL tree may lead to balance factor other than 0, 1, or −1. The two cases that make the balance factor of an AVL tree other than 0, 1, and −1 are as follows.

In the first case, the deletion of a node from the right sub-tree may lead to balance factor of the root becoming 2. In order to handle such situation, the root of the right sub-tree is shifted to the root. The rotation is shown in Fig. A8.18. When a node is deleted from the right sub-tree of the root, the balance factor of the root becomes 2 (Fig. A8.19). In such cases, the rotation (R1) shown in Fig. A8.20 is carried out.

There is another case where in nodes are removed from the left sub-tree of the left child, and the right sub-tree of its right child. In such cases, the right child of the left child of the root becomes the root node, the left child of the root becomes the left child of the root in the new tree and the root in the old tree becomes the right child of the new root. The sub-tree of the left child of the root in the old tree remains its sub-tree. The right sub-tree of its right child becomes the left sub-tree of the right child of the root and the left sub-tree of the right child of the left child of the root in the old tree becomes
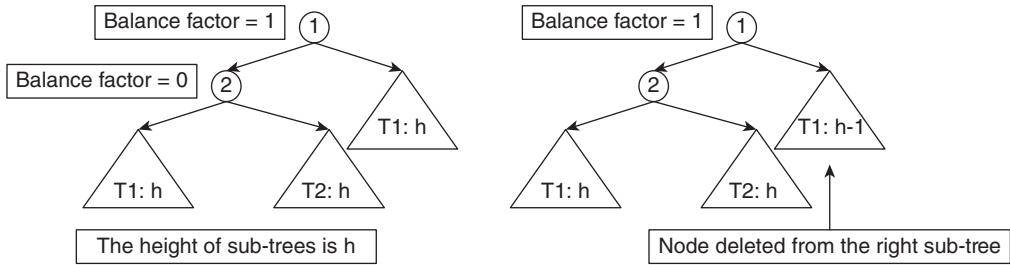
Figure A8.17   Balance factor of the root becomes 2 due to the deletion of element from the right sub-tree
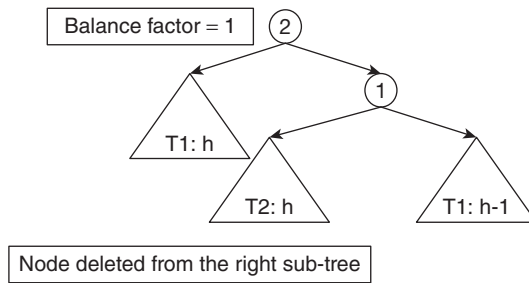
Figure A8.18   R1 rotation
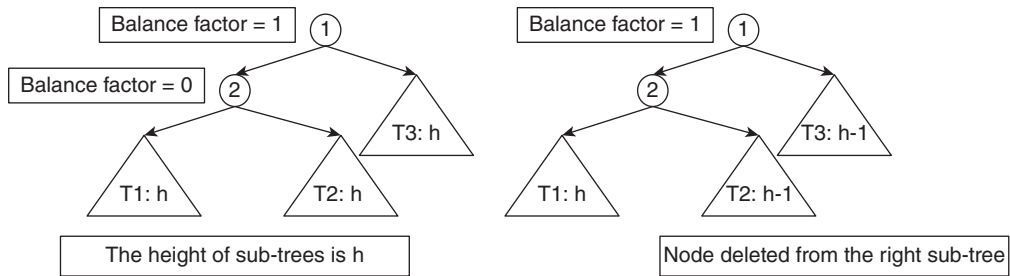
Figure A8.19   Balance factor of the root becomes 2 due to the deletion of element from the right sub-tree and the right sub-tree of the left child of the root
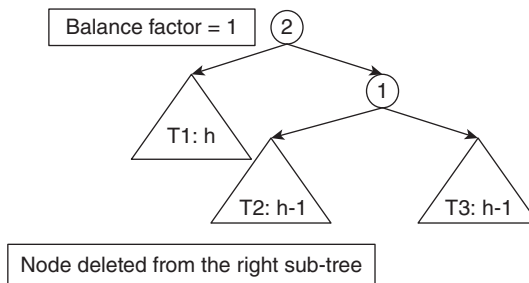

Figure A8.20   R1 rotation

the right sub-tree of the left child of the new root. The fourth sub-tree is placed at the remaining position. This is referred to as the R-1 rotation.

A8.5 CONCLUSION

The appendix discusses various strategies of searching. The advantages and disadvantages of each technique have been presented. The chapter also covers the data structures used for effective searching. The insertion and deletion in a BST and deletion of a node in an AVL tree have also been discussed in the appendix. The reader is expected to go through the web resources of the book to look for the corresponding programs.

EXERCISES

I. Multiple Choice Questions

1. Which of the following is true for a binary search tree?
   (a) The value at the root is less than any element on the right sub-tree
   (b) The value at the root is more than any element on the left sub-tree
   (c) All the above
   (d) None of the above
2. Which of the following is the correct formula for the number of nodes in a BST?
   (a) Number of elements in the left sub-tree + Number of elements in the right sub-tree
   (b) Number of elements in the left sub-tree + Number of elements in the right sub-tree +1
   (c) Number of elements in the left sub-tree + Number of elements in the right sub-tree −1
   (d) None of the above
3. Which is the successor of an element?
   (a) The leftmost element of the right sub-tree
   (b) The rightmost element of the left sub-tree
   (c) None of the above
   (d) Can be any of the above
4. Which is the maximum element in a BST?
   (a) The leftmost element   (c) Can be any of the above
   (b) The rightmost element   (d) None of the above
5. Which is the minimum element in a BST?
   (a) The leftmost element   (c) Can be any of the above
   (b) The rightmost element   (d) None of the above
6. What is the complexity of searching an element in an unsorted array?
   (a) $O(n)$ (c) $O(n^2)$
   (b) $O(\log n)$ (d) None of the above

7. Which of the following is the best complexity of searching an element in a sorted array?
   (a) $O(n)$ (c) $O(n^2)$
   (b) $O(\log n)$ (d) None of the above

8. Binary search cannot be used in which of the following case?
   (a) When the elements in the array are sorted
   (b) When the elements in the array are unsorted
   (c) In both the cases
   (d) None of the above

9. Which of the following can be used in searching an element from a graph?
   (a) Breadth first search   (c) Both
   (b) Depth first search   (d) None of the above

10. What is the complexity of inserting an element in a BST having n elements?
    (a) $O(n)$ (c) Depends on the type of BST
    (b) $O(\log n)$ (d) None of the above

11. Which of the following is used if deletion from an AVL tree leads to balance factor becoming 2?
    (a) R0   (c) R-1
    (b) R1   (d) All of the above

12. Which of the following is not a valid rotation in the case of deletion from an AVL tree?
    (a) R0   (c) R-1
    (b) R1   (d) R2

13. What is the complexity of insertion in an AVL tree?
    (a) $O(n)$ (c) depends on the case
    (b) $O(\log n)$ (d) None of the above

14. What is the complexity of lookup in an AVL tree?
    (a) $O(n)$ (c) depends on the case
    (b) $O(\log n)$ (d) None of the above

15. Which of the following is a type of BST?
    (a) AVL   (c) 2-3-4
    (b) Red–black   (d) All of the above

II. Review Questions

1. Which is more efficient, linear search and/or binary search?
2. Which data structure is most suitable to string the given values in an ordered fashion?
3. Write an algorithm for deleting a node from a BST.
4. Write an algorithm for inserting a node in a BST.
5. Write an algorithm for finding the image of a BST.
6. Write an algorithm for finding the node having the least value in a BST.

7. Write an algorithm for finding the node having largest value in a BST.
8. Write an algorithm for finding the number of nodes in a BST.
9. Can we have a duplicate value in a BST?
10. Explain deletion of a node from an AVL tree.

III. Application-Based Questions

1. The image of a BST is found as follows. The left sub-tree of the given BST is made the right and the right is made the left recursively. For example, the image of the tree depicted in Fig. A8.21 is depicted in the same figure. Write an algorithm to find the image of BST.
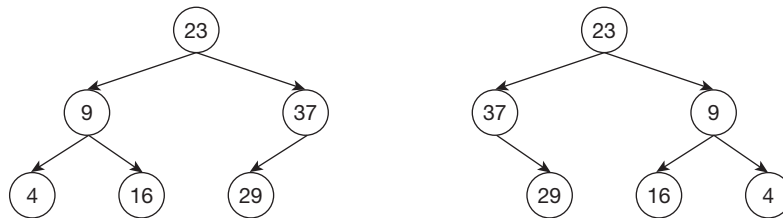
Figure A8.21   A BST and its image

2. Trace the steps of finding the image of the BST depicted in Fig. A8.22
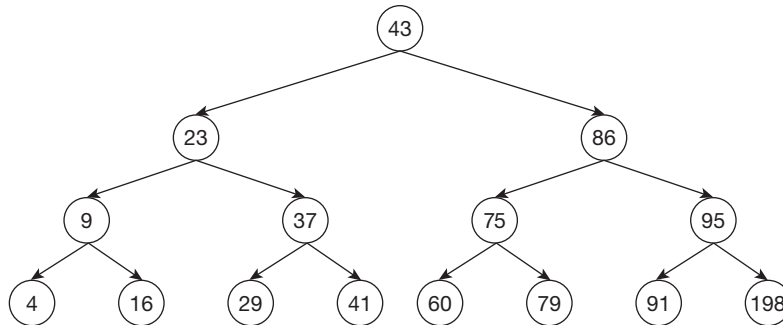
Figure A8.22   Binary search tree for Illustration A8.1

3. In the above question, find the number of pointers that needs to be rearranged in order to carry out the task.
4. In the case where the tree in Question 2 is represented as in an array, suggest a non-recursive procedure to find the image.
5. Can the complexity of the above algorithm reduce by using auxiliary arrays?
6. Design an algorithm to find the number of leaves in a BST.
7. Design an algorithm to find the number of internal nodes in a BST.
8. In the tree of Fig. A8.22 insert the following values (trace the steps of the procedure)
   (a) 81   (c) 19   (e) 32
   (b) 10   (d) 39

9. In the tree of Fig. A8.22 delete the following values (trace the steps)

    (a) 23   (c) 95   (e) 86

    (b) 37   (d) 75

10. In the BST of Fig. A8.22, find the node having the least value.

11. Create an AVL tree from the following values

    (a) 2   (d) 9   (g) 5

    (b) 3   (e) 10   (h) 8

    (c) 6   (f) 1

12. In the AVL tree formed in Question 11, insert the following values

    (a) 4   (b) 7   (c) 12

13. From the final AVL, formed in Question 12, delete the following values

    (a) 1   (b) 10   (c) 6

14. Construct a binary search tree from the values given in Question 11.

15. Perform the additions and deletions of question 12 and 13 from the tree of question 14 and compare the complexity with that of the AVL tree.

Answers to MCQs

1. (c)	4. (b)	7. (b)	10. (c)	13. (b)
2. (b)	5. (a)	8. (b)	11. (d)	14. (b)
3. (a)	6. (a)	9. (c)	12. (d)	15. (b), (c)

APPENDIX   A 9

Analysis of Sorting Algorithms

OBJECTIVES

After studying this appendix, the reader will be able to

- Enlist the characteristics of a sorting algorithm
- Give various scenarios for which sorting algorithm should be analysed
- Carry out lab exercises

A9.1 INTRODUCTION

Sorting is the rearrangement of a given set of elements in order of their keys. A good sorting algorithm should have the following characteristics:

- The keys that are equal should not be reordered. This characteristic makes an algorithm stable.
- The number of swaps in the worst case should be as low as possible. The reader is requested to go through Chapters 8 and 9. It can be inferred from the algorithms that this number should not exceed $O(n)$.
- The sorting algorithm should execute faster when the given list is sorted or almost sorted. This characteristic is referred to as adaptability.
- The number of comparisons should be as low as possible.
- The amount of extra space needed by the algorithm should be as small as possible.

Chapter 8 discussed various linear and quadratic sorting algorithms. The complexities of algorithms discussed in the chapter were either $O(n)$ or $O(n^2)$. Chapter 9 discussed quick sort and merge sort. The worst-case complexity of quick sort is $O(n^2)$ and the average-case complexity is $O(n \log n)$. In the case of merge sort, the complexity is $O(n \log n)$. The complexity of various algorithms has been summarized in Chapter 8. The worst-case complexity is not the only deciding criteria for an algorithm. There are many factors like the behaviour when the array is almost sorted, and so on. However, the reader is expected to understand the advantages of each algorithm to be able to select the appropriate algorithm in a situation.

An algorithm should be analysed for the following cases in order to access it:

- Sorted
- Sorted reverse
- Random
- Repeated values

The reader is expected to perform the above analysis for the following lab exercises. The data of results of their execution are also given (on the machines stated in the following exercises). The reader should answer the following questions by analysing the given data.

A9.2 LAB 1: QUICK SORT

A9.2.1 Goal: Implement and Analyse Quick Sort for Small Input Size (Exactly Reverse of What We Should Have Done)

The implementation of quick sort in an Intel i3 machine with a 4GB RAM, 500 GB hard disk (Windows 8.1 Operating System), the following results were obtained when the input was sorted. The list to be sorted contains 75 numbers. The following snapshot shows the time elapsed in 4 executions of the program:

The following snapshot shows the results of the execution of quick sort when the list is reversed:

The following snapshot shows the results of the execution of the program when the list is random:


```
F:\>java QuickSort1
1 2 3 4 5 6 7 8 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 3
1 32 33 34 35 36 37 38 39 40 41 42 43 44 45 47 48 49 50 51 52 53 54 55 57 58 59
60 61 62 63 64 65 65 66 67 68 69 70 71 72 73 74 75 Elaspsed Time=15

F:\>java QuickSort1
1 2 3 4 5 6 7 8 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 3
1 32 33 34 35 36 37 38 39 40 41 42 43 44 45 47 48 49 50 51 52 53 54 55 57 58 59
60 61 62 63 64 65 65 66 67 68 69 70 71 72 73 74 75 Elaspsed Time=31

F:\>java QuickSort1
1 2 3 4 5 6 7 8 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 3
1 32 33 34 35 36 37 38 39 40 41 42 43 44 45 47 48 49 50 51 52 53 54 55 57 58 59
60 61 62 63 64 65 65 66 67 68 69 70 71 72 73 74 75 Elaspsed Time=16

F:\>java QuickSort1
1 2 3 4 5 6 7 8 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 3
1 32 33 34 35 36 37 38 39 40 41 42 43 44 45 47 48 49 50 51 52 53 54 55 57 58 59
60 61 62 63 64 65 65 66 67 68 69 70 71 72 73 74 75 Elaspsed Time=15
```

The time is not just that required in the linking and loading of a program. A compiled program becomes a process. The process is put in the scheduling queue by the operating system. As per the policy and the number of programs executing at that time, the process is allocated the processor. The total time of execution, therefore, consists of all the above things and can therefore vary greatly. Moreover, in the implementation, the running time was calculated in milliseconds, which could have been a source of error. However, the above data gives a good idea of how the average running time varies by changing the ethos of the input.

A9.2.2 Related Problems

1. Analyse the above results and discuss whether quick sort is really better than any of the $O(n^2)$ algorithm a small input size.
2. What do you think is the reason of getting such variations in the result?
3. Run the above algorithm for an input size of 1000, 3000, 5000, 10,000, and 20,000 (5 runs in each case), report the results, and analyse them.
4. Find the average and standard deviation of the 5 trials (in each case) and plot their graph with the input size.
5. Check if the results would be different, if an i5 is used in the above case?

A9.3 LAB 2: SELECTION SORT

A9.3.1 Goal: Implement and Analyse Selection Sort

The asymptotic complexity gives an idea of the variation of the time with the size of the input, but it cannot compare the algorithms with the same asymptotic complexity. For example, the asymptotic complexity of both selection sort and bubble sort is $O(n^2)$; however, selection sort performs better than bubble sort.

Table A9.1 presents the running time of standard selection time program in C on a Pentium(R), 1.8 GHz machine with a 64-bit OS (Windows 8), 500 GB hard disk.

Table A9.1 Results of execution of selection sort

Input size	Experiments					Sum	Average	Standard deviation
	1	2	3	4	5			
500	0	54.945	54.945	0	0	109.89	21.978	30.09462
1000	0	100	109.89	109.89	109.89	429.67	85.934	48.22907
1500	274.725	219.78	219.78	219.78	219.78	1153.845	230.769	24.57215
2000	329.67	329.67	329.67	384.615	329.67	1703.295	340.659	24.57215
2500	494.505	494.505	494.505	494.505	494.505	2472.525	494.505	0
3000	569.349	604.349	659.349	659.349	659.349	3151.745	630.349	41.59327
3500	824.175	824.175	824.175	769.23	824.175	4065.93	813.186	24.57215
4000	989.01	1043.956	989.01	989.01	989.01	4999.996	999.9992	24.5726
4500	1263.736	1263.736	1263.736	1263.736	1263.791	6318.735	1263.747	0.024597
5000	1483.516	1483.516	1483.516	1483.516	1483.516	7417.58	1483.516	0
8000	3571.428	3571.428	3571.428	3571.483	3571.428	17857.2	3571.439	0.024597
9000	4450.549	4450.549	4450.494	4450.494	4450.549	22252.64	4450.527	0.030125
9500	4945.055	5000	4945.055	4945.055	4945.055	24780.22	4956.044	24.57215
10,000	5439.348	5604.395	5494.505	5494.505	5494.505	27527.26	5505.452	60.24734

The corresponding graph of the average and the standard deviation of the running time with the input size is as follows (Fig. A9.1).
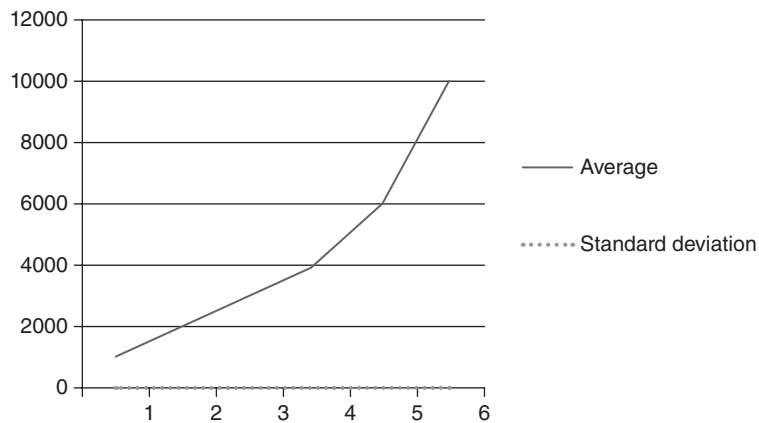
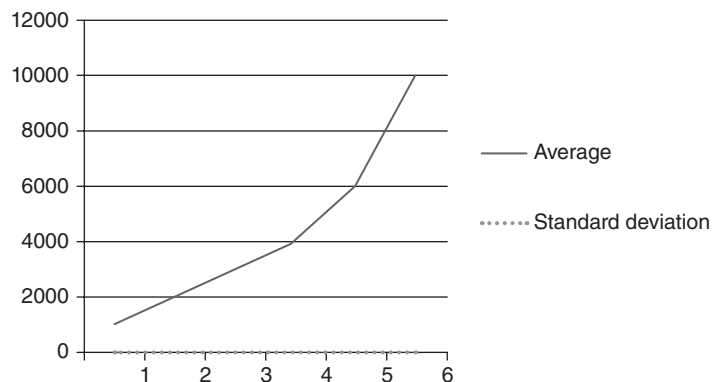**Figure A9.1** Variation of average and standard deviation of running time with the input size in the case of selection sort

A9.3.2 Related Problems

1. Analyse the above results and discuss whether selection sort is really better than other $O(n^2)$ algorithms?
2. What do you think is the reason of getting such variations in the result?

3. Run the above algorithm for an input size of 20,000 and 30,000 (5 runs in each case), report the results, and analyse them.
4. Find the average and standard deviation of the 5 trials (in each case) and plot their graph with the input size.
5. Check if the results would be different, if an Intel i5 is used in the above case?
6. Implement the version of selection sort that selects the smallest element by divide and conquer and then replace it with the element at the ith position for i varying from 0 to $n - 2$.

A9.4 LAB 3: INSERTION SORT

A9.4.1 Goal: Implement and Analyse Insertion Sort

Table A9.2 presents the running time of standard Insertion Sort program in C on a Pentium(R), 1.8 GHz machine with a 64-bit OS (Windows 8), 500 GB hard disk.

Table A9.2 Results of execution of insertion sort

Input size	Experiments					Sum	Average	Standard deviation
	1	2	3	4	5			
1000	0.274	0.219	0.274	0.164	0.274	1.205	0.241	0.049193
2000	1	0.659	0.659	0.549	0.659	3.526	0.7052	0.171544
3000	1.208	1.318	1.153	1.318	1.318	6.315	1.263	0.077782
4000	2.252	2.252	2.307	2.197	2.307	11.315	2.263	0.046016
5000	4.835	4.835	4.89	4.725	4.835	24.12	4.824	0.060249
8000	8.241	8.351	8.461	8.186	8.274	41.513	8.3026	0.106819
10,000	13.131	12.857	13.296	13.076	13.912	66.272	13.2544	0.399734

The corresponding graph of the average and the standard deviation of the running time with the input size is as follows (Fig. A9.2).

Figure A9.2 Variation of average and standard deviation of running time with the input size in the case of insertion sort.

A9.4.2 Related Problems

1. Analyse the above results and discuss whether insertion sort is really better than other $O(n^2)$ algorithms?
2. What do you think is the reason of getting such variations in the result?
3. Run the above algorithm for an input size of 20,000 and 30,000 (5 runs in each case), report the results, and analyse them.
4. Find the average and standard deviation of the 5 trials (in each case) and plot their graph with the input size.
5. Check if the results would be different, if an Intel i5 is used in the above case?
6. Implement the version of insertion sort that inserts a sub-list in a sorted list in an adaptive manner.

A9.5 LAB 4: BUBBLE SORT

A9.5.1 Goal: Implement and Analyse Bubble Sort

Table A9.3 presents the running time of standard bubble sort program in C on a Pentium(R), 1.8 GHz machine with a 64 bit OS (Windows 8), and 500 GB hard disk.

Table A9.3 Results of execution of insertion sort

Input size	Experiments				Sum	Average	Standard deviation
	1	2	3	4			
1000	0.164	0.164	0.109	0.109	0.546	0.1365	0.031754
2000	0.384	0.329	0.329	0.384	1.426	0.3565	0.031754
3000	0.549	0.604	0.549	0.659	2.361	0.59025	0.052658
4000	1.043	0.934	0.879	0.934	3.79	0.9475	0.068743
6000	1.758	1.813	1.978	1.813	7.362	1.8405	0.095263
10,000	4.45	4.45	4.395	4.395	17.69	4.4225	0.031754
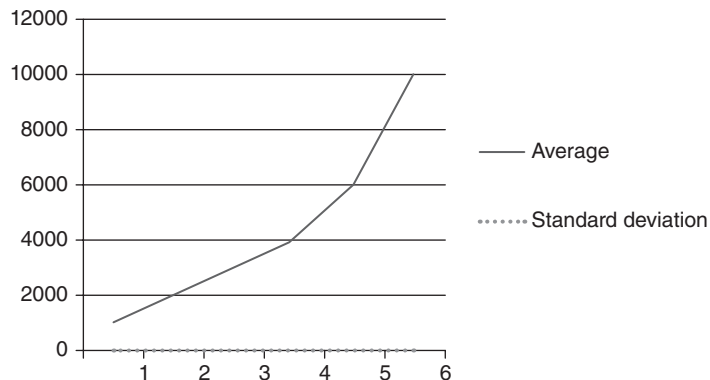
Figure A9.3 Variation of average and standard deviation with the input size in case of insertion sort.

The corresponding graph (Fig. A9.3) of the average and the standard deviation of the running time with the input size is as follows.

A9.5.2 Related Problems

1. Analyse the above results and discuss whether bubble sort is really better than other $O(n^2)$ algorithms.
2. What do you think is the reason of getting such variations in the result?
3. Run the above algorithm for an input size of 20,000 and 30,000 (5 runs in each case), report the results, and analyse them.
4. Find the average and standard deviation of the 5 trials (in each case) and plot their graph with the input size.
5. Check if the results would be different, if an Intel i5 is used in the above case?

A9.6 PROBLEMS BASED ON SORTING

Question 1: There are n letters and corresponding envelopes. Each letter has a unique identification number (which is an integer). The envelopes are also marked with the same identification numbers. Now each letter is to be inserted into the correct envelope. Develop an algorithm to accomplish the task. Analyse the time complexity of the algorithm.

Solution 1: The easiest approach would be to select a letter and find the correct envelope for it. Since, there are n envelopes, finding the correct envelope of the first letter would require n comparisons. The process would require $(n - 1)$ comparisons for the second letter and so on. The total number of comparisons in this fashion would be $(n + (n - 1) + (n - 2) \ldots 1)$, that is, $n(n - 1)/2$. Hence, $O(n^2)$ comparisons would be required to accomplish the task. The above approach is the brute force approach, as every possible permutation is being checked.

Solution 2: The second approach is to sort the letters (and envelopes in order of their identification numbers) and then carry out the required task (of putting the letters in the correct envelopes) in $O(n)$ time. The sorting may be carried out using an efficient method like heapsort. This would result in the accomplishment of the task in $O(n \log n)$ time as against $O(n^2)$, in the previous example.

Solution 3: If the identification numbers are between 1 and some finite (not very large) number, then the task can be accomplished using counting sort. This reduces the complexity to $O(n)$. The letters (and the envelopes) are sorted using counting sort. This is followed by the insertion of the requisite letter in the corresponding envelope.

Question 2: CTU, a deemed university in the capital of some developing country, holds elections for the president for its student union. There are three candidates in the fray. Each candidate has been assigned an ID. The students enter the ID of the candidate he/she wishes to vote for. The votes are stored in an array. The problem is to find out which candidate wins.

Solution: The problem can be tackled using counting sort. The input array contains the three IDs repeated unknown number of times. An array B can be made having three locations. The following algorithm would achieve the above task.

Algorithm A9.1 Winner

Input: A
Output: The ID of the candidate who wins

```
Winner (A[], ID1, ID2, ID3)
{
B[0]=B[1]=B[2]=0;
for (i=0; i<n ; i++)
        {
        if (A[i]==ID1)
                B[0]++;
        else if(A[i]==ID2)
                B[1]++;
        else if(A[i]==ID3)
                B[2]++;
        }
if (B[0]>B[1])&&(B[0]>B[2])
        {
        return ID1;
        }
else if((B[1]>B[0]) &&(B[1]>B[2]))
        {
        return ID2;
        }
else
        {
        return ID3;
        }
}
```

EXERCISES

I. Multiple Choice Questions

1. If an input is placed at its appropriate position in a binary search tree, then traversing the tree in in-order is
    (a) Tree sort
    (c) Insertion sort
    (b) Selection sort
    (d) None of the above
2. What is the worst-case complexity of selection sort?
    (a) $O(n^2)$
    (c) $O(n)$
    (b) $O(n \log n)$
    (d) None of the above

3. In a version of selection sort that selects the smallest element by divide and conquer and then replaces it with the element at the ith position for i varying from 0 to $n - 2$, the complexity is
   (a) $O(n2)$ (c) $O(n)$
   (b) O(n log n)   (d) None of the above

4. Which of the following performs better, in general?
   (a) Selection sort   (c) Both are equally good
   (b) Bubble sort   (d) Cannot say

5. What is the best-case complexity of merge sort?
   (a) $O(n \log n)$ (c) $O(n)$
   (b) $O(n^2)$ (d) None of the above

6. Which of the following is not stable?
   (a) Bubble sort   (c) Selection sort
   (b) Insertion sort   (d) All of the above

7. Which of the following is stable?
   (a) Merge sort   (c) Both
   (b) Heapsort   (d) None of the above

8. Which of the following requires auxiliary memory $O(\log n)$
   (a) Quick sort   (c) Selection sort
   (b) Bubble sort   (d) None of the above

9. Which of the following is true for a randomized quick sort?
   (a) The input is randomized   (c) Any of the above
   (b) The pivot is selected randomly   (d) None of the above

10. Merge sort can be implemented
    (a) By using recursion   (c) Any of the above
    (b) By iteration   (d) None of the above

Answers to MCQs

1. (a)	3. (b)	5. (a)	7. (a)	9. (c)
2. (a)	4. (b)	6. (c)	8. (d)	10. (c)

APPENDIX A10

Problems

A10.1 INTRODUCTION

The appendix has been designed to test the skills of the reader. The problems cover almost all the major topics in this book. Some of the problems given in this appendix have been solved in more than one way. The reason for doing so is to encourage the reader to develop more than one algorithm for a given problem and choose the best after careful consideration and analysis. The reader is expected to go through the problems, develop his own solution, and then analyse them. Some of the solutions to the problems given below are not absolute (e.g. Problem 10.4). The reader must also develop algorithms for the unsolved problems and implement them. The given algorithms must also be implemented and tested for all possible classes of inputs. The analysis done by implementing and running an algorithm would also help in understanding the hardware and operating system issues.

A10.2 PROBLEMS

A10.2.1 To Design an $O(n)$ Algorithm to Find the nth Fibonacci Term

Solution: The first two terms of Fibonacci series are 1 and 1. Any other term can be obtained by adding the previous two terms. The recursive version of the problem was introduced in Section 4.1 of Chapter 4. The complexity of the recursive Fibonacci series is $O(2^n)$. Algorithm A10.1 finds the nth Fibonacci term in $O(n)$ time.

Algorithm A10.1 Fibonacci_term

Input: n, the index
Output: a[n-1], the nth Fibonacci term
```
Fibonacci_term( int n)
// a[] is an array
{
a[0] = 1;
```

```
a[1] = 1;
for(i = 2; i<n; i++)
    {
    a[i] = a[i-1]+a[i-2];
    }
//print the nth Fibonacci term
Print: a[n-1];
}
```

Complexity: $O(n)$, as the algorithm has a single loop. The space complexity of the algorithm is also $O(n)$ as it uses an array of length n.

The above algorithm is better than the recursive version as it takes lesser time. The above algorithm is also an excellent example of how a simple change can reduce the complexity of the algorithm, both in terms of time and space.

Related Questions

1. Write a program in C to implement the above algorithm and note the time of execution if the value of *n* is

    (a) 5   (d) 1000
    (b) 10   (e) 10,000
    (c) 100

2. Compare the running time with the running time of the algorithm to calculate the *n*th Fibonacci term using recursion.

3. Compare the space complexity of the two versions (with and without recursion).

A10.2.2 To Find Whether a Strictly Binary Tree is a Heap

Solution: In Section 6.3 of Chapter 6, it was stated that if a node is stored at index '*i*' of an array, then the left child of the node would be stored at '$2i + 1$', and the right child would be stored at '$2i + 2$'. In Algorithm A10.2, the array is traversed from $i = 0$ to $i = n/2$, at each position if the values stored at the indices '$2i + 1$' and '$2i + 2$' are less than that at '*i*', then the tree is a heap.

Algorithm A10.2   Check (a, n)

Input: a, an array in which the tree is stored, n is the number of elements in the array
Output: I the tree is a heap, "Heap" is printed, otherwise the algorithm prints "Not a heap"
```
Check (a, n)
{
FLAG = 0;
for( i=0; i<n/2; i++)
    {
    if ((a[2i + 1] < a[i])&&(a[2i + 2] < a[i]))
```

```
        {
        }
    else
        {
        FLAG = 1;
        break; // this takes the control out of the loop
        }
    }
if (FLAG == 0)
    {
    Print: " Heap";
    }
else
    {
    Print: "Not a heap";
    }
}
```

Complexity: Since the algorithm has a single loop, the complexity is $O(n)$.

Extrapolate: The reader is encouraged to write this algorithm for a tree stored in a doubly linked list. If ptr is a leaf node of the tree, then ptr->left= NULL and ptr->right->NULL. In all other cases, the data at ptr (ptr->data) should be greater than the data at (ptr->left) and that at (ptr->right).

Related Questions

1. Write an algorithm to find the minimum element from a Max-heap.
2. Write an algorithm to find the maximum element from a Min-heap.
3. Write an algorithm to find the sum of elements in a heap.
4. Write an algorithm to find whether a given strictly binary tree is a binary search tree.
5. Write an algorithm to find the sum of elements of a binary search tree.

A10.2.3 To Develop an $O(n)$ Algorithm to Sort Numbers Between 0 to ($n-1$)

The algorithm uses an array of size n. If an item is placed at the index denoted by the item itself, the array would then contain elements in the sorted order.

Algorithm A10.3 uses an array of length n. The getItem() function gives an incoming element. The function returns a NULL when there is no input.

Algorithm A10.3   Efficient sorting

Input: n, the number of items
Output: The array a[] would contain elements in the sorted order, after the algorithm has been executed.

```
Sort_n(n)
{
```

```
for (i=0; i<n; i++)
    {
    a[i] = -1;// Here (-1) denotes a NULL
    }
while(True)
    {
    item = getItem();
    if (item != NULL)
        {
        a[item]= item;
        }
    else
        {
        break;
        //the statement takes the control out of the loop
        }
    }
}
```

Complexity: The loop runs x ($\leq n$) times. The complexity is, therefore, $O(n)$.

The algorithm, though good in terms of time complexity has a grave limitation. If the number of items is too large, then it would not be advisable to use the above algorithm. The problem is the same as that faced in the case of sequential search.

Related Questions

1. In the above problem, consider that the number of elements is ($n/2$) and the elements are in the range $0 - (n - 1)$. How would you modify the above algorithm to that having better space complexity? (probably on the expense of time complexity).
2. In the above problem, how would you search for a given number in $\ln(n)$ time?
3. Modify the algorithm to compress the array, so that there is no (-1) (or NULL) in the modified array.

A10.3   DIVISION OF A LIST INTO TWO PARTS WHOSE SUM HAS MINIMUM DIFFERENCE ■

Problem: A list of n numbers is given. It is desired to separate the numbers into two halves such that if the sum of elements of the first half is S_1 and that of the second half is S_2, then $|S_1 - S_2|$ is minimum. Assume that the number of elements, n, is a multiple of 2.

Discussion: The question is tricky. As a matter of fact, an exact solution would require brute force analysis, which is computationally very expensive. So, let us look at some of the solutions that would help us to accomplish the task (almost). The first solution presented as follows, though easy to understand, does not scale well in many cases.

Solution: Put the first two elements at $a[0]$ and $a[n/2]$ positions. Initially, the sum of the elements of the left sub-array would be $S1 = a[0]$ and $S2 = a[n/2]$. For each element ($=x$)

that comes, find whether |*S1* + x – *S2*| is lesser than |*S1* – x – *S2*|. If it is the case, then put *x* in the first sub-array, otherwise in the second.

Algorithm A10.4   Minimum_Difference (a)

Input: a, the given array and n, the number of elements
Output: The array which contains elements such that the difference of the sums of the sub arrays on the left and right is minimum.

```
Minimum_Differnce(n)
    {
    a[0]= getItem();
    a[n/2]=getItem();
    i=0;
    j=n/2;
    S1=a[i];
    S2=a[j];
    while(True)
        {
        int item=getItem();
        if(item != NULL)
            {
            if(|S1+x –S2| <= |S1 – x- S2|)
                {
                i++;
                a[i]=x;
                S1+=x;
                }
            else
                {
                j++;
                a[j]=x;
                S2+= x;
                }
            }
        else
            {
            break;
            }
        }
    }
```

Complexity: The algorithm contains a loop which runs *n* times; hence, the complexity of the algorithm is *O(n)*.

Analysis: When all the elements are equal, the algorithm delivers. However, in other cases, it does not perform that well. In order to understand this, try solving the following questions.

Related Questions

1. The following series of number is given as an input to the above algorithm.

$$1, 2, 3, 4, \ldots, n$$

Implement the above algorithm in C and observe the result for the following values of n

   (a) 5   (c) 100

   (b) 10   (d) 1000

2. In the above question, if the series is changed to 3, 100, 6, 98, 9, 96, … ,observe that the odd terms have common difference (−2) and the even terms have common difference (3). Run the program for the following values of n.

   (a) 5   (c) 15

   (b) 10   (d) 25

3. Now, give random numbers as input to the above program and observe the result for at least 10 trials (take $n = 100$), and report the result.

4. Suggest another algorithm to solve the above problem which gives a better result.

A10.4 COMPLEXITY-RELATED PROBLEMS

What is the complexity of the following?

```
for( i = 0; i<n; i++)
    {
    for( j=n; j>=1; j/=2)
        {
        for(k=n; k>=1; k=k/4)
            {
            //body requires θ(1) time
            }
        }
    }
```

Complexity: The innermost loop runs $\log_4 n$ times. Initially, the value of k is n, it becomes $k/4$ next time, and so on. When $\left(\dfrac{n}{4}\right)^i = 1$ that is, $\log_4 n = i$. The second loop runs $\log_2 n$ times. The outermost loop runs $O(n)$ times. The complexity of the nested loop is, therefore, $O(\log_2 n \times \log_4 n \times n)$ or $O(n(\log n)^2)$.

Related Problems

1. If the comments inside the loops are replaced by the following statement
    print: (i*j*k);
    (a) What would be the output?
    (b) Can you accomplish the task accomplished by the above code using a method having lesser complexity?

2. A computer takes 5 minutes to solve an instance of longest common subsequence discussed in Chapter 11. The computer is then replaced with another which is 100 times faster than the previous one. What would be the time taken to solve the same problem if the complexity of the problem is $O(n^3)$?

3. In the above question what would be the time taken if the complexity is $O(2^n)$?

4. What is meant by $O(n^{O(n)})$? If an algorithm has complexity $O(n^{O(1)})$, is it better than that having complexity $O(n^{O(n)})$?

5. Which is better—an algorithm having complexity log n! or that having complexity $O((\log n)^2)$?

A10.5 ALGORITHM TO STORE SUBSETS HAVING TWO ELEMENTS

A set of n elements is given. The problem is to find all possible subsets having two elements. Algorithm A10.5 stores the result in a 2-dimensional array b, having two columns and $n(n − 1)/2$ rows. The first column stores the first element of the subset and the second column stores the second element. The number of rows in b is ${}_2^n C$, that is, $\dfrac{n(n-1)}{2}$.

The index k, which is initially 0, increments at each iteration. The indices i and j are the counters of the two loops.

Algorithm A10.5   Subset 2

Input: a, the array which contains the elements of the given set; n, the number of elements in the set
Output: b, the two-dimensional array containing the subsets

```
Subset2(a, n)
{
k=0;
for( i=0; i<n;i++)
    {
    for(j=(i+1); j<n; j++)
        {
        b[0][k]=a[i];
        b[1][k]=a[j];
        k++;
        }
    }
return b;
}
```

Complexity: The inner loop runs n times and the inner runs for $(n − 1 − i)$ times. The total number of runs would be

$$(n-1)+(n-2)+(n-3)+\cdots+1 = n \times \frac{n-1}{2}$$

Thus, the complexity is $O(n^2)$.

Note: The above approach is easy to understand and implement. However, had the number of items in each subset been three, the complexity would have become $O(n^3)$, had the number been four, the complexity would have been $O(n^4)$, and so on. In order to understand the concept, the reader may solve the questions that follow.

Related Questions

1. Design an algorithm that prints all the subsets, of a set having n elements, which have three elements.
2. In the above questions had the number of elements in each subset is m.
3. Find the complexity of all the subsets of a given set.

A10.6 DIVIDE AND CONQUER

Divide and conquer divides the given problem into sub-problems, solves the sub-problems and then combines the results to give the final result. The approach, in general, reduces the time complexity. The topic has been discussed in Chapter 9. However, the approach cannot be used always used. In the cases where the division of the problem of size n leads to sub-problems of size which is almost n, divide and conquer should not be used. In addition, in the cases where divide and conquer leads to n problems of size (n/b), the technique should not be used.

A10.6.1 Non-recursive Binary Search

A sorted array 'a' is given. The first index of 'a' is 'low' the last index is 'high'. An element 'item' is to be found in the array using the concept of divide and conquer. The problem has already been discussed in Section 9.2 of Chapter 9. Algorithm A10.6 presents the non-recursive version of binary search.

Algorithm A10.6 Non-recursive binary search

Input: a, array containing elements from index low to high. The item to be searched is 'item'.
Output: The algorithm prints "Found", if 'item' is found in the array 'a', otherwise it prints "Not Found".

```
Binary_Search_Non_Recursive (a, item, low, high)
{
low = 0;
high = n-1;
While (low==high)
    {
    mid= (low + high)/2;
```

```
    if (a [low] == item)
        {
        Print: "Found";
        break;
        }
    else if( a[high] == item)
        {
        Print: "Found";
        break;
        }
    else if( a[mid] ==item)
        {
        Print: "Found";
        break;
        }
    else if(item < a[mid])
        {
        low= low+1;
        high = mid -1;
        }
    else if( item> a[mid])
        {
        low= mid +1;
        high =- high -1;
        }
    else
        {
        Print: "Not Found";
        }
    }
}
```

Complexity: The complexity of the algorithm is $O(\log n)$.

The last element of each row is highest in the row. So, we apply binary search in the last row. If 'item' is found in the last row, then the algorithm prints "Found", otherwise the search is directed to the row, identified by the single element that crops up in the search. Binary_Search is then applied to that row.

A10.6.2 Binary Search in a 2-Dimensional array

Algorithm A10.7 Binary_Search_2D

Input:
1. a[n, n], a two dimensional array containing n rows and n columns, where n is a power of 2.
2. 'item', which is to be searched in the above array.

Output: The algorithm prints "Found", when the element (item) is found in the array otherwise it prints "Not Found".

Condition: The given array is sorted such that an element in a particular column in ith row is less than the element in that column in the jth row, where $j > i$. In addition, an element in the ith column of a particular row is less than the item in the jth column of the same row if $j > i$.

Example:

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

```
Binary_Search_2D(a,0,n-1, 0, m-1)
//The arguments denotes the respective indices of rows and columns
    {
    if((n==1)&&(m==1))
        {
        if (a[0,0]==item)
            {
            Print: "Found";
            exit();
            //exit() terminates the program
            }
        }
    else if(n==1)
        {
        //copy the elements of a to a single dimensional array b
        Binary_Search( b, o, n-1);
        }
    else
        {
        if (item == a[0, n-1])
            {
            Print: "Found";
            exit();
            }
        else if(item == a[n-1, n-1])
            {
            Print: "Found";
            exit();
            }
        else if(item == a[n/2, n-1];
```

```
        {
        Print: "Found";
        exit();
        }
    else if(item < a[n/2, n-1, 0, n-1])
        {
        Binary_Search_2D(a, 0, n/2 -1, o, n-1];
        }
    else if (item > a[n/2, n-1])
        {
        Binary_Search(a, n/2-1, n-1, o, n-1])
        }
    }
}
```

Complexity: The algorithm applies binary search in the last column, which takes $O(\log n)$ time in the row selected by the previous algorithm, which again takes $O(\log n)$ time. Hence, the complexity of the algorithm is $O(\log n)$.

A10.6.3 Complexity of Divide and Conquer

Note: The reader is advised to go through Master theorem (Chapter 9) before attempting the following problems.

Problem: A problem is solved using divide and conquer such that each time it is divided into 8 problems of size ($n/2$). The cost of combining the results of the sub-problems is $O(n^2)$. What is the complexity of the algorithm?

The recursive equation representing the complexity of the above is

$$T(n) = 8 \times T\left(\frac{n}{2}\right) + O(n^2)$$

Here, the value of 'a' (see Master theorem, Chapter 9) is 8 and that of 'b' is 2. The value of $n^{\log_b a} = n^{\log_b 8} = n^3$. Since the value of $f(n)$ is $O(n^2)$, the complexity of the algorithm is $O(n^3)$.

Related Questions

1. Design a search algorithm that divides a sorted array into four parts. If the element is found at low, high, mid1, mid2, or mid3, then the algorithm should print "Found", otherwise it should select the appropriate sub-array and continue the search in that array.
2. What would be the complexity of the above algorithm?
3. What is the problem if the number of divisions in search is too high (as in binary search divides array into four parts, the above problem divides it into four parts, and so on).
4. Design a variant of merge sort wherein the array is divided into m parts instead of 2. Each of the m sub-arrays should have (n/m) elements.

5. What is the complexity of the algorithm developed in Question 4.
6. Write a non-recursive algorithm for merge sort.
7. Write a non-recursive algorithm for quick sort.
8. Can the algorithm given in Section A10.7.2 be modified, so that each call of the algorithm results in the application of the algorithm in a $\dfrac{n}{2} \times \dfrac{n}{2}$ array?
9. Can the algorithm given in Section A10.7.1 be applied to an array in which the elements are not ordered?
10. If $T(n) = 27T\left(\dfrac{n}{3}\right) + n^3$, $nT(n) = 27T\left(\dfrac{n}{3}\right) + n^3$, $n > 1$ and $T(1) = 1$, find $T(243)$.

A10.7 APPLICATIONS OF DYNAMIC PROGRAMMING

Problem 1: A sequence is such that any term is the product of the previous two terms. The first two terms of the sequence are 1 and 2. Find the sequence and analyse the algorithm.

Solution 1: The problem is similar to Fibonacci series. The solution can be easily obtained using recursion. Algorithm A10.8 presents a recursion-based solution to the above problem.

Algorithm A10.8   Problem1_algo

Input: a[1] = 1 and a[2] = 2, a[] being the array in which the terms would be stored.
Output: a[], the array containing the desired sequence.

```
//a[] is a global array, a[0] = 1 and a[1] = 2, n is the number of elements in
the sequence;
//Prob (int) calculates the ith term of the sequence and Main calls Prob
recursively
Algorithm Prob ( int i) returns int
{
if (i==1)
    return 1;
else if (i==2)
    return 2;
else
    return (Prob(i-1) + Prob (i-2));
}
Main returns a[]
{
for (i=0; i<n; i++)
    {
    a[i] = Prod(i);
    }
return a;
}
```

Analysis: The complexity of the algorithm can be calculated as follows. The calculation of the nth term, in this case, would require the evaluation of the $(n - 1)$ and the $(n - 2)$th term. The time complexity can be stated as follows.

$$T(n) = T(n - 1) + T(n - 2), T(1) = 1 \text{ and } T(2) = 2.$$

Notice that the equation is similar to that obtained in the Fibonacci series. The complexity is therefore near to $O(2^n)$.

Solution 2: The other approach, using the concept of bottom-up approach of the dynamic programming, accomplishes the above task in $O(n)$ time. The approach is as follows.

```
Prob ()
{
a[0] = 1;
a[1] = 2;
for (i=2; i<n; i++)
    {
    a[i] = a[i-1] * a[i-2];
    }
return a;
}
```

Analysis: The above algorithm uses a single loop and hence the complexity is $O(n)$.

Problem 2: Given a list of n numbers, find the sum of the sub-list of neighbouring elements having maximum sum.

Solution 1: The first approach uses a brute force method to accomplish the above task. The approach requires $O(n^3)$ time. The algorithm is as follows.

Algorithm A10.9 Neighbouring_sub_list_sum

Input: a[], the list and n, the number of elements in the list
Output: integer which is the sum of the sub-list of neighbouring elements having maximum sum

```
MaximumSumOfNeighbours(a[], n)
//a[] is an array having n elements
{
/*sum[] is an array containing all the possible sums
t=0;
for(k=0; k<n; k++)
{
for(i=k; i<n; i++)
    {
    sum[t]=0;
    for(j=0; j<=i; j++)
        {
```

```
            sum[t]+=a[j];
            }
        t++;
        }
    }
    Max = sum[0];
    for(i=1; i<t; i++)
        {
        if(a[t]>Max)
            {
            Max = a[t];
            }
        }
    return Max;
    }
```

Analysis: The above algorithm has three levels of nesting (of loops). The complexity is therefore $O(n^3)$.

Solution 2: The other approach uses the concept of longest common subsequence, explained in Section 11.3 of Chapter 11. The reader is expected to develop the algorithm using the approach.

Related Questions

1. In the above problem if the sum of all the terms is to be calculated, what would be the complexity?
2. A sequence is such that any term is 9 times the product of the previous three terms. The first three terms of the sequence are 1, 2, and 3. Find the sequence using recursion and analyse the algorithm.
3. Solve the above problem using dynamic approach.
4. Given a list of n numbers, find the sum of the sub-list having maximum sum (note that the elements need not to be contiguous).
5. Given a list of n numbers, find the sum of the sub-list of neighbouring elements having minimum sum.
6. Given a list of n numbers, find the sum of the sub-list having maximum product.
7. Given a list of n numbers, find the sum of the sub-list of neighbouring elements having maximum product.
8. Can an array be divided into two parts such that the sum of the two parts is same?
9. Can an array be divided into two parts such that the product of the two parts is same?
10. Develop an algorithm that finds the sub-list having maximum median.

Note: For problems on NP and number theoretic, refer to the web resources of this book.

Bibliography

BOOKS

Arora, S., Barak, B., *Computational Complexity: A Modern Approach*, Cambridge University Press, 2009.

Attwood, Parry Smith, Phukan, *Introduction to Bioinformatics*, Pearson, 2009.

Bishop, C.M., *Neural Networks for Pattern Recognition*, Oxford University Press, Oxford, England, 1995.

Bishop, C.M., *Pattern Recognition and Machine Learning*, Springer-Verlag, New York, 2008.

Brigham, E.O., *The Fast Fourier Transform and its Applications*, Prentice Hall, Englewood Cliffs, NJ, 1988.

Cormen, L., Rivest, S., *Introduction to Algorithms*, Second Edition, Prentice Hall of India, 1990.

Dantzig, G.B., *Linear Programming and Extensions*, Princeton University Press, 1963.

Dave and Dave, *Design and Analysis of Algorithms*, Pearson, 2008.

David, G., *The Theory of Linear Economic Models*, McGraw-Hill, 1960.

Duhamel, P., B. Piron, and J.M. Etcheto, *On Computing the Inverse DFT*, IEEE Trans. Acoust., Speech and Sig. Processing 36(2), 285–286, 1988.

Goldberg, D.E., *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, Reading, MA, 1989.

Goldberg, D.E., *The Design of Innovation: Lessons from and for Competent Genetic Algorithms*, Addison-Wesley, Reading, MA, 2002.

Horowitz, et al., *Algorithms*, Second Edition, University Press, 2007.

Jones and Pevzner, *An Introduction to Bioinformatics Algorithms*, MIT Press, 2004.

Kanitkar, *Data Structures Through C*, BPB Publications, 2003.

Kleinberg, T., *Algorithm Design*, Pearson, 2011.

Knuth, D., *Sorting and Searching: The Art of Computer Programming*, Vol. 3, Second ed., Addison–Wesley, 1998.

Levitin, *Introduction to Design and Analysis of Algorithms*, Perason, 2009.

Motwani, R. and P. Raghavan, *Randomized Algorithms*, Cambridge University Press, New York (NY), 1995.

Neapolitan, N., *Foundations of Algorithms*, Fourth Edition, Jones & Barlett, 2013.

Oppenheim, A.V., R.W. Schafer, and J.R. Buck, *Discrete-time Signal Processing*, Upper Saddle River, NJ, Prentice Hall, 1999.

Papadimitriou, C., *Computational Complexity: Polynomial Space*, 1st ed., Chapter 19, Addison Wesley, pp. 455–490, 1993.

Papadimitriou, C., *Computational Complexity: Randomized Computation*, 1st ed., Chapter 11, Addison Wesley, 1993, pp. 241–278.

Rich, E., K. Knight, *Artificial Intelligence*, McGraw-Hill, 1991.

Russell, S.J. and P. Norvig, *Artificial Intelligence: A Modern Approach*, 2nd ed., Pearson Education, 2003.

Sahini, H., *Fundamentals of Data Structures*, Galgotia Booksource, 1999.

Samuel, K., *Mathematical Methods and Theory in Games*: *Programming and Economics*, Volume 1, Addison-Wesley, 1959.

Sharma, *Data Structures Using C*, Pearson, 2013.

Sipser, M., *Introduction to the Theory of Computation*, PWS Publishing, Section 8.2–8.3, The Class PSPACE, PSPACE Completeness, 1997, pp. 281–294.

Sipser, M., *Introduction to the Theory of Computation: Space Complexity*, 2nd ed., Chapter 8, Thomson Course Technology, 2006.

Smith, S.W., The Discrete Fourier Transform, *The Scientist and Engineer's Guide to Digital Signal Processing*, Second ed., Chapter 8, California Technical Publishing, San Diego, CA, 1999.

Strang, G., *Introduction to Linear Algebra*, 4th ed., Wellesley-Cambridge Press, Wellesley, MA, February 2009.

Strayer, J.K., *Linear Programming and Applications*, Springer-Verlag, 1989.

Tenenbaum, et al., *Data Structures Using C*, Pearson, 2006.

Ullman, J. D., NP-complete Scheduling Problems, *Journal of Computer and System Sciences*, 10, 384–393.

Vašek C., *Linear Programming*, W. H. Freeman & Co., 1983.

Weiss, *Data Structure and Algorithm Analysis in C++*, Pearson, 2013.

Williamson and Shmoys, *The Design of Approximation Algorithms*, Cambridge University Press, 2012.

LECTURE NOTES

Artificial-Life, Inc.: Smart-Bots: Solutions for the Networked Economy, http://www.artificial-life.com/products/papers/ALifeSolutions.pdf, *July 2002.*

Aspnes, J., *Lecture Notes on Randomized Algorithms*, Available on http://cs-www.cs.yale.edu/homes/aspnes/classes/469/notes.pdf.

Diehl, S., *VRML — Virtual Reality Modeling Language, in: Gesellschaft für Informatik: Informatik-Lexikon, Stuttgart 1997* http://www.gi-ev.de/informatik/lexikon/inf-lex-vrml.shtml#dokanf, *July 2002.*

Lecture notes on Approximation Algorithms, Yale, Available on http://www.cs.yale.edu/homes/aspnes/pinewiki/ApproximationAlgorithms.html?highlight=%28CategoryAlgorithmNotes%29

Lecture notes on Decrease and Conquer, Available on http://www.cs.utsa.edu/~bylander/cs3343/chapter5handout.pdf

R Fiebrink, *Lecture Notes on Amortized Analysis*, Princeton University, Available on http://www.cs.princeton.edu/~fiebrink/423/AmortizedAnalysisExplained_Fiebrink.pdf.

Nandy, S.C., Lecture Notes on Randomized Algorithms, Available on http://www.tcs.tifr.res.in/~workshop/nitrkl_igga/randomized-lecture.pdf.

PAPERS

Aaronson, S., *Is P versus NP formally independent?*, Bulletin of the European Association for Theoretical Computer Science 81 (Oct. 2003).

Agarwal, P.K., J. Xie, J. Yang, and H. Yu., Scalable Continuous Query Processing by Tracking Hotspots, *Proceedings of the 32*nd *International Conference on Very Large Databases*, 32, 2006, pp. 31–42.

Agrawal, M., N. Kayal, and N. Saxena, *PRIMEs In Annals of Mathematics* 160, 2 (2004) 781–793.

Applegate, D., R. Bixby, V. Chvátal, and W. Cook, On the Solution of Traveling Salesman Problems, *Documenta Mathematica*, Extra Volume ICM III (1998), 645–656.

Baker, T., J. Gill, and R. Solovay, Relativizations of the P = NP Question, *SIAM Journal on Computing* 4, 4 (1975), 431–442.

Bedau, M.A., Artificial life: organization, adaptation, and complexity from the bottom up, *Science Direct-Trends in Cognitive Sciences*, Vol. 7, Issue 11, November 2003, pp. 505–512.

Berry, T. and S. Ravindran, A fast string matching algorithm and experimental results. In: Holub, J., M. Simánek, (Eds.), Proceedings of the Prague Stringology ClubWorkshop 1999, Collaborative Report DC-99-05, Czech Technical University, Prague, Czech Republic, pp. 16–26, 2001.

Bhasin, H. and N. Singla, *Article: Genetic based Algorithm for N-Puzzle Problem*, 51(22) (2012) 44–50.

Boyer, R., J. Moore, A Fast String Searching Algorithm, *Communication of the ACM*, Vol. 20(10) (1977) 762–772.

Brown, M.R. and R.E. Tarjan, Design and Analysis of a Data Structure for Representing Sorted Lists, *SIAM Journal on Computing* 9(3) (1980) 594–614.

Cook, S. The Complexity of Theorem-proving Procedures. In: *Proceedings of the 3rd ACM Symposium on the Theory of Computing*, ACM, NY, 1971, 151–158.

Cooley, J., P. Lewis, and P. Welch, The finite Fourier transform, *IEEE Trans. Audio Electroacoustics* 17(2) (1969) 77–85.

Crochemore, M., A. Czumaj, L. Gasieniec, S. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter, Speeding Up Two String Matching Algorithms, *Algorithmica*, Vol. 12, No. 4–5, 1994, pp. 247–267.

Edmonds, J., Paths, Trees and Owers, *Canadian Journal of Mathematics* 17 (1965), 449–467.

Faro, S., T. Lecroq, The Exact Online String Matching Problem: A Review of the Most Recent Results, *ACM Computing Surveys (CSUR) Surveys Homepage Archive*, Vol. 45, Issue 2, Article No. 13, February 2013.

Fredman, M.L. and R.E. Tarjan, Fibonacci Heaps and their uses in Improved Network Optimization Algorithms, *Journal of the ACM* 34(3) (1987) 596–615.

Garey, M. and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company, NY, 1979.

Grover, L., A fast quantum mechanical algorithm for database search. In: *Proceedings of the 28th ACM Symposium on the Theory of Computing*, ACM, NY, 1996, pp. 212–219.

Gurevich, S., R. Hadani, and N. Sochen, The finite harmonic oscillator and its applications to sequences, communication and radar, *IEEE Transactions on Information Theory* 54 (9) (2008) 4239–4253.

Hayles, N.K., *How We Became Posthuman, Virtual Bodies in Cybernetics, Literature, and Informatics,* Chicago, 1999.

Hong, Y., X. Ke, and C. Yong, An improved Wu-Manber Multiple Patterns Matching Algorithm, *Performance, Computing, and Communications Conference*, IPCCC 2006, 25th IEEE International, 6, 2006, p. 680.

Huddleston, S. Naher, and K. Melhorn, A New Data Structure for Representing Sorted Lists, *Acta Informatica*, 17, 1982, pp. 157–184.

Huddleston, S., and K. Melhorn, Robust Balancing in B-trees, 5th GI Conference on Theoretical Computer Science, *Lecture Notes in Computer Science*, 104, Springer Verlag, New York, 1981, pp. 234–44.

Impagliazzo, R. and Wigderson, A.P = BPP if E requires exponential circuits: Derandomizing the XOR lemma, In: *Proceedings of the 29th ACM Symposium on the Theory of Computing*, ACM, NY, 1997, pp. 220–229.

Karp, R., Reducibility Among Combinatorial Problems, *Complexity of Computer Computations*, R. Miller and J. Thatcher (Eds.), Plenum Press, 1972, pp. 85–103.

Knuth, D., J. Morris, and V. Pratt, Fast Pattern Matching in Strings, *SIAM Journal on Computing*, Vol. 6(1) (1977) 322–350.

Levin, L., Average Case Complete Problems, *SIAM Journal on Computing* 15 (1986), 285–286.

Luger, G.F., A. William, Stubblefield, *Artificial Intelligence: Structures and Strategies for Complex Problem Solving*, Addison-Wesley Longman Publishing Co. Boston, MA, 1997.

Luscombe, Greenbaum, and Gerstein, What is Bioinformatics? An introduction and Overview, Yearbook of Medical Informatics, 2001.

Michailidis, P., and K. Margaritis, On-line String Matching Algorithms: Survey and Experimental Results, *International Journal of Computer Mathematics*, Vol. 76, Issue 4, 2001.

Rabin, M.O., Probabilistic Algorithm for Testing Primality, *Journal of Number Theory* 12 (1980) 128–138.

Razborov, A., Lower Bounds on the Monotone Complexity of Some Boolean Functions, *Soviet Mathematics-Doklady* 31 (1985) 485–493.

SaiKrishna, V., A. Rasool, and N. Khare, String Matching and its Applications in: Diversified Fields, *International Journal of Computer Science Issues*, Volume 9, Issue 1, Jan 2012, pp. 219–226.

Sudan, M., Probabilistically Checkable Proofs, *Commun. ACM* 52, 3 (March 2009) 76–84.

Turing, A., On Computable Numbers, With an Application to the Etscheidungs Problem, *Proceedings of the London Mathematical Society* 42 (1936), 230–265.

Vargas-Rubio, J.G. and B. Santhanam, On the Multiangle Centered Discrete Fractional Fourier Transform, *IEEE Sig. Proc. Lett.* 12(4) (2005) 273–276.

Williams, G., *Approximation Algorithms: Introduction to Optimization, Decision Support and Search Methodologies*. In: Burke and Kendall (Eds.), Kluwer, 2005. (Invited Survey).

INDEX

Index Terms	Links	

D

E

Index Terms	**Links**	

Index Terms	Links		
In-order traversal	116		
Insertion sort	179	662	
Internal algorithms	170		
Intractable	419		
Inverse method	567		
Inverse of a matrix	561		
Isomorphic graphs	146		
Isomorphism	306		

J

Job assignment	306	
Job scheduling	635	
Job sequencing	228	

K

Knapsack problem	305	319	498
Knuth–Morris–Pratt automata	403		
Kruskal's algorithm	231		
Kuratowski's theorem	148		

L

Lateness	635	
LIFO search	311	
Linear search	6	80
Linked list	82	
Load balancing	340	
Logarithm	18	
Longest common subsequence	262	395
Longest path problem (LPP)	420	
Look-ahead method	467	
Lower triangular matrix	555	

Index Terms	Links	

M

N

Index Terms	Links		

O

P

Index Terms	Links

Index Terms	Links	
Related problems	661	
Relaxation	327	
Reversing the order	81	
Ribonucleic acid	516	517
Ribosomal RNA	518	
RL and LR rotations	134	
Roulette wheel selection	496	
Row matrix	553	
RP problems	337	
RR rotation	134	
RSA algorithm	391	

S

Sample space	599	
Satisfiability	422	
Scalar matrix	554	
Scalar multiplication	556	
Scheduling problems	634	
Selection	496	
Selection sort	172	660
Sequence alignment	524	
Sequence detection	524	
Sequence–structure deficit	522	
Sequencing	521	
Sequential method	467	
Set cover	422	
Set packing	422	
Shortest path algorithms	3	
Shortest path problem (SPP)	420	
Sigmoid function	491	
SIMD	468	

Index Terms	Links		
Simplex method	576		
Single-source shortest path	243		
Skew-symmetric matrix	558		
Slack variables	577		
Sorting	81	169	
Spanning tree	161	222	
Sparse matrix	82		
Stable sort	171		
Stack	90		
Standard form	577		
Standard random access machine model	469		
Static implementation of stack	90		
Static optimization	327		
Strassen's matrix	211		
Strictly binary tree	109		
String matching	395	396	
Structures of protein	521		
Subset sum problem	249	289	421
Subset sum using GA	499		
Substitution	43		
Subtraction of matrices	556		
Sudoku problems	292		
Suffix tree	409		
Sum of roots of unity	593		
Surplus variables	577		
Surrogate inequalities	328		
Swaps	170		
switch case	8		
Symmetric key cryptography	389		
Symmetric matrix	557		
Symmetry property	33		
Synapse	488	489	

Index Terms	Links			

T

U

Index Terms	Links	

V

W