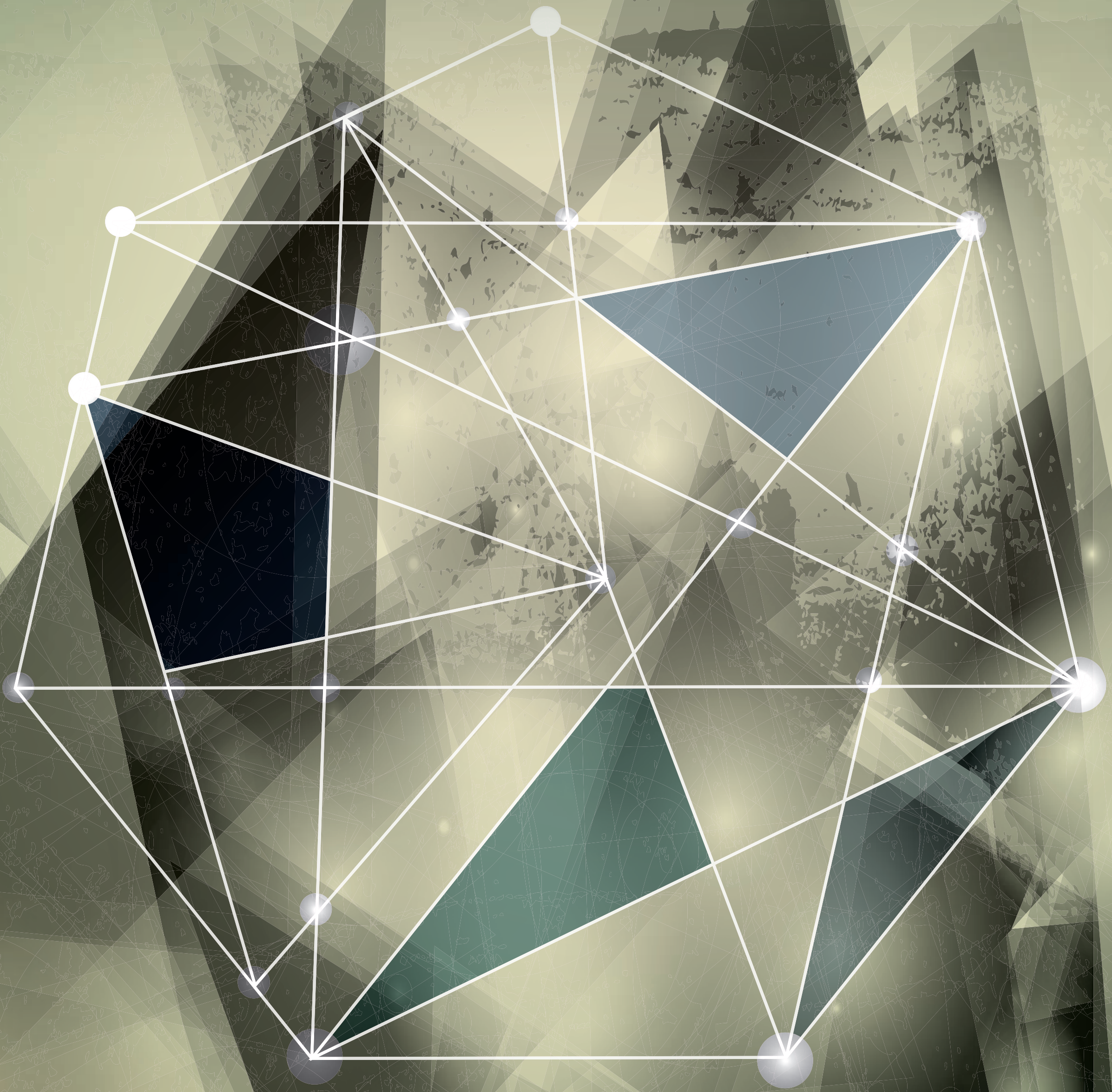
SECOND EDITION

ANALYSIS AND DESIGN OF ALGORITHMS



BY AMRINDER ARORA

ANALYSIS AND DESIGN OF ALGORITHMS

SECOND EDITION

BY AMRINDER ARORA
The George Washington University

cognella
academic publishing

First published in the United States of America in 2015 by Cognella, Inc.

Printed in the United States of America

ISBN: 978-1-63487-021-4 (pbk)/ 978-1-63487-073-3 (hc)/ 978-1-63487-022-1 (br)

cognella
academic publishing

www.cognella.com 800-200-3908

CONTENTS

*"There's no secret about success.
Did you ever know a successful man who didn't tell you all about it?"*
Kin Hubbard

"Eighty percent of success is showing up."
Woody Allen

TABLE OF FIGURES

Figure 8: Adversary's strategy for maximizing the number of comparisons for finding both maximum and minimum numbers. Adversary thinks of numbers in terms of buckets Q, W, L, and X.

ACKNOWLEDGEMENTS

This book would never have been possible if the universe had not conspired in a very specific way. Starting with my own education, I am forever indebted to my parents and my sister for believing in me and providing the environment of learning. My advisor and now colleague Hyeong-Ah Choi provided me the kind of training that makes endeavors of this kind even remotely possible. In the recent years, my wife and three bouncy kids—Roman, Jessica and Nayan—have exercised a lot of patience in affording me the time to write and revise this book.

I would also like to thank the very patient staff at Cognella with all the designs, redesigns, re-redesigns and "proofs" of so many versions.

A very special thanks to David Balash, also of the George Washington University, who guest authored the chapter 11 (Theory of Lower Bounds). David was an excellent student and remains a good friend. He is the owner and chief computer scientist of the IT consulting company One Ten Logic and an active member of the computer science community at the university.

Most importantly, I would like to thank countless students who gave me the much needed feedback during the lectures and various study sessions. You are the collaborators—front and center—and this book is ultimately dedicated to you all.

PREFACE

We live in a speed world, in which the model of learning has transformed from "let me learn it all" model to "I will learn it as needed" model. Students no longer read voluminous books, and instead rely on the lectures and course outline to get the overall picture, and then get the specific answers they need from Wikipedia, Google, watching videos on YouTube, and visiting many question/answer and forums websites. Thus, the books have lost their role of being the leading edge and instead, have become the trailing edge wherein they compete with these non-traditional sources. While there are many algorithms books currently available, many of them were written prior to this complete transformation of the learning model.

This book was written with a specific purpose in mind—a full course in design and analysis of algorithms in no more than 150 easy to read pages that can be read from cover to cover in less than 4 hours. Keeping this strict limit was necessary to maintain the viability of this book actually being read, as opposed to becoming a reference material. Naturally, such a small limit on the number of pages forces us to hand select some of the material. Towards the end, I felt the desire to increase the page limit. Thankfully, in this edition at least, I have resisted that temptation. I venture to guess that other books have been written with a similar limit in mind, but over the coming editions, grow to include more "essential" material. You are well advised to grab this edition of the book before the author also falls to such temptation in the coming editions.

This book is divided into 4 broad sections.

- Basics—Data structures, asymptotic notation, etc.
- Design—Algorithmic design techniques
- Analysis—NP-completeness and proving inherent complexity of problems
- Summary—Conclusions and auxiliary material

In the "Design" section, we discuss the following algorithmic design techniques.

- Divide and Conquer
- Greedy Method
- Dynamic Programming
- Graph traversal methods
- Branch and bound

One of the key observations that I have made from teaching the graduate course in algorithms class over multiple years is the rigor required by this class. Many students come into this class with some aspect of their mathematical background missing. Other students who have been working for a few years find that some of the mathematical material seems "vaguely familiar" and nothing else. Those essential mathematical topics are frequently mentioned in the book—the students need to return to those topics often. Those topics will ensure that while you can read this book in a few hours from end to end, you will need to spend many more hours making the best of the material in your capacity of a computer scientist.

The students are encouraged to use this book as an accompaniment to their study sessions. Algorithms, in some respects even more than other topics, requires a clarity of thought that can perhaps more easily be achieved by speaking and explaining, than by listening and reading. Every time you are asked to explain the topic of dynamic programming to someone, your knowledge and understanding of the topic increases. [More of this is covered in Appendix A.]

The best way to read this book (and many other books) is to remember to finesse the concepts presented in this book, and apply them in your everyday work. Read the chapters in different frames of mind, at leisure and consider different variations of problems presented herein. Observe how minor variations affect the complexity of those problems.

Another observation that I have made is that there is an almost direct correlation between grades and number of classes attended. There is also a strong correlation between grades and time spent on course, which manifests in form of emails, and other activities done by the student. There is an even stronger correlation between grades, homework assignments and projects—not many students finished with a top grade after losing easy points on assignments or projects. Perhaps Woody Allen was referring to algorithms when he said that 80% of success is showing up.

SECTION 1

THE BASICS

The coming 3 chapters cover the basics. We discuss what is an algorithm, study and review the asymptotic notation and data structures that are used repetitively in the rest of the book. Many of the readers may be well versed with these foundation elements. For this reasons, these chapters are very brief. It is highly recommended that you read these chapters regardless. You may be surprised by some concepts mentioned therein. You can also refer to other books and background materials, such as [1] [2] [3] [4] [5] [6] [7] [8] [9] and [10], etc.

INTRODUCTION AND BARE ESSENTIALS

CHAPTER 1

1.1 WHAT IS AN ALGORITHM?

An algorithm is generally defined in one of two following ways:

- A precise statement to solve a problem on a computer
- A sequence of definite instructions to do a certain job

Wikipedia says: "(In mathematics, computing, and related subjects) An **algorithm** is an effective method for solving a problem using a finite sequence of instructions."

Algorithms have existed for a long time, with Euclid's algorithm for finding the greatest common divisor having been described back around 300 BC. Incidentally, it is still known as a fairly effective algorithm, and is a standard first programming assignment. The algorithm can be described in English, without using any mathematical notation as follows. You are given two numbers, and you need to find the greatest common divisor of these two numbers. For example, given *35* and *20*, the greatest common divisor is *5*, and given *27* and *48*, the greatest common divisor is *3*. The algorithm goes as follows: divide the larger number with the smaller number, and obtain the remainder. If the remainder is zero, then the smaller number is the greatest common divisor. Otherwise, consider the remainder as your new "smaller" number, the previous smaller number as your new larger number, and go back to the divide step. [The unsaid part is that this will always terminate and that eventually we will find that the remainder is zero.]

Describing even a simple algorithm in English can be tricky and prone to errors in boundary conditions. Therefore, going forward we will be using a standard mathematical notation to describe algorithms.

Let us use insertion sort as our next example. The goal of this algorithm is to sort the given list of numbers in the increasing order. For example, given a list *[5, 8, 2, 10]*, the algorithm should produce the sorted list of *[2, 5, 8, 10]*.

Insertion sort works by growing a sorted list, and by inserting a new number in the list in every iteration (hence the name insertion sort). Consider the following pseudo-code for describing insertion sort.

```
// Given an array A of n numbers
// Sorts the array A
for j = 2 to n {
   key = A[j]
   i = j - 1
   // A[j] is added in the sorted sequence A[1.. j-1]
   while ((i ≥ 0) and (A[i] > key)) {
      A[i + 1] = A[i]
      i = i - 1
   }
   A[i] = key
}
```

From this pseudo-code, we can attempt to estimate the best case running time, the worst case running time and the average case running time. We observe that the **for** loop runs $n-1$ times, but the **while** loop can run a variable number of times based on the actual values in the array. In the worst case scenario, it may run i times, and in the best case, it may run 0 times.

For our third example, let us turn to the searching problem. We are given a **sorted** list of numbers, and we are asked to find if a certain number exists in the list or not. The commonly known Binary Search algorithm compares the middle element of the array with the given number, and depending upon the result, focuses on either the left half or the right half of the array.

```
// Given a sorted array A
// Finds if a given “value” exists in A or not
BinarySearch(A[0..N-1], value, low, high)
{
   if (high < low)
      return -1                          // not found
   mid = (low + high) / 2
   if (A[mid] > value)
      return BinarySearch(A, value, low, mid-1) else
      if (A[mid] < value)
      return BinarySearch(A, value, mid+1, high)
      else
      return mid                         // found
}
```

1.2 PRACTICAL APPLICATIONS OF ALGORITHMS

Practical applications of algorithms are found quite abundantly and can be very diverse. Here are some practical applications:

- Find relevant web pages for a given search term
- Analyze thousands of casual pictures taken by hundreds of different cameras to track an individual
- Given millions of "x knew y at time t" statements and millions of stock transactions, detect insider-trading
- Given a grocery list and the layout of a store, find the quickest way to collect all items
- Given gas prices in a city and the places that you need to go, find a path that minimizes your cost
- Given a person's song history and a large song database, find a song that they may like hearing next.
- Given many recipes and ingredients, find the maximum number of dishes you can make
- Given a candidate's answers to *10* questions, find the next *5* questions that maximize your confidence in assessing their level
- Given thousands of image files, separate the ones taken at night from the ones taken during the day

- Given a list of train stations and a map of tracks, create a schedule that minimizes the sum of travel time across all users.

1.3 WHAT IS MEANT BY "ANALYZING AN ALGORITHM"?

Assuming a given algorithm is functionally correct, analyzing it typically requires us to answer the following two questions:

(i) How long will the algorithm take to run, in best case, worst case and average case?
(ii) How much memory will it require, in best case, worst case and average case?

We may also want to know if there are some inputs on which the answer varies considerably, and what the average case values would be if the input is limited to certain range.

Depending upon the type of algorithm, analyzing an algorithm may have many other meanings as well.

- If the algorithm is an approximation algorithm, analyzing an algorithm requires us to find the approximation ratio.
- If the algorithm is an online algorithm, analysis involves competitive ratio.
- If the algorithm is a prediction algorithm, analysis involves many metrics such as accuracy and precision.

1.4 WHY SHOULD WE ANALYZE ALGORITHMS?

For a given problem, there may be many algorithms that solve the problem correctly. One algorithm may be more complicated than other, but the other may be more readable and more eloquent. Similarly, one algorithm may be very efficient on certain inputs, and very inefficient on some other inputs. Having a mechanism to analyze the algorithm in terms of one standard notation allows us to compare many different algorithms.

Another very practical reason to analyze algorithms is to perform a priori estimation of performance. Algorithms are packaged in form of software programs which run on a wide variety of devices (computers, mobile phones, tablets, TVs, GPS receivers). Those software programs are marketed using a variety of eye catching advertisements and marketing buzz words. Having a mathematical analysis of the algorithms implemented by those software programs allows us to understand their performance parameters.

1.5 HOW TO ANALYZE A GIVEN ALGORITHM (PROGRAM)

To analyze a given algorithm, firstly we agree on a model of computation (the *computation* model). A computation model that closely reflects today's machines is the Random-Access Machine (the *RAM model)*. In this model, math operations (addition, subtraction, division and multiplication) take one unit of time, simple logic operations (comparison of two numbers) and read and write operations take one unit of time also. Further, we can access any memory location (including registers, etc.) in a unit time as well. We will be using this computation model in the rest of the book.

Here are some simple observations that we can make in analyzing the time complexity of given programs.

- When analyzing an **if-then-else** condition, consider the arm that takes the longer time. However, in some cases, doing so repeatedly may yield a result that is a significant over-approximation. Therefore, a more careful analysis may be required in those situations.
- When considering a **while loop** (or equivalently, a **for loop** or **repeat loop**), multiply the number of times the loop runs with the time complexity of the function inside the loop.
- When considering **nested loops**, we need to multiply the number of times each loop runs, with the time complexity of the function inside the innermost loop.

1.6 PRE-REQUISITES

This book significantly depends on the reader having the appropriate Math background. The following topics are assumed to be well understood. In case a deeper explanation is required for these topics, excellent textbooks are available for these topics, although going through these materials will take a semester or so. So, if you find yourself lacking in these topics, you may want to take a different class first, and then return to algorithms in a future semester.

- Sets and functions

- Logs and exponents
- Recurrence relations
- Mathematic series, such as arithmetic progression, geometric progression, arithmetic-geometric progression and their sums

1.7 POP QUIZ FOR THE PRE-REQUISITES

The following questions can serve as a quick check of readiness to consume the material in this book. You should be able to answer all of the questions in order to extract the maximum advantage out of this book. *(The material in this book does not cover what may be needed to answer these questions.)*

1. Consider A and B are two sets, such that $|A| = 50$, and $|A - B| = 20$, and $|B| = 85$. Find the value of $|B - A|$.
2. Given that $log_{10}2 = 0.3010$ and $log_{10}3 = 0.4771$, find the value of $log_6 10$.
3. Given that $T(n) = T(n-1) + n^2$ can you find a closed-form expression for $T(n)$?
4. What is the sum of the following series:

   $\sum_{i=1}^{n} i\ 2^i$
5. What is the sum of the following series:

   $\sum_{i=1}^{n} i^2\, 2^i$
6. Which of the following two terms is larger:

   $\sum_{1}^{n} i^2$ or $\sum_{1}^{n\char`^2} i$

ASYMPTOTIC ANALYSIS AND NOTATION

CHAPTER 2

Asymptotic analysis is a method of analyzing the performance of algorithms when applied to very large inputs. The goal of asymptotic analysis is to arrive at the asymptotic notation—a simple articulation of the space or time performance of the algorithm, for example to say an algorithm has $\theta(n)$ or $\theta(n^2)$ or $\theta(n \log n)$ time complexity.

There are usually two aspects of the asymptotic analysis—(i) take a given algorithm, given in pseudo-code or another format and to arrive at a closed form expression, and (ii) reduce the closed form expression to a simpler format.

For example, consider an algorithm that takes an array as an input, compares each pair of elements, iterates the array three times, and checks the last element seven times. In that case, the running time of this algorithm may be written

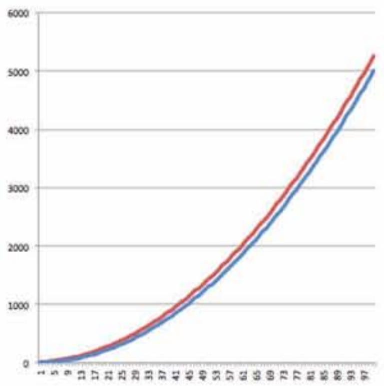


Figure 1: A visual comparison of $n^2/2$ and $n(n-1)/2 + 3n + 7$ curves. The dropping of the lower level terms is the typical first step in asymptotic analysis.

as $n(n-1)/2 + 3n + 7$. This is the closed form expression that we may have arrived at after a visual inspection of the algorithm.

In the second phase, we simplify this polynomial expression. As n becomes larger, as we plot the performance of this algorithm, we observe that the final two terms ($3n$ and 7) do not affect the curve. In fact, drawing that curve against the curve of much simpler expression $n^2/2$, we can observe that the two curves are nearly the same. This represents the first step in simplifying the closed form expression—simply drop the "lower level" terms. The second step is a similar simplification—simply drop the constant multiple of the highest-level term, for example, using n^2 instead of $n^2/2$. This idea allows us to ignore some constant time jumps in performance—for example, something that makes an algorithm twice as fast.

Asymptotic notation is an important tool at our disposal to articulate and communicate the running time and space usage of algorithms. An algorithm may be complicated and its running time may depend upon the input. Even on a given input, it may be difficult to specify exactly how long an algorithm will take. The asymptotic notation allows us to specify those attributes in very simple terms such as being able to say, "Algorithm A has $O(n)$ running time", something that we could easily tell someone over the phone.[1]

The implicit purpose of the asymptotic notation is that it allows us to compare algorithms in terms of their running time or space usage. Even though the algorithms may have many idiosyncrasies, using asymptotic analysis, and after we arrive at asymptotic notation, we can compare which algorithm performs better as the input size grows.

The concise idea: Asymptotic analysis is the technique. Asymptotic notation is the output, which allows us to easily communicate the performance of algorithms.

2.1 BIG O NOTATION

We define $O(g(n))$ to be the set of all functions $f(n)$ such that there exist constants n_0 and c such that $0 \leq f(n) \leq c\, g(n)$ for all $n \geq n_0$. (Asymptotic analysis is usually used for positive functions only, so we assume that $f(n) \geq 0$.)

Thus, by very definition, $O(g(n))$ is a **set** of functions. However, we abuse this notion sometimes by saying $f(n) = O(g(n))$ when we mean that $f(n)$ is in

1 We sometimes use the **telephone conversation** analogy. In its simplistic form, the telephone conversation includes questions and answers that we can ask each other over an old fashioned telephone line, without the benefit of any video, screen sharing, etc. It is an attempt to articulate our concepts in simple terms, and ignore all details that can indeed be ignored.

the set $O(g(n))$. [Perhaps the fact that the mathematical symbol ∈ is not easily typeset may have played a role in this abuse of notation becoming prevalent.]

For example, $n = O(2n)$, $2n = O(n)$ and $n = O(n^2)$. To prove each of these statements, the reader should find the constants n_0 and c such that $f(n) \leq c\,g(n)$ for all $n \geq n_0$. We can prove all 3 of these assertions, and many others by making the observation that if $f(n)$ can be written in linear terms as $f(n) = a_0 n^0 + a_1 n^1 + \ldots + a_m n^m$, then we can write $f(n) = O(n^m)$.

2.2 BIG OMEGA NOTATION

We write that $f(n) = \Omega(g(n))$ if there exist constants n_0 and c such that $f(n) \geq c\,g(n)$ for all $n \geq n_0$.

We observe that big omega notation is the inverse of the Big O notation. That is, $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$.

2.3 SMALL OH (o) NOTATION

We define $o(g(n))$ to be the set of all functions $f(n)$ such that for ANY constant $c > 0$, there exists n_0 such that $0 \leq f(n) < c\,g(n)$ for all $n \geq n_0$. As in the case of other asymptotic notations, we frequently write $f(n) = o(g(n))$ when we mean that $f(n)$ is in the set $o(g(n))$.

For example, $n = o(n^2)$.

One way of proving that $f(n) = o(g(n))$ is to show the existence of constant n_0 for any constant $c > 0$. An equivalent way is to prove that $lim_{n\to\infty} f(n)/g(n) = 0$. One very helpful tool in evaluating limits is L'Hopital's rule, which states that assuming certain conditions apply, $lim_{n\to\infty} f(n)/g(n) = lim_{n\to\infty} f'(n)/g'(n)$, where $f'(n)$ and $g'(n)$ represent the first derivatives of functions $f(n)$ and $g(n)$ respectively.

For example, using L'Hopital's rule, we can easily observe that $lim_{n\to\infty} (log(n)\hat{}3)/n = 0$. We can evaluate this limit as follows.

$$\begin{aligned} lim_{n\to\infty} (log(n)\hat{}3)/n &= lim_{n\to\infty} (2\,log(n)\hat{}2)/n && \text{// Apply L'Hopital's rule} \\ &= lim_{n\to\infty} (4\,log(n))/n && \text{// Apply L'Hopital's rule again} \\ &= lim_{n\to\infty} 4/n && \text{// Apply L'Hopital's rule yet again} \\ &= 0. \end{aligned}$$

Therefore, we can conclude that $log(n)\hat{}3 = o(n)$.

2.4 SMALL OMEGA (ω) NOTATION

We write that $f(n) = \omega(g(n))$ if for any constant $c > 0$, there exists n_0 such that $f(n) \geq c\, g(n) \geq 0$, for all $n \geq n_0$.

For example, $n^3 = \omega(n^2)$.

We observe that the small omega notation is inverse of Small Oh notation. That is, $f(n) = o(g(n))$ if and only if $g(n) = \omega(f(n))$.

2.5 THETA NOTATION

Theta notation is used to identify functions that can be considered asymptotically equivalent. That is, function $f(n) = \theta(\mathbf{g(n)})$ if and only if $f(n) = O(g(n))$ and $g(n) = O(f(n))$.

2.6 MAIN DIFFERENCE BETWEEN BIG O AND SMALL o

The subtle difference between big O and small o (and equivalently between big Omega and small omega) functions may be sometimes hard to spot. They seem to convey almost the same thing. However, a crucial difference between big O and small o is that in the case of big O, we have the liberty to choose both the constants c and n_0. In the case of small o, our statement is much stronger. We are in fact saying that for *any* constant c, we can find a constant n_0 such that the inequality holds. This is a considerable difference. When we say that $f(n) = o(g(n))$, we are explicitly saying that even if we multiply the function $f(n)$ by any large constant, the function $g(n)$ will ultimately grow larger than $f(n)$ for large values of n.

2.7 ANALOGY WITH COMPARISON FUNCTIONS FOR REAL NUMBERS

It can sometimes be helpful to think of asymptotic functions as we think about the relationship between real numbers. Given two real numbers x and y, we can ask these 5 questions: $x \leq y$, $x < y$, $x = y$, $x > y$, $x \geq y$. Answering one of these questions may automatically answer another of those 5 questions, or it may not. For example, if $x \leq y$, then we know that $x > y$ is not true, but we do not know if $x < y$ or not. Analogously, when we make a claim that $f(n) = O(g(n))$, we do not know if $f(n) = o(g(n))$ or not.

Tip: Using the analogy of less than equal relationship (≤) for big O, strictly less than relationship (<) for small o, equal relationship (=) for theta, larger than equal relationship (≥) for big Omega and strictly greater relationship (>) for small omega can sometimes be helpful.

Once the analogy has been established, we can extend it to ask which of the properties of real numbers apply to asymptotic functions. For example, "the greater than or equal to" is a transitive property. If $x \geq y$ and $y \geq z$, then $x \geq z$. Is the same statement true for big O notation as well? From the definition of big O notation, we observe that it is indeed true, and we claim the following.

	SMALL o	BIG O	Ø	BIG Ω	SMALL OMEGA
TRANSITIVITY	Yes	Yes	Yes	Yes	Yes
REFLEXIVITY	No	Yes	Yes	Yes	No
SYMMETRY	No	No	Yes	No	No

As mentioned earlier, there is transpose symmetry between big O with big Ω functions, and between small o and small ω functions. That is, if $f(n) = O(g(n))$, then $g(n) = \Omega(f(n))$ and similarly, if $f(n) = o(g(n))$ then $g(n) = \omega(f(n))$.

Finally, we investigate the property of **trichotomy**. For real numbers x and y, we can always say that either $x < y$ or $x = y$ or $x > y$. For asymptotic functions $f(n)$ and $g(n)$, we may be unable to prove any asymptotic function between $f(n)$ and $g(n)$. For example, consider $f(n) = sin(n)$ and $g(n) = cos(n)$.

2.8 HOME EXERCISES

1. What is the time complexity of the following program?

```
j = 1
while (j < n) {
    k = 2
    while (k < n) {
        Sum += a[j]*b[k]
        k = k * k
    }
    j++
}
```

2. What is the time complexity of the following programs:

```
j = 1                           j = 2
while (j < n) {                 while (j < n) {
  k = j                           k = 2
  while (k < n) {                 while (k < n) {
    If (k is odd)                   Sum += a[k]*b[k]
       k++                          k = k * k
    else                          }
       k += 0.01 * n              k = log n
  }                               j += j/2
  j += 0.1 * n                  }
}
```

3. Given $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$, prove that $f_1(n) + f_2(n) = O(g_1(n) + g_2(n))$.
4. Given $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$, prove that $f_1(n)\, f_2(n) = O(g_1(n)\, g_2(n))$.
5. For each of the following questions, answer how the given two functions compare asymptotically.

   a. Polynomial, logs and exponential:
      i. n^{17} and 2^n
      ii. $(n+3)^6$ and $(1.05)^n$
      iii. $(1.05)^n$ and $(1.06)^n$
      iv. $2^{n\text{\textasciicircum}2}$ and 10^n
      v. n^{100} and $2^{\sqrt{n}}$
      vi. n^2 and $(\log n)^{80}$
      vii. $n \log n$ and $n^{1.1} \log \log \log n$
      viii. n and $(\log n)^3 + (\log \log n)^4$
   b. Factorial / Combinatorial
      i. $n!$ and n^6
      ii. $C(n,n/3)$ and n^4
   c. Floors and Ceilings
      i. $(\text{ceil } n)^3$ and $(\text{floor } n)^4$

DATA STRUCTURES

CHAPTER 3

In this chapter, we provide a brief overview of the various data structures that will be helpful in the coming chapters. As a quick recapitulation, ***a data structure is a structure to hold data, which allows a specific set of operations to be performed on the data set.***

There are two general problems that one can come across when dealing with data structures:

1. **Choosing an appropriate data structure**: Given a data set and the operations to be supported, **choose** a data structure that allows those operations to be done efficiently.
2. **Designing a good data structure:** Given a data set and the operations to be supported, **design** a data structure (organization) that allows those operations to be done efficiently.

Given a problem, we typically start by choosing the right data structure. If no data structure is found, then we may need to design a data structure that meets our requirements of efficiency.

3.1 RECORD

A record allows the packaging of related data elements (called fields). Most high level languages allow the user to define customized records. In C#/Java/PHP/VB.NET and other object oriented programming languages, this is called "class"[1]. In C, this is called "struct". Older languages such as COBOL, and newer languages such as Ruby and Groovy support this concept as well.

3.2 LINKED LIST

A **(singly) linked list** is a sequence of records, where every record has a pointer to the next record. A special pointer called "first" has the reference to the first record.

Most implementations of linked lists though are doubly linked due to very low additional overhead of making a linked list double linked.

A **doubly linked list** is a sequence of records, where every record has a pointer to the next record, and a pointer to the previous record. Special pointers called "first" and "last" have references to the first and the last records.

3.3 STACK

A stack is a data structure that allows us to add a data item, to remove a data item, and to peek at the last item that was added. When removing an item, the last item that was added (pushed) is the one that gets removed (popped). For that reason, stack is also referred to as a last in first out (LIFO) data structure. Here is a brief description of these operations:

- Push (an object)—This adds the given object to the stack, which is now on top of the stack.
- Pop()—This returns the object that was on top of the stack, and removes it from the stack.
- Top()—This returns an object, without actually removing it from the stack. This is sometimes also referred to as peek(), because we are just taking a peek at the top of the stack, but we are not actually removing it.

Implementation of Stacks

Stacks exist as common data structures in the libraries of most high-level programming languages. For example, in Java, Stack is provided in the java.util. Stack class.[2] If we need to implement Stacks as a programming exercise, we

1 In Object Oriented paradigm, definition of "class" packages data elements, as well as related operations.

2 In Java, it is now recommended that we use a double ended queue (java.util.deque) instead of the stack class.

can use a variety of mechanisms. For example, to implement stacks using an array, we can use an array S[1:N], and use a special pointer to the "top" of the stack. When pushing something on the stack, we can increment the pointer. When popping, we can decrement the pointer.

Similarly, to implement stacks using a double linked list, we can use a special pointer to the "top" of the stack. When pushing something on the stack, we can advance the "top" pointer. When popping, we can move the "top" pointer back one step.

3.4 QUEUE

Analogous to a Stack, a queue is a first in first out (FIFO) data structure that allows two basic operations:

- dequeue(): Returns the first element
- enqueue(*t*): Adds an element t to the end of the queue.

It is quite appropriate to use a Queue data structure to model a physical world queue such as a customer service queue. The customers come and enter the queue at the end ("enqueue") and as a customer service representative frees up, they can serve a customer ("dequeue").

Implementation of Queues

Queues are typically included in standard libraries. For example, in Java, the Queue is defined in java.util.Queue as an interface, with many implementations available. To implement a queue using a doubly linked list, we can use two special pointers—tail and head. The queue can then be represented as tail → .... → head. When enqueuing an item, we move tail pointer one step to the left. When dequeuing an item, we move head one step to the left.

To implement a queue using an array, we can use "head" and "tail" indexes. When enqueuing an item, we decrement the tail index. When dequeuing an item, we decrement the head index.

3.5 SET

A set is a data structure that ensures that only a single copy of an element is included. Two main methods are provided in this data structure:

- add(*x*): Adds the element *x*. If the element *x* is already present, it is not added again.
- contains(*x*): Checks if the set contains the element *x* or not.

If a set is implemented using a simple linked list, both of these methods require the entire list to be traversed in the worst case, and thereby require $O(n)$ time. For this reason, sets are typically implemented using either a binary tree (more on this in the coming sections) or using hashing.

Set implementations are included in most high-level languages. In Java, Set is an interface with TreeSet and HashSet as two implementation classes.

3.6 MAP

A map (also called an Associative Array) is a data structure that allows storing of elements using a given key value. Three main methods are provided in this data structure:

- put(k,v): Adds the key k and associates it to value v. If the key k is already present, the value v may replace the previously associated value.
- get(k): Returns the value associated with key k. If key k is not present, this method returns an empty value (null).
- contains(k): Checks if the map contains the key k or not.

Similar to a set, maps are typically implemented using either a binary tree or using hashing. In Java, Map is an interface with TreeMap and HashMap as two implementation classes.

Collections API in Java

In Java, sets and lists (and occasionally maps) are typically considered as part of the Collections API. Most of the interfaces provide additional methods such as size(), which returns the number of elements in that set, list or map. Some of the implementation classes are better suited to handle multiple threads accessing that collection.

3.7 GRAPH AND TREE DATA STRUCTURES

3.7.1 Graph

A graph $G = (V,E)$ is an ordered pair, defined by a set V of vertices, and set E of edges. Each edge in E connects two vertices v_1 and v_2, which are in V. A graph can be directed or undirected. If the graph is directed, then each edge is defined as an ordered pair (v_1, v_2) and the edge is said to exist from v_1 **to** v_2, and not from v_2 to v_1. If the graph is undirected, then each edge is defined as an unordered pair $\{v_1, v_2\}$ and the edge is said to exist **between** the vertices

v_1 and v_2, or equivalently between v_2 and v_1. Here are some more definitions typically used in graph theory:

- If (x,y) is a directed edge, then x is said to be **adjacent** to y, and y is adjacent from x.
- In the case of undirected graphs, if $\{x,y\}$ is an edge, we simply say that x and y are **adjacent** (or x is adjacent to y, or y is adjacent to x). Also, we can say that x is the neighbor of y.
- The **indegree** of a node x is the number of nodes adjacent to x.
- The **outdegree** of a node x is the number of nodes adjacent from x.
- The **degree** of a node x in an undirected graph is the number of neighbors of x.
- A **path** from a node x to a node y in a graph is a sequence of node x, x_1, x_2, ..., x_n, y, such that x is adjacent to x_1, x_1 is adjacent to x_2, ..., and x_n is adjacent to y.
- The **length of a path** is the number of edges that comprise that path.
- A **cycle** is a path that begins and ends at the same node.
- The **distance** from node x to node y is the length of the shortest path from x to y.
- We say a graph is **connected** if there is at least one path between every pair of vertices.

Graph Representations

Graphs are generally represented in one of the following two ways:

(i) Adjacency Matrix: Using a matrix $A[1 .. n, 1 .. n]$ where $A[i,j] = 1$ if (i,j) is an edge, and is 0 otherwise. If the graph is undirected, then the adjacency matrix is symmetric about the main diagonal.

(ii) Adjacency List: Using an array $Adj[1 .. n]$ of pointers, where $Adj[i]$ is a linked list of nodes which are adjacent to i.

Tradeoffs between space and time complexity when considering adjacency matrix and adjacency list representations: The matrix representation requires more memory, since it has a matrix cell for each *possible* edge, whether that edge exists or not. In adjacency list representation, the space used is directly proportional to the number of edges. If the graph is sparse (very few edges), then adjacency list may be a more efficient choice.

However, the time required to look up whether an edge exists or not is higher in the case of an adjacency list representation. For this reason, an adjacency list representation is typically used only in case of sparse matrices or when memory space is at a premium.

3.7.2 Tree

A tree is a connected acyclic graph (i.e., it has no cycles). It is easy to prove that a tree with *n* vertices must have *n–1* edges. It is also easy to prove that following three characterizations of trees are equivalent:

(i) A tree is a connected acyclic graph
(ii) A tree is a connected graph on *n* vertices and *n–1* edges
(iii) A tree is a graph with *n* vertices and *n–1* edges that does not have any cycles.

We further observe that in a tree, there is a unique path between every pair of vertices.

We refer to a tree as a **rooted tree** if one of the nodes is designated as a root (the top node). Following definitions are generally used in the context of rooted trees:

- A *leaf node* is a node that has no children
- *Ancestors of a node x* are all the nodes on the path from *x* to the root, including *x* and the root
- *Subtree rooted at x* is the tree consisting of *x*, its children and their children, and so on and so forth all the way down
- *Height* of a tree is the maximum distance from the root to any node.

3.7.3 Binary Tree

A binary tree is a (rooted) tree where every node has at most two children: left child and right child. The subtree rooted at the left child node is then referred to as the left subtree and the subtree rooted at the right child node is similarly referred to as the right subtree.

Within the context of binary trees, following two related concepts are often used.

A **full binary tree** is a binary tree where every non-leaf has two children and all the leaves are at the same level. If the root is defined to be at level *0*, then the number of nodes in level *j* is 2^j. A full binary tree of height *h* has $2^{h+1} - 1$ nodes.

An **almost complete binary tree** of *n* nodes is the binary tree in which all the leaves are at the bottom two adjacent levels and there are no gaps between the leaves.

We observe that a simple array *a[1 .. n]* can represent a full or an almost complete binary tree using the following simple arithmetic:

- *a[1]* is the root of the tree
- For the node *a[j]*, the child nodes are *a[2j]* and *a[2j + 1]*
- For the node *a[j]*, the parent node is *a[j/2]*, where division is considered integer division.

3.7.4 Binary Search Trees

A binary search tree (BST) is a binary tree where every node contains a value, and for every node x, all the nodes of the left subtree of x have values $\leq x$, and all nodes in the right subtree of x have values $\geq x$.

BST supports 3 operations: insert(x), delete(x) and search(x). The height of search trees represents a bound on the searching time, so keeping the height bounded is a common objective in binary search trees. Further, since the data is typically mutable, it is important that the operations insert and delete operations maintain the balanced height property, and yet, do not have a significant overhead in maintaining that property.

Red Black and AVL trees are interesting implementations of height balanced binary search trees that allow searches, sequential access, insertions, and deletions in logarithmic time.

3.7.5 B-tree

B-tree (which is different from a binary tree!) is a one of the self-balancing search trees. B-tree is a generalization of a binary search tree in that a node can have more than two children, and that different nodes can have different number of children.

Due to the inherent flexibility present in their structure, B-trees are very well suited to handle mutable data, and for that reason, B-trees are commonly used in actual system implementations of databases and file systems. While the implementation of a B-tree is more complex than that of a binary search tree such as a red black tree, the overall performance is generally better with B-trees.

3.7.6 2-3 Trees

A 2–3 tree is a tree where every internal node (that is, a node with children) has either two children (2-node) and one data element or three children (3-nodes) and two data elements. Leaf nodes have no children and one or two data elements. 2-3 Trees can be considered a special instance of B-trees.

3.8 HEAPS

A min heap (or a max heap) is a data structure that enforces a priority on the items. Smaller items are retrieved from the min heap before larger items. It supports three basic operations.

- Insert(x): Inserts the item x and ensures that heap property of the data structure is maintained.
- Minimum(): Returns the minimum element, but does not remove it from the heap.
- Extract_Min(): Returns the minimum element and removes it from the heap data structure.

[Correspondingly, a max heap supports insert(x), Maximum() and Extract_Max() operations.]

Heaps are often implemented using a tree and in that case they are referred to as heap trees. In the case of minimum heap trees, the following constraint is satisfied: all children of a node must be larger than or equal to the parent node. The children nodes do not themselves need to be in any particular order.

In case a heap is implemented using a binary tree, it is referred to as a binary heap.

A heap may be implemented using a non-binary tree, that is, a tree that allows more than two child nodes. Similarly, a heap may be implemented using a collection of trees, for example, a Fibonacci heap.

A heap is also referred to as "priority queue", and is one of the implementation classes of the Queue data structure defined in the preceding sections.

3.9 HOME EXERCISES

1. Write pseudo code for following operations, considering that heap is an almost complete binary tree implementation using an array
   a. Insert(x)
   b. Extract_min()
2. Prove by induction that a full binary tree of height h has $2^{h+1}-1$ nodes.
3. Write pseudo code for following operations, considering that the binary search is an almost complete binary tree implementation using an array
   c. Insert(x)
   d. Delete(x)
   e. Search(x)

4. **(Trichotomy—Extended)** Given two functions $f(n)$ and $g(n)$, both strictly increasing with n, is it possible that $f(n)$ and $g(n)$ cannot be compared asymptotically? Either prove that such two functions can always be compared asymptotically, or give a counter example, such that neither $f(n)$ is in $O(g(n))$ nor is $g(n)$ in $O(f(n))$.
5. Suppose an array is being used to handle a list of objects, such as customers. In such a case, how can we support delete operation for a customer? Two obvious mechanisms are: (i) leave "holes" (null elements) in the array, and (ii) shift the elements to the left when an element gets deleted. What are the pros and cons of these two approaches? Can you think of other approaches?

SECTION II

ALGORITHM DESIGN TECHNIQUES

In this section, we begin our core objective on how to design efficient algorithms. We study the following design techniques: (i) divide and conquer, (ii) greedy method, (iii) dynamic programming, and (iv) branch and bound. We will also review graph traversal methods, which do not form a distinct algorithm design technique, but are a very helpful component nonetheless.

There is no silver bullet for deciding which algorithmic design technique to apply, but there are numerous hints and patterns that we can utilize in deciding which algorithm design technique to use. **Our goal will be to investigate, understand and memorialize those patterns, and become adept at discerning those hints.**

DIVIDE AND CONQUER

CHAPTER 4

Divide and conquer is an algorithmic design technique to solve complex problems by breaking a problem instance into smaller instances of the problem and combining the results. It is a recursive methodology and involves calling the same algorithm on smaller instances of the same problem.

The phrase "divide and conquer" is used in other contexts as well, such as management and politics. It may also refer to an informal way of breaking down a task, such as "organizing a party" into subtasks such as "decide the guest list", "prepare invitation cards" and "decide catering menu". However, in Computer Science, divide and conquer is a formal paradigm and the preceding example of party organization task would not be considered a divide and conquer approach within the computer algorithms context. When using the divide and conquer algorithm design technique, the smaller instances of the problem created during the divide step must be instances of the *same* problem.

Two typically used accompaniments of divide and conquer are:

(i) Induction for proving correctness, and
(ii) Recurrence relation solving for computing time (and/or space) complexity

Since divide and conquer algorithms are recursive algorithms at heart, we begin by defining a recursive algorithm: **A recursive algorithm is an algorithm that calls itself.** A hackneyed joke about recursion is: "In order to learn recursion, you must first learn recursion."

We start with the example of the quintessential recursive algorithm to sort a given array.

```
// A generic recursive algorithm to sort an array
Algorithm sort (Array a)
Begin
   If (input is small enough) {
      Solve using a different method and return
   }
   sort (subarray consisting of first half of a)
   sort (subarray consisting of second half of a)
   do_something_else();
End
```

When analyzing algorithms constructed using the divide and conquer technique, we will frequently come across recurrence relations. A brief overview of the same is provided next.

4.1 SOLVING RECURRENCE RELATIONS

(Definition) A **recurrence relation** is an equation that recursively defines a sequence, using some initial terms that are given. For example, a recurrence relation may be given as:

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

For this example, we can easily calculate $T(n)$ to be $n(n+1)/2$ using the concept of arithmetic progression.

Consider a different example: $T(n) = T(n/2) + T(n/2) + n^2$ for $n > 1$; division is assumed to be integer division. [When $T(1)$ is not specified, we usually assume $T(1)$ to be any constant value, such as c or 1.] Given this definition, we want to find a closed form expression for $T(n)$.

Another example of recurrence relations is: ***$T(n) = a\ T(n/b) + f(n)$***. The recurrence relations of this form appear very frequently in divide and conquer algorithms. In this case, ***a*** is the number of sub-problems that you create in the "divide" step, and ***n/b*** is the size of the individual sub-problems created and ***$f(n)$*** is the time spent in dividing and merging. For example:

- $T(n) = T(n/2) + c$: A divide and conquer algorithm wherein each step creates one sub-problem of half the size of the original problem, and where a constant time of c units is spent in dividing and merging
- $T(n) = 3\ T(n/4) + 1$: A divide and conquer algorithm wherein each step creates 3 sub-problems of one-fourth the size of the original problem, and where a unit time is spent in dividing and merging

Intuitively, all other things being equal, an algorithm is more efficient if it creates fewer sub-problems or if the sub-problems are of smaller size. We will attempt to validate this intuition when we learn to solve the recurrence relations next.

There are three general approaches for solving recurrence relations:

- Substitution method: Guess a solution and prove by induction
- Recursion tree unfolding method: Expand and analyze the pattern
- Master theorem: A cookbook method that allows us to solve some special cases without intricate analysis by characterizing the given recurrence relation as one of predefined cases.

4.1.1 Substitution Method

In this method, we "guess" a solution, and prove it using principle of mathematical induction. This may be best explained using an example.

Consider a given recurrence relation, such as, $T(n) = 2\ T(n/2) + cn$. In order to use the substitution method, we first "guess" the solution. Suppose we first guess that the solution is $T(n) = O(n \log n)$.

To prove this using induction, we first assume $T(m) \leq k\ m \log m$ for all $m < n$. *[We observe that while we have not assigned an exact value to the constant k, we have picked the constant k nevertheless. If we are only able to prove that $T(n) \leq k'\ n \log n$, for some value of $k' > k$, that will not be sufficient to prove our hypothesis.]*

Therefore, using the recurrence relation and the induction hypothesis we obtain that:

$$\begin{aligned} T(n) &= 2\,T(n/2) + cn \\ &\le 2\,kn/2 \log(n/2) + cn \\ &= kn \log n - (k-c)n && // \log(n/2) = \log n - 1 \\ &\le k\,n \log n, \text{ as long as } k \ge c. \end{aligned}$$

Therefore, there exists a constant k for which the hypothesis is true.

However, this induction hypothesis does not satisfy the base case of induction, if $T(1) > 0$, since $log(1)$ is 0. Therefore, we need to adjust the induction hypothesis, to satisfy the base case. In this specific example, we can say: $T(m) \le km \log m + k'm$. Using the recurrence relation, we then obtain that $T(n) \le 2\,[k\,n/2 \log n/2 + k'\,n/2\,] + cn$, which is less than or equal to $k\,n \log n + k'n$ as long as $k \ge c$ and the base case is also satisfied as long as $k' \ge T(1)$. Thus, we conclude that $T(n) = O(n \log n)$.

4.1.2 Recursion Tree (Unfolding) Method

In this method, we "unfold" (that is, expand) the recurrence relation, analyze the pattern that emerges, and use mathematical series to solve the recurrence.

For example, let us consider $T(n) = T(n/2) + 1$. If we expand this recurrence, we obtain that $T(n/2) = T(n/2^2) + 1$, and therefore: $T(n) = T(n/2^2) + 2$. Expanding it one more time, we obtain that $T(n) = T(n/2^3) + 3$. In more general terms, we obtain that $T(n) = T(n/2^k) + k$. When $k = log_2 n$, we have $T(n) = T(1) + log_2 n$, or in other terms, $T(n) = \theta(log\ n)$.

As another example, let us consider $T(n) = 3\,T(n/2) + 1$. If we expand this recurrence, we obtain that $T(n/2) = 3\,T(n/2^2) + 1$. That is, $T(n) = 3^2\,T(n/2^2) + 3 + 1$. Expanding this k times, we obtain $T(n) = 3^k\,T(n/2^k) + 3^{k-1} + 3^{k-2} + .. + 1$. Using geometric progression, this can be rewritten as, $T(n) = 3^k\,T(n/2^k) + (3^k - 1)/2$. Assuming $T(1) = c$ and using $k = log_2 n$, we obtain $T(n) = 3^k\,(c+1/2)$, that is, $T(n) = \theta(3^k)$. Since $3\wedge log_2 n$ is the same as $n \wedge log_2 3$, we conclude that $T(n) = \theta(n \wedge log_2 3)$.

Tip: We frequently observe n^log$_b$a as a component in our solutions when solving recurrence relations of the form T(n) = a T(n/b) + f(n).

4.1.3 Master Theorem

Master theorem is a cookbook solution method for the recurrence relations of the form $T(n) = a\ T(n/b) + f(n)$, that examines values a, b and $f(n)$ to determine a closed form expression for $T(n)$. While the substitution method and the recursion tree unfolding method can be used for recurrence relations of other forms as well, master theorem is a method expressly for the recurrence relations of the form $T(n) = a\ T(n/b) + f(n)$.

Before stating the actual master theorem, we can observe from the previous sections that the expression $n\char`^log_b a$ appears rather frequently in such recurrence relations.

Specifically, master theorem states:

Case 1.	If $f(n) = \theta(n^c)$ where $c < log_b\ a$, then $T(n) = \theta(n\char`^log_b\ a)$
Case 2.	If it is true, for some constant $k \geq 0$, that $f(n) = \theta(n^c\ log^k\ n)$ where $c = log_b\ a$, then $T(n) = \theta(n^c\ log^{k+1}\ n)$
Case 3.	If it is true that $\theta(n^c)$ where $c > log_b\ a$, then $T(n) = \theta(n^c)$

Intuition behind the master theorem

While this is not a formal proof of the master theorem, we can use the recursion tree unfolding method to develop some intuition for this method.

If we unfold the recurrence relation $T(n) = a\ T(n/b) + f(n)$, we get an expression like:

$$T(n) = a^k\ T(n/b^k) + f(n) + a\ f(n/b) + \dots + a^k\ f(n/b^k)$$

For $k \approx log_b n,\ n/b^k = 1$, and we can assume $T(n/b^k) = c$.

$$\begin{aligned}\text{Then, } T(n) &= a\char`^(log_b n) + f(n) + af(n/b) + \dots + a^k\ f(n/b^k)\\ &= c\ .\ n\char`^(log_b a) + f(n) + af(n/b) + \dots + a^k\ f(n/b^k)\end{aligned}$$

We note that there are about $log_b n$ terms. We observe that there are three cases, which correspond to the cases specified in the master theorem:

Case 1.	If $f(n)$ is very small, say a constant, then the first term dominates, the other terms can be ignored
Case 2.	If $f(n) = \theta(n\char`^(log_b a))$, then there are $log\ n$ terms, and each term is same as $f(n)$. Specifically, we observe that $af(n/b) = f(n)$ in this case. Therefore, $T(n) = f(n)\ log\ n$.
Case 3.	If $f(n)$ is too large, then $f(n)$ terms dominate. Specifically, from all of the $f(n)$ terms, f(n) itself dominates.

Applying Master Theorem to Example Recurrences

1. $T(n) = 2\ T(n/2) + n\ log\ n$
   $a = b = 2$
   $f(n) = n\ log\ n$
   $log_b a = 1$
   $f(n) = \theta(n\char`^(log_b a)\ log\ n)$

Therefore, by Master Theorem,
$T(n) = \theta(n \log^2 n)$

2. $T(n) = 3\,T(n/2) + n^2$
$a = 3, b = 2$
$f(n) = n^2$
$\log_b a = \log_2 3$
f(n) = θ(n2) where $2 > \log_b a$
Therefore, $T(n) = \theta(n^2)$

4.2 THE DIVIDE AND CONQUER TEMPLATE

The divide and conquer algorithms typically begin by first generalizing the problem to a notation that can solve problems other than the one that was directly posed. We refer to this as the "generalization" step. It may be a bit counterintuitive that a generalization, that is, making the problem harder, actually helps in solving it, but finding the right generalization is sometimes the key to solve a problem efficiently. One reason that a generalization may be helpful is that when invoking the algorithm recursively, the problem instances can be different, and a more general problem structure allows us to invoke the same algorithm for those different problems.

This is what Wikipedia says about this step: " ... it is often necessary to replace the original problem by a more general or complicated problem in order to get the recursion going, and there is no systematic method for finding the proper generalization."

Here is the general template.

Step 1.	If the input J is small, solve it directly using a brute force, or another trivial method.
Step 2.	Divide input J into two or more parts $J_1, J_2, \ldots$
Step 3.	Call the algorithm recursively on individual inputs $(J_1, J_2, \ldots)$ to get sub-solutions $(S_1, S_2, \ldots)$
Step 4.	Merge the sub-solutions $S_1, S_2, \ldots$ into a global solution S

4.3 BINARY SEARCH

We consider the problem of searching for a given number in a sorted array. The well-known binary search algorithm operates by comparing the given number to the midpoint of the array, and then deciding which half of the array to search further.

```
// Given a sorted array A and key, searches for that
key
BinarySearch (A, low, high, key)
   If (high - low < 5) {
      // Search iteratively through the array and
      return
   }
   Mid = (low + high) / 2
   If (key < A[mid]) {
      Return BinarySearch (A, low, mid, key)
   } else {
      Return BinarySearch (A, mid+1, high, key)
   }
```

In each iteration, the binary search algorithm is able to eliminate one half of the array. The recurrence relation can therefore be written as: $T(n) = T(n/2) + c$, where the constant c represents the cost of computing the mid-point and comparing it to the given key.

By using the recursion tree unfolding method, we can obtain a closed form expression of $T(n) = \theta(\log n)$ for this recurrence relation.

Therefore, the binary search algorithm runs in $\theta(\log n)$ time.

4.4 MERGE SORT

Consider the classic problem of sorting a given array, which may be invoked with the simple command: Sort(Array a). To apply the divide and conquer template, we first generalize the problem as follows: Given an array *and indexes i and j (start and end)*, sort *that portion* of the array.

The Merge Sort algorithm solves this problem by recursively sorting the first and the second halves of the given array, and then merging the sorted sections. The pseudo-code is given here:

```
// Given an unsorted array and bounds
// Recursive algorithm MergeSort sorts that array
segment
Algorithm MergeSort (input: Array a, int i, int j) {
   if (j - i < THRESHOLD) {
      InsertionSort(a,i,j)
   }
   int k=(i+j)/2
   MergeSort(a,i,k)
   MergeSort(a,k+1,j)
   Merge(a,i,k,k+1,j)
}
```

4.4.1 Merging

In the last line of the merge sort algorithm, we need to merge two lists represented by two different sections of the array. That leads us to the question of merging two lists effectively. We consider the following intuitive method:

- Compare the first elements of the two lists
- The smaller of the two elements is added to the merged list
- We keep repeating while both the lists are non-empty. If one of the lists becomes empty, we append the non-empty list to the merged list.

In the worst case, merging two lists of x and y elements respectively using this merging algorithm may require $x + y - 1$ comparisons.

4.4.2 Time Complexity of Merge Sort

Time complexity of merge sort can be written using the following recurrence relation.

$$T(n) = 2T(n/2) + \theta(n)$$

Using the methods for solving recurrence equations discussed previously, we can conclude that $T(n) = \theta(n \log n)$.

4.5 QUICKSORT

Invented in 1960 by C. A. R. Hoare, Quicksort is a very widely used and well-studied sorting algorithm[1]. It is a fairly easy to implement algorithm and sorts the array in-place, that is, without requiring extra space.

Quicksort begins by selecting a "partition" element, and then partitioning the array into "left" and "right" portions (not necessarily equal) based on the partition element. Quicksort is also a divide and conquer algorithm, and involves two recursive calls to sort the left and right sides. The pseudo-code for Quicksort is given next:

```
// Algorithm quicksort to sort [p..r] portion of array A
quicksort(A,p,r)
   if (p < r) {
      find a partition element, q // The "central"
      problem
      partition (A,p,r,q)
      quicksort(A,p,q-1)
      quicksort(A,q+1,r)
   }
```

4.5.1 Central problem in Quicksort

The central problem in Quicksort is how to find a **good partition element**.

Once a partition element is found, it is straightforward to partition efficiently around that partition element so that the partitioning element (q) is its final position, every element smaller than q is to the left of q, and every element larger than q is to the right of q. A simple algorithm for partitioning can be given as: evaluate each element in the array A and move elements less than or equal to q to the array B, and move elements greater than q to array C. Copy all the elements of the array B, the element q and the array C to form the new partitioned array A'. A better algorithm for partitioning can use two pointers to start from each end of the array and swap the elements that are in wrong place relative to element q.

4.5.2 Time Complexity Analysis of Quicksort

From the pseudo-code of the partition routine, we can observe that the partitioning algorithm runs in linear time. The recurrence relation can then be written as:

1 In his 1975 doctoral thesis at Stanford University, Sedgewick analyzed many variations of Quicksort algorithm.

$T(n) = T(n_1) + T(n_2) + cn$, where $n_1 + n_2 = n - 1$

The values n1 and n2, and therefore, the recurrence relation, depends on the kind of split caused by the partition element. In the worst case, we may see a very bad split every time, and the recurrence relation can be written as: $T(n) = T(n-1) + cn$, assuming that $T(0) = 0$. This leads to the closed form expression for the worst case complexity of $O(n^2)$. In the best case, we may see a perfectly even split every time, and in that case, the recurrence relation can be written as: $T(n) = 2\ T(n/2) + cn$, which our analysis of Merge Sort algorithm leads us to the closed form expression of $T(n) = O(n \log n)$. However, the probability of observing a perfectly even split or a perfectly uneven split every time is very low. Therefore, we need to analyze Quicksort in the average case to assess its running time.

We make the following observation about partitioning. If a partition element is selected randomly, it has a $1/n$ probability of being the n-th largest element. Stated differently, it has a *50%* chance of being between the *25th* and *75th* percentile elements. Let us define such a partition to be a "good" partition element. If we find a partition element that is not good (along the lines of this definition), then we may simply discard it, and select a different partition element. Since probabilistically, there is a *50%* chance of it being a good partition element, the expected number of times that we have to try before finding a good partition element is *2*. In that case, we can write the average case recurrence relation as: $T(n) = T(n_1) + T(n_2) + 2cn$, where we are guaranteed that both n_1 and n_2 are between $n/4$ and $3n/4$. In this case, the number of recursive calls before we reach an array of size *1* is limited to $\log_{4/3} n$. At each level, the time spent in the partitioning phase is no more than $2cn$. Therefore, in the average case, the total running time is $O(n \log n)$.

Average case can also be analyzed using recurrence relations by writing $T(n)$ as follows:

$$T(n) = cn + \sum_{i=1}^{n} \left(T(i) + T(n-1-i)\right)$$

This yields the same $O(n \log n)$ answer, although the analysis is slightly more complicated.

Comparing Merge Sort and Quicksort

Merge Sort and Quicksort are both divide and conquer algorithms. However, Quicksort spends more time upfront (in partitioning), and after the recursive calls, there is no need to merge. Merge Sort makes the recursive calls with little preliminary work, and then spends the time in merging the results from the recursive calls.

4.6 MEDIAN FINDING

Median finding problem is defined as follows. We are given an **unsorted** array of numbers, and we need to find the median number. A simple solution is to sort the array, and then find the middle element (or the average of the two middle elements in case the array length is even). This solution requires $O(n \log n)$ time for the sorting phase. Is there a more efficient algorithm?

We first generalize the median finding problem to instead find the k-th smallest element in the given range of the array. Finding the median of the entire array is akin to finding the $(n/2)^{th}$-smallest element. The generalized problem is also referred to as "selection" problem or "order statistics".

Having generalized the problem, we can use the ideas presented in the average case analysis of Quicksort algorithm to design divide and conquer algorithms to find the median in linear, that is, $O(n)$ time. There are two distinct algorithm, which we describe next.

4.6.1 QuickSelect Algorithm—Probabilistic Version

The probabilistic version of the QuickSelect algorithm is defined as follows.

QuickSelectProbabilistic *(A, i, j, k)*: Returns k-th smallest element in the *(i,j)* range of given array.

1. Partition the given array using a random partition element.
2. Repeat step 1, until the partition is "good", that is, each of the partitions contains at least a quarter of the elements. Suppose the index of the good partition element in array A is q.
3. If $k > q$ of the left partition, invoke the algorithm recursively on the right partition, otherwise invoke the algorithm recursively on the left partition.

Asymptotic Analysis of Probabilistic QuickSelect

When a random partition element is used on an array of size n, there is a $1/n$ probability that the partition element will split the array into two sections of sizes j and $n-j-1$, for each value of j from 0 to $n-1$. Therefore, there is a 50% probability that the partition will be "good", that is, each of the partitions contains at least a quarter of the elements.

Therefore, the expected number of times that step 1 needs to be repeated before we find a good partition is 2. (Why? Consider an event that happens with a probability p. Say the expected number of times before you observe the event is E. Then, we can write, $E = p.1 + (1-p)\cdot(E+1)$ by observing that after one attempt, the event either occurs with probability p, and with probability

$1-p$ it doesn't occur and therefore we need E more attempts to observe the event. That equation then leads to $pE = 1$.)

The recursive call will then be made on an array of size no more than $3n/4$ where n was the original length. Therefore, the recurrence relation can be written as:

$$T(n) = 2cn + T(3n/4)$$

Using any of the methods for solving recurrence relations, we can observe that $T(n) = O(n)$.

4.6.2 QuickSelect Algorithm—Median of Medians Version

Next variation of the QuickSelect algorithm uses the same recursion logic in the divide and conquer but uses a slightly different mechanism to achieve a good partition. Instead of using a random partition element and probabilistic analysis, we can use the approach defined as follows.

QuickSelect (A, k): Returns k-th smallest element in given array

1. Divide the array into groups of 5. There are $n/5$ groups.
2. Sort the small groups using insertion sort. Since each group is of 5 elements, it takes a constant amount of time to sort those groups.
3. Collect all the $n/5$ medians from the $n/5$ groups.
4. Find the median of the medians by recursively calling the QuickSelect algorithm. We observe that there are $3n/10$ of elements smaller than the median of medians and $3n/10$ of elements larger than median of medians.
5. Partition the array on the median of medians.
6. Similar to the probabilistic algorithm, invoke the algorithm recursively on the left or the right partition depending on the value of k and the size of the partition.

Asymptotic Analysis of Median of Medians QuickSelect

We observe that the steps 1, 2, 3 and 5 all require linear amount of time. Step 4 is a recursive call to find the median of medians. Step 6 is another recursive call after removing at least $3n/10$ of elements. The recurrence relation can thus be written as:

$$T(n) = cn + T(n/5) + T(7n/10)$$

We can prove that $T(n)$ is linear, that is $O(n)$, by using substitution method. Our induction hypothesis is that $T(m) \leq 10\ cm$ for all values of $m < n$.

Therefore,

$$T(n/5) \le 2cn$$
$$T(7n/10) \le 7cn$$
$$T(n) \le cn + 2cn + 7cn = 10\ cn$$

Therefore, the equality holds for n, and by principle of mathematical induction, inequality holds for all values of n.

4.7 CLOSEST PAIR OF POINTS

The closest pair of points problem is defined as follows: Given n points on the two-dimensional plane, find the pair of points such that the distance between them is smaller than any other pair of points.[2]

A naïve algorithm to solve this problem requires $O(n^2)$ time to compute all pair-wise differences and then find the closest pair. A divide and conquer algorithm that finds the closest pair in $O(n \log n)$ time can be constructed as follows:

1. Split the set of points into two equal-sized subsets by finding the median of the x-dimensions of the points. (From our learnings from the previous sections, this can be done in linear time.) We define the vertical line on that median x-dimension to be the "dividing vertical" of the set of points.
2. Solve the problem recursively subsets to the left and right subsets. Define δ to be the minimum among the minimum distances in the left and the right subsets.
3. Find the minimal distance among the pair of points in which one point lies on the left of the dividing vertical and the second point lies to the right. We only need to examine the points that are within a distance δ of the dividing vertical, since otherwise the distance between them cannot be less than δ. Define the minimal distance thus found to be δ'.
4. The final answer is the minimum among δ and δ'.

The main complication in this divide and conquer algorithm is in Step 3. If we assume that the Step 3 can be accomplished in linear time, then the overall recurrence relation will look like: $T(n) = 2\ T(n/2) + cn$, which we can solve to get $T(n) = O(n \log n)$.

We argue that Step 3 can indeed be done in linear time by making the following set of observations:

2 This problem is different from a related problem wherein we are given a point, and we want to find the nearest neighbor from that given point.

1. Each point p on the left side of the vertical line only needs to be compared with a set of points S_p that are: (i) on the right side of the vertical line, (ii) within δ x-distance from the dividing vertical line, and (iii) δ y-distance of the point p. There cannot be more than six points in set S_p, otherwise there would be a pair of points that would have distance less than δ, contradicting the definition of δ.
2. Since there are only $n/2$ points on one side of the vertical line, that implies that there are only $3n$ pairwise comparisons to be made in Step 3 of the algorithm.

This is the main idea of Step 3. Some more implementation details, such as, how to find the set of points S_p without increasing the time complexity are left as an exercise to the reader.

4.8 MATRIX MULTIPLICATION

Given matrices A and B of size $n \times n$ each, the product C of the two matrices is defined as follows. C_{ij}, the element in the i-th row and j-th column of C is obtained by pairwise multiplying the elements of i-th row of A with j-th column in B. One may compute each entry in the third matrix one at a time. This leads to a straight forward algorithm to compute each element of the product matrix in $O(n)$ time, and the entire product matrix in $O(n^3)$ time.

For many years, this straightforward algorithm was the best-known algorithm. In 1969, Volker Strassen published the first algorithm that pointed out that the standard approach is not optimal.

Strassen's divide and conquer algorithm for matrix multiplication is described as follows:

1. Firstly, we assume that A and B are square matrices of size 2^k, that is, n is a power of 2. If they are not of this size, we fill the missing rows and columns with zeroes. This structure allows us to divide the problem into sub-problems.
2. We partition A into 4 equally sized block matrices as: A_{11}, A_{12}, A_{21} and A_{22}. Similarly we partition B into B_{11}, B_{12}, B_{21}, B_{22} and C into C_{11}, C_{12}, C_{21}, C_{22}. We observe that each of the smaller block matrices is of size 2^{k-1}.
3. ***[Only an observation, not an actual step in the algorithm:]*** We can compute the product matrix C as $C_{11} = A_{11} B_{11} + A_{12} B_{21}$. Similarly, $C_{12} = A_{11} B_{12} + A_{12} B_{22}$, $C_{21} = A_{21} B_{11} + A_{22} B_{21}$ and $C_{22} = A_{21} B_{12} + A_{22} B_{22}$.

4. The trick that is used in this algorithm is that instead of computing the *8* products, we compute *7* intermediate products, defined as follows.
   - $M_1 = (A_{11} + A_{22}) (B_{11} + B_{22})$
   - $M_2 = (A_{21} + A_{22}) B_{11}$
   - $M_3 = A_{11} (B_{12} - B_{22})$
   - $M_4 = A_{22} (B_{21} - B_{11})$
   - $M_5 = (A_{11} + A_{12}) B_{22}$
   - $M_6 = (A_{21} - A_{11}) (B_{11} + B_{12})$
   - $M_7 = (A_{12} - A_{22}) (B_{21} + B_{22})$
5. We can now express the final product matrix in terms of intermediate matrices as follows:
   - $C_{11} = M_1 + M_4 - M_5 + M_7$
   - $C_{12} = M_3 + M_5$
   - $C_{21} = M_2 + M_4$
   - $C_{22} = M_1 - M_2 + M_3 + M_6$

4.8.1 Time Complexity Analysis of Strassen's Algorithm

Strassen's algorithm involves *7* recursive calls for the multiplication operations and a constant number of addition/subtraction operations. Addition/subtraction operations involving $n \times n$ matrices require $O(n^2)$ time, as each element can be calculated in a constant time. The recurrence relation can then be written as:

$$T(n) = 7\ T(n/2) + c\ n^2$$

We can solve this recurrence relation to $T(n) = O(n\hat{}log_2 7)$, that is, $O(n^{2.81})$.

Therefore, Strassen's algorithm is asymptotically faster than the straightforward $O(n^3)$ algorithm. Many other algorithms have been designed since Strassen's breakthrough algorithm that are asymptotically superior; an example being an $O(n^{2.375})$ algorithm by Coppersmith and Winograd.

4.8.2 Understanding Strassen's Algorithm

The definitions of intermediate matrices in Strassen's algorithm are not intuitive, and may appear to be ad hoc. In order to build our intuition regarding Strassen's algorithm, we can start with an easier problem of multiplying complex numbers. Suppose we are given two complex numbers $z_1 = x + iy$ and $z_2 = u + iv$. The product $z_3 = z_1 \cdot z_2$ is then defined as $(xu - vy) + i(xv + uy)$. This product requires *4* individual multiplications and *2* addition/subtraction operations. A different way to compute z_3 can be by first computing *3* products $p_1 = (x+y)(u+v)$, $p_2 = xu$ and $p_3 = vy$. Then, z_3 can be written as $(p_2 - p_3) + i(p_1 - p_2 - p_3)$. This alternate method involving *3* individual multiplications and *5* addition/subtraction operations shows the concept of rearranging terms to reduce costly multiplication operations.

Suppose we rename the A_{11}, A_{12}, A_{21} and A_{22} matrices in Step 2 of Strassen's algorithm to a, b, c and d, and rename matrices B_{11}, B_{12}, B_{21}, B_{22} to (x, y, z and w). Then, our problem can be restated as: computing expressions $(a\,x + b\,z)$, $(a\,y + b\,w)$, $(c\,x + d\,z)$ and $(c\,y + d\,w)$ using only *7* multiplication operations.

4.9 SUMMARY

Divide and Conquer is a formal algorithm design technique with numerous practical algorithms. This technique often involves the use of recurrence relations to represent their computational complexity, which can be solved using master theorem, substitution method or simply by unfolding and analyzing the expression.

For sorting an array of numbers, Merge Sort runs in $O(n \log n)$ time. Quicksort also runs in $O(n \log n)$ time **on average**. While we mostly focus on worst case analysis, we observe that in case of Quicksort, average case analysis is actually closer to best case analysis.

A linear time algorithm to find the median (or k-th largest or smallest element) in an unsorted array exists that operates without sorting. While asymptotically the algorithm is linear time, the constant hidden in the linear expression is rather large, necessitating caution when using this algorithm.

4.10 HOME EXERCISES

1. Review the merging algorithm described in Section 4.4.1. Can you argue that it is optimal, that is, there exist two lists that will require these many comparisons? Alternatively, can you find a better merging algorithm?
2. Given an unsorted list of integers a_1, a_2, ... a_n, design an algorithm that checks if there is a pair a_i, a_j that adds up to exactly M. The time complexity of your algorithm should be $O(n \log n)$ or better.
3. In the QuickSelect algorithm—probabilistic version, if we define a "good" partition to be such that each partition is at least one-third of the original size, is the resulting algorithm still linear time?
4. In the QuickSelect algorithm—median of medians version, if we use groups of size *7* instead of size *5*, is the resulting algorithm still linear

time? Is the resulting algorithm better, or worse? What if we use groups of size *3*?

5. You are given a sequence of n numbers $A(1), \ldots, A(n)$, with a very special property, that adjacent numbers are "off" by no more than 1. For example, here is a valid sequence: *[100, 99, 99, 98, 99, 100, 100, 101, 102, 103, 104, 105, 104, 103, 103, 103, 102]*. Say the first number in the sequence is x, the last one is y, and you are given a number z, such that $x < z < y$. You have to find the location of z in the sequence (or report that it does not exist). The time complexity of your algorithm should be $O(\log n)$ or better.
6. Consider the "closest pair of points" problem. Suppose we simply sort the points by their x-dimensions in the first step, in $O(n \log n)$ time instead of using the linear time median finding algorithm. How does this change the time complexity of the entire algorithm?
7. Given an array of n unsorted numbers, give a linear time algorithm to find if there exists a number in the array that exists at least 10% of the time, that is, at least $n/10$ times. (For example, if the array has 1000 elements, a number that appears 100 or more times.)
8. Design and implement an algorithm for function **power(integer *a*, integer *n*)** that computes a^n, in $O(\log n)$ time.
9. Consider n people standing in a circle, marked from 1 to n. Suppose every second standing person is asked to sit down, and this process continues in the circle until only one person is left standing. What is the initial index of the last person left standing?
10. Consider the problem of multiplying two complex numbers $z_1 = u + iv$ and $z_2 = w + ix$, where the product is $z_1 z_2 = (uw - xv) + i(vw + ux)$. We observe that the product requires 4 multiplications: uw, xv, vw and ux. Can you rewrite this in a way so that it only involves 3 multiplications instead of 4?

GREEDY METHOD

CHAPTER 5

In this chapter, we describe greedy method—a technique to build a complete solution by making a sequence of "best selection" steps. The exact "best selection" depends upon actual problem, but the important distinction is that the focus is simply on "what is best step from this point", as opposed to characterizing the "best solution overall".

Applications of greedy method are *very* broad, with examples in numerous fields. We list some of these applications and also specify the optimality of the greedy approach for that problem.

- Sorting (Selection Sort, which is suboptimal)
- Merging sorted lists (Optimal)
- Knapsack (Optimality depends upon the type)
- Minimum Spanning Tree (Kruskal's algorithm, Optimal)
- Character Encoding (Hoffman Encoding, Optimal)

Firstly though, we learn about an important property, using which can try to characterize the problems in which greedy method works and in which case it doesn't.

5.1 OPTIMAL SUBSTRUCTURE

A problem is said to have optimal substructure if the sub-solutions of an optimal solution of the problem are themselves optimal solutions for their sub-problems.

This can be best observed by way of an example. Suppose we have found the shortest path from node a to node b in a graph, and suppose that shortest path is $[a, x_1, x_2 \dots x_i \dots x_j \dots\ x_n, b]$.

Then we can prove that the portion of x_i to x_j on that path is a shortest path from x_i to x_j as well. (Proof outline: Hypothetically, if there existed another shorter path from x_i to x_j, then we could construct a shorter path from a to b by using the shorter path from x_i to x_j, thus contradicting the claim that the said path is the shortest path between nodes a and b.)

5.2 SORTING USING GREEDY METHOD

A simple sorting algorithm using greedy method can be described as follows:

- Select the minimum element
- Move it to the beginning
- Continue doing this for the remaining array

```
// Given an unsorted array of numbers
// Sorts the array using the greedy method
Algorithm SelectionSort (Array a[1..n])
For i = 1 to n-1
   For j = i+1 to n
      If (a[i] > a[j]) swap (a[i], a[j])
```

In the *i*-th iteration of the outer loop, the *i*-th smallest element is selected and placed at *a[i]* (hence the name **selection sort**). The algorithm swaps the elements *a[i]* and *a[j]* only if *a[i]* is strictly larger than *a[j]*; equal elements are not swapped. Therefore, the selection sort algorithm is **stable**, that is, equal elements stay in the same relative order as in the input list. Stability is an important characteristic of sorting algorithms in scenarios where the elements have more than one sortable attribute.

Due to the two nested loops, the SelectionSort algorithm takes $O(n^2)$ time. Considering the other sorting algorithms discussed in the Divide and Conquer section that require $O(n \log n)$, we know that this algorithm is not optimal in terms of the time that it takes.

5.3 MERGING SORTED LISTS

We consider an interesting problem next. We are given many sorted lists, and we want to merge all the lists into one consolidated list, by merging two lists at a time using the merge process defined in Merge Sort algorithm. **We need to determine the order (that is, the sequence) in which to merge the lists.**

A brief example can demonstrate that the order can significantly affect the overall number of comparisons required to merge the lists. Suppose we have three lists of sizes $2M$, $5M$, $10M$, and let us consider two alternate merge sequences:

i. If we merge the lists of sizes $5M$ and $10M$ first, then using the merge process described in the merge sort algorithm, we observe that it requires $5M + 10M - 1$, that is, approximately $15M$ comparisons in the worst case. After that, we have two lists of sizes $2M$ and $15M$. To merge those lists requires approximately $17M$ comparisons, thereby resulting in $32M$ total comparisons.
ii. If we merge the list of size 2M with the lists of size 5M, that requires approximately 7M comparisons, and after that we have two lists of size 7M and 10M. Merging those lists requires approximately 17M comparisons, thereby resulting in 24M total comparisons.

Therefore, simply using the second sequence saves approximately 25% in terms of total number of comparisons in this specific example.

5.3.1 Greedy Algorithm

The greedy approach can be to merge the two shortest remaining lists. After that point, the new list is inserted into the pool of lists, and the process is repeated. (As an example of the greedy technique, the algorithm makes its choice only on the basis of two shortest lists, and the decision is not influenced by the sizes of the larger lists.)

To implement this greedy algorithm, we need a data structure that allows us to:

(i) Remove the two smallest elements
(ii) Add an element
(iii) Repeat these steps until we have only one element

5.3.2 Implementation

Although we can implement these steps using a variety of data structures such as a binary search tree, we can also use a slightly simpler data structure of a heap.

```
// Finds the merge sequence, given an array
// that contains the sizes of the lists
Algorithm MergeSequence (Input: Array L[1..n], L[i] is
the size of the i-th list)
   Build the original heap
   For i = 1 to n-1
      Remove two smallest elements from the heap
         // 2 log (n) time for two delete operations
      Add a new element corresponding to the merged
      list
         // log(n) time for one insert operation
```

The total time complexity of the merge sequencing algorithm is: *O(n log n)*, where *n* is the number of lists to merge. We should observe carefully that this time complexity is in terms of *n*, the number of lists to merge. The number *n* has no correlation with the number of elements in any of those lists. Similarly, the goal of the merge sequence algorithm is only to create the correct merge sequence – the actual merging of lists can happen after the sequence has been determined.

5.3.3 Proof of Correctness

The greedy algorithm suggested above can be shown to be optimal using the similar proof by contradiction as outlined in the optimal substructure.

5.4 KNAPSACK PROBLEM

Next we present a central problem in optimization that manifests almost every day, albeit in different form each time. This problem is generally referred to as a "knapsack" problem, in which we are given a knapsack of fixed weight capacity and a set of items of varying weight and reward values. Our goal is to select a subset of items such that the sum of the selected items does not exceed the weight capacity of the knapsack and the reward achieved from the selected items is maximized.

Problem Definition
Input: A weight capacity *C*, and *n* items of weights *W[1:n]* and reward value *R[1:n]*.
To do: Determine which items to select so that the total weight of the selected items is ≤ *C*, and the total reward value is maximized.
An Example
We are given *4* items, with weight values *[104, 291, 200, 213]* and corresponding rewards values *[500, 1000, 550, 800]*. The weight capacity of the knapsack is *500*.

We observe that we cannot select the items *2* and *4* together. Even though those items have high rewards values (*1000* and *800* respectively), their total weight is *504*, which exceeds the capacity of the knapsack. If we select the items *2* and *3*, we can achieve a total reward of *1550*.

An Intuitive Ordering and the Essence of the Problem
The items that have a higher reward to weight ratio present a more attractive choice to be considered in the knapsack. However, for the example shown above, the reward to weight ratio of the *4* items are *[4.81, 3.44, 2.75, 3.76]*. Therefore, the items *1* and *4* are more attractive than items *2* and *3* based on their reward to weight ratios.

As the example shows, we may have a scenario in which the knapsack is not full, and ignoring some items with the high reward to weight ratio may achieve a total reward value. This is the essence of the knapsack problem.

5.4.1 Greedy Algorithm for Fractional Knapsack Problem

We consider a fractional version of the problem, in which we have the option of selecting fractional portions of the items. This is a significant relaxation of the original knapsack problem and instantly, the problem of the knapsack not filling up completely is eliminated. This enables a greedy algorithm, which simply orders the items using reward to weight ratio, to achieve the maximum possible reward value.

The greedy algorithm can then be described as follows.

1. Sort the items using their reward to weight ratio.
2. Starting with the item with the highest reward to weight ratio, continue selecting the item until the weight of the selected items equals the weight capacity of the knapsack.

For the example presented above, the greedy algorithm finds the solution in the following steps:

- (Initial state) No item selected, selected reward = *0*, selected weight = *0*, remaining weight capacity = *500*
- All of item *1*, item reward = *500*, selected reward = *500*, selected weight = *104*, remaining weight capacity = *396*
- All of item *4*, item reward = *800*, selected reward = *1300*, selected weight = *317*, remaining weight capacity = *183*
- *183/291* fraction of item *2*, item reward = *628.87*, selected reward = *1928.87*, selected weight = *500*, remaining weight capacity = *0*.

The algorithm terminates when there is no weight capacity remaining, with a total reward value of *1928.87*.

5.4.2 Practical Applications and Variations

Knapsack problem can be used to model many kinds of optimization problems, such as, advertising channel selection. The reward value of the channel can correspond to the expected revenue to be realized by using that channel. Many different variations appear frequently based on the actual constraints pertaining to the business domain. Here are some examples:

1. An item cannot be selected partially (either select an advertising channel or not)
2. An item can be selected multiple times
3. An item can only be selected if another specific item is selected
4. An item cannot be selected if another specific item is selected
5. An item can only be selected if all of the other specified items are selected.
6. Only one of a specific group of items can be selected.

5.4.3 General Solution Using Integer Linear Programming

We can model a general knapsack problem using an integer linear programming formulation. Let *x[i]* represent whether or not the item *i* is selected, where *x[i]* is *0* or *1*. The weight of the part taken from item *i* is *x[i]*W[i]*. The corresponding reward is *x[i]*R[i]*. The problem then is to find the values of the array *x[1:n]* so that *x[1]R[1] + x[2]R[2] + ... + x[n]R[n]* is maximized subject to the constraint that *x[1]W[1] + x[2]W[2] + ... + x[n]W[n] ≤ C*. If an item can be selected multiple times, we can allow *x[i]* to be any integer. If we are considering multiple constraint knapsack problem, we can include those constraints in the integer linear programming model as well.

5.5 MINIMUM SPANNING TREE

Given a graph (as defined in Section 3.7), we define a spanning tree of a graph to be a tree that contains all nodes in the graph (i.e., *spans the entire graph*), and the edges of the tree are a subset of the edges of the graph. In other words, if we select $n-1$ edges from the graph that create a tree, the nodes of the graph and the $n-1$ selected edges constitute a **spanning tree**. Clearly, this is possible only if the graph is a connected graph.

The **weight of the spanning tree** is defined as the sum of weights of edges in the tree.

A spanning tree with the minimum weight amongst all the spanning trees is called **a minimum spanning tree**. Since many spanning trees may have the same minimum weight, minimum spanning tree may not be unique.

5.5.1 Problem Definition

Problem: Minimum Spanning Tree (MST)
Input: Given a weighted connected graph $G = (V,E)$, where $w[i,j]$ is the weight of edge between vertices i and j. We also use notation $w(e)$ to represent the cost of edge e wherever convenient.

Objective: Find *a* minimum-weight spanning tree of G. [Since the minimum spanning tree does not have to be unique, we can find any minimum spanning tree.]

5.5.2 Kruskal's Greedy Algorithm

There are three commonly known greedy algorithms for finding the minimum spanning tree: Kruskal's algorithm, Prim's algorithm and Boruvka's algorithm. All three are interesting algorithms, however, we study Kruskal's algorithm here, primarily because it shows the greedy property and the local choice property clearly. As a bonus, Kruskal's algorithm also allows us to introduce a new data structure.

Every greedy algorithm can use a different greedy choice or policy. Kruskal's algorithm uses the following selection policy: Select the minimum weighted edge that does NOT create a cycle. The algorithm can be described as follows.

```
// Given a graph and its weight matrix
// Find a minimum spanning tree
Algorithm KruskalMST (in:G, W[1:n,1:n]; out:T)
   Sort edges by weight: e[1], e[2], .. e[m].
   Initialize counter j = 1
   Initialize tree T to empty
   While (number of edges in Tree < n-1) {
      If adding the edge e[j] does not create a cycle,
        add edge e[j] to tree T
      Increment j
   }
```

Using Disjoint-Set Data Structure

There are two operations in Kruskal's algorithm that require our careful consideration. To check whether adding an edge creates a cycle or not, we need to determine if the two end points of the edge are already connected. Similarly, when we add an edge, we need to update our records about which vertices are connected to each other.[1] These two questions can be answered efficiently if we consider "sets" of vertices, where a set is defined by the collection of vertices that are connected to each other. If we want to find whether or not two vertices are connected to each other, it is akin to asking if the two vertices are in the same set.

The data structure that supports the two stated operations is generally referred to as the **disjoint-set** data structure or the **union-find** data structure. The data structure supports the following two operations:

- Find(x): This returns the "set" that contains the element x
- Union(x,y): Merges the two sets containing elements x and y

Union Find data structure (and the algorithm that manages the data structure) has many different variations, but here are some common characteristics of those implementations:

- Each set is identified by a "leader" node.
- Each node has a pointer to its parent node. For a leader node, this pointer points to itself.
- When we call "find" on a node, we recursively navigate to its parent node until we find the leader node. We return the leader node to identify the set containing that element.

1 It is easy to observe that besides the two end points of the edge, many other vertices also become connected when an edge is added.

- Each leader maintains a rank property (or height, depending upon the implementation variation) as well.
- When we perform a union operation, we make the tree with smaller rank (or height) to be a child of the tree with the larger rank (or height). If the two trees are of equal rank, we choose one randomly and increment the rank of the tree that is assigned to be the parent.

We provide a more detailed explanation of the time complexity Union Find data structure in Appendix D, but for the purpose of time complexity analysis of Kruskal's algorithm, it suffices for us to know that each of the union and find operations take $O(\log n)$ time where n is the total number of elements in the disjoint set data structure.

Time complexity analysis of Kruskal's Algorithm
We use the standard notation of using n for the number of nodes and m for the number of edges. In the initialization step, we need $O(m \log m)$ time to sort the edges by weight. Since m is no more than n^2, $\log (m)$ is bounded by $2 \log (n)$. Therefore, we can write this step to take $O(m \log n)$ time.[2]

Inside the while loop, we use two Find operations to check if adding an edge will create a cycle or not. This requires $O(m \log n)$ time, since we can execute this step at most once per edge.

When adding an edge, we use one Union Operation. Since we add $n-1$ edges, this step requires $O(n \log n)$ time.

Therefore, in total, we require $O(m \log n + n \log n)$ time. Since m is larger than n, we can write this as $O(m \log n)$ time.

Proof of Correctness of Kruskal's Algorithm
To prove that Kruskal's algorithm finds a minimum spanning tree, we can use a proof by contradiction. To prove this statement, we need to prove two distinct claims:

Claim 1. The tree produced by Kruskal's algorithm on a connected weighted graph G, is in fact a spanning tree.

Proof: This is true because:

- ***T* is acyclic**: This is true by construction (We add an edge only if it does not create a cycle.)
- ***T* is spanning.** Every vertex v is included in T, because the incident edges of v must have been considered in the algorithm. The least weighted

2 For the same reason, it is quite rare to see $\log m$ factor when studying asymptotic analysis of graph algorithms. It is almost always replaced by $\log n$.

edge amongst that set of edges would have been included because it could not have created a cycle.

- ***T* is connected, i.e., *T* is not a forest.** Suppose that T is not connected. Then T has at least two components. Since graph G itself is connected, then the two components of T must be connected by some edges in G, not in T. The least weight edge amongst this set of edges would have been included in T because it could not have created a cycle. This contradicts the hypothesis that T is not connected.

Claim 2: T is a spanning tree of minimum weight.

Proof: We can prove this using contradiction. Let T' be a minimum-weight spanning tree, such that amongst all the minimum spanning trees, T' has the highest number of edges in common with T. If $T = T'$, then T is a minimum weight spanning tree. If $T \neq T'$, then there exist an edge $e \in T'$ that is not in T. Further, $T \cup \{e\}$ contains a cycle C such that:

a. Every edge in C has weight no more than $w(e)$. (This follows from the sequence in which the edges were added to T.)
b. There is some edge f in C that is not in T'. (Because T' is an MST and does not contain the cycle C.)

Consider the tree T'' constructed by removing edge e, and adding edge f. That is, $T'' = T' \setminus \{e\} \cup \{f\}$. Then, we observe that:

a. T'' is a spanning tree.
b. T'' has more edges in common with T than T'.
c. Weight(T'') $\leq$ Weight(T'). (This is true because $w(f) \leq w(e)$.)

Therefore T'' is also a minimum spanning tree, and it has one more edge in common with T compared to T'. This contradicts the hypothesis that T' is an MST with the highest number of edges in common with T.

5.6 A WORD OF CAUTION (DON'T BE GREEDY WITH GREEDY!)

Greedy algorithms are very easy to apply, and therefore, are prone to overuse. Numerous times students answer a question using greedy algorithm, because they are generally easy to design. However, many times greedy algorithms are a wrong choice, since they produce suboptimal results.

This is a key take away: **it is not justified to use greedy algorithm to produce a suboptimal solution, where another algorithmic technique (such as Divide & Conquer) would have resulted in an optimal solution.**

Greedy algorithms may be simple to design and may be "efficient", that is, they may have time complexity that is asymptotically less than other algorithms. However, **optimality is usually more important than efficiency**. As an example, suppose you are trying to maximize the number of flights that you can schedule using *3* aircrafts. The number of flights represents the "business value" of the schedule. The time complexity merely represents a "cost of computation" of that schedule. If one algorithm (greedy) runs in *5* minutes, but only schedules *7* flights, and another algorithm (based on dynamic programming) runs in *2* hours, but schedules *8* flights, the second algorithm is likely superior.

5.7 HOME EXERCISES

1. Interval scheduling: Suppose you are given a list of lectures with their start time and end times. How can you choose the maximum number of non-overlapping lectures?
2. You are asked to be the organizer for *n* parties and are provided with their start and end times. (For example: P1: 7 AM – 9 AM; P2: 8 AM to 3 PM; P3: 4 AM to 8 AM.) You can only be organizing one party at a time, so you need to choose. For every party that you organize, you are given a fixed reward *(1000$)* irrespective of the length of the party. How do you select the parties to **maximize your reward**? What is the time complexity of your algorithm in terms of *n*?
3. Given a set of symbols and their frequency of usage, find a binary code for each symbol, such that:
   a. Binary code for any symbol is not the prefix of the binary code of another symbol.
   b. The weighted length of codes for all the symbols (weighted by the usage frequency) is minimized.
4. Argue whether or not you would apply Greedy technique to the following problems:
   a. Chess
   b. Sorting
   c. Shortest path computation
   d. Knapsack

HISTORICAL NOTE

A famous greedy algorithm was invented by David Huffman in 1951, while he was a student at MIT and chose to do a term paper (instead of taking a final exam, of course). The professor, Robert M. Fano, assigned a term paper on the problem of finding the most efficient binary code. Huffman, unable to prove any codes were the most efficient, was about to give up when he hit upon the idea of using a frequency-sorted binary tree and quickly proved this method the most efficient. In doing so, the student outdid his professor, who had worked with information theory inventor Claude Shannon to develop a similar code. By building the tree from the bottom up instead of the top down, Huffman avoided the major flaw of the suboptimal Shannon-Fano coding. The resulting algorithm, conveniently named "Huffman coding", is referenced in one of the home exercises above.

DYNAMIC PROGRAMMING

CHAPTER 6

Dynamic Programming (commonly abbreviated as DP) is an algorithm design technique that builds its solution by constructing solutions to smaller instances of the same problem. It is similar to divide and conquer technique, but it is also different in a subtle but significant way: DP builds the solution to many sub-problems that may or may not be required (bottom up traversal), rather than by solving specifically the sub-problems that are required (top down traversal).[1]

An example of difference between top down and bottom up traversal may be observed by using a program to calculate Fibonacci numbers. As we may recall, Fibonacci Numbers are defined recursively as follows:

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

1 It may appear counter-intuitive to solve sub-problems that may not be needed, and in a way, that is the beauty of Dynamic Programming. Next few sections will expand on this intuition and counter-intuition.

A simple recursive program to compute n-th Fibonacci number can be written as follows:

```
function fib(int n) {
   if (n ≤ 2) {
       return 1;
   }
   return fib(n-1) + fib(n-2);
}
```

To estimate the time complexity of this algorithm, we can simply run this program and plot the time (in milliseconds, for example) against n. The following table is an output of such program, although your own results may vary depending upon the actual computer and programming environment.

n	TIME (MSEC)
10	0
20	1
30	8
40	922
50	113770

We can construct an alternate solution to compute Fibonacci numbers as follows:

```
long a[] = new long[n + 1];
a[1] = a[2] = 1;
for (int i = 3; i ≤ n; i++) {
     a[i] = a[i - 1] + a[i - 2];
}
return a[n];
```

This solution also looks quite similar, and we can compare the two solutions by finding the computation time of the two programs for same values of n (and running the two programs on the same computer).

N	TIME (MSEC)	
	RECURSIVE	DP
10	*0*	*0*
20	*1*	*0*
30	*8*	*0*
40	*922*	*0*
50	*113770*	*0*

We can easily observe that the dynamic programming solution outperforms the recursive program quite spectacularly, even though the programs look quite similar. This simple example highlights Dynamic Programming as a computation technique that stores the results of sub-solutions to avoid repeated solving of sub-problems. DP is an especially useful technique when we can observe two properties:

- Optimal substructure
- Overlapping sub-problems

In the coming sections, we study more about these two properties.

6.1 OPTIMAL SUBSTRUCTURE

As discussed in the case of greedy method, a problem is said to have optimal substructure if the sub-solutions of an optimal solution of the problem are themselves optimal solutions for their sub-problems.

The example presented above discusses that if we find the shortest path from node *a* to node *b* to be $[a, x_1, x_2 \ldots x_i \ldots x_j \ldots x_n, b]$, then the portion of x_i to x_j on that path is a shortest path from x_i to x_j as well.

6.2 OVERLAPPING SUB-PROBLEMS

We observe that the calculation of *5*th Fibonacci number, that is, fib(*5*) requires computation of *3*rd and *4*th Fibonacci numbers, that is, fib(*4*) and fib(*3*). However, fib(*4*) requires computation of fib(*3*) and fib(*2*).

Therefore, we observe that fib(*3*) and fib(*4*) are not disjoint (In fact fib(*3*) is entirely included within fib(*4*)). Therefore, when calculating fib(*5*), we are solving overlapping sub-problems.

Similar overlapping sub-problems situation can be observed in many board games (such as, chess), selection problems, and shortest distance problems.

6.3 DYNAMIC PROGRAMMING TEMPLATE

Next, we describe the dynamic programming template that can be consistently used to apply dynamic programming design technique for a given problem. The template consists of four steps.

Step 1. **Notation**: In this step, we develop a mathematical notation that can express any solution and any sub-solution for the problem at hand. This is typically the hardest step and during the course of this template, we may need to revisit the notation to add more parameters to ensure that the notation is sufficiently generalized.

Step 2. **Optimality**: In this step, we prove that the Optimal Substructure (Principle of Optimality) holds. This allows us prove that the dynamic programming will in fact produce result that is optimal.

Step 3. **Recurrence**: In this step, we develop a recurrence relation that relates a solution to its sub-solutions, using the math notation of step 1. If we observe that recurrence relation cannot easily be formed, we may need to generalize the notation further. Similarly if we observe that some parameters of the notation do not play any role in the recurrence, we may eliminate them from the notation.

Step 4. **Algorithm**: Writing the algorithm is usually straight forward. We iterate over all the parameters of the recurrence relation to compute the results for the actual problem that needed to be solved.

This template will become clear over the next few sections as we consider specific problems and use the DP template for those problems.

Mnemonic NORA (Notation – Optimality – Recurrence – Algorithm) can sometimes be helpful in remembering the template.

6.4 MATRIX CHAIN MULTIPLICATION

Next, we consider a matrix chain multiplication problem[2]. We are given n matrices $A_1, A, \dots A_n$ of dimensions $r_1 \times c_1, r_2 \times c_2, \dots, r_n \times c_n$, respectively. Our goal

2 Although it may sound similar, matrix chain multiplication is a different problem from matrix multiplication problem that we studied in an earlier section on divide and conquer algorithms.

is to compute the product $A_1 A_2 \ldots A_n$. Specifically, **we need to determine the order in which we should multiply the matrices pairwise.**

Before we attempt to solve the problem, three observations are in order.

Observation 1. For two matrices A_1 and A_2 of dimensions $r_1 \times c_1$, $r_2 \times c_2$ to be multipliable, c_1 must be equal to r_2. Therefore, extending that argument to matrix chain multiplication problem, c_j must be equal to r_{j+1} for $1 \leq j < n$ for the problem to be well defined.

Observation 2. The cost/time to multiply two matrices of sizes, $(a \times b)$ and $(b \times c)$ can be computed as follows: The product matrix is of dimensions $(a \times c)$, and to compute each element of the product matrix, we need to perform b scalar multiplication operations, and b additions operations. Therefore, the cost/time can be given as ***abck***, where ***k*** represents the cost for: (one scalar multiplication + one scalar addition). Since we are only comparing relative costs, we can drop the constant ***k*** in our further discussions and define the cost/time simply as ***abc***, that is, the product of their dimensions.

Observation 3. Given three matrices, A_1, A_2 and A_3, we can multiply them as $((A_1 A_2) A_3)$ or as $(A_1(A_2 A_3))$. In either case, we observe an equivalent final product matrix, even though the cost can be different. For example, when computing $(A_1 A_2 A_3)$ as $((A_1 A_2)A_3)$, the cost will be $(r_1 c_1 c_2 + r_1 c_2 c_3)$. Similarly, when computing $(A_1 A_2 A_3)$ as $(A_1(A_2 A_3))$, the cost will be $(r_2 c_2 c_3 + r_1 r_2 c_3)$. This observation is the crux of the problem – how to select the order of multiplication which minimizes the cost.

An example can further help us develop our understanding of this problem. Suppose we are given *3* matrices:

- A_1 of dimensions 3×5
- A_2 of dimensions 5×7
- A_3 of dimensions 7×2

The sequence $(A_1 A_2) A_3$ takes $3*5*7 + 3*7*2 = 147$ operations, while the sequence $A_1(A_2 A_3)$ takes $5*7*2 + 3*5*2 = 100$ operations even though, both calculations return same product matrix as the final result.

Next, we apply the DP template to the matrix chain multiplication, following the four steps from the template in the process.

6.4.1 Notation

We develop a notation M_{ij} to denote the cost of multiplying the chain of matrices $A_i \dots A_j$. We observe that this notation can represent the cost of the solution or any sub-solution at hand. We can also define $M(i,i)$ to be 0 for all i, since no multiplication is required. Ultimately, we need to find $M(1,n)$ – the cost of multiplying all n matrices.

6.4.2 Principle of Optimality

To prove that the principle of optimality holds, we observe that every way of multiplying a sequence of matrices can be represented by a binary tree, where the leaves are the matrices, and the internal nodes are intermediary products.

Suppose tree T corresponds to a computation sequence tree for $A_i \dots A_j$. Tree T has a left sub-tree L and a right sub-tree R, where L corresponds to multiplying $B = A_i \dots A_k$, and R to multiplying $C = A_{k+1} \dots A_j$, for some integer k ($i \le k \le j-1$). Then, the cost of tree T is given by: cost(L) + cost(R) + cost(BC). We need to prove that if T is an optimal tree, then L is an optimal tree of $A_i \dots A_k$ and R is an optimal tree for $A_{k+1} \dots A_j$.

We can prove this by contradiction.

Suppose we are able to find a tree L' strictly better than L. That is, cost(L') < cost(L). Then, we can derive T' by replacing L with L'. Cost of tree T' is then given as cost(L') + cost(R) + cost(BC), which is lower than cost(T). This contradicts the optimality of T.

We can arrive at a similar contradiction if we suppose that we are able to find a tree R' better than R.

Therefore, if T is an optimal tree, then L is an optimal tree of $A_i \dots A_k$ and R is an optimal tree for $A_{k+1} \dots A_j$, and the principle of optimality holds.

6.4.3 Recurrence Relation

Next, we develop a recurrence relation that relates a solution M_{ij} to sub-solutions, where the sub-solutions are defined as M_{ix} and M_{xj}, where $i \le x \le j$.

We observe that given a value of k, the cost M_{ij} is given as $M_{ik} + M_{k+1,j} + r_i c_k c_j$. Therefore, we can minimize M_{ij} by finding the value of k that minimizes ($M_{ik} + M_{k+1,j} + r_i c_k c_j$), subject to the constraint $i \le k \le j-1$.

6.4.4 Algorithm

In order to define the algorithm, we use the array notation *M[i,j]* instead of the subscript notation M_{ij} as it lends itself better to an algorithmic representation. In this array, only the upper diagonal values of this array will be used, that is, values such that $i \leq j$.

The algorithm can then be defined as follows.

```
// Matrix Chain Multiplication – Dynamic Programming
// Given a sequence of n matrices and their dimensions
// Find a sequence of multiplication that minimizes
cost
Algorithm MCM_DP (in:n, r[1:n], c[1:n])
   // Initialization Steps
   For i = 1 to n
       M[i,i] = 0
   For i = 1 to n-1
       M[i,i+1] = r[i] * c[i] * c[i+1]

   // Iteration Step
   For j = 2 to n-1
       For i = 1 to n-j
           M[i,i+j] = mink {M[i,k] + r[i]*c[k]*c[j] +
           M[k+1,i+j]}
```

From the algorithm, we can observe that in the iteration step, the loop runs $O(n^2)$ time, where n is the number of matrices given. Each step requires $O(n)$ time to condition over the value of k. Thus, the overall time complexity of this algorithm is $O(n^3)$.

Implementation Note: The algorithm as defined only finds the minimum cost, not the actual sequence that we set out to find! However, we can easily adjust this algorithm to also save the value of k in a different array (say B, for break point) in the iteration step to memorialize the break point for each value of *(i,j)*. Using that value of k, we can find the actual sequence of matrices that we need to multiply. For example, *B(1,n)* may be *31*, which represents that we need to multiply matrices *1* through *31* and then *32* through n, and then multiply the two product matrices. Similarly, we can lookup *B(1,31)* to identify the breakpoint between *1* and *31* and continue this approach to find the exact sequence. Once the array B has been populated, the exact sequence can be looked up in linear time.

Finally, we make a mental note that this algorithm simply finds the sequence in which to multiply the matrices – not to be confused with matrix multiplication algorithm itself. The matrices may be very large or very small and the time to actually multiply them will depend on their dimensions.

6.5 ALL PAIRS SHORTEST PATH (APSP)

Next, we consider the all pairs shortest paths problem. We are given a weighted graph where the nodes are labeled *1..n*, and the weight of the graph is represented by matrix *W*, where *W[i,j]* represents the cost of the direct edge between nodes *i* and *j*.[3] The graph may be directed or undirected, that is, *W[i,j]* may or may not be the same as *W[j,i]*.

Our objective is to find the distance between every pair of nodes.

As in the case of prior problem, we will apply the dynamic programming template by following the four steps.

6.5.1 Notation

As our first try, we could try to use the notation *D(i,j)* to represent the length of the shortest path from *i* to *j*. However, this notation by itself does not allow us to represent the solution recursively in terms of its sub-solutions. For example, we do not know if *D(1,3)* is a sub-solution for *D(2,5)*, vice versa, or if neither of those is the case.

The exact notation, one that may be a bit counter-intuitive, but one that does allow us to represent the solution recursively in terms of its sub-solutions is the following:

Let $D^{(k)}(i,j)$ denote the length of the shortest path from node *i* to node *j* using nodes *{1 ... k}* as intermediate nodes. We observe that this notation does not say that we **need** to use *k* intermediate nodes. Rather, it says that we are allowed to use only the **set of *{1 ... k}* nodes** as intermediate nodes**.** We may use none, or one, or two, or any number of intermediate nodes from this set, but we are not allowed to use any node outside of this set as an intermediate node.

As a degenerate case of this notation, we observe that $D^{(0)}(i,j)$ signifies that we are not allowed to use any intermediate nodes, and therefore, $D^{(0)}(i,j)= W[i,j]$.

6.5.2 Principle of Optimality

It is easy to prove that the principle of optimality holds in this case, as a portion of a shortest path must be a shortest path as well. This is true even if we are constrained to use a selected set of nodes as our intermediate nodes.

3 In case no direct edge exists between nodes *i* and *j*, then *W[i,j]* can be set to a very high value, such as infinity.

6.5.3 Recurrence Relation

The notation $D^{(k)}(i, j)$ as defined above allows us to form a very simple recurrence relation by observing that shortest path from node i to node j using nodes $\{1 \dots k\}$ as intermediate nodes either uses the node k as an intermediate node or it does not. If it does not use the node k, then $D^{(k)}(i,j)$ must be the same as $D^{(k-1)}(i,j)$. In case it does use the node k, then $D^{(k)}(i,j)$ can be written as the sum of paths from i to k, and from k to j, and each of those paths would only use the set of nodes $\{1 \dots k-1\}$ as the intermediate nodes. Therefore, the simple recurrence relations can be written as follows:

$$D^{(k)}(i,j)=min\{D^{(k-1)}(i,j), D^{(k-1)}(i,k) + D^{(k-1)}(k,j)\}$$

6.5.4 Algorithm

The recurrence relation can then be easily written in form of an iterative algorithm. As usual, this step is an easy step of the DP template.

```
// All Pairs Shortest Path – Dynamic Programming
// Given a graph with vertices {1…n} and the weight
matrix
// Find shortest paths between all pairs of nodes
Algorithm APSP_DP (in: n, W[i,j])
   // Initialization step
   for i=1 to n
      for j=1 to n
          D(0)(i,j) := W[i,j]

   // Build the solution in steps
   for k=1 to n
      for i=1 to n
          for j=1 to n
              D(k)(i,j)=min{D(k-1)(i,j), D(k-1)(i,k) +
      D(k-1)(k,j)}
```

6.5.5 Algorithm Analysis

Due to the three nested loops, it is trivial to observe that the time complexity of the algorithm is $O(n^3)$, where n is the number of the vertices in the graph. We observe that the m, the number of edges in the graph does not appear in the algorithm or in the time complexity.

The space complexity of the algorithm is slightly more interesting. The array D, as defined above has three dimensions: i, j and k, and this suggests that we require $O(n^3)$ space for the algorithm. However, we observe that the computation of $D^{(k)}$ only depends upon $D^{(k-1)}$ array and therefore, once $D^{(k)}$ has been computed, there is no need for us to access $D^{(k-1)}$. Thus, we can save space by not keeping the old values of D array, and simply keep two copies of the array D: the "*previousD*" and the "*currentD*". Then, we can simply using a counter to store the value of k. Based on the value of counter, we can interpret the superscript of the *previousD* and *currentD* matrices.

Therefore, the algorithm can be tweaked to run in $O(n^2)$ space.

6.6 MAXIMUM VALUE CONTIGUOUS SUBSEQUENCE (MVCS)

Next, we study an interesting problem that highlights the importance of coming up with an appropriate notation when using dynamic programming.

The problem is quite simply defined. We are given an Array $A(1..n)$, and we need to find a subarray $A(i..j)$, such that the sum of the elements in the subarray is maximized.

We can observe that if there were no negative elements in the array, then we could just select the entire array and return that as the subarray. Therefore, this problem is meaningful only when we have negative numbers in the array.

As is often helpful, we begin with a brute force solution that finds the sum of all contiguous subarrays.

6.6.1 Algorithm MVCS1

```
// Brute force solution that finds the sums of all
// subarrays, and chooses the maximum
Algorithm MVCS1_Bruteforce (int[] A)
MaxValue = -infinity
for i = 1 to n {
    for j = i to n {
        currSum = findSubArraySum(A,i,j)
        if CurrSum > MaxValue, then MaxValue = CurrSum
    }
}
Return MaxValue

Procedure findSubArraySum(int[] A, int i, int j)
double sum = 0
for int k = i to j {
  sum = sum + A[k]
}
Return sum
```

To calculate the time complexity, we observe that there are three nested loops (two inside the algorithm, and one in the subroutine). Thus, this brute force solution runs in $O(n^3)$ time. Can we do better? We observe that the routine to calculate the sums of subarrays can build upon the previous sum by adding the new element, and that idea is used in the next version of the algorithm.

6.6.2 Algorithm MVCS2

```
// Second version of the MVCS algorithm - Significant
// improvement in calculating the subarray sums
Algorithm MVCS2(int[] A)
InitSubArraySums(); // Calculates all the sums (once)
MaxValue = -infinity
for i = 1 to n {
   for j = i to n {
      currSum = SubArraySum[i][j]
      If currSum > MaxValue, then MaxValue = currSum
   }
}
Return MaxValue

// Procedure that initializes all the sub array sums
Procedure InitSubArraySums() {
   for int i = 1 to n {
      double sum = 0
      for int k = i to n {
        sum = sum + A(k)
        SubArraySum[i][k] = sum
      }
   }
}
```

MVCS2 algorithm runs in $O(n^2)$ time, and is a significant improvement over the MVCS1 algorithm. However, an even faster dynamic programming algorithm exists, which we consider next.

6.6.3 Algorithm MVCS3

Consider the following notation.

Suppose *MVCS(i)* represents the maximum value contiguous subarray that ***ends*** at position i. By our definition, this subarray must include the position i. *MVCS(i)* may or may not include any elements before i.

The principle of optimality holds: if *MVCS(i)* includes any elements before *i*, then *MVCS(i–1)* must be the maximum value contiguous subarray ending at position *i–1*.

The recurrence relation can be defined as follows:

MVCS(i) = max {MVCS(i–1) + A[i], A[i]}

Based on this recurrence relation, we can write the third version of the MVCS algorithm as follows.[4]

```
// MVCS algorithm, using Dynamic Programming.
// MVCS(i) represents the value of maximum value
contiguous
// subarray ending at (and including) position i.
Algorithm MVCS3(int[] A)
MVCS[1] = A[1]
MaxValue = MVCS[1]
for i = 2 to n {
   If (MVCS[i-1] > 0) {
      MVCS[i] = MVCS[i-1]+A[i]
   } else {
      MVCS[i] = A[i]
   }
   If MVCS[i] > MaxValue, then MaxValue = MVCS[i]
}
Return MaxValue
```

This algorithm contains a single loop and runs in *O(n)* time, which is a remarkable improvement over the MVCS2 algorithm.

6.7 LONGEST INCREASING SUBSEQUENCE (LIS)

As the MVCS problem showed, coming up with the right notation can sometimes be the crux of the dynamic programming solution. We may observe the same in another interesting problem that we consider next.

We are given an array *A(1 ..n)* of numbers, and we want to find a subsequence (not necessarily contiguous) that is strictly increasing.

For example, given an array *[1, 7, 2, 8, 4, 1, 6, 11, 3, 15, 5, 12, 14]*, we can find the subsequence *[1 ..2 ..3 ..15]* that is strictly increasing, and consists of

4 When we transition from a logical notation to an algorithm, some things may change. For example, we use an array notation, MVCS[i], in the algorithm. Similarly, in the algorithm, MVCS presents the value of the subarray, rather than the subarray itself.

4 elements. Can you find another subsequence that has *5*, or perhaps *6* elements? For this problem, just the number of the elements in the increasing subsequence is of interest, not the values or the sums of those elements.

Before giving an outline of the dynamic programming solution, we make two quick observations:

(i) The size of the largest increasing subsequence can be no more than n, the size of the input array.

(ii) Consider a binary string of size n. Each such binary string represents a subsequence (*1* representing the elements included *in* the subsequence and *0* representing the elements left *out* of the subsequence), and there are 2^n binary strings. Therefore, a brute force algorithm for this problem that examines all subsequences requires exponential time, and may not be a viable option for inputs of size larger than say *20*. This is in stark contrast to the MVCS problem where a naïve brute force algorithm only requires $O(n^3)$ time.

6.7.1 Dynamic Programming Algorithm for LIS

Suppose $X(i)$ represents the size of longest strictly increasing subsequence that ***ends*** at position i. By our definition, such a sequence must include $A[i]$, and therefore, each $X(i)$ is at least *1*.

We can then write the recurrence relation to calculate the X values as follows.

$X(1) = 1$
$X(i) = max\ \{X(j)\} + 1$, such that $j < i$, and $A[j] < A[i]$

6.7.2 Time Complexity

We observe that to calculate each $X(i)$, we need $O(n)$ time, to examine the $A[j]$ values against $A[i]$, for all $j < i$.

Therefore, we can calculate all $X(i)$ values in $O(n^2)$ time, and then find the largest one in $O(n)$ time. Therefore, the overall time complexity of the algorithm is $O(n^2)$.

Considering that the brute force algorithm runs in exponential time, $O(n^2)$ time is quite impressive. An even faster $O(n \log n)$ algorithm is possible that uses binary search to find the largest value when updating $X(i)$ in the recurrence step.

6.8 SUMMARY

Dynamic Programming is an algorithm design technique that seeks to prevent computation of same problem instances multiple times. Dynamic programming template consists of four steps: developing a notation, proving the principle of optimality, writing the recurrence relation, and finally, writing the algorithm. From these four steps, developing a notation is often the hardest part of the problem as an incorrect notation may lead to an inefficient or incorrect algorithm.

6.9 HOME EXERCISES

1. In the context of matrix chain multiplication problem, consider a divide and conquer algorithm that finds the optimal value of "k" by defining the same recurrence relation as used in the dynamic programming algorithm. Find the time complexity of such a divide and conquer algorithm.
2. In the context of matrix chain multiplication problem, consider a greedy algorithm that simply chooses to first multiply two matrices that minimize the cost of that multiplication operation. **Give a specific example sequence of matrix dimensions** in which the greedy algorithm does not minimize the overall cost of matrix chain multiplication.
3. Implement the matrix chain multiplication DP algorithm, by adjusting the algorithm to find the actual multiplication sequence using the ideas described in the implementation note.
4. **("Barbie's Array of Diamonds")** Barbie has n diamonds. Each diamond has two attributes: shiny value and weight value. Barbie wants to create a "diamond line" in which each diamond is both shinier and heavier than the previous one. She may not be able to use all her diamonds, but wants to maximize the number of diamonds in this diamond line. Give a polynomial time algorithm for creating a diamond line with maximum number of diamonds. Assume that her initial list of diamonds is not in any specific order.
5. **("Love (Skip) thy neighbor")** Given a list of n positive numbers, your objective is to select the set of numbers that maximizes the sum of selected numbers, given the constraint that we cannot select two numbers that are located adjacent to each other. Describe a **linear time algorithm** for this problem.
6. **("Around the block party planning")** Consider a row of n houses represented as an array: $A[1 .. n]$, where the phrase "next door neighbor"

having its natural meaning. Each resident is assigned a "fun factor" *F[1 ..n]*, which represents how much fun they bring to a party. Your goal is to maximize the fun of a party that you are arranging, but with the constraint that you cannot select three consecutive neighbors. (So for example, if you invite the *A[5]* and *A[6]* family, you cannot invite the *A[4]* or *A[7]* families.) Give an efficient algorithm to select the guest list.

7. In the context of Maximum Value Contiguous problem, (i) What can be a greedy algorithm? (ii) What can be a Divide and Conquer algorithm?
8. **("Canoeing on the cheap")** You are canoeing down a river and there are n renting posts along the way. Before starting your journey, you are given, for each $1 \leq i \leq j \leq n$, the fee $f(i,j)$ for renting a canoe from post i to post j. These fees are arbitrary. For example it is possible that $f(1,3)= 10$ and $f(1,4) = 5$. You begin at trading post 1 and must end at trading post n (using rented canoes). Your goal is to minimize the rental cost. Give the most efficient algorithm you can for this problem. Prove that your algorithm yields an optimal solution and analyze the time complexity.
9. **("Magical eggs and tiny floors[5]")** You are given 4 eggs and a 30 floor building. You need to figure out the highest floor an egg can be dropped without breaking, assuming that (i) all eggs are identical, (ii) if an egg breaks after being dropped from one floor, then the egg will also break if dropped from all higher floors, and (iii) if an egg does not break after being thrown from a certain floor, it retains all of its strength and you can continue to use that egg. Your goal is to minimize the number of throws. From which floor do you drop the first egg? How do you handle this problem given generally m eggs and an n-floor building?
10. Given two strings (sequences of characters), the longest common subsequence (LCS) problem is to find the longest subsequence (not necessarily contiguous) that exists in both of the input strings. For example, given strings "mangoes" and "mementos", the subsequence "mnos" is common in both and is in fact the longest common subsequence. Given two strings of sizes n_1 and n_2 respectively, find a dynamic programming algorithm to find the longest common subsequence in $O(n_1 n_2)$ time.
11. **("Maximum Value But Limited Neighbors")** You are given an array $a[1 ..n]$ of positive numbers and an integer k. You have to produce an array $b[1 ..n]$, such that: (i) For each j, $b[j]$ is 0 or 1, (ii) Array b has

5 This problem is found on many internet forums, and was apparently also used in a Google placement interview.

adjacent *1*s at most *k* times, and (iii) $\Sigma_{j=1}^{n}\left(a[j]*b[j]\right)$ is maximized. For example, given an array *[100, 300, 400, 50]* and integer $k = 1$, the array *b* can be: *[0 1 1 0]*, which maximizes the sum to be *700*. Or, given an array *[10, 100, 300, 400, 50, 4500, 200, 30, 90]* and $k = 2$, the array *b* can be *[1, 0, 1, 1, 0, 1, 1, 0, 1]* which maximizes the sum to *5500*.

HISTORICAL NOTE

The term "Dynamic Programming" was originally used in the 1940s by Richard Bellman to describe the process of solving problems where one needs to find the best decisions one after another. By 1953, he had refined this to the modern meaning, which refers specifically to nesting smaller decision problems inside larger decisions. Programming in this context refers to a tabular method of computing, not computer programming using Java or C!

EXTRA READING

We can mix the divide and conquer and dynamic programming design techniques using Memoization[6]. Memoization aims to prevent recomputations by storing (memorizing) the return values of function calls. You can read more at: http://en.wikipedia.org/wiki/Memoization

Many interesting problems for dynamic programming can be found at the MIT CSAIL website: http://people.csail.mit.edu/bdean/6.046/dp/

6 The missing "r" is not a typo! It is indeed memoization, and not memorization.

GRAPH TRAVERSAL TECHNIQUES

CHAPTER 7

After a long day with algorithms, some light relaxation may be helpful. So, here is a puzzle. Suppose you have two water pitchers, one capable of holding *8* cotylas[1] of water, and the other capable of holding *5* cotylas[2]. The pitchers are irregularly shaped and without markings, so you can't determine how much water is in either pitcher unless it is completely full or completely empty. You also have a faucet, and as much water as you'd like. Can you get exactly *3* cotylas of water? That seems straightforward—fill up the *8* cotyla pitcher, and transfer the water to the *5* cotyla pitcher. What remains in the first pitcher is *3* cotylas.

However, can you obtain *1* cotyla? How about *2* cotylas? *4* cotylas? *6* cotylas? *7.5* cotylas?

Each of those questions can be answered using simple reasoning and constructive proofs.

However, in a more generalized sense, we could ask ourselves the question, given two pitchers with capacities A and B, can we get exactly C units of water? To answer that general question, we can model a **"move of water"** as follows. Suppose the two pitchers contain x

1 In a saner world, the pitcher would be holding 5 gallons of water or 5 litres of water. However, since the Metric and the British imperial/American systems refuse to converge, as a sign of small protest, I am using a unit of volume from ancient times.
2 The cotyla was a measure of capacity among the Romans and Greeks. It is equivalent to about *280* ml.

and y units of water. We can represent the state of the system as (x,y). Here is a list of all other states of the system that we can reach from the current state:

- $(0, y)/(x, 0)$ // Empty first/second
- $(A, y)/(x, B)$ // Fill first/second
- $(A, x + y - A)$ // Pour water from second to first, $x + y > A$
- $(x + y, 0)$ // Pour water from second to first, $x + y \leq A$
- $(x + y - B, B)$ // Pour water from first to second, $x + y > B$
- $(0, x + y)$ // Pour water from first to second, $x + y \leq B$

Using these observations, we can define an algorithm to solve the puzzles of this form as follows:

```
// General algorithm to solve the puzzles of the form
```

```
// Given two pitchers of capacities A and B and infinite
// supply of water, can we get exactly C units?
Algorithm solvePuzzle (int A, int B, int C)
   While (desired state not found) {
     Make a transition step and reach a new state
     If new state is what we were looking for {
        exclaim ("Eureka Eureka")
     }
     If we have spent enough time {
        giveup ()
        // And optionally..
        giveFlimsyPhilosophicalProof("State not
        possible")
     }
   }
```

However, the "algorithm" as we have defined above contains some loopholes. The main loophole essentially is that we have not yet defined a mechanism to reach a new state. If we start with two pitchers of water, and keep moving the water from the first to the second in one step and from the second to the first in the next step, we may never reach any state other than those two states. While we can try to account for this specific cyclical condition in the algorithm, similar cyclical condition involving four or five or more states may be very difficult to identify. What we really need is a systematic way to explore all possible states.

That sets the stage for us to introduce graph traversal techniques: **A graph search (or traversal) technique is a method to systematically visit all the nodes of the graph.**

In order to use a graph traversal technique, we need to model the solution space in the form of a mathematical graph, wherein each vertex (or node) of the graph represents some meaningful state of the solution space, and each edge represents navigation from one state to another.

There are two basic graph traversal techniques:

(i) Depth-First Search (DFS)
(ii) Breadth-First Search (BFS)

Before we study the two techniques, we discuss the concept of edge classification, which is applicable to both the traversal techniques.

7.1 CLASSIFICATION OF EDGES

Given a graph $G = (V,E)$, the edges of the graph can be classified in context of the forest G' produced by the traversal of G. Each edge in E can be classified in one of four possible ways:

- Tree edges (aka Discovery Edges): Edge (u,v) is called a tree edge, if node v is first discovered by exploring edge (u,v).
- Back edges: Edge (u,v) is called a back edge if it connects a vertex u to an ancestor v, that is, v was discovered before u, and there is a path of tree edges from v to u.
- Forward edge: Edge (u,v) is called a forward edge, if it connects a vertex u to a descendent v, that is, u was discovered before v, and there is a set of tree edges from u to v.
- Cross edges: All other edges that are not classified as tree edges, back edges or forward edges are classified as cross edges. This catch-all definition ensures that all edges are classified even though we have not yet specified the traversal techniques themselves.

7.2 DEPTH FIRST SEARCH (DFS)

Depth First Search (DFS) graph traversal algorithm can be conceptually outlined as follows:

1. Select an unvisited node s, visit it, and treat as the current node
2. Find an unvisited neighbor of the current node, visit it, and make it the new current node;
3. If the current node has no unvisited neighbors, backtrack to the parent of the current node, and make that the new current node.

4. Repeat the steps 2 and 3 until no more nodes can be visited.
5. If any unvisited nodes remain, repeat from step 1.

7.2.1 DFS Implementation

Use of backtracking in the conceptual outline, step # 3 suggests that a stack is a good data structure for DFS implementation.

```
// Depth first search - graph traversal algorithm
Algorithm DFS(input: graph G)
   Stack T, int s, int x
   While (G has an unvisited node) do {
     s = an unvisited node
     visit(s)
     T.push(s)
     While (T is not empty) {
        x = T.top()
        if (x has an unvisited neighbor y) {
          visit(y)
          T.push(y)
        } else {
          T.pop()
        }
     }
   }
```

An alternate algorithmic view that also marks the start and finish times for the exploration of each node can be considered as follows.

```
dfs (Graph $G$) {
    // all vertices of $G$ are first painted white
    while there is a white node $u$ in $G$ {
      dfs-visit($G$, $u$)
    }
}
dfs-visit (Graph $G$, Vertex $u$) {
    the vertex $u$ is painted gray
    $u.s = time{+}{+}$  // $u$ has now been discovered
    for all white successors $v$ of $u$ {
      dfs-visit($G$, $v$)
    }
    $u$ is painted black
    $u.f = time{+}{+}$ // Exploration of $u$ has finished
}
```

Parenthesis Theorem: We can make an interesting observation—if the start of node u is marked as "(u", and the end as "u)", then the overall parenthetical expression is *well-formed*. For example: *(u (v (z z) (w w) v) u)*. We never have a mal-formed expression. This is clear from the recursive nature of the algorithm.

7.2.2 Time Complexity of DFS Algorithm

To calculate the time complexity of the DFS algorithm, we observe that every node is "visited" exactly once during the course of the algorithm. Also, every edge (x,y) is "crossed" twice: one time when node y is checked from x to see if it is visited (if not visited, then y would be visited from x), and another time, when we back track from y to x.

Therefore, the time of DFS is $O(n+|E|)$, or $O(n+m)$.

If the graph is connected, the time is $O(m)$ because the graph has at least $n-1$ edges, and so $n+m \leq 2m+1$, and the time complexity can be written simply as $O(m)$.

7.2.3 DFS Edge Classification Theorem

Theorem: In DFS, every edge of undirected graph G is either a tree edge or a back edge. (In other words, no forward or cross edges exist in G' produced by DFS traversal of G).

Proof: We give a proof by contradiction that cross edges cannot exist G'. Let (x,y) be a cross edge, that is, x and y are in separate sub trees of the DFS tree. We consider two cases:

Case 1: x was visited before y.

We observe that when a search for unvisited neighbors of x was conducted and none found, we backtracked from x, and did not return to x.

However, since y is a neighbor of x and y is not visited at time t, as per DFS, we would have visited y from x before the algorithm backtracks from x. That would make y a descendent of x. This is a contradiction.

Case 2: y was visited before x.

Similar reasoning applies as before, with the roles of x and y reversed.

Therefore, in both case 1 and case 2 we reach a contradiction. Therefore, no such cross edge (x, y) can exist in a DFS tree.

A similar proof that no forward edge can exist in a DFS traversal is left to the reader.

7.3 FIRST APPLICATION OF DFS: CONNECTIVITY

We generalize the problem of establishing graph connectivity to counting the number of connected components of the given graph. If the number of components is one, then the graph is connected. If the number of components is greater than one, then the graph is not connected.

To count the number of components, we use a counter to count the number of times the outer while-loop iterates in the DFS algorithm (or, the number of times the **dfs-visit** routine gets invoked in the alternate view). The counter value when the algorithm terminates is equal to the number of connected components of the input graph G. This is because the body of the outer loop, that is, every iteration, fully traverses the connected component that contains the node v.

Therefore, the DFS algorithm becomes a connectedness-testing algorithm that counts the number of components of a graph in $O(n+m)$ time. We observe that if the number of components is greater than one, the algorithm can be used to identify the various connected components as well.

7.4 SECOND APPLICATION OF DFS: MINIMUM SPANNING TREES IN UNIFORMLY WEIGHTED GRAPHS

A trivial application of depth first search traversal algorithm is to find a minimum spanning tree given a uniformly weighted graph, that is, a graph that has all edges with weight equal to one. Since all spanning trees have n nodes and $n-1$ edges, all spanning trees are of the same weight.

Thus, to find a minimum spanning tree in such graphs, it suffices to find any spanning tree.

DFS yields a spanning tree (if the input graph is connected, otherwise, it is a spanning forest). For a uniformly weighted graph, that tree is a minimum spanning tree. The time to compute the tree is $O(m)$, which is better than the $O(m \log n)$ time minimum spanning tree algorithm for general weighted graphs.

7.5 THIRD APPLICATION OF DFS: BICONNECTIVITY

Next we study biconnectivity—a non-trivial application of depth first search graph traversal algorithm. Biconnectivity informally means that every pair of nodes in a graph is connected using at least two different paths. To formally define biconnectivity, we use the following two definitions:

- A node in a connected graph is called an **articulation point** if the deletion of that node disconnects the graph.

- A connected graph is called **biconnected** if it has no articulation points. That is, the deletion of any single node does not cause the graph to get disconnected.

The Biconnectivity Problem can then be formally defined as follows.

Problem: Biconnectivity
Input: A connected graph *G*
Output: Determine whether or not the graph is biconnected. If the graph is not biconnected, find all the articulation points.

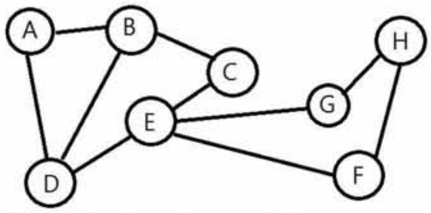


Figure 2: Example graph for a biconnectivity problem. This graph is not biconnected as deletion of node E makes the graph disconnected.

An example input for the biconnected graph is shown in Figure 2. This graph is not biconnected as the node E is an articulation point *(single point of failure).* If there was one more edge, say between C and H, then *E* would not be an articulation point. Biconnectivity is of special interest in the case of networks, as an articulation point is a single point of failure.

We observe that a non-root node is an articulation point if and only that node has a subtree from which no backward edge that ends at a proper ancestor of *x*.

In order to determine if a given graph is biconnected, we introduce following two labels for each node *i*: *DFN[i]* and *L[i]*.

- *DFN[i]*: sequence in which *i* is visited. Thus, the first node visited (i.e., the root) has its *DFN = 1*. The second node visited has a *DFN = 2*, and so on.
- *L[i]*: Lowest DFN number of node which can be reached from node *i* using zero or more tree edges, and then a single back edge; or *DFN[i]*, whichever is lower.

The DFN labels are easy to compute during the depth first search traversal using a simple counter. To determine the *L[i]*, we notice that:

```
L[x]=min{
 DFN[x],
 {DFN[y] | (x, y) is a back edge},
 {L[w] | for each child w of x}
}
```

The entire biconnectivity algorithm using depth first search using these ideas is provided below.

```
// Biconnectivity algorithm, using depth first search
// Uses: Stack T for DFS; labels DFN and L for
biconnectivity
Algorithm Biconnectivity_DFS(input: graph G)
Stack T;
Integer num = 1;
Integer DFN[1:n], L[1:n], Parent[1:n]
Node s = an unvisited node
L[s] = DFN[s] = num++
mark s as visited
T.push(s)
While (stack T is not empty) do
   Node x = top(T)
   if (x has an unvisited neighbor y) then
      mark y as visited
      T.push(y)
      DFN[y] = num++
      Parent[y] = x
      L[y] = DFN[y]
   else
      pop(T)
   for (every neighbor y of x) do
      if (y != parent[x] and DFN[y] < DFN[x]) then
/* y is an ancestor of x, and (x,y) is a back edge*/
         L[x] = min(L[x],DFN[y])
      else if (x = Parent[y]) then
          L[x] = min(L[x],L[y])
         if (L[y] ≥ DFN[x] and x is not root) then
           return x as an articulation point
if (s has more than one child) then
   return s as an articulation point
return true // Graph G is biconnected
```

7.6 BREADTH FIRST SEARCH (BFS)

Breadth first search uses a fundamentally different procedure compared to depth first search. As the name suggests, we explore the breadth of a node before the depth.

Like DFS, it starts with any node, which is designated as the root node. When exploring a node, all unvisited neighbors are designated as the child nodes of this node, and all child nodes are "scanned" before doing a "deep dive" into any of those nodes. Once all the neighbors have been scanned, then the control shifts to the child nodes, and all of the child nodes are explored before any of the "grand child" nodes are explored. Therefore, this traversal mechanism ensures that nodes are explored in the order of their "level", that is, their distance from the designated root node.

1. Select an unvisited node *s*, visit it, have it be the root in a BFS tree being formed. Its level is called the current level.
2. From each node *x* in the current level, in the order in which the level nodes were visited, visit all the unvisited neighbors of *x*. The newly visited nodes from this level form a new level that becomes the next current level.
3. Repeat the previous step until no more nodes can be visited.
4. If there are still unvisited nodes, repeat from Step 1.

We observations that the first node visited in each level is the first node from which to proceed to visit new nodes. This first in first out nature of the algorithm suggests that a queue is the proper data structure to remember the order of the steps.

7.6.1 BFS pseudocode

```
Procedure BFS(input: graph G)
Queue Q;  Integer s, x
while (G has an unvisited node) do
   s := an unvisited node
   visit(s)
   Enqueue(s,Q)
   While (Q is not empty) do
      x := Dequeue(Q)
      For (unvisited neighbor y of x) do
          visit(y)
          Enqueue(y,Q)
```

7.6.2 Applications of BFS

The applications of the breadth first search are virtually unlimited. As an example, consider a robot that is looking for signs of life, in an infinite maze (such as the surface of a planet, with the topology of the planet defining the "walls" of the maze). How can the robot explore the search space systematically, without venturing too far from its home base?

Similarly, there are many optimization problems in which the search space is not a physical search space like a planet, but is a virtual search space comprising of the system parameters. As an example, consider the problem of building a new airplane. The length, the height, the weight, the wingspan are all parameters, and there are constraints on these parameters. These constraints may not be linear and may not lend themselves to a simple optimization mechanism such as linear optimization. Instead, the solution space needs to be explored systematically and each point in the solution space needs to be evaluated.

7.7 HOME EXERCISES

1. Refer to the classification of edges discussed in Section 7.1.
   a. What types of edges can be found in a depth first search traversal of an undirected graph? Specifically, why can a forward edge not exist in a depth first search traversal of an undirected graph?
   b. What types of edges can be found in a breadth first search traversal of an undirected graph?
   c. What types of edges can exist in each of those cases if we consider directed graphs?
2. **Cyclic and acyclic graphs:** A graph is called acyclic if it does not have any cycles. Prove that a directed graph is acyclic if the depth first traversal of the graph does not yield any back edges.
3. **Topological sort:** Given a directed acyclic graph $G = (V,E)$, a topological sort T is an ordering of vertices, such that, for each directed edge (u,v) in E, u comes before v in T.
   a. Prove that if the exploration of node u is completed before the exploration of node v in a depth first search traversal of G, then there exists a topological ordering in which u comes before v.
   b. Using the above proof, modify the DFS algorithm to produce a topological ordering that adds vertices to a list as their exploration is finished.
4. When you delete a non-leaf node of a tree, you create more than 1 subtree. Given a tree with n nodes, give an algorithm to find a non-leaf node v, such that deletion of node v leaves no subtree with more than $n/2$ nodes.

BRANCH AND BOUND

CHAPTER 8

Branch and Bound is a general optimization technique that can be applied where other algorithmic design techniques fail. Sometimes abbreviated as B&B, branch and bound is a systematic method for solving optimization problems. As a technique, B&B is much slower than other algorithm design techniques and often leads to exponential time complexities in the worst case. However, the strength of branch and bound is that it can solve many problems that are **intrinsically hard**[1], and if applied carefully, can lead to algorithms that run reasonably fast on average.

B&B method is credited to a 1960 paper by Ailsa Land and Alison Doig [11].

General idea of B&B is a BFS-like search for the optimal solution, where each node in the BFS tree represents a solution, that is, a point in the solution space. Solution space may be infinite, or it may be finite with an exponential number of points. Thus, a general, a complete BFS search would require an exponential amount of time. **One key attribute of B&B is that not all nodes get explored fully.** Rather, using carefully selected criteria, we can determine which node to expand and when, and which nodes to discard without further evaluation.

1 We will study the intrinsic hardness of problems in a later chapter on NP-completeness.

Problem specific insights are used to do as much pruning as possible, and the pruning strategy has a significant impact on the running time of the algorithm.

Since B&B is usually used for hard problems, it is often acceptable to return solutions that are not optimal, but are within some performance bound of the optimal. A **termination criterion** is used to tell the algorithm when to stop based on the solution that has been found. Such solutions are referred to as approximate solutions, and the corresponding algorithms are generally referred to as *approximation algorithms.*

8.1 EXAMPLE PROBLEMS

We present some example problems to explore how B&B design technique can be applied for different applications.

Job Assignment Problem: We are given n jobs and n resources, and an $n \times n$ cost matrix A where A_{ij} is the cost for resource i to perform job j. Our objective is to find a one-to-one matching of the n resources to the n jobs so that the total cost is minimized.
Traveling Salesperson Problem (TSP): Given a complete graph with n vertices, the salesperson wishes to make a tour, visiting each city exactly once and finishing at the city he starts from. Cost of going from city i to city $j = c(i,j)$.
0/1 Knapsack Problem: Given a set of items, each with a weight and a value, determine which items to include in a collection so that the total weight is less than a given limit and the total value is as large as possible. The 0/1 version of the knapsack problem is different from fractional knapsack in that we can only choose to take, or not take, an item—we cannot choose a fraction of an item.

8.2 BRANCH AND BOUND TEMPLATE

The B&B template consists of the following steps. The most difficult step typically is to model the solution space so that can be visualized in the form of a graph. It is important to observe that as in the case of the 0/1 knapsack problem, the problem and the solution definition may not have any mention or hint of a graph. *Rather, the graph that we need to explore is a theoretical graph that models the solution space for the problem.*

[The steps that we outline below are considering a maximization problem. Steps can altered slightly in case of a minimization problem.]

Step 1. Model the solution space in terms of a graph, where each node represents a partial or a complete solution, and each edge represents a step, a decision or a constraint in the solution building process.

Step 2. Develop a strategy to assign an "upper bound" and a "lower bound" for a node. A lower bound refers to one possible solution and the upper bound refers to the maximum possible solution. Strategy depends on the insights of the actual problem and the example problems that we cover in the coming sections will highlight this aspect.

Step 3. Conduct a breadth first search of the graph with the following modifications: (i) Bound each node. (ii) Prune (discard) a node if the upper bound on that node is no larger than a lower bound of another node in the same level. (iii) Explore (branch) the nodes in the descending order of their lower bounds[2].

Step 4. (Termination Step) Terminate the search if solution meets required performance bounds.

8.3 APPLYING B&B TO 0/1 KNAPSACK PROBLEM

The 0/1 knapsack problem is defined as follows: Given a set of items, each with a weight and a value, determine which items to include in a collection so that the total weight is less than a given limit and the total value is as large as possible. We must decide either to take an item, or to not take an item—we cannot take a fraction of any item.

This is a maximization problem as the objective is to maximize the total value.

As the following example shows, a greedy solution that takes the items in the order of maximum reward to weight ratio may not be optimal. Suppose we have 3 Items. Item 1 worth *60*$ weighs *10* lbs, item 2 worth *100*$ weighs *20* lbs, item 3 worth *120*$ weighs *30* lbs, and the knapsack can hold *50* lbs.

Greedy solution orders the items in the order (item 1, item 2 and item 3). So, it takes item 1 and item 2, and realizes a total reward of $*160*. However, there is unused capacity in the knapsack of *20* lbs, but the weight of item 3 exceeds the remaining capacity. An optimal solution can instead take items 2 and 3, and realize a reward of $*220*.

In order to apply B&B template, we apply the following steps of the B&B template:

Step 1. Modeling of the graph: We consider a node of the graph to denote whether or not an item has been selected. The root node signifies no items have been taken, and it has two nodes:

2 A different order for branching is certainly possible

Y1, and N1 to signify that the item 1 has been taken, and item 1 has not been taken respectively.

Step 2. From any node, we can use the greedy algorithm to find a possible solution and therefore establish a lower bound on that node. Similarly, we can use a greedy algorithm and relax the 0/1 constraint to allow the greedy algorithm to select fractional values, and therefore establish a theoretical upper bound on that node. We observe that the theoretical upper bound may not be feasible, but we can safely say that no value higher than the fractional knapsack result is possible.

Step 3. We can conduct a breadth first search of the graph thus created, and we can explore the nodes either in the order of their lower bounds, or in the order of their upper bounds. We can use the bounds established in the Step 2 to prune the graph and to limit the exploration of a node.

Step 4. Depending upon the business context, we can establish 5% or 2% or a different value as an acceptable approximation limit as termination criteria. This allows us to stop the exploration once we find a node within acceptable difference of the theoretical upper bound. This is justifiable, since 0/1 knapsack is an intrinsically hard problem.

8.4 APPLYING B&B TO JOB ASSIGNMENT PROBLEM

In this section, we apply B&B technique to the job assignment problem, which is defined as follows: We are given n jobs and n resources, and an $n \times n$ cost matrix A where A_{ij} is the cost for resource i to perform job j. Our objective is to find a one-to-one matching of the n resources to the n jobs so that the total cost is minimized.

The B&B template itself is largely the same, and we will focus our efforts for this problem in defining upper and lower bounds, instead of the template itself.

We observe that this is a minimization problem, and therefore, we will be changing the interpretation of upper and lower bounds to reflect the nature of the optimization. The lower bound will now be a theoretical lower bound, and the upper bound will be based on an actual result.

Upper bound: Since any permutation is a valid assignment (even if not a very good one), we can use an identity permutation (that is, job *i* is done by resource *i*) as a general upper bound.
Lower bound: The lower bound asks: how much will it cost *at minimum* to do the job assignment? We can establish a lower bound on the job assignment problem based on the following two observations:

(i) Each job must be done—so if we add minimum cost per job, then that must be minimum cost.
(ii) Each person must do a job—so if we add minimum cost per resource, then that must be minimum cost

Since each of the lower bounds holds independently, we can take the **maximum** of these two minimums, as a good "lower bound".

8.5 HOME EXERCISES

1. How can we apply B&B to Traveling Salesperson Problem? Define how the solution space is modeled as a graph. Further, define lower and upper bounds on a node. Consider two cases: (i) The original problem is based on a graph in Euclidian space and therefore satisfies the triangle inequality, and (ii) The original problem does not satisfy the triangle inequality.
2. You are given a Boolean formula involving variables X_1, X_2, ... X_n. The Boolean formula is of form (C_1 AND C_2 AND C_3 ... AND C_m), where each clause is a disjunction (logical "or" function) of the X variables. You have to assign true/false values to the variables so as to maximize the number of clauses that evaluate to true. Present a branch and bound approach for this optimization problem.

MISCELLANEOUS NOTES

A blog post by Prof. Dick Lipton, a Professor of Computer Science at Georgia Tech, says this about Branch and Bound: *"A branch-and-bound algorithm searches the entire space of candidate solutions, with one extra trick: it throws out large parts of the search space by using previous estimates on the quantity being optimized."* His December 2012 blog post "Branch and Bound—Why Does It Work?" explores the theoretical underpinnings of the B&B framework.

SECTION III

INTRINSIC HARDNESS OF PROBLEMS

Thus far, we have broadened our horizon in two significant ways. Firstly, we have moved beyond discussing exact time complexity of algorithms to asymptotic notation for the time complexity of algorithms. In other words, we are interested in classes of algorithms, and we group the algorithms that are asymptotically similar. Secondly, we have learnt algorithm design techniques, which can lead to many algorithms for specific problems.

With those two significant generalizations, we can proceed to explore intrinsic hardness of problems. Our objective is to study classes of problems based on their intrinsic hardness.

We begin this section with the following quote, often attributed to William James, a late 19th/early 20th century American psychologist and philosopher.

"Any question can be made immaterial by subsuming all its answers under a common head ... . The sovereign road to indifference, whether to evils or to goods, lies in the thought of the higher genus."

NP COMPLETENESS

CHAPTER 9

The quintessential book on NP-completeness by Garey and Johnson [12] begins by exploring the following plausible situation: Suppose you are given a very hard problem at work, and you are asked to solve it. You try and do not reach anything useful, to the point that you feel that your job is on the line. How do you then proceed?

As the aforementioned book describes, there are a few excellent alternatives.

Firstly, if you are able to *reduce* one of the famous open problems to the problem at hand, then you can conclude (and convey) that the problem at hand is a hard problem and there are experts who have been attempting to solve the famous open problem and have not been able to solve the problem at hand either (even though they haven't tried the problem at hand *directly*).

Secondly, if you are able to *reduce* the problem at hand to one of the other well-known problems, then you can utilize the solutions to the well-known problem in building the solution to the problem at hand.

In this and the following chapter, we will address both of these ideas.

In order to develop our formal understanding, we need to define a formal **model of computation** in terms of a **Turing Machine**. We begin with a refresher of Turing Machine. Following that, we will define P and NP, two important classes of problems, and discuss their interrelationships.

9.1 TURING MACHINE REFRESHER

A (deterministic) Turing machine is a device that manipulates symbols on a strip of tape according to a table of rules. Described in 1936 by Alan Turing who called it an "a-machine" (automatic machine), the Turing machine is not intended as practical computing technology, but rather as a hypothetical device representing a computing machine.

The excellent textbook on automata theory by Hopcroft, Motwani and Ullman [13] formally defines a (one-tape) Turing machine as a 7-tuple $M = \langle Q, T, B, S, q_0, F, \delta \rangle$, where:

- Q is a finite, non-empty set of *states*
- T is a finite, non-empty set of the *tape alphabet/symbols*
- $B \in T$ is the *blank symbol*—the symbol that exists on the tape by default, and the only symbol that can occur on the tape infinitely often at any step during the computation
- $\sum \subseteq T$—$\{B\}$ is the set of input symbols
- $q_0 \in Q$ is the initial state
- $F \subseteq Q$ is the set of final or accepting states
- δ: $Q - F \times T \rightarrow Q \times T \times$ {Left, Right} is a partial function called the transition function, where Left is left shift and Right is right shift of the input tape.

Given a word (i.e., a string) ***w*** written on the input tape, the machine invokes the transition function at each character of the input to move from one state to another, writes the symbol on the input tape, and moves the input tape either to left or to right. If the machine M ends in one of the final states after reading the entire word ***w***, then the machine M is said to accept the word ***w***. Otherwise, the machine M is set to reject the word ***w***.

A **non-deterministic Turing Machine** is defined similar to the machine M above, except the transition function δ maps each non-final state and tape symbol to a **set** of moves, where each move is a combination of state, tape symbol and {Left, Right}. The non-deterministic Turing Machine can make *any* of the moves in the set.

9.2 EQUIVALENCY OF A "PROBLEM" AND A "LANGUAGE"

Traditionally, we use the word "problem" to define our input and output pairs. For example, we define the **sorting problem** as follows; given an input list of n numbers, create an output list of n numbers, such that, (i) each number from the input list is represented exactly once in the output list, and (ii) the output list is in a certain order.

In formal computation models, we can also use the word "language" for the aforementioned scenario. The input list can be encoded as a string on the input tape of the Turing Machine M, the machine M can modify the contents of the input tape to produce the output. When the input has been processed successfully, the machine M can enter an accepting final state and halt. In case of invalid input, the machine M can enter a non-accepting final state and halt. The string of the valid input is then said to belong to the language L corresponding to the sorting problem. The string of the invalid input is then said to not belong in the language L.

Similarly, consider a decision problem: given an array of numbers and a number x, does the number exist in the array (the **search problem**)? The input to the problem (the array and the number x) can be encoded as a String w. If the answer to the problem instance is Yes, then the string w is set to belong to the language L corresponding to the search problem. If the answer to the problem instance is No, then the string w does not belong to L.

Therefore, we can observe that the words "problem" and "language" can be used interchangeably within this context. Given a Turing Machine M, we can refer to the **language *L* accepted by the machine *M***, or, equivalently, to the **problem *p* solved by the machine *M***.

In summary: Given a problem p, we can devise an encoding mechanism to characterize the language L corresponding to problem p. If we are able to design a Turing Machine M that accepts L, then the machine M is said to solve the problem p.

9.3 CLASSES P AND NP

Having defined the equivalence between problems and languages, we can now define classes of problems as classes (sets) of languages. Two very commonly used classes of problems are classes P and NP, defined as follows:

- P = $\{L/L$ is accepted by a deterministic Turing Machine in polynomial time$\}$. That is, there exists a Turing Machine, that takes at most $O(n^c)$ steps to accept a string of length n, for each string in L.
- NP = $\{L/L$ is accepted by a non-deterministic Turing Machine in polynomial time$\}$. That is, there exists a non-deterministic Turing Machine, that takes at most $O(n^c)$ steps on each computation path to accept a string of length n, for each string in L.

We observe that the only difference in the two definitions is whether the Turing Machine is deterministic or non-deterministic. We further observe that **P and NP are sets (classes) of languages**, and using our prior discussion about relation between problems and languages, we can equivalently say that **P and NP are classes of problems**.

Since every deterministic Turing Machine can also be considered a non-deterministic Turing Machine (with a single move option in each case), every problem in P is also in NP. It is also easy to observe that the class P (and therefore, the class NP) is infinite.

9.3.1 Is P = NP?

Whether the classes P and NP are in fact the same, or if they are distinct, is one of the most important unanswered questions since 1960s. Observing the similar definitions of P and NP, the question of P vs. NP can be restated as: Are non-deterministic Turing machines really more powerful (efficient) than deterministic ones? In other words, does non-determinism help in the case of Turing Machines?

We do know that in the case of finite state automata, there is no significant difference between the capabilities of deterministic or non-deterministic finite state automata. We also know that in the case of Pushdown Automata, there is indeed a difference between the capabilities of deterministic and non-deterministic pushdown automata. However, in the case of Turing Machines, we simply do not know the answer (yet).

Many optimization problems appear amenable only to brute force, i.e., (near) exhaustive enumeration, and many theoretical computer scientists are of the opinion that the classes P and NP are distinct. Nevertheless, there is no known answer for this question in public knowledge.[1]

Jack Edmonds, regarded as one of the most important contributors to the field of combinatorial optimization, and author of the blossom algorithm for constructing maximum matchings on graphs, said in 1966:

The classes of problems which are respectively known and not known to have good algorithms are of great theoretical interest [...] I conjecture that there is no good algorithm for the traveling salesman problem. My reasons are the same as for any mathematical conjecture: (1) It's a legitimate mathematical possibility, and (2) I do not know.

1 The premise of the movie "Travelling Salesman" (2012) is that the classes P and NP are the same, and some "well-connected" people know how to solve hard problems in polynomial time. (You can use this as yet another confirmation that NP-completeness is an exciting Hollywood topic.)

9.4 NP-COMPLETENESS

Given any class of problems, we can informally define a "central" problem in that class to be a problem that (i) belongs to that class, and (ii) the solution to that problem serves as a strong foundation for other problems in that class. (We use a similar informal definition for an item in any set, for example, by saying that Tom Cruise is a "central" or a "quintessential" Hollywood actor.)

Slightly more formally, we can define a problem p to be "complete" in the class C, if (i) problem p belongs in class C, and (ii) problem p can be used to solve **all** of the problems in class C using resources (time or space) that are comparable to solving the problem p itself.

Specifically, for the class NP, we define a problem X to be an **NP-Complete** problem if: (i) X is in NP, and (ii) Every problem in class NP is reducible to X in polynomial time.

9.4.1 Cook-Levin Theorem

To show that a problem is NP-complete, we need to show a reduction from *every* problem in NP to that problem. Since the class NP is an infinite class of problems, we cannot enumerate all the problems in NP, much less show a reduction from every problem in NP to the given problem. Therefore, the existence of an NP-complete problem must be shown using a proof that is not dependent on individual reductions. A famous theorem, discovered independently by two computer scientists, and now commonly known as the Cook-Levin theorem does exactly that. Before describing the Cook-Levin theorem, we describe the closely related Boolean satisfiability problem.

Boolean Satisfiability (SAT) problem: We are given a boolean formula consisting of multiple conjunctions (AND clauses) and disjunctions (OR clauses), and we have to find whether there is an assignment of true/false values to the literals, such that the entire formula is true. For example, consider the following boolean formula:

$(x_1$ or x_2 or $x_3)$ and $(x_1$ or $n(x_2)$ or $n(x_3))$ and $(n(x_1)$ or x_2 or $n(x_3))$ and $(n(x_1)$ or $n(x_2)$ or $n(x_3))$ and $(n(x_1)$ or x_2 or $x_3)$ and $(x_1$ or $n(x_2)$ or $x_3)$ and $(x_1$ or $n(x_2)$ or $n(x_3))$ and $(n(x_1)$ or x_2 or $n(x_3))$

Here $n(x_1)$ represents the negation of x_1. That is, if x_1 is true, then $n(x_1)$ is false, and vice versa.

So, if we assign $x_1 = x_2 = x_3 =$ true, then overall clause becomes:

(T or T or T) and (T or F or F) and (F or T or F) and (F or F or F) and
(F or T or T) and (T or F or T) and (T or F or F) and (F or T or F),
which becomes: T and T and T and F and T and T and T and T, which is false.

Therefore, this assignment does not satisfy this clause. Some other true/false assignment to variables may satisfy this clause, or the formula may not be satisfiable.

CNF and **CSAT**: A boolean formula is said to be in Conjunctive Normal Form (CNF) if it consists of conjunction of clauses, and each clause is a disjunction of literals or their negations. The boolean formula used in the SAT example above is in conjunctive normal form.

Boolean Satisfiability problem where the input is a boolean formula in Conjunctive Normal Form is commonly referred to as Conjunctive SAT, or CSAT.

Cook-Levin Theorem: Suppose we are given a NTM N and a string w of length n which is decided by N in $f(n)$ or fewer nondeterministic steps. Then, there is an explicit CNF formula f of length $O(f(n)^3)$ which is satisfiable if and only if N accepts w. In particular, when $f(n)$ is a polynomial, f has polynomial length in terms of n. Therefore every language in NP reduces to CSAT in polynomial time.

Thus, Cook-Levin theorem proves that every problem in NP is reducible to CSAT in polynomial time. We observe that CSAT is trivially in NP, and therefore, CSAT is an NP-complete problem. The set of NP-complete problems is commonly written as NPC, and the Cook-Levin theorem can also be written as: CSAT $\in$ NPC.

9.4.2 Proving NP-Completeness for a Given Problem

The proof of Cook-Levin theorem is quite involved. However, we can use the Cook-Levin theorem to prove that a given problem X is NP-complete simply by showing a reduction from CSAT to X. This gives us a ready template to show that the given problem X is NP-Complete.

Template to prove that problem X is NP-Complete:

1. Show that X is in NP, i.e., a polynomial time verifier exists for X.
2. Select CSAT or another **known** NP-complete problem, S.
3. Show a polynomial algorithm to reduce S to X, by transforming an instance of S into an instance of X. (A schematic of a reduction is shown in Figure 3.)

So far we have introduced only CSAT[2] as an NP-complete problem, but our list will grow soon, and therefore the template generalizes the step 2 to

2 There are many related versions of satisfiability problem—general boolean satisfiability (SAT), conjunctive satisfiability (CSAT) and conjunctive satisfiability where each clause has exactly three literals (3SAT).

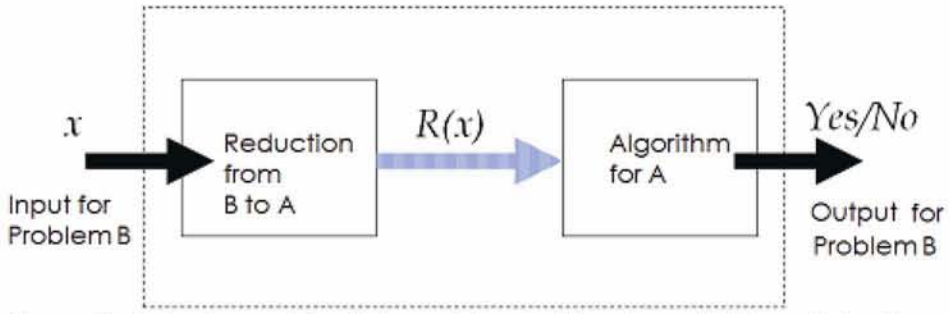


Figure 3: Reducing problem B to problem A. If such a reduction exists, then we can conclude that problem A, is at least as hard as problem B (assuming the reduction itself does not dominate the algorithm for problem A).

identify a suitable NP-complete problem. Steps 2 and 3 are interrelated, and many times involve a trial and error process. We may select a problem but if we are not able to find a suitable transformation, we may need to go back to Step 2 and select a different NP-complete problem.

SOX RESTOX Mnemonic: It is a common mistake to reduce problem A to problem B, when attempting to prove the hardness of problem A. For example, when showing the NP-completeness of problem X using SAT, we need to reduce SAT to X (and not the other way around!) We suggest the mnemonic SOX RESTOX: **S O**utside the Bo**x**; **Re**duce **S TO X**. (Here, S may be SAT or another known NP-complete problem.)

9.5 EXAMPLE NP-COMPLETE PROBLEMS

In this section, we discuss more NP-Complete problems. For some of them, we also present hints or outlines of proofs of their NP-completeness.

9.5.1 Clique

A clique is a set of vertices such that there is an edge between each pair of vertices in that set. (The concept of clique is similarly used in context of people—a clique is a group of people who all know each other.) Clique problem, also sometimes written as k-Clique problem, is that given a graph G, we would like to identify whether or not G has a clique of size k.

Represented in language terms, the problem can be equivalently written as: CLIQUE = $\{<G,k> \mid G$ is a graph with a clique of size $k\}$

It is easy to observe that the Clique problem is in NP. A non-deterministic Turing Machine can randomly select k vertices and then check whether or not the k vertices are connected. If the graph G indeed has a clique of size k, then

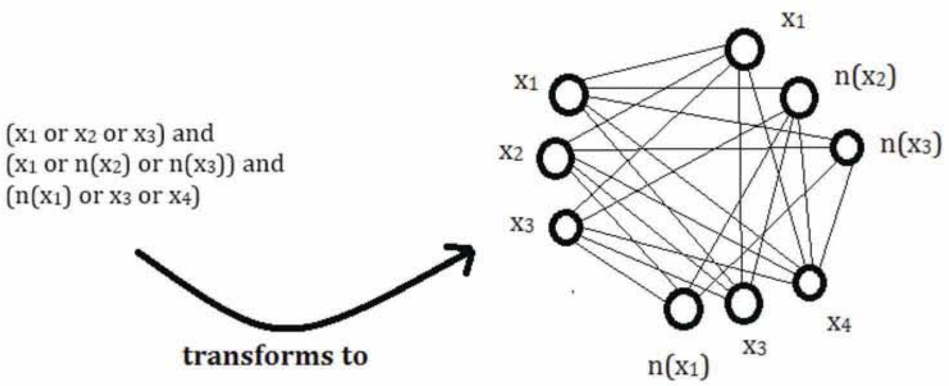


Figure 4: Transforming an instance of CSAT problem into an instance of clique problem. A vertex is created for each literal, and each vertex is connected to all vertices in other clauses, except the vertices that correspond to their negations. For example, the top left vertex corresponding to first *x1* is connected to all 6 vertices in other clauses, except $n(x_1)$.

the machine will terminate in polynomial time in at least one of the many possible execution paths. Therefore, Clique problem is in NP.

We now proceed to prove that there is a polynomial time reduction from *every* problem in NP to the clique problem, in other words, that clique problem is **NP-hard**. We will do so by showing a reduction from the CSAT problem to the clique problem.[3]

Proof that Clique problem is NP-hard: To reduce the CSAT problem to a clique problem, we employ the following transition. Given an instance of CSAT with k clauses, we make a vertex for each literal, such as x_1. If the literal appears multiple times, we make multiple vertices corresponding to each appearance. We connect each vertex to the literals in other clauses that are not the negation of this literal. An example of this transformation is shown in Figure 4. We observe that if there exists a k-clique in this graph, then the k vertices must be in different groups and therefore correspond to literals from different conjunctive clauses. Further, since no vertex is connected to its complement, we can set the literals corresponding to the vertices in the clique to true, and thereby obtain a satisfying assignment.

Therefore, if we had a solver for the k-clique problem, we could use it to solve CSAT problem, and therefore k-clique problem is as hard as, or harder than the CSAT problem. In other words, k-clique is an NP-hard problem.

3 This reduction presents an oft-repeated pattern. We will need to transform a problem in boolean logic into a problem in graph theory! In other transformations, we may need to transcend similar differences.

9.5.2 Independent Set (IS)

Very closely related to the clique problem is the independent set problem—an independent set is a set of vertices such that there are no edges between any two vertices in the set. In language terms, we can define the problem to be:

INDEPENDENT SET = {<*G*,*k*> | where *G* has an independent set of size *k* }

We can easily observe that Independent Set problem is an NP-hard problem since it is the *dual* of the clique problem. Given a graph *G*, we can construct the complement graph *G'* that has the same set of vertices, and an edge exists in *G'* if and only if the corresponding edge does not exist in graph *G*. A clique of size *k* exists in *G* if and only if an independent set of size *k* exists in *G'*.

Therefore, we can reduce the clique problem to an independent set problem simply by creating its complement graph.

9.5.3 Vertex Cover

Vertex Cover problem is defined as follows. Given a graph *G*, we would like to find a smallest set of vertices, such that every edge in *G* is incident upon at least one of the vertices in the selected set.

Vertex Cover problem is very related to the Independent Set problem, and we can observe that given a vertex cover, the remaining set of vertices is an independent set. Therefore, if we can find the minimum vertex cover, we can also find the maximum independent set. We use this idea to reduce the independent set problem to the vertex cover problem, which is shown in Figure 5.

9.5.4 Hamiltonian Path and Hamiltonian Cycle

Hamiltonian path problem and Hamiltonian cycle are two related problems, which attempt to find if there is a path (a cycle, in the case of Hamiltonian cycle problem) that includes every vertex exactly once. For example, given a graph on *4* vertices *{a, b, c, d}* and four edges *{{a,b}, {b,c}, {c,d}, {a,d}}*, we can easily

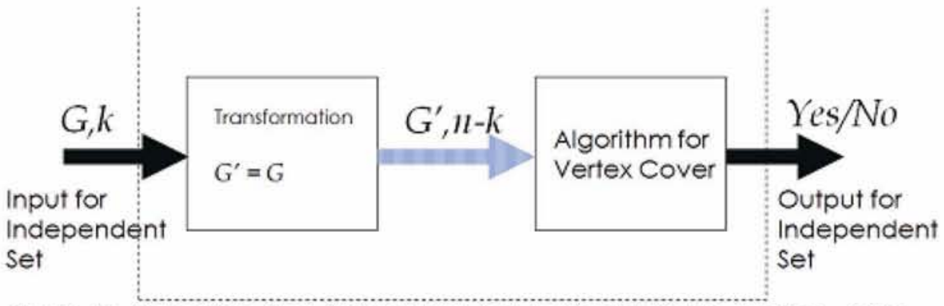

G,k
Input for Independent Set
Transformation
G' = G
G',n-k
Algorithm for Vertex Cover
Yes/No
Output for Independent Set

Figure 5: Reduction from Independent Set to Vertex Cover problem. Since Independent Set is known to be an NP-hard problem, this proves that Vertex Cover is NP-hard problem also.

observe that this graph has a Hamiltonian cycle *(a, b, c, d, a)* and a Hamiltonian path *(a, b, c, d)*.

Hamiltonian path and Hamiltonian cycle are both computationally difficult problems, and are in fact both NP-complete. We do not give a proof of their NP-completeness here, rather we provide ideas for reducing each problem to the other.

Reducing Hamiltonian Path problem to Hamiltonian Cycle problem: We can easily reduce the Hamiltonian Path to the Hamiltonian Cycle problem using the following transformation. Given a graph $G = (V,E)$, construct a graph $G' = (V', E')$, such that:

$V' = V$ union $\{z\}$, where z is a new vertex
$E' = E$ union $\{(z, v) \mid$ all v in $V\}$

That is, G' has one more vertex, and n more edges.

Claim: G has a Hamiltonian Path if and only if G' has a Hamiltonian Cycle.

Proof: We prove both sides of this claim.

1. If G has a Hamiltonian path, say $[v_1, v_2, .. v_n]$, then G' has a Hamiltonian cycle, which is $[z, v_1, v_2, .. v_{n,} z]$
2. If G' has a Hamiltonian cycle, then the cycle must enter and leave each vertex exactly once. The Hamiltonian cycle can be considered to begin with z and return to z without loss of generality. If we remove the two edges of the Hamiltonian cycle connected to z, the remaining set of edges were all present in the original graph G, and therefore, the remaining path is a Hamiltonian path in G.

Reducing Hamiltonian Cycle problem to Hamiltonian Path problem: The transformation used in reducing Hamiltonian Cycle to the Hamiltonian Path is slightly more involved. We present the transformation, and also provide some guidance on how to think through such transformations.

Given a graph $G = (V,E)$, we would like construct a graph $G' = (V',E')$, such that G has a Hamiltonian Cycle if and only if G' has a Hamiltonian Path.

Our first instinct may be to remove an edge to transform a cycle to a path, however, we have to execute the transformation without knowing the actual Hamiltonian Cycle (and indeed, without knowing if a Hamiltonian Cycle even exists in the graph G or not).

The second attempt can involve adding a new vertex connected to a random vertex in G. This attempt does indeed have merit in the sense that the transformed graph G' will have a Hamiltonian Path if G has a Hamiltonian Cycle.

However, G' may have a Hamiltonian Path even if G only had a Hamiltonian Path and we connected the new vertex to one end of the Hamiltonian Path.

To improve on this attempt, we add three vertices and some edges as defined below:

$$V' = V \cup \{z, w, x\}$$
$$E' = E \cup \{\{z, v_1\}, \{w, adj(v_1)\}, \{x, w\}\}$$

We connect the new vertex z to a randomly selected vertex v_1, and connect the new vertex w to all vertices adjacent to v_1. We connect the new vertex x to the new vertex w.

Therefore, the transformed G' has 3 more vertices, and a few more edges.

Claim: G has a Hamiltonian Cycle if and only if G' has a Hamiltonian Path.

Proof: We prove both sides of this claim.

Suppose G has a Hamiltonian cycle, then without loss of generality, we can view the Hamiltonian Cycle as beginning from and returning to v_1. Suppose the Hamiltonian Cycle is $[v_1, v_2 \dots v_n, v_1]$. Since node v_n is adjacent to node v_1, therefore, there must be an edge $\{v_n, w\}$ in G'. We can construct a Hamiltonian Path in G' as $[z, v_1, v_2 \dots v_n, w, x]$. Therefore G' has a Hamiltonian Path.

Suppose G' has a Hamiltonian Path. Clearly, this path must begin and end at vertices z and x, since both of those vertices are only connected to one vertex each (v_1 and w respectively). Therefore, this path must look like $[z, v_1 \dots. w, x]$. Suppose the last vertex before w is a vertex v_k. We observe that in G', node w is only connected to vertices that are adjacent to v_1 in G, therefore, the node v_k must be adjacent to node v_1 in graph G. We can construct a Hamiltonian Cycle in G as $[v_1 \dots v_k, v_1]$ by reusing the portion of Hamiltonian Path between nodes v_1 and v_k in G'. Therefore G has a Hamiltonian Cycle.

9.5.5 Traveling Salesperson Problem (TSP)

The famous "Traveling Salesperson Problem" (and now a major motion picture, see [14]) is defined as follows. You are given a list of n cities and distances between each pair of cities. You need to visit all n cities exactly once, and return to your start city, and want to make the shortest trip possible.

A decision version of this problem can be specified as: Given a graph G, does a trip exist that is of total cost less than equal to a given value c?

There is no known polynomial time algorithm for this problem either. So, this problem is not known to be in class P. However, given a machine that can guess a tour, we can easily verify if the given tour is a valid tour with cost less than c or not. Therefore, the decision version of this problem is in class NP.

We do not present a proof of NP-completeness of TSP or show any reductions. However, we do observe that the brute force algorithm that evaluates every possible tour needs to evaluate $(n-1)$! permutations[4]. Even for a small value of n, such as 30, this solution is computationally infeasible. For example, on a 10 GHz processor, such an algorithm will take more than 200 billion centuries. In a later section, we will discuss a different exponential time algorithm which takes $O(n^2 2^n)$. Although that time complexity is also exponential, that algorithm can solve the same problem instance in under a minute.

9.6 NP-COMPLETE VS. NP-HARD

NP-completeness is defined as a combination of two properties: (i) of being in the class NP, and (ii) of being provably as hard as any other problem in NP.

If, for a problem P, we can only prove the second property, we say that the problem is NP-hard. However, until we can prove that the same problem is in NP, we cannot claim that the said problem is NP-complete.

An example of this is the optimization version of the Traveling Salesperson Problem. This problem can be proven to be NP-hard. However, we do not know how to design a non-deterministic Turing Machine which can solve this problem in polynomial time. Therefore, this problem is NP-Hard, but not known to be NP-complete.

9.7 SUMMARY

In this chapter we studied P and NP – two classes of problems. P is the class of problems that can be solved in polynomial time using a deterministic Turing Machine. NP is the class of problems that can be solved in polynomial time using a non-deterministic Turing Machine. NP-complete is a subset of problems that are the hardest problems in class NP. Any problem in NP can be reduced to any NP-complete problem in polynomial time[5].

We do not know whether classes P and NP are the same or not. We can know the answer if we manage to solve any NP-complete problem in polynomial time, or if we manage to prove that a certain NP-compete problem cannot be solved in polynomial time using a deterministic Turing Machine.

To prove a problem X is NP-complete, we need to prove two properties:

(i) Show X is in NP

4 First vertex of the tour can be fixed arbitrarily without loss of generality.

5 This is not an outcome, rather the very definition of an NP-complete problem.

(ii) Select a well-known NP-complete problem, such as SAT, and reduce SAT to X. (The SOX RESTOX mnemonic may help in remembering the direction of reduction we need to show.)

From a few reductions that we discussed, we may observe that reducing problems can be hard and takes practice. Many reductions need to transform problem instances of one type (such as a boolean logic problem) to a problem instance of another type (such as a graph problem). Many contemporary research publications containing novel reductions have shown the NP-completeness of thousands of problems and their variations.

Although no polynomial time algorithm is known for any NP-complete problem, and may in fact not exist, there are reasonable and practical strategies that one can employ to solve NP-complete problems. We will discuss some of those strategies in the next chapter.

9.8 HOME EXERCISES

1. Show a polynomial time reduction from the Clique problem to the Vertex Cover problem.
2. Given a graph $G = (V, E)$, a dominating set for the graph G is a subset D of V such that every vertex not in D is adjacent to at least one member of D. The "Dominating Set" problem is defined as given a graph G and an integer k, to determine if the graph G has a dominating set of size k. Prove that the Dominating Set problem is an NP-complete problem. (Hint: Show a reduction from Vertex Cover problem to the Dominating set problem.)
3. Finding a Spanning Tree is an easily solvable polynomial time problem. Consider a "k-degree constrained Spanning Tree", wherein we have to find a spanning tree such that no vertex in the spanning tree has degree more than k. Show that the k-degree constrained spanning tree problem is NP-complete.
4. Prove that the following problem is NP-complete: Given a graph G, and an integer k, find whether or not graph G has a simple cycle consisting of k edges. A simple cycle is defined as one that does not have any repeating vertices.

SLAYING THE NP-HARDNESS DRAGON

CHAPTER 10

Consider the scenario that you have encountered a problem at work and proved it to be NP-complete. Now what? Certainly, you can safely defend that you have not found a polynomial time algorithm for the problem (not yet, anyway), since many other distinguished computer scientists have toiled for many years without finding a polynomial time solution for NP-complete problems. Further, it is entirely possible that a polynomial time solution for the problem does not exist.

However, the problem still needs a solution. Very rarely do legitimate business problems go away simply because they are difficult! Instead, much more likely is that by proving the problem's intrinsic hardness, you may be able to negotiate (i) a higher budget, (ii) acceptance of a less-than-perfect solution, (iii) solving a more limited problem, or a combination thereof.

There are many strategies to consider when solving hard computation problems, NP-complete or otherwise. Here are at least four of them.

Strategy 1. **(Solve within a context)** Look for simplifications in the context that render the problem solvable more quickly

Strategy 2. **(Find a better exponential algorithm)** Look for improvements in running time that make the exponential "bearable"—for example, an $O(1.5^n)$ algorithm is far better than a $O(n!)$ algorithm, although they are both exponential

Strategy 3. **(Find an approximation algorithm)** Understand performance bounds that are acceptable practically, and use approximation algorithms

Strategy 4. **(Use parallel computing and deploy more hardware)** Use parallel processing, cloud computing, and server farms (Google, Facebook, and government agencies do it)

In the coming few sections, we provide more details and examples for some of these strategies.

10.1 STRATEGY 1: SOLVING WITHIN A CONTEXT

This strategy focuses on finding the simplifications that occur in the *context* in which the problem is being solved. For example, consider a general scheduling problem where you are trying to book appointments, given available timeslots for multiple resources. Depending upon the exact details of the problem, the problem may be hard in general sense. However, the context of the problem may make it easier. For example, if you find that the scheduling problem is being solved for a dental practice and all appointments are of one of three durations, that may make the problem easier to solve.

Consider another example. Suppose we are trying to solve the Maximum Independent Set (MIS) problem. For general graphs, this problem is NP-complete. Again, the context of the problem may make it easier. For example, if you find that graph that you are being provided is a tree, you can solve the problem in a polynomial time very easily. We provide a recursive formulation for the MIS problem, based on which a dynamic programming algorithm can easily be constructed.

10.1.1 Maximum Independent Set in Trees

Consider a tree (or subtree) rooted at a node v. We use the following notations to capture the size of the independent sets for the subtree rooted at node v.

$NMIS(v)$: Size of the maximum independent set for the tree rooted at node v, such that v is **not** in the set.
$MIS(v)$: Size of the maximum independent set for the tree rooted at node v.
We observe that in the base case, that is, if v is a leaf node, $NMIS(v) = 0$, $MIS(v) = 1$.

We observe that in the recursive case, the following equations hold:
$NMIS(v) = \sum_u MIS(u)$ for all child nodes u of v (If the current node v is not included, then we can include maximum independent sets from all subtrees)

$$MIS(v) = \max \left\{ \begin{array}{c} \left(1+\sum_u NMIS(u) \mid u \text{ is a child node of } v\right), \\ NMIS(v) \end{array} \right\}$$

(The first operand refers to the case that the current node v is included. In that case, we can include the maximum independent sets from all subtrees, as long as those independent sets do not contain the root nodes themselves, as those nodes are adjacent to node v. The second operand refers to the case that the current node v is not included.)

Based on this recursive formulation, a dynamic programming algorithm can easily be constructed that evaluates the *NMIS* and *MIS* values for each node, starting with the bottom of the tree. (We can first use a breadth first search algorithm to number the nodes and start evaluating these values in the reverse order of their BFS numbering.)

10.2 STRATEGY 2: FINDING AN ALGORITHM WITH A LOWER EXPONENT

Consider the famous traveling salesperson problem, in which we are given a weighted undirected graph $G = (V,E)$ and we need to find a shortest tour that starts at a vertex, visits each vertex exactly once, and returns to the start vertex. Distance (also called weight) between adjacent vertices (u,v) is given in form of the function $d(u,v)$. As discussed in Section 9.5.5, a brute force algorithm that examines all possible tours needs to examine $n!$ tours, where n is the number of vertices. By considering the fact that the first node in the tour can be fixed arbitrarily, we can reduce it to slightly better $(n-1)!$ tours. Computing the cost of each tour takes O(n) time, resulting in an O($n!$) overall algorithm. This algorithm is computationally infeasible for even small values n, such as 25 or 30. A much more efficient algorithm using Dynamic Programming is possible, which we present next.

10.2.1 Traveling Salesperson Problem—A Dynamic Programming Algorithm

Given $S \subseteq V$, let $C(S, j)$ denote the shortest path that starts at 1, visits all nodes in S, and ends at j. (1 and j must be in S.)

Since the set S must contain at least two elements, we can define the base case as $|S| = 2$, and in that case, we can easily compute $C(S, k)$ as $d(1,k)$, for k in $\{2 \dots n\}$.

In the recursive case, we consider that $|S| > 2$.

In this case, $C(S, k)$ can be computed in terms of $C(S',k)$ where $S' = S - \{k\}$. Specifically, the shortest path from 1 to k can be constructed by extending one of the shortest paths from 1 to another node in the set S. Therefore, we can write the recursive equation as:

$C(S, k) = min \{C(S', j) + d(j, k)$ / for all $j \in S'$ where $S' = S - \{k\}\}$

Each value $C(S,k)$ can be computed in $O(n)$ time. Since there are a total of $n\ 2^n$ combinations for all possibilities of (S,k), we can compute all possible values in $O(n^2\ 2^n)$ time. Finally, computing the shortest tour is simply a matter of finding the minimum of all $C(V,v) + d(v,1)$ over all possible vertices v, where V is the set of vertices of the graph.

10.3 STRATEGY 3: USING AN APPROXIMATION ALGORITHM

This strategy is perhaps the most important strategy for solving NP-complete problems[1]. Since it is unknown if a polynomial time algorithm for such a problem exists, we can trade optimality for efficiency and settle for polynomial time sub-optimal solutions. Ideally, the approximation is optimal up to a small constant factor (for example, within 5% of the optimal solution).

Definition: Given a computation problem, an ***r***-approximation algorithm is an algorithm that returns a feasible solution with value at most ***r*** times "worse" than optimal. The interpretation of "worse" depends upon the nature of the problem. For example, for a maximization problem, worse means smaller. Therefore, a **2**-approximation algorithm returns a feasible solution that is at least one half the size of the optimal solution. Similarly, for a minimization problem, worse signifies larger. Therefore, a **2**-approximation algorithm for a minimization problem may return a feasible solution that is at most twice the size of the optimal solution.

We can reasonably raise the following important question: An approximation algorithm makes sense in the case of hard problems where we do not know how to compute the optimal solution; in that case, how can we compare our solution to the optimal solution that we cannot compute?

1 Approximation Algorithms are a complete field of study of their own, with many available textbooks, and a graduate class focused on Approximation Algorithms can often be found as well.

The answer lies in the theoretical bounds that we studied in the chapter on Branch & Bound. Considering a minimization problem, a theoretical lower bound can be found by making a relaxation to the problem and then observing that the optimal solution cannot be any lower than the solution to the relaxed version of the problem. For example, in Section 8.4, we observed that for a job-assignment problem, the cost of an optimal assignment can be no lower than the sum of the minimum costs for each job. Therefore, if we can find a feasible solution that costs *1.5* times the sum of minimum costs for each job, we can conclude that this feasible solution is no more than *1.5* times the optimal cost. While the job-assignment problem is not an NP-complete problem, this general idea of using a theoretical bound is often used while using the approximation algorithm strategy.

10.3.1 Traveling Salesperson Problem—An Approximation Algorithm

Once again, we return to the traveling salesperson problem, in which we are given a weighted undirected graph $G = (V,E)$ and we need to find a shortest tour that starts at a vertex, visits each vertex exactly once, and returns to the start vertex. Distance between adjacent vertices (u,v) is given in form of the function $d(u,v)$, which we also refer to as the weight of the edge.

We observe the following lower bound. Consider the shortest salesperson tour. Given that tour, we can remove an edge to obtain a spanning tree. The weight of that spanning tree must be at least the weight of a minimum spanning tree (by the very definition of a minimum spanning tree). Therefore, the weight of the shortest salesperson tour must be at least the weight of the minimum spanning tree.

Further, we observe that if we are given a minimum spanning tree, we can create a salesperson tour in the following manner. We run the depth-first traversal on the minimum spanning tree, which assigns each vertex a DFS traversal number. We use that sequence as the basis for creating the traveling salesperson tour. For example, if given the Minimum Spanning Tree shown in Figure 6, we can construct a TSP tour as $(1, 2, 3, 4, 5, 1)$. This uses the edge $(3,4)$, which was not there in the MST, but if the triangle

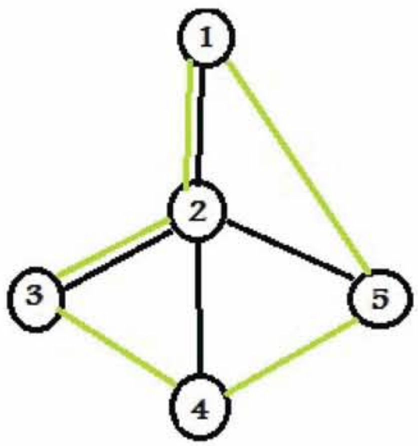


Figure 6: Construction of a TSP tour given a Spanning Tree: we can construct a tour by following the vertices in sequence of their DFS numbers. If triangle inequality holds, then the weight of the TSP tour is at most twice the weight of the spanning tree.

inequality holds, then $d(3,4) \leq d(2,3) + d(2,4)$. Similarly, $d(4,5) \leq d(2,4) + d(2,5)$, and $d(1,5) \leq d(1,2) + d(2,5)$. If we take the sum of weights of all the edges of the tour, we observe that the total weight of the tour is no more than the twice the weight of the Minimum Spanning Tree.

Finally, we observe from Section 5.4, that Kruskal's algorithm runs in $O(m \log n)$ time. By combining the two observations about the TSP and the cost of the lower bound, we have a polynomial time algorithm that guarantees a 2-approximation algorithm for the TSP problem, assuming the triangle inequality holds[2].

10.4 SUMMARY

While proving a problem to be NP-complete may be interesting, it is rarely the end of the problem itself. The problem still needs to be solved and the NP-completeness of the problem may allow us to justify an exponential time algorithm, an approximation algorithm, or merely the purchase of more computing power to solve the problem.

There are many strategies to consider when solving hard problems, such as those that are provably NP-hard. Some of the main strategies that we considered were:

Strategy 1: Look for simplifications and the context of the problem, and solve the problem within that limited context

Strategy 2: Look for improvements in running time that make the exponential "bearable"

Strategy 3: Understand performance bounds that are acceptable practically, and use approximation algorithms

Strategy 4: Use parallel processing, cloud computing, and more processing power

As part of strategy 3, we also touched upon the exciting field of approximation algorithms, and reviewed the theoretical bounds that we had touched upon earlier in Chapter 8 on Branch and Bound Algorithm design technique.

When we encounter **"real world"** problems, we often observe that they are harder *and* easier than mathematical computation problem: they are harder because modeling the problem is part of solving the problem; they are easier because we can use the quirks in the data to fine tune our solutions.

Lower Bounds and NP-completeness analysis help in guiding our efforts. While algorithmic techniques are the solution pillars and the building blocks of our ideas, multiple techniques often need to be used together to create lasting solutions.

2 Triangle inequality need not always hold. For example, if the weight of the edge represents the cost of airfare between two cities, no such constraint may exist on the weights, as airfares notoriously defy common logic.

THEORY OF LOWER BOUNDS

CHAPTER 11

Chapter Author: David Balash

A lower bound is a theoretical limit on the number of operations required to solve a particular problem. For example, we may claim that to evaluate a certain function, we must perform at least n^2 addition operations. Thus, a lower bound quantifies the **intrinsic complexity of a given problem**, independent of the algorithm used to solve it. In more general terms, a lower bound may specify a limit on any computation resource (time, space, operations, circuits, etc.) to solve a certain problem.[1]

The lower bounds are interesting because they provide both a performance goal for the efficiency of an algorithm and protection from seeking unachievable results. A lower bound is said to be **tight** if there is a known algorithm with the same efficiency. If a gap exists between a lower bound and the efficiency of the fastest known algorithm, then either a faster algorithm can be designed or a better lower bound can be proved.

In order to define a lower bound, we need to select a computation model, including a set of allowable operations and their respective costs.

1 This aspect of the theory of lower bounds may be philosophically repugnant to some. By a lower bound, we are essentially saying that it is not possible to do better than a given result. Albert Einstein is once reported to have said, "If you believe something is impossible, don't stop a guy from trying it." And yet, Einstein also gave us a theoretical limit on the velocity of an object. Here, we are trying to prove our own theoretical limits, although not nearly as exciting!

In the commonly used algebraic computation model, we assume that we can perform any common mathematical operation (addition, subtraction, multiplication, and division) involving real numbers in a unit-constant time, and we can perform any comparison operation in a unit-constant time.

We can categorize mechanisms to prove lower bounds along the following lines:

1. Trivial lower bounds can be obtained by counting the size of input that must be examined and the size of output that must be computed. For example, one can argue that it is not possible to find the largest of given n numbers in better than n time, since at the very least we must read all the n inputs.
2. Lower bounds can be obtained using a decision tree argument. In a decision tree, each internal node represents some partial conclusion and a question (decision) about the input, and each leaf a possible outcome. Algorithm execution is represented by a path from the root to a leaf. The length of the longest such path corresponds to the worst-case number of decisions an algorithm must compute. The number of possible outcomes (decisions) at each node of the tree determines the branching factor of the tree, usually a constant, say b. The height h of a decision tree with N leaves must be at least $[\log_b N]$. The key point is that a decision tree with a given number of leaves must be tall enough to have that many leaves. The lower bound on the running time of an algorithm represented by a decision tree is the lower bound on the heights of all decision trees in which each possible outcome appears as a reachable leaf. A decision tree is an information-theoretic argument, because it seeks to establish a lower bound based on the amount of information it has to produce to solve a problem correctly.
3. Lower bounds can also be established with an adversary argument. An adversary argument makes use of a malicious, but honest, all-knowing adversary who is allowed to choose an input for the algorithm. The adversary will actively work against the algorithm to maximize the amount of work required, with the condition that the adversary cannot contradict itself. For example, the adversary cannot say $a > b$, $b > c$ and then also say $c > a$.

11.1 LOWER BOUND ON SORTING

Let us use a decision tree to establish a lower bound for the problem of comparison-based sorting of an arbitrary array containing n distinct elements. In the comparison-based model of computation, the only allowable operation is comparing pairs of elements. The decision tree for this problem will contain a leaf for each permutation of the original input. These permutations correspond to correctly completed sort orders for particular input sequences. The number of possible outcomes for this problem is the number of permutations of the input. We recall from our study of combinatorics that the number of permutations of *n* distinct objects is *n!* (that is, *n(n–1)(n–2) … 3 2 1*). The internal nodes of the decision tree are comparisons between two elements of the input. This gives us a branching factor of *b=2*; therefore, the decision tree for a comparison-based sort is a binary tree. It is an intrinsic property of a binary tree that a tree of height *h* has no more than 2^h leaves, and, since each of the *n!* permutations of the input is a leaf, we have $n! \leq$ *number of leaves* $\leq 2^h$ and, after taking logarithms (base *2*), this gives

$$h \geq [\lg(n!)]$$
$$= \Omega(n \log n)$$

Note the approximation, for $n > 1$

$$n! \geq n\left(n-1\right)\left(n-2\right)\cdots\frac{n}{2} \geq (\frac{n}{2})^{n/}$$

$\log n! \geq (n/2) \log (n/2)$.

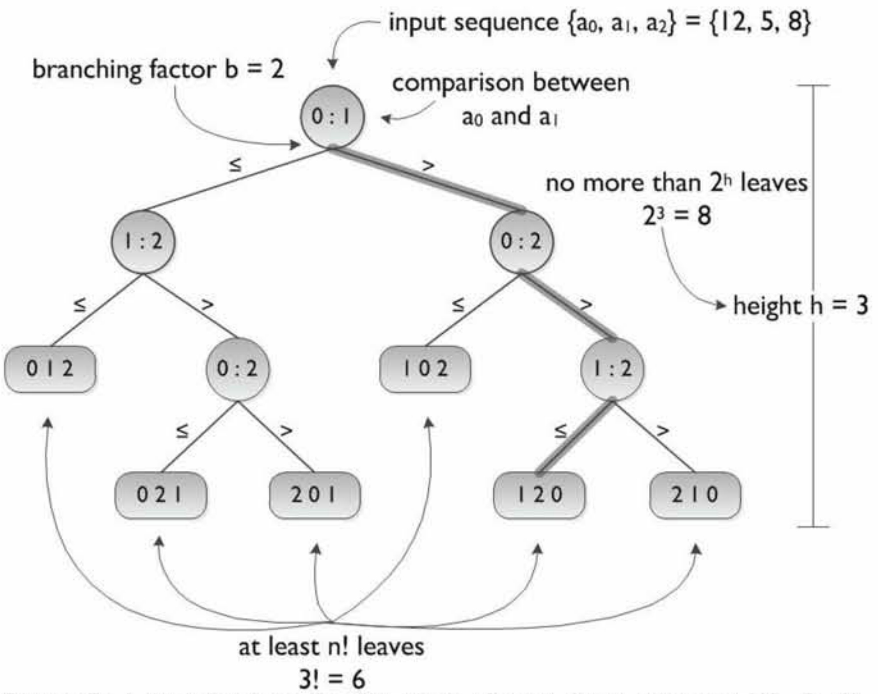


Figure 7: A decision tree for the three-element input sequence {12, 5, 8}

As a result, for any comparison-based sorting algorithm A, there exists an input I of size n such that A makes at least $\Omega(n \log n)$ comparisons to sort I. Therefore, the comparison-based sorting algorithms that we know of—merge sort, heap sort, quick sort (average case)—are all asymptotically optimal. The lower bound for the comparison-based sorting problem is tight, because there are known algorithms with the same efficiency.

The lower bound tells us that a worst-case input exists that requires $\Omega(n \log n)$ comparisons. However, a comparison-based sorting algorithm can still run in linear time in special scenarios. For example, we can use counting sort or radix sort if each number is within a small range (for example, *1 ... 1000*).

11.2 LOWER BOUND ON SEARCHING

The problem of comparison-based searching for an element in an unordered array of size n has a lower bound of n, since every array element must be read in the worst case. The worst-case input is an array that does not contain the search element. This is an example of the trivial lower bound.

The problem of comparison-based searching for an element in an ordered array of size n has a lower bound of $\Omega(\log n)$. The decision tree for the problem must have at least $n+1$ leaf nodes, because there are n possible occurrences of the search element and the possibility that the array does not contain the element. So, the height of the decision tree is at least $[\lg(n+1)]$. The lower bound for this problem is tight, because the binary search algorithm has the same efficiency.

The problem of comparison-based construction of a binary search tree from an arbitrary array of n elements has a lower bound of $\Omega(n \log n)$, the same lower bound as comparison-based sorting of an arbitrary array.

11.3 FINDING MINIMUM, MAXIMUM, AND MEDIAN

The lower bound for finding the maximum (or the minimum) of n elements, based on comparing pairs of elements, is $n-1$ comparisons. To observe this, we can think of a tennis tournament. How many games must be played before a champion is declared? Each tennis player in the tournament except the champion must lose at least one game. If fewer than $n-1$ games are played, there will be two or more players who have never lost a game, and therefore we cannot declare a champion yet. We note that this argument is independent of the sequence of games (comparisons). This is important, as the lower bound must be independent of the knowledge of any algorithm for the problem.

Median: We determined in Section 4.6 that $O(n)$ time is sufficient to find the median of a list of n integers. Using a trivial lower bound, we observe that we need $\Omega(n)$ time just to read the input.

11.4 AN ADVERSARY ARGUMENT FOR FINDING BOTH MINIMUM AND MAXIMUM NUMBERS IN AN ARRAY

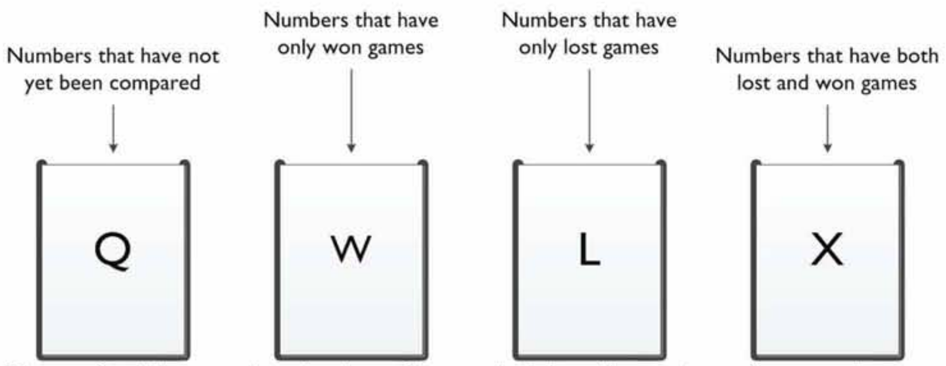
If we are given an array of n distinct unsorted numbers, how can we find both the minimum and the maximum using as few comparisons as possible? The first attempt at an algorithm could be to find the minimum, and then to find the maximum from the remaining, in a total of $(n - 1) + (n - 2)$ comparisons. A better algorithm would first compare pairs of numbers, in $n/2$ comparisons, and collect $n/2$ "winners" and $n/2$ "losers." Then, it would find the maximum from the winners in $(n/2) - 1$ comparisons, and the minimum from the losers in $(n/2) - 1$. That is, a total of $(3n/2) - 2$ comparisons.

That much is straightforward. A much more interesting question is whether or not the algorithm presented above is optimal. In other words, can we say that *any* algorithm that is based on comparisons will need at least $(3n/2) - 2$ comparisons, regardless of the sequence in which it does the comparisons?

We prove that for any given algorithm (let us call it algorithm A), there is a sequence of numbers for which A takes at least $(3n/2) - 2$ comparisons to find both the maximum and minimum numbers. Such a proof can be constructed with an adversary argument.

The adversary thinks in terms of numbers in *four* buckets:

Bucket Q: Contains numbers that have not yet been compared to anything.
Bucket W: Contains numbers that have only won games that they have played. So, they are candidates for being the maximum.
Bucket L: Contains numbers that have only lost games that they have played. So, they are candidates for being the minimum.
Bucket X: Contains numbers that have both won and lost games they have played. So, they are not candidates for being the minimum or the maximum.
The buckets start in the following state when the algorithm A starts:
Bucket Q: n numbers. Buckets W, L, X: 0 numbers each.
Finally, when algorithm A finishes, this must be the state of the buckets:
Bucket Q: 0 numbers. Bucket W, L: 1 number each. Bucket X: $n–2$ numbers.
The adversary follows these rules for creating outcomes of comparisons:


Numbers that have not yet been compared
Numbers that have only won games
Numbers that have only lost games
Numbers that have both lost and won games
Q
W
L
X
Figure 8: Adversary's strategy for maximizing the number of comparisons for finding both maximum and minimum numbers. Adversary thinks of numbers in terms of buckets Q, W, L, and X.

C1: When comparing two numbers in bucket Q, one number will go into bucket W, and one number will go into bucket L. (Net change: *−2, +1, +1, 0*)
C2: When comparing a number from bucket W to a number from bucket Q, the number from W will win and will remain in W. The number from bucket Q will go into L. (Net change: *−1, 0, +1, 0*)
C3: When comparing a number from bucket L to a number from bucket Q, the number from the L will lose and will remain in L. The number from bucket Q will go into W. (Net change: *−1, +1, 0, 0*)
C4: When comparing a number from bucket W to a number from bucket X, the number from W will win and will remain in W. The number from bucket X will remain in X. (Net change: *0, 0, 0, 0*)
C5: When comparing a number from bucket L to a number from bucket X, the number from L will lose and will remain in L. The number from bucket X will remain in X. (Net change: *0, 0, 0, 0*)
C6: When comparing a number from bucket W to a number from bucket L, the number from L will lose and will remain in L. Number from the W bucket will remain in W. (Net change: *0, 0, 0, 0*)
C7: When comparing two numbers from bucket W, one will remain in W, and one will go into X. (Net change: *0, −1, 0, +1*)
C8: When comparing two numbers from bucket L, one will remain in L, and one will go into X. (Net change: *0, 0, −1, +1*)
These rules ensure the following points:

- A maximum of two numbers can be removed from bucket Q during each comparison, and during this comparison, no number goes into bucket X. This happens in comparison C1.

- During each comparison, a maximum of one number can enter bucket X. This happens in comparisons C7 and C8, and in those comparisons, there is no change in the size of bucket Q.

So, we observe that we need $n–2$ comparisons (of type C7/C8) simply to fill up bucket X (we need $n–2$ numbers there by the end). Further, since those comparisons do not involve bucket Q, we need at least an additional $(n/2)$ comparisons just to empty out bucket Q (it starts with n numbers and needs to have 0 numbers). So, in total, we need to make at least $(3n/2)–2$ comparisons just to make sure bucket Q and bucket X have the desired state before algorithm can draw its conclusion.

11.5 MATRIX MULTIPLICATION

A trivial lower bound for the problem of multiplying two n row by n column matrices is $\Omega(n^2)$ because $2n^2$ elements in the input matrices must be read and n^2 elements of the product must be computed. Determining whether or not the $\Omega(n^2)$ lower bound is tight continues to be an active area of research.

The simplest algorithm for matrix multiplication takes $\Theta(n^3)$ time, but many improvements to matrix multiplication algorithms have been made, starting with Volker Strassen, who, in a 1969 paper, famously demonstrated that the simple $\Theta(n^3)$ algorithm was suboptimal. Strassen's algorithm, also discussed in Section 4.8, reduces the number of multiplications required and runs in $O(n^{2.807})$ time. In 1987, Don Coppersmith and Shmuel Winograd improved upon Strassen's algorithm and developed a $O(n^{2.3755})$ time algorithm. Recent smaller improvements have reduced the running time to $O(n^{2.3727})$. The debate continues on whether a faster algorithm matching the $\Omega(n^2)$ lower bound can be designed, or if a performance barrier exists and a better lower bound must be proved [15].

11.6 LOWER BOUNDS ON GRAPH PROBLEMS

A trivial lower bound for many graph problems is $\Omega(|V| + |E|)$, or simply the time to read the input when the graph is stored in an adjacency list. For many graph problems, this is the best-known lower bound.

11.7 LOWER BOUNDS ON PUZZLES

Information-theoretic arguments with the aid of a decision tree can be used to solve puzzles involving lower bounds. The classic twelve-coin puzzle is an excellent example because the solution depends upon the amount of information we must produce in order to solve the problem correctly.

Given twelve coins, of which one is *either heavier or lighter*, and a two-pan equal-arm balance, how many weighings are *necessary* to find the unique coin and to determine whether the unique coin is heavier or lighter?

To find a lower bound on the number of weighings, first consider the number of possible outcomes. Since any of the *12* coins can be heavier, and any of the *12* coins can be lighter, there are *24* possible outcomes. We must use the balance to provide enough information to specify any of the *24* possible outcomes. A weighing provides three possible answers—left-pan heavy, neutral balance, or right-pan heavy—resulting in a branching factor of $b=3$. Thus the height of the decision tree must be at least $[log_2(24)] = 3$. Therefore, at least three weighings are necessary.[2]

11.8 SUMMARY

To establish lower bounds we analyze the problem itself and not individual algorithms. Lower bounds expose the intrinsic difficulty of a problem and apply to all algorithms that use the same model of computation when solving the problem correctly. We use lower bounds to determine if the performance of a known algorithm is optimal, or if there is room for improvement. Trivial lower bounds, based on the size of the input that must be read, can be found for many problems; however, trivial lower bounds may not always be tight. Information-theoretic arguments can be made with the help of a decision tree. A decision tree uses the branching factor and the size of the solution space to establish a lower bound. Adversary arguments rely on a malicious adversary who forces an algorithm into the most time-consuming path.

There are limitations on the power of algorithms, and the study of lower bounds may have many of us asking: Can this be as good as it gets? Using sophisticated arguments and based on our understanding of the underlying computation models, we can sometimes establish strong lower bounds on the performance of any algorithm to solve a specific problem.

2 Consider the same question involving *13* coins, one of which is heavier or lighter. The lower bound using decision tree approach still evaluates to $[log_2(24)]$, that is, *3*, but *3* weighings are not sufficient.

SECTION IV

CONCLUSIONS AND AUXILIARY MATERIALS

In this section, we present some concluding remarks, some suggestions for extension topics, and some references for further reading.

WRAPPING UP

CHAPTER 12

One of the key conclusions the reader should draw from this book is that algorithms matter. They are all around us, being used in fields as banal as trash-pickup scheduling, as fantastic as exploring the landscape of a distant planet, or as useful as finding patterns of genomic text that may make a person more or less susceptible to a dangerous kind of cancer.

By using the power of asymptotic notation, we can compare different algorithms in terms that ignore minor differences and focus on the growth of those functions as the input size grows.

While the study of all algorithms is outside the scope of this (or any other) book, the algorithms can be categorized according to relatively finite algorithm design techniques, such as divide & conquer, greedy, dynamic programming, searching & backtracking, and branch & bound. By thinking along the lines of algorithm design techniques, we can approach a problem from multiple perspectives in order to find an efficient algorithm. However, we remind ourselves that these algorithm design techniques are merely tools to organize our thinking. A comprehensive working solution is likely to involve components of all of these design techniques.

While the intention of algorithm design techniques is to always improve the performance of the solution for a particular problem, we are sometimes limited by the inherent complexity of the problem.

Indeed there are lower bounds to problems that we cannot expect to beat without changing the underlying computation model.

We can organize various problems in classes, depending upon their inherent computation complexity. For example, using the resources of time and space, we can define classes of problems such that they require at most a polynomial time, or polynomial space with respect to the input size.

The **ultimate litmus test** of this book is in the form of following questions that each reader can answer on their own in the context of each computation or algorithmic problem they face: How would you have approached the problems before you had the benefit of this book? How do you approach those problems now?

12.1 SPECIALIZE, BUT NOT OVER SPECIALIZE

While we live in an era of super specialization, it is important that we don't over specialize and box ourselves into too specific a field. Algorithms present a specialized field themselves, and many people focus on specific classes of problems, such as online algorithms, scheduling algorithms, and approximation algorithms. The reader is ultimately the best judge for how much specialization is appropriate.

The following quote may be relevant.

"A human being should be able to change a diaper, plan an invasion, butcher a hog, conn a ship, design a building, write a sonnet, balance accounts, build a wall, set a bone, comfort the dying, take orders, give orders, cooperate, act alone, solve equations, analyze a new problem, pitch manure, program a computer, cook a tasty meal, fight efficiently, die gallantly. Specialization is for insects."

—Robert Heinlein

12.2 SUGGESTED PROJECTS

Since this book came about by teaching the graduate class on algorithms for several years, here is a selected list of projects that were used by students in those years.

1. Implementation of All Pairs–Shortest Path algorithm: Use file-based input and output and use a library that models the graph objects, instead of generating user interface design. This allows an enhanced

focus on algorithmic aspects of the problem, as opposed to the visual aspects of the problem. Ensure the implementation uses only n^2 amount of space; that is, it uses *1* matrix instead of *2* or *n* matrices. Ensure the implementation is able to create actual paths, not merely distances, without increasing the time complexity of the algorithm.

2. Proof of Cook's theorem: Proof of Cook's theorem involves many aspects of reduction and transforming a problem instance for any non-deterministic Turing Machine into a boolean clause.
3. Union Find Analysis: Analysis of the Union Find algorithm, including path compression, contains amortized cost-analysis techniques, such as credit-debit, which are otherwise not covered in this book.
4. Implementation of Strassen's Matrix Multiplication Algorithm, or another matrix multiplication algorithm that is $o(n^3)$ algorithm.
5. Implementation of Closest Pair of Points Divide and Conquer algorithm: While theoretically this algorithm has an $O(n \log n)$ implementation, it involves a few challenges in details that can change the time complexity to $O(n^2)$ unless implemented correctly.
6. Comparison of different Convex Hull algorithms: Graham scan with Akl-Toussaint Heuristic and Chan's algorithm are both interesting algorithms.

HOW DO WE LEARN?

APPENDIX A

Hermann Ebbinghaus, a German psychologist, described in 1885 the exponential nature of forgetting. In his study (of which he himself was also a participant), subjects memorize a list of meaningless, three-letter words. Ebbinghaus tracked how quickly his subjects forgot the words, and this is now known as the Ebbinghaus or Forgetting Curve, which can be concisely approximated as $R=e^{-t/S}$, where R is the amount of information retained, S is the relative strength of memory, and t is the elapsed time.

Overcoming the Ebbinghaus Curve: Amount of information retained changes upon each successive time the information is reviewed. We can overcome the forgetting curve simply by reviewing in a systematic fashion:

- 10 min—After class by completing, organizing, and comprehending by rewriting or typing notes (e.g., Cornell right column)
- 24 hours—Next day, by rereading notes, condensing them to main ideas, and creating questions (e.g., Cornell left column)
- 1 week—Before class the following week or earlier, by reviewing and self-evaluating your recall
- 1 month—When you review material about a month after reading it, you often draw new insights from it, and at that point, may be able to retain that information for the rest of your life

Does a study of three-letter words apply to Logic and Computer Science? One can argue that in computer science, the problem is with applying, not memorizing, and it is an issue of logic and reasoning, and not remembering or forgetting! In order to evaluate that line of reason, let us take a moment to learn about and from sea slugs.

A sea slug has been described in the excellent book *Genome* [16] in not-too-generous terms, using the following phrases: "A more contemptibly basic animal is hard to imagine." "It displays an enviable lack of neurosis." "It just exists." "Its life is a cinch."

And yet, we observe that when it comes to learning, learn it can. When a jet of water is blown upon its gill, the sea slug withdraws the gill. However, if jet of water is repeatedly blown on the gill, the withdrawal gradually ceases. *(It stops responding to the false alarm. It "habituates.")*

If given an electric shock once, before water is blown on the gill, the sea slug learns to withdraw its gill even further than usual ("sensitization").

When it receives only a very gentle puff of water paired with an electric shock, it withdraws its gill. Thereafter, a gentle puff alone, without any shock, results in a rapid gill withdrawal. (In other words, the slug is capable of "associative learning.")

It learns using all three mechanisms by which dogs or people learn. This is not the same as calculus or relativity, but it is learning just the same. While we may think that our learning needs and models are significantly different, underlying learning mechanisms are still the same.

The following quote, attributed to William Glasser, articulates what portion of material we absorb, depending on what activity we perform with it: "We learn 10% of what we read, 20% of what we hear, 30% of what we see, 50% of what we both see and hear, 70% of what is discussed with others, 80% of what we experience personally, and 95% of what we teach someone else."

The take-away point is that computer science is learning, like other forms of learning. We can always learn better by improving how we learn. As you read the material, make up some questions. Try to answer them. Notice how subtle changes affect which algorithm design technique works and which ones don't work.

MORE GRAPH THEORY AREAS

APPENDIX B

Graph theory is an area that is very rich in challenging problems. Besides some NP-complete problems presented in Section 9.5, consider the following problems.

Graph (Vertex) Coloring and Chromatic Number
Given a graph and an integer k, can the vertices of given graph be colored using k colors so that no two adjacent vertices receive the same color? (We define a coloring in which no two adjacent vertices receive the same color as a **valid coloring** of the graph. Further, we define the minimum integer k for which a valid vertex coloring exists as the **chromatic number** of the graph.)

Some insights into the coloring problem can be obtained based on the degree of vertices. Let Δ be the maximum degree in the graph, and let d be the minimum degree in the graph.

Following claims may sound intuitive, but are all false.

False Claim 1. If $\Delta \leq 3$, then the graph is 3-colorable.
False Claim 2. If $\delta > 3$, then the graph is not 3-colorable.
False Claim 3. If the graph does not have a clique of 4-vertices, then the graph is 3-colorable.

For specific values of k, we can consider specific decision problems. For example, for k = 2, the decision problem becomes:

Is a given graph 2-colorable?
This problem can be solved in polynomial time. However, for all values of $k > 2$, the problem is NP-complete. Specifically, the following problems are NP-complete:
Is a given graph 3-colorable?
Is a given graph 4-colorable?
We can observe that the decision problem for $k=2$ can be easily solved by assigning a random vertex color 1, and all adjacent vertices color 2. We continue this process and either we are able to assign color 1 or color 2 to all vertices, or we reach a contradiction in which case the graph is not 2-colorable.

Graph (Edge) Coloring
The graph edge coloring problem is defined analogous to the graph vertex coloring problem: Given a graph and an integer k, can the edges of given graph be colored using k colors so that no two adjacent edges receive the same color?

Clearly, to have a valid edge coloring of a graph, we need at least Δ colors, as all the edges around a maximum degree vertex need to be assigned different colors. However, it is much less obvious that $\Delta + 1$ colors are always sufficient for a valid edge coloring. Perhaps the most surprising result is that the decision problem as to whether Δ colors are sufficient or not is NP-complete.

Steiner Tree
Given a graph G and set T of points (vertices), interconnect them by a network of shortest length, where the length is the sum of the lengths of all edges. The Steiner tree problem has applications in circuit layout or network design. Most versions of the Steiner tree problem are NP-complete.

MINIMUM SPANNING TREE

APPENDIX C

Let us revisit the Minimum Spanning Tree (MST) problem discussed in Section 5.5. This problem has wide applications in many areas—any time we want to visit all vertices in a graph at minimum cost, the minimum spanning tree problem may be relevant. For example, wire routing on printed circuit boards, sewer pipe layout, and road planning are all applications of the Minimum Spanning Tree problem. Further, as discussed in Section 10.3.1, MST also provides a heuristic for traveling salesperson problems.

One of the first algorithms for finding an MST was Borůvka's Algorithm, developed by Czech scientist Otakar Borůvka in 1926 for an efficient electrical coverage of Moravia.

Kruskal's algorithm for finding an MST was presented in Section 5.5.

Another famous algorithm for finding an MST is Prim's Algorithm. Prim's algorithm starts with a single vertex, chosen randomly from the graph. In each step, it grows the tree by one edge (and one vertex). Specifically it adds the minimum-weight edge such that one end point is in the tree and the other end point is not in the tree. (When adding that edge, the other end point is assumed to be added implicitly as well.) A pseudo-code version of Prim's algorithm is shown next. It uses a priority queue Q, which is initialized in the beginning of the algorithm. The performance of Prim's algorithm depends on how we implement the priority queue Q.

```
for each $u$ in Q do                                  // line 1
   key[$u$] ← ∞
   π[$u$] ← Nil
   Q ← V[G]                                           // line 4
Find a random vertex v in V[G]                        // line 5
key[v] = 0                                            // line 6

while Q is not empty do
   $u$ ← EXTRACT_MIN (Q)                              // line 8
   for each $v$ in Adj[$u$] do
      if v is in Q and $w(u, v)$ < key [$v$] then
         π[$v$] ← $u$                                 // line 11
         key[$v$] ← $w(u, v)$                         // line 12
```

We observe that there are two heap operations — EXTRACT_MIN (invoked in line 8) and DECREASE_KEY (implicitly invoked in lines 6 and 12). The while-loop is executed n, that is, $|V|$ times, and the for-loop in lines 9–12 is executed m, that is, $|E|$ times all together.

If we implement the priority queue using a binary heap, then each EXTRACT_MIN operation takes $O(\log n)$ time, and the total time for all calls to EXTRACT_MIN operation is $O(n \log n)$. Further, the DECREASE_KEY operation takes $O(\log n)$ time. Thus, the total time for Prim's algorithm using a binary heap is $O(n \log n + m \log n)$; that is, $O(m \log n)$.

We can improve the time complexity by implementing the priority queue using a Fibonacci heap data structure. A Fibonacci heap is a collection of trees satisfying the minimum-heap property; that is, the key of a child is always greater than or equal to the key of the parent. With a Fibonacci heap of n elements, the DECREASE_KEY operation takes $O(1)$ amortized time and an EXTRACT_MIN operation takes $O(\log n)$ amortized time. Therefore, the total time for Prim's algorithm using a Fibonacci heap to implement the priority queue Q is $O(m + n \log n)$.

TIME COMPLEXITY OF UNION FIND DATA STRUCTURE

APPENDIX D

The Union Find data structure for managing disjoint set operations is described in Section 5.5.2. The three main operations supported by the Union Find data structure are:

- makeSet (x): Makes a new set containing a single element x
- union (x , y): Merges the two sets containing elements x and y
- find (x): Finds the set containing the element x

In a Union Find data structure, sets of vertices are stored in trees, where the root node of the tree is the representative node and the label of the set. When performing a find operation, we navigate from a node to its parent until we reach the root node of that set.

There are two important aspects of Union Find that significantly improve the time complexity.

Union by Rank: Each set maintains a rank index, which is initialized to 0 for singleton sets. While doing the union operation, the root node that has the smaller rank becomes a child node of the root node with the larger rank. In case there is a tie, the tie is broken arbitrarily and the rank index of the node that is selected to be the parent is incremented. Clearly, the union operation takes $O(1)$ time.

Path Compression: When performing a find operation, we recursively navigate from a child to its parent until we reach the root node. Once we have identified the root node, we make another pass through the path and "flatten" the tree by making all the intermediate nodes point directly to the root node. The idea behind path compression is that while this makes another pass at navigating the path, it makes the subsequent find operations significantly faster by shortening the path to the root. There are other variations of the path compression, some of which do not involve a second traversal of the path. For example, we can make each node point to its "grandparent," thus making the path half as long.

Next, we introduce a very fast-growing mathematical function, the inverse of which appears in the time complexity of the Union Find data structure.

Ackermann's function: Ackermann's function $(A(i,j))$ is defined as follows.

$A(1,j) = 2^j, j \geq 1$
$A(i,1) = A(i-1, 2), i \geq 2$
$A(i,j) = A(i-1, A(i, j-1)), i, j \geq 2$

Ackermann's function grows very rapidly as i and j are increased. For example, $A(2,4) = 2^{65,536}$ and $A(4,1) = A(2,16)$, which is much larger than $A(2,4)$.

The inverse of Ackermann's function is called the alpha function, which is defined as follows.

$alpha(p,q) = min\{z \geq 1 \mid A(z, p/q) > log_2 q\}, p \geq q \geq 1$

The inverse function grows very slowly.

$alpha(p,q) < 5$ until $q = 2^{A(4,1)}$

Thus, for all practical purposes, $alpha(p,q) < 5$.

Theorem 12.2 [Tarjan and Van Leeuwen]
Let $T(f,u)$ be the maximum time required to process any intermixed sequence of f finds and u unions. Assume that $u \geq n/2$.

$a\,(n + f\,alpha(f+n, n)) \leq T(f,u) \leq b(n + f\,alpha(f+n, n))$

where a and b are constants.

These bounds apply when we start with singleton sets and use either the weight or the height rule for unions and any one of the path compression methods for a find.

FACILITY LOCATION PROBLEM

APPENDIX E

The facility location problem is a central problem in operations research related to optimal placement of facilities. Here, "optimal" may refer to minimizing transportation or communication costs, response times, etc. We dedicate this special section only to introduce the facility location problem for the following reasons:

(i) It is a hard problem computationally, and a central problem in computer science and operations research.
(ii) It is a very practical problem and manifests in many different application areas.
(iii) The formulation of the problem itself is tricky, as the objectives may be very ambiguous. In fact, in many practical scenarios, it may be significantly harder to agree on the objectives than to actually solve the problem.

Some of the areas in which the facility location problem has significant applications are:

(i) *Operations*: Stores and Warehouses. Where do we build our warehouses so that they are close to our stores? How many should we build to attain efficiency? In this application, accuracy far outweighs speed of making the decision.

(ii) *Content Delivery Network*: High-traffic sites such as CNN, Yahoo, Rediff, etc. serve contents using a Content Delivery Network (such as Akamai or Amazon S3). Where should the next delivery node/ document cache be located in order to serve the customers? In this application, speed outweighs accuracy, as the dynamics of demand change quickly.

(iii) *Urban services*: Urban emergency services such as ambulances, fire trucks and police cars need to be located strategically. Where should ambulances be located so as to service each request as quickly as possible?

(iv) *Communication*: Where should the next communication tower be located? This application can have many variations — towers may be positioned anywhere (continuous); towers may be positioned in five available slots (discrete); or, towers may be located on the roadside (network).

A Mathematical Model

Input: We're given a weighted, connected graph. Each vertex represents a client having some demand and each edge represents a connection. Distance or transportation cost or communication cost can be modeled as weights on the edges and demand can be modeled as weights on the vertices.

Output: We need to identify k vertices within our graph, at which we will place facilities to serve all the other clients. At which vertices do we place our k facilities in order to minimize maximum distance for any client?

Constraints: There can be additional constraints such as "avoid placing hazardous materials near housing."

The metric k-center problem: Given n cities with specified distances, one wants to build k warehouses in different cities and minimize the distance of each city from its *closest* warehouse. The problem as defined in the "metric space" implies that distances obey triangle inequality. (This problem is NP-hard.)

STRING MATCHING

APPENDIX F

The problem of string matching or string searching is to find if a pattern $P[1..m]$ occurs within text $T[1..n]$. Typically, the length of the pattern (m) is much smaller than the length of the text (n).

A naïve string matching algorithm involves matching each position in the pattern to each position in the text. We match the pattern with the text as far as it matches. If it doesn't match, we shift the pattern by one character. The time complexity of this solution is $O(mn)$, where m is the size of the pattern and n is the size of the text. Consider a case in which $n = 3 \times 10^9$ and $m = 40{,}000$: how long does that solution take?[1]

One of the basic ideas in more sophisticated string matching algorithms is that we can try to understand how the pattern matches shifts against itself. If we know how the pattern matches against itself, we can slide the pattern ahead by more than just one character as in the naïve algorithm. KMP (Knuth Morris Pratt) String Searching Algorithm is a famous linear-time string matching algorithm that achieves an $O(m+n)$ running time. It uses an auxiliary function $pi[1..m]$ precomputed from P in $O(m)$ time.

Another idea is that while matching the pattern against the text, if we encounter a character in text that does not appear in the pattern, then we can shift the entire pattern to the right of that character. For example, given pattern P: $pappa\underline{r}$ and text T: $pappa\underline{x}papparrassanuaragh$, we observe that the letter x does not appear in P and we can shift the entire pattern P to the right of that x.

1 Hint: The values of n and m may be related to the size of the human genome and a typical gene, respectively.

Some string searching algorithms start matching from the right instead of the left. If we combine that idea with the previous observation, if we find a mismatch, we can jump m characters at a time. Consequently, in some circumstances, we may even be able to achieve an $O(n/m)$ running time if there are sufficient mismatches.

INDEX

WORKS CITED

[1] M. Ben-Or, "Lower Bounds for Algebraic Computation Trees," in *ACM Symposium on Theory of Computing*, New York, 1983.

[2] R. Sedgewick and P. Flajolet, An Introduction to the Analysis of Algorithms, 2nd ed., Upper Saddle River, NJ: Addison Wesley, 2013.

[3] K. Berman and J. Paul, Fundamentals of Sequential and Parallel Algorithms, Boston, MA: PWS Publishing Company, 1997.

[4] S. Dasgupta, C. Papadimitriou and U. Vazirani, Algorithms, New York, NY: McGraw-Hill, 2008.

[5] T. Cormen, C. Leiserson, R. Rivest and C. Stein, Introduction to Algorithms, 3rd ed., Cambridge, MA: MIT Press, 2009.

[6] E. Horowitz, S. Sahni and S. Rajasekaran, Computer Algorithms, New York, NY: Computer Science Press, 1998.

[7] D. Knuth, The Art of Computer Programming, 2nd ed., vol. 3: Sorting and Searching, Addison-Wesley, 1998.

[8] A. Levitin, The Design & Analysis of Algorithms, Boston, MA: Addison-Wesley, 2007.

[9] J. Pierce, An Introduction to Information Theory, 2nd ed., New York, NY: Dover, 1980.

[10] R. Sedgewick and K. Wayne, Algorithms, 4th ed., Upper Saddle River, NJ: Addison-Wesley, 2011.

[11] A. Land and A. Doig, "An Automated Method of Solving Discrete Programming Problems," *Econometrica,* vol. 28, no. 3, pp. 497–520, 1960.

[12] M. R. Garey and D. S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, 1st ed., W.H. Freeman & Co, 1979.
[13] J. E. Hopcroft, R. Motwani and J. D. Ullman, Automata Theory, Languages, and Computation, 3rd ed., Addison Wesley, 2006.
[14] T. Lanzone, "Travelling Salesman, Movie," Fretboard Pictures, [Online]. Available: http://www.travellingsalesmanmovie.com/.
[15] V. Williams, "An overview of the recent progress on matrix multiplication," *ACM SIGACT News,* vol. 43, no. 4, pp. 57–59, December 2012.
[16] M. Ridley, Genome: The Autobiography of a Species in 23 Chapters, Harper Perennial, 2006.