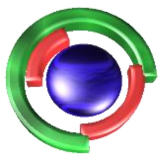
FUTURE VISION BIE

One Stop for All Study Materials & Lab Programs



Future Vision

By K B Hemanth Raj

Scan the QR Code to Visit the Web Page

Or

Visit : https://hemanthrajhemu.github.io

Gain Access to All Study Materials according to VTU,
CSE – Computer Science Engineering,
ISE – Information Science Engineering,
ECE - Electronics and Communication Engineering
& MORE...

Join Telegram to get Instant Updates: https://bit.ly/VTU_TELEGRAM

Contact: MAIL: futurevisionbie@gmail.com

INSTAGRAM: www.instagram.com/hemanthraj_hemu/

INSTAGRAM: www.instagram.com/futurevisionbie/

WHATSAPP SHARE: https://bit.ly/FVBIESHARE

Introduction to The Design & Analysis of Algorithms

2ND EDITION

Anany Levitin
Villanova University

Boston San Francisco New York
London Toronto Sydney Tokyo Singapore Madrid
Mexico City Munich Paris Cape Town Hong Kong Montreal

https://hemanthrajhemu.github.io

Contents

Preface xvii

1 Introduction 1

1.1 What is an Algorithm? 3
Exercises 1.1 8

1.2 Fundamentals of Algorithmic Problem Solving 9
Understanding the Problem 9
Ascertaining the Capabilities of a Computational Device 11
Choosing between Exact and Approximate Problem Solving 11
Deciding on Appropriate Data Structures 12
Algorithm Design Techniques 12
Methods of Specifying an Algorithm 12
Proving an Algorithm's Correctness 13
Analyzing an Algorithm 14
Coding an Algorithm 15
Exercises 1.2 17

1.3 Important Problem Types 19
Sorting 19
Searching 20
String Processing 21
Graph Problems 21
Combinatorial Problems 22
Geometric Problems 22
Numerical Problems 23
Exercises 1.3 23

https://hemanthrajhemu.github.io

1.4 Fundamental Data Structures 26
Linear Data Structures 26
Graphs 28
Trees 32
Sets and Dictionaries 36
Exercises 1.4 38
Summary 39

2 Fundamentals of the Analysis of Algorithm Efficiency 41

2.1 Analysis Framework 42
Measuring an Input's Size 43
Units for Measuring Running Time 44
Orders of Growth 45
Worst-Case, Best-Case, and Average-Case Efficiencies 47
Recapitulation of the Analysis Framework 50
Exercises 2.1 50

2.2 Asymptotic Notations and Basic Efficiency Classes 52
Informal Introduction 52
O-notation 53
Ω-notation 54
Θ-notation 55
Useful Property Involving the Asymptotic Notations 56
Using Limits for Comparing Orders of Growth 57
Basic Efficiency Classes 58
Exercises 2.2 59

2.3 Mathematical Analysis of Nonrecursive Algorithms 61
Exercises 2.3 67

2.4 Mathematical Analysis of Recursive Algorithms 69
Exercises 2.4 76

2.5 Example: Fibonacci Numbers 78
Explicit Formula for the nth Fibonacci Number 79
Algorithms for Computing Fibonacci Numbers 80
Exercises 2.5 83

https://hemanthrajhemu.github.io

2.6 Empirical Analysis of Algorithms 84
Exercises 2.6 90

2.7 Algorithm Visualization 91
Summary 95

3 Brute Force 97

3.1 Selection Sort and Bubble Sort 98
Selection Sort 98
Bubble Sort 100
Exercises 3.1 102

3.2 Sequential Search and Brute-Force String Matching 103
Sequential Search 103
Brute-Force String Matching 104
Exercises 3.2 105

3.3 Closest-Pair and Convex-Hull Problems by Brute Force 107
Closest-Pair Problem 107
Convex-Hull Problem 108
Exercises 3.3 112

3.4 Exhaustive Search 114
Traveling Salesman Problem 114
Knapsack Problem 115
Assignment Problem 116
Exercises 3.4 119
Summary 120

4 Divide-and-Conquer 123

4.1 Mergesort 125
Exercises 4.1 128

4.2 Quicksort 129
Exercises 4.2 134

4.3 Binary Search 135
Exercises 4.3 138

https://hemanthrajhemu.github.io

1

Introduction

> Two ideas lie gleaming on the jeweler's velvet. The first is the calculus, the second, the algorithm. The calculus and the rich body of mathematical analysis to which it gave rise made modern science possible; but it has been the algorithm that has made possible the modern world.
>
> —David Berlinski, *The Advent of the Algorithm,* 2000

Why do you need to study algorithms? If you are going to be a computer professional, there are both practical and theoretical reasons to study algorithms. From a practical standpoint, you have to know a standard set of important algorithms from different areas of computing; in addition, you should be able to design new algorithms and analyze their efficiency. From the theoretical standpoint, the study of algorithms, sometimes called ***algorithmics***, has come to be recognized as the cornerstone of computer science. David Harel, in his delightful book pointedly titled *Algorithmics: the Spirit of Computing,* put it as follows:

> Algorithmics is more than a branch of computer science. It is the core of computer science, and, in all fairness, can be said to be relevant to most of science, business, and technology. [Har92], p. 6.

But even if you are not a student in a computing-related program, there are compelling reasons to study algorithms. To put it bluntly, computer programs would not exist without algorithms. And with computer applications becoming indispensable in almost all aspects of our professional and personal lives, studying algorithms becomes a necessity for more and more people.

Another reason for studying algorithms is their usefulness in developing analytical skills. After all, algorithms can be seen as special kinds of solutions to problems—not answers but precisely defined procedures for getting answers. Consequently, specific algorithm design techniques can be interpreted as problem-solving strategies that can be useful regardless of whether a computer is involved. Of course, the precision inherently imposed by algorithmic thinking limits the kinds of problems that can be solved with an algorithm. You will not find, for example, an algorithm for living a happy life or becoming rich and famous. On

https://hemanthrajhemu.github.io

the other hand, this required precision has an important educational advantage. Donald Knuth, one of the most prominent computer scientists in the history of algorithmics, put it as follows:

> A person well-trained in computer science knows how to deal with algorithms: how to construct them, manipulate them, understand them, analyze them. This knowledge is preparation for much more than writing good computer programs; it is a general-purpose mental tool that will be a definite aid to the understanding of other subjects, whether they be chemistry, linguistics, or music, etc. The reason for this may be understood in the following way: It has often been said that a person does not really understand something until after teaching it to someone else. Actually, a person does not *really* understand something until after teaching it to a *computer*, i.e., expressing it as an algorithm . . . An attempt to formalize things as algorithms leads to a much deeper understanding than if we simply try to comprehend things in the traditional way. [Knu96], p. 9.

We take up the notion of algorithm in Section 1.1. As examples, we use three algorithms for the same problem: computing the greatest common divisor. There are several reasons for this choice. First, it deals with a problem familiar to everybody from their middle-school days. Second, it makes the important point that the same problem can often be solved by several algorithms. Quite typically, these algorithms differ in their idea, level of sophistication, and efficiency. Third, one of these algorithms deserves to be introduced first, both because of its age—it appeared in Euclid's famous treatise more than two thousand years ago—and its enduring power and importance. Finally, the middle-school procedure for computing the greatest common divisor allows us to highlight a critical requirement every algorithm must satisfy.

Section 1.2 deals with algorithmic problem solving. There we discuss several important issues related to the design and analysis of algorithms. The different aspects of algorithmic problem solving range from analysis of the problem and the means of expressing an algorithm to establishing its correctness and analyzing its efficiency. The section does not contain a magic recipe for designing an algorithm for an arbitrary problem. It is a well-established fact that such a recipe does not exist. Still, the material of Section 1.2 should be useful for organizing your work on designing and analyzing algorithms.

Section 1.3 is devoted to a few problem types that have proven to be particularly important to the study of algorithms and their application. In fact, there are textbooks (e.g., [Sed88]) organized around such problem types. I hold the view—shared by many others—that an organization based on algorithm design techniques is superior. In any case, it is very important to be aware of the principal problem types. Not only are they the most commonly encountered problem types in real-life applications, they are used throughout the book to demonstrate particular algorithm design techniques.

https://hemanthrajhemu.github.io

Section 1.4 contains a review of fundamental data structures. It is meant to serve as a reference rather than a deliberate discussion of this topic. If you need a more detailed exposition, there is a wealth of good books on the subject, most of them tailored to a particular programming language.

1.1 What is an Algorithm?

Although there is no universally agreed-on wording to describe this notion, there is general agreement about what the concept means:

> An ***algorithm*** is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.

This definition can be illustrated by a simple diagram (Figure 1.1).

The reference to "instructions" in the definition implies that there is something or someone capable of understanding and following the instructions given. We call this a "computer," keeping in mind that before the electronic computer was invented, the word "computer" meant a human being involved in performing numeric calculations. Nowadays, of course, "computers" are those ubiquitous electronic devices that have become indispensable in almost everything we do. Note, however, that although the majority of algorithms are indeed intended for eventual computer implementation, the notion of algorithm does not depend on such an assumption.

As examples illustrating the notion of algorithm, we consider in this section three methods for solving the same problem: computing the greatest common divisor of two integers. These examples will help us to illustrate several important points:

- The nonambiguity requirement for each step of an algorithm cannot be compromised.

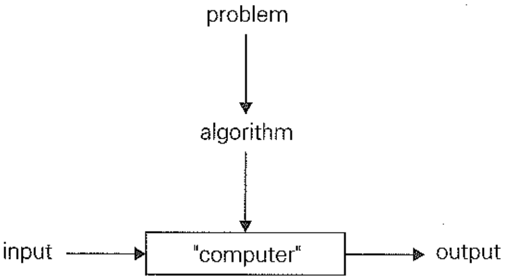


FIGURE 1.1 Notion of algorithm

https://hemanthrajhemu.github.io

- The range of inputs for which an algorithm works has to be specified carefully.
- The same algorithm can be represented in several different ways.
- Several algorithms for solving the same problem may exist.
- Algorithms for the same problem can be based on very different ideas and can solve the problem with dramatically different speeds.

Recall that the greatest common divisor of two nonnegative, not-both-zero integers m and n, denoted $\gcd(m, n)$, is defined as the largest integer that divides both m and n evenly, i.e., with a remainder of zero. Euclid of Alexandria (third century B.C.) outlined an algorithm for solving this problem in one of the volumes of his *Elements*, most famous for its systematic exposition of geometry. In modern terms, ***Euclid's algorithm*** is based on applying repeatedly the equality

$$\gcd(m, n) = \gcd(n, m \bmod n)$$

(where $m \bmod n$ is the remainder of the division of m by n) until $m \bmod n$ is equal to 0; since $\gcd(m, 0) = m$ (why?), the last value of m is also the greatest common divisor of the initial m and n.

For example, $\gcd(60, 24)$ can be computed as follows:

$$\gcd(60, 24) = \gcd(24, 12) = \gcd(12, 0) = 12.$$

(If you are not impressed by this algorithm, try finding the greatest common divisor of larger numbers such as those in Problem 5 of Exercises 1.1.)

Here is a more structured description of this algorithm:

Euclid's algorithm for computing $\gcd(m, n)$

Step 1 If $n = 0$, return the value of m as the answer and stop; otherwise, proceed to Step 2.

Step 2 Divide m by n and assign the value of the remainder to r.

Step 3 Assign the value of n to m and the value of r to n. Go to Step 1.

Alternatively, we can express the same algorithm in a pseudocode:

```
ALGORITHM  Euclid(m, n)
    //Computes gcd(m, n) by Euclid's algorithm
    //Input: Two nonnegative, not-both-zero integers m and n
    //Output: Greatest common divisor of m and n
    while n ≠ 0 do
        r ← m mod n
        m ← n
        n ← r
    return m
```

How do we know that Euclid's algorithm eventually comes to a stop? This follows from the observation that the second number of the pair gets smaller with

https://hemanthrajhemu.github.io

each iteration and it cannot become negative. Indeed, the new value of n on the next iteration is $m \bmod n$, which is always smaller than n. Hence, the value of the second number in the pair eventually becomes 0, and the algorithm stops.

Just as with many other problems, there are several algorithms for computing the greatest common divisor. Let us look at the other two methods for this problem. The first is simply based on the definition of the greatest common divisor of m and n as the largest integer that divides both numbers evenly. Obviously, such a common divisor cannot be greater than the smaller of these numbers, which we will denote by $t = \min\{m, n\}$. So we can start by checking whether t divides both m and n: if it does, t is the answer; if it does not, we simply decrease t by 1 and try again. (How do we know that the process will eventually stop?) For example, for numbers 60 and 24, the algorithm will try first 24, then 23, and so on until it reaches 12, where it stops.

Consecutive integer checking algorithm for computing $\gcd(m, n)$

Step 1 Assign the value of $\min\{m, n\}$ to t.

Step 2 Divide m by t. If the remainder of this division is 0, go to Step 3; otherwise, go to Step 4.

Step 3 Divide n by t. If the remainder of this division is 0, return the value of t as the answer and stop; otherwise, proceed to Step 4.

Step 4 Decrease the value of t by 1. Go to Step 2.

Note that unlike Euclid's algorithm, this algorithm, in the form presented, does not work correctly when one of its input numbers is zero. This example illustrates why it is so important to specify the range of an algorithm's inputs explicitly and carefully.

The third procedure for finding the greatest common divisor should be familiar to you from middle school.

Middle-school procedure for computing $\gcd(m, n)$

Step 1 Find the prime factors of m.

Step 2 Find the prime factors of n.

Step 3 Identify all the common factors in the two prime expansions found in Step 1 and Step 2. (If p is a common factor occurring p_m and p_n times in m and n, respectively, it should be repeated $\min\{p_m, p_n\}$ times.)

Step 4 Compute the product of all the common factors and return it as the greatest common divisor of the numbers given.

Thus, for the numbers 60 and 24, we get

$$60 = 2 \cdot 2 \cdot 3 \cdot 5$$
$$24 = 2 \cdot 2 \cdot 2 \cdot 3$$
$$\gcd(60, 24) = 2 \cdot 2 \cdot 3 = 12.$$

https://hemanthrajhemu.github.io

Nostalgia for the days when we learned this method should not prevent us from noting that the last procedure is much more complex and slower than Euclid's algorithm. (We will discuss methods for finding and comparing running times of algorithms in the next chapter.) In addition to inferior efficiency, the middle-school procedure does not qualify, in the form presented, as a legitimate algorithm. Why? Because the prime factorization steps are not defined unambiguously: they require a list of prime numbers, and I strongly suspect that your middle-school teacher did not explain how to obtain such a list. You undoubtedly agree that this is not a matter of unnecessary nitpicking. Unless this issue is resolved, we cannot, say, write a program implementing this procedure. (Incidentally, Step 3 is also not defined clearly enough. Its ambiguity is much easier to rectify than that of the factorization steps, however. How would you find common elements in two sorted lists?)

So let us introduce a simple algorithm for generating consecutive primes not exceeding any given integer n. It was probably invented in ancient Greece and is known as the ***sieve of Eratosthenes*** (ca. 200 B.C.). The algorithm starts by initializing a list of prime candidates with consecutive integers from 2 to n. Then, on the first iteration of the algorithm, it eliminates from the list all multiples of 2, i.e., 4, 6, and so on. Then it moves to the next item on the list, which is 3, and eliminates its multiples. (In this straightforward version, there is an overhead because some numbers, such as 6, are eliminated more than once.) No pass for number 4 is needed: since 4 itself and all its multiples are also multiples of 2, they were already eliminated on a previous pass. (By similar reasoning, we need not consider multiples of any eliminated number.) The next remaining number on the list, which is used on the third pass, is 5. The algorithm continues in this fashion until no more numbers can be eliminated from the list. The remaining integers of the list are the primes needed.

As an example, consider the application of the algorithm to finding the list of primes not exceeding $n = 25$:

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
2	3		5		7		9		11		13		15		17		19		21		23		25
2	**3**		5		7				11		13				17		19				23		25
2	3		**5**		7				11		13				17		19				23		

For this example, no more passes are needed because they would eliminate numbers already eliminated on previous iterations of the algorithm. The remaining numbers on the list are the consecutive primes less than or equal to 25.

In general, what is the largest number p whose multiples can still remain on the list? Before we answer this question, let us first note that if p is a number whose multiples are being eliminated on the current pass, then the first multiple we should consider is $p \cdot p$ because all its smaller multiples $2p, \ldots, (p-1)p$ have been eliminated on earlier passes through the list. This observation helps to avoid

https://hemanthrajhemu.github.io

eliminating the same number more than once. Obviously, $p \cdot p$ should not be greater than n, and therefore p cannot exceed $\sqrt{n}$ rounded down (denoted $\lfloor \sqrt{n} \rfloor$ using the so-called floor function). We assume in the following pseudocode that there is a function available for computing $\lfloor \sqrt{n} \rfloor$; alternatively, we could check the inequality $p \cdot p \le n$ as the loop continuation condition there.

ALGORITHM *Sieve(n)*

```
//Implements the sieve of Eratosthenes
//Input: An integer n ≥ 2
//Output: Array L of all prime numbers less than or equal to n
for p ← 2 to n do A[p] ← p
for p ← 2 to ⌊√n⌋ do    //see note before pseudocode
    if A[p] ≠ 0    //p hasn't been eliminated on previous passes
        j ← p * p
        while j ≤ n do
            A[j] ← 0    //mark element as eliminated
            j ← j + p
//copy the remaining elements of A to array L of the primes
i ← 0
for p ← 2 to n do
    if A[p] ≠ 0
        L[i] ← A[p]
        i ← i + 1
return L
```

So now we can incorporate the sieve of Eratosthenes into the middle-school procedure to get a legitimate algorithm for computing the greatest common divisor of two positive integers. Note that special care needs to be exercised if one or both input numbers are equal to 1: because mathematicians do not consider 1 to be a prime number, strictly speaking, the method does not work for such inputs.

Before we leave this section, one more comment is in order. The examples considered in this section notwithstanding, the majority of algorithms in use today—even those that are implemented as computer programs—do not deal with mathematical problems. Look around for algorithms helping us through our daily routines, both professional and personal. May this ubiquity of algorithms in today's world strengthen your resolve to learn more about these fascinating engines of the information age.

https://hemanthrajhemu.github.io

Exercises 1.1

1. Do some research on al-Khorezmi (also al-Khwarizmi), the man from whose name the word "algorithm" is derived. In particular, you should learn what the origins of the words "algorithm" and "algebra" have in common.

2. Given that the official purpose of the U.S. patent system is the promotion of the "useful arts," do you think algorithms are patentable in this country? Should they be?

3. **a.** Write down driving directions for going from your school to your home with the precision required by an algorithm.

   b. Write down a recipe for cooking your favorite dish with the precision required by an algorithm.

4. Design an algorithm for computing $\lfloor\sqrt{n}\rfloor$ for any positive integer n. Besides assignment and comparison, your algorithm may only use the four basic arithmetical operations.

5. **a.** Find gcd(31415, 14142) by applying Euclid's algorithm.

   b. Estimate how many times faster it will be to find gcd(31415, 14142) by Euclid's algorithm compared with the algorithm based on checking consecutive integers from $\min\{m, n\}$ down to $\gcd(m, n)$.

6. Prove the equality $\gcd(m, n) = \gcd(n, m \bmod n)$ for every pair of positive integers m and n.

7. What does Euclid's algorithm do for a pair of numbers in which the first number is smaller than the second one? What is the largest number of times this can happen during the algorithm's execution on such an input?

8. **a.** What is the smallest number of divisions made by Euclid's algorithm among all inputs $1 \le m, n \le 10$?

   b. What is the largest number of divisions made by Euclid's algorithm among all inputs $1 \le m, n \le 10$?

9. **a.** Euclid's algorithm, as presented in Euclid's treatise, uses subtractions rather than integer divisions. Write a pseudocode for this version of Euclid's algorithm.

   b. *Euclid's game* (see [Bog]) starts with two unequal positive numbers on the board. Two players move in turn. On each move, a player has to write on the board a positive number equal to the difference of two numbers already on the board; this number must be new, i.e., different from all the numbers already on the board. The player who cannot move loses the game. Should you choose to move first or second in this game?

10. The ***extended Euclid's algorithm*** determines not only the greatest common divisor d of two positive integers m and n but also integers (not necessarily positive) x and y, such that $mx + ny = d$.

https://hemanthrajhemu.github.io

a. Look up a description of the extended Euclid's algorithm (see, e.g., [KnuI], p. 13) and implement it in the language of your choice.

b. Modify your program for finding integer solutions to the Diophantine equation $ax + by = c$ with any set of integer coefficients a, b, and c.

11. *Locker doors* There are n lockers in a hallway, numbered sequentially from 1 to n. Initially all the locker doors are closed. You make n passes by the lockers, each time starting with locker #1. On the ith pass, $i = 1, 2, \dots, n$, you toggle the door of every ith locker: if the door is closed, you open it; if it is open, you close it. For example, after the first pass every door is open; on the second pass you only toggle the even-numbered lockers (#2, #4, . . .) so that after the second pass the even doors are closed and the odd ones are open; the third time through, you close the door of locker #3 (opened from the first pass), open the door of locker #6 (closed from the second pass), and so on. After the last pass, which locker doors are open and which are closed? How many of them are open?

1.2 Fundamentals of Algorithmic Problem Solving

Let us start by reiterating an important point made in the introduction to this chapter:

We can consider algorithms to be procedural solutions to problems.

These solutions are not answers but specific instructions for getting answers. It is this emphasis on precisely defined constructive procedures that makes computer science distinct from other disciplines. In particular, this distinguishes it from theoretical mathematics whose practitioners are typically satisfied with just proving the existence of a solution to a problem and, possibly, investigating the solution's properties.

We now list and briefly discuss a sequence of steps one typically goes through in designing and analyzing an algorithm (Figure 1.2).

Understanding the Problem

From a practical perspective, the first thing you need to do before designing an algorithm is to understand completely the problem given. Read the problem's description carefully and ask questions if you have any doubts about the problem, do a few small examples by hand, think about special cases, and ask questions again if needed.

There are a few types of problems that arise in computing applications quite often. We review them in the next section. If the problem in question is one of them, you might be able to use a known algorithm for solving it. Of course, it helps to understand how such an algorithm works and to know its strengths and

https://hemanthrajhemu.github.io

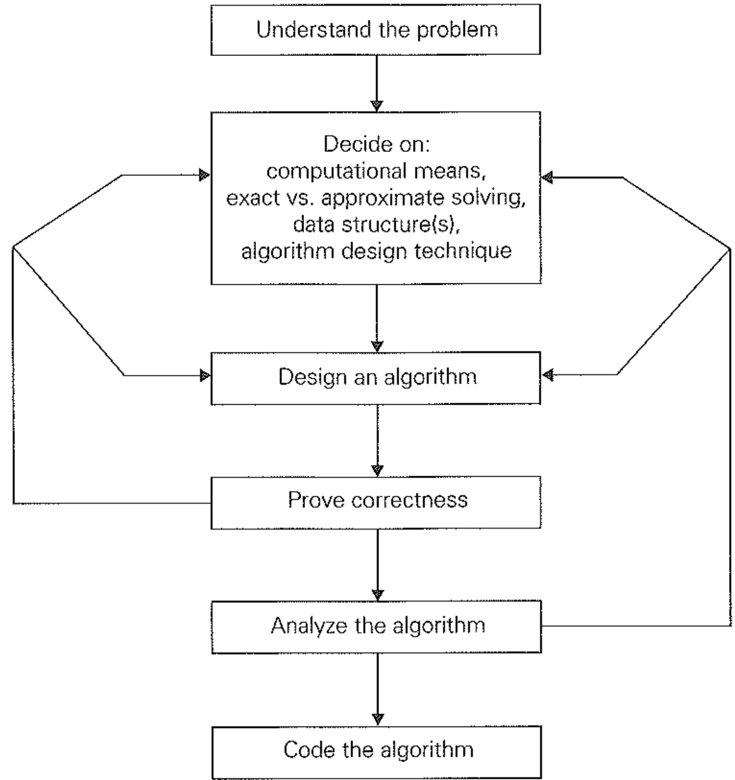


FIGURE 1.2 Algorithm design and analysis process

weaknesses, especially if you have to choose among several available algorithms. But often, you will not find a readily available algorithm and will have to design your own. The sequence of steps outlined in this section should help you in this exciting but not always easy task.

An input to an algorithm specifies an ***instance*** of the problem the algorithm solves. It is very important to specify exactly the range of instances the algorithm needs to handle. (As an example, recall the variations in the range of instances for the three greatest common divisor algorithms discussed in the previous section.) If you fail to do this, your algorithm may work correctly for a majority of inputs but crash on some "boundary" value. Remember that a correct algorithm is not one that works most of the time, but one that works correctly for *all* legitimate inputs.

Do not skimp on this first step of the algorithmic problem-solving process; if you do, you will run the risk of unnecessary rework.

https://hemanthrajhemu.github.io

Ascertaining the Capabilities of a Computational Device

Once you completely understand a problem, you need to ascertain the capabilities of the computational device the algorithm is intended for. The vast majority of algorithms in use today are still destined to be programmed for a computer closely resembling the von Neumann machine—a computer architecture outlined by the prominent Hungarian-American mathematician John von Neumann (1903–1957), in collaboration with A. Burks and H. Goldstine, in 1946. The essence of this architecture is captured by the so-called ***random-access machine*** (***RAM***). Its central assumption is that instructions are executed one after another, one operation at a time. Accordingly, algorithms designed to be executed on such machines are called ***sequential algorithms***.

The central assumption of the RAM model does not hold for some newer computers that can execute operations concurrently, i.e., in parallel. Algorithms that take advantage of this capability are called ***parallel algorithms***. Still, studying the classic techniques for design and analysis of algorithms under the RAM model remains the cornerstone of algorithmics for the foreseeable future.

Should you worry about the speed and amount of memory of a computer at your disposal? If you are designing an algorithm as a scientific exercise, the answer is a qualified no. As you will see in Section 2.1, most computer scientists prefer to study algorithms in terms independent of specification parameters for a particular computer. If you are designing an algorithm as a practical tool, the answer may depend on a problem you need to solve. Even "slow" computers of today are almost unimaginably fast. Consequently, in many situations, you need not worry about a computer being too slow for the task. There are important problems, however, that are very complex by their nature, have to process huge volumes of data, or deal with applications where time is critical. In such situations, it is imperative to be aware of the speed and memory available on a particular computer system.

Choosing between Exact and Approximate Problem Solving

The next principal decision is to choose between solving the problem exactly or solving it approximately. In the former case, an algorithm is called an ***exact algorithm***; in the latter case, an algorithm is called an ***approximation algorithm***. Why would one opt for an approximation algorithm? First, there are important problems that simply cannot be solved exactly for most of their instances; examples include extracting square roots, solving nonlinear equations, and evaluating definite integrals. Second, available algorithms for solving a problem exactly can be unacceptably slow because of the problem's intrinsic complexity. This happens, in particular, for many problems involving a very large number of choices; you will see examples of such difficult problems in Chapters 3, 11, and 12. Third, an approximation algorithm can be a part of a more sophisticated algorithm that solves a problem exactly.

https://hemanthrajhemu.github.io

Deciding on Appropriate Data Structures

Some algorithms do not demand any ingenuity in representing their inputs. But others are, in fact, predicated on ingenious data structures. In addition, some of the algorithm design techniques we shall discuss in Chapters 6 and 7 depend intimately on structuring or restructuring data specifying a problem's instance. Many years ago, an influential textbook proclaimed the fundamental importance of both algorithms and data structures for computer programming by its very title: *Algorithms + Data Structures = Programs* [Wir76]. In the new world of object-oriented programming, data structures remain crucially important for both design and analysis of algorithms. We review basic data structures in Section 1.4.

Algorithm Design Techniques

Now, with all the components of the algorithmic problem solving in place, how do you design an algorithm to solve a given problem? This is the main question this book seeks to answer by teaching you several general design techniques.

What is an algorithm design technique?

> An ***algorithm design technique*** (or "strategy" or "paradigm") is a general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing.

Check this book's table of contents and you will see that a majority of its chapters are devoted to individual design techniques. They distill a few key ideas that have proven to be useful in designing algorithms. Learning these techniques is of utmost importance for the following reasons.

First, they provide guidance for designing algorithms for new problems, i.e., problems for which there is no known satisfactory algorithm. Therefore—to use the language of a famous proverb—learning such techniques is akin to learning to fish as opposed to being given a fish caught by somebody else. It is not true, of course, that each of these general techniques will be necessarily applicable to every problem you may encounter. But taken together, they do constitute a powerful collection of tools that you will find quite handy in your studies and work.

Second, algorithms are the cornerstone of computer science. Every science is interested in classifying its principal subject, and computer science is no exception. Algorithm design techniques make it possible to classify algorithms according to an underlying design idea; therefore, they can serve as a natural way to both categorize and study algorithms.

Methods of Specifying an Algorithm

Once you have designed an algorithm, you need to specify it in some fashion. In Section 1.1, to give you an example, we described Euclid's algorithm in words (in a free and also a step-by-step form) and in pseudocode. These are the two options that are most widely used nowadays for specifying algorithms.

https://hemanthrajhemu.github.io

Using a natural language has an obvious appeal; however, the inherent ambiguity of any natural language makes a succinct and clear description of algorithms surprisingly difficult. Nevertheless, being able to do this is an important skill that you should strive to develop in the process of learning algorithms.

A ***pseudocode*** is a mixture of a natural language and programming language-like constructs. A pseudocode is usually more precise than a natural language, and its usage often yields more succinct algorithm descriptions. Surprisingly, computer scientists have never agreed on a single form of pseudocode, leaving textbook authors to design their own "dialects." Fortunately, these dialects are so close to each other that anyone familiar with a modern programming language should be able to understand them all.

This book's dialect was selected to cause minimal difficulty for a reader. For the sake of simplicity, we omit declarations of variables and use indentation to show the scope of such statements as **for**, **if**, and **while**. As you saw in the previous section, we use an arrow $\leftarrow$ for the assignment operation and two slashes // for comments.

In the earlier days of computing, the dominant vehicle for specifying algorithms was a ***flowchart***, a method of expressing an algorithm by a collection of connected geometric shapes containing descriptions of the algorithm's steps. This representation technique has proved to be inconvenient for all but very simple algorithms; nowadays, it can be found only in old algorithm books.

The state of the art of computing has not yet reached a point where an algorithm's description—whether in a natural language or a pseudocode—can be fed into an electronic computer directly. Instead, it needs to be converted into a computer program written in a particular computer language. We can look at such a program as yet another way of specifying the algorithm, although it is preferable to consider it as the algorithm's implementation.

Proving an Algorithm's Correctness

Once an algorithm has been specified, you have to prove its ***correctness***. That is, you have to prove that the algorithm yields a required result for every legitimate input in a finite amount of time. For example, correctness of Euclid's algorithm for computing the greatest common divisor stems from correctness of the equality $\gcd(m, n) = \gcd(n, m \bmod n)$ (which, in turn, needs a proof; see Problem 6 in Exercises 1.1), the simple observation that the second number gets smaller on every iteration of the algorithm, and the fact that the algorithm stops when the second number becomes 0.

For some algorithms, a proof of correctness is quite easy; for others, it can be quite complex. A common technique for proving correctness is to use mathematical induction because an algorithm's iterations provide a natural sequence of steps needed for such proofs. It might be worth mentioning that although tracing the algorithm's performance for a few specific inputs can be a very worthwhile activity, it cannot prove the algorithm's correctness conclusively. But in order to

https://hemanthrajhemu.github.io

show that an algorithm is incorrect, you need just one instance of its input for which the algorithm fails. If the algorithm is found to be incorrect, you need to either redesign it under the same decisions regarding the data structures, the design technique, and so on, or, in a more dramatic reversal, to reconsider one or more of those decisions (see Figure 1.2).

The notion of correctness for approximation algorithms is less straightforward than it is for exact algorithms. For an approximation algorithm, we usually would like to be able to show that the error produced by the algorithm does not exceed a predefined limit. You can find examples of such investigations in Chapter 12.

Analyzing an Algorithm

We usually want our algorithms to possess several qualities. After correctness, by far the most important is efficiency. In fact, there are two kinds of algorithm efficiency: time efficiency and space efficiency. ***Time efficiency*** indicates how fast the algorithm runs; ***space efficiency*** indicates how much extra memory the algorithm needs. A general framework and specific techniques for analyzing an algorithm's efficiency appear in Chapter 2.

Another desirable characteristic of an algorithm is ***simplicity***. Unlike efficiency, which can be precisely defined and investigated with mathematical rigor, simplicity, like beauty, is to a considerable degree in the eye of the beholder. For example, most people would agree that Euclid's algorithm is simpler than the middle-school procedure for computing $\gcd(m, n)$, but it is not clear whether Euclid's algorithm is simpler than the consecutive integer checking algorithm. Still, simplicity is an important algorithm characteristic to strive for. Why? Because simpler algorithms are easier to understand and easier to program; consequently, the resulting programs usually contain fewer bugs. There is also the undeniable aesthetic appeal of simplicity. Sometimes simpler algorithms are also more efficient than more complicated alternatives. Unfortunately, it is not always true, in which case a judicious compromise needs to be made.

Yet another desirable characteristic of an algorithm is ***generality***. There are, in fact, two issues here: generality of the problem the algorithm solves and the range of inputs it accepts. On the first issue, note that it is sometimes easier to design an algorithm for a problem posed in more general terms. Consider, for example, the problem of determining whether two integers are relatively prime, i.e., whether their only common divisor is equal to 1. It is easier to design an algorithm for a more general problem of computing the greatest common divisor of two integers and, to solve the former problem, check whether the gcd is 1 or not. There are situations, however, where designing a more general algorithm is unnecessary or difficult or even impossible. For example, it is unnecessary to sort a list of n numbers to find its median, which is its $\lceil n/2 \rceil$th smallest element. To give another example, the standard formula for roots of a quadratic equation cannot be generalized to handle polynomials of arbitrary degrees.

https://hemanthrajhemu.github.io

As to the range of inputs, your main concern should be designing an algorithm that can handle a range of inputs that is natural for the problem at hand. For example, excluding integers equal to 1 as possible inputs for a greatest common divisor algorithm would be quite unnatural. On the other hand, although the standard formula for the roots of a quadratic equation holds for complex coefficients, we would normally not implement it on this level of generality unless this capability is explicitly required.

If you are not satisfied with the algorithm's efficiency, simplicity, or generality, you must return to the drawing board and redesign the algorithm. In fact, even if your evaluation is positive, it is still worth searching for other algorithmic solutions. Recall the three different algorithms in the previous section for computing the greatest common divisor; generally, you should not expect to get the best algorithm on the first try. At the very least, you should try to fine-tune the algorithm you already have. For example, we made several improvements in our implementation of the sieve of Eratosthenes compared with its initial outline in Section 1.1. (Can you identify them?) You will do well if you keep in mind the following observation of Antoine de Saint-Exupéry, the French writer, pilot, and aircraft designer: "A designer knows he has arrived at perfection not when there is no longer anything to add, but when there is no longer anything to take away."[1]

Coding an Algorithm

Most algorithms are destined to be ultimately implemented as computer programs. Programming an algorithm presents both a peril and an opportunity. The peril lies in the possibility of making the transition from an algorithm to a program either incorrectly or very inefficiently. Some influential computer scientists strongly believe that unless the correctness of a computer program is proven with full mathematical rigor, the program cannot be considered correct. They have developed special techniques for doing such proofs (see [Gri81]), but the power of these techniques of formal verification is limited so far to very small programs. As a practical matter, the validity of programs is still established by testing. Testing of computer programs is an art rather than a science, but that does not mean that there is nothing in it to learn. Look up books devoted to testing and debugging; even more important, test and debug your program thoroughly whenever you implement an algorithm.

Also note that throughout the book, we assume that inputs to algorithms fall within their specified ranges and hence require no verification. When implementing algorithms as programs to be used in actual applications, you should provide such verifications.

1. I found this call for design simplicity in an essay collection by Jon Bentley [Ben00]; the essays deal with a variety of issues in algorithm design and implementation, and are justifiably titled *Programming Pearls*. I wholeheartedly recommend writings of both Jon Bentley and Antoine de Saint-Exupéry.

https://hemanthrajhemu.github.io

Of course, implementing an algorithm correctly is necessary but not sufficient: you would not like to diminish your algorithm's power by an inefficient implementation. Modern compilers do provide a certain safety net in this regard, especially when they are used in their code optimization mode. Still, you need to be aware of such standard tricks as computing a loop's invariant (an expression that does not change its value) outside the loop, collecting common subexpressions, replacing expensive operations by cheap ones, and so on. (See [Ker99] and [Ben00] for a good discussion of code tuning and other issues related to algorithm programming.) Typically, such improvements can speed up a program only by a constant factor, whereas a better algorithm can make a difference in running time by orders of magnitude. But once an algorithm is selected, a 10–50% speedup may be worth an effort.

A working program provides an additional opportunity in allowing an empirical analysis of the underlying algorithm. The analysis is based on timing the program on several inputs and then analyzing the results obtained. We discuss the advantages and disadvantages of this approach to analyzing algorithms in Section 2.6.

In conclusion, let us emphasize again the main lesson of the process depicted in Figure 1.2:

As a rule, a good algorithm is a result of repeated effort and rework.

Even if you have been fortunate enough to get an algorithmic idea that seems perfect, you should still try to see whether it can be improved.

Actually, this is good news since it makes the ultimate result so much more enjoyable. (Yes, I did think of naming this book *The Joy of Algorithms*.) On the other hand, how does one know when to stop? In the real world, more often than not the project's schedule or the patience of your boss will stop you. And so it should be: perfection is expensive and in fact not always called for. Designing an algorithm is an engineering-like activity that calls for compromises among competing goals under the constraints of available resources, with the designer's time being one of the resources.

In the academic world, the question leads to an interesting but usually difficult investigation of an algorithm's ***optimality***. Actually, this question is not about the efficiency of an algorithm but about the complexity of the problem it solves: what is the minimum amount of effort *any* algorithm will need to exert to solve the problem in question? For some problems, the answer to this question is known. For example, any algorithm that sorts an array by comparing values of its elements needs about $n \log_2 n$ comparisons for some arrays of size n (see Section 11.2). But for many seemingly easy problems, such as matrix multiplication, computer scientists do not yet have a final answer.

Another important issue of algorithmic problem solving is the question of whether or not every problem can be solved by an algorithm. We are not talking here about problems that do not have a solution, such as finding real roots of

https://hemanthrajhemu.github.io

a quadratic equation with a negative discriminant. For such cases, an output indicating that the problem does not have a solution is all we can and should expect from an algorithm. Nor are we talking about ambiguously stated problems. Even some unambiguous problems that must have a simple yes or no answer are "undecidable," i.e., unsolvable by any algorithm. An important example of such a problem appears in Section 11.3. Fortunately, a vast majority of problems in practical computing *can* be solved by an algorithm.

Before leaving this section, let us be sure that you do not have the misconception—possibly caused by the somewhat mechanical nature of the diagram of Figure 1.2—that designing an algorithm is a dull activity. There is nothing further from the truth: inventing (or discovering?) algorithms is a very creative and rewarding process. This book is designed to convince you that this is the case.

Exercises 1.2

1. *Old World puzzle* A peasant finds himself on a riverbank with a wolf, a goat, and a head of cabbage. He needs to transport all three to the other side of the river in his boat. However, the boat has room for only the peasant himself and one other item (either the wolf, the goat, or the cabbage). In his absence, the wolf would eat the goat, and the goat would eat the cabbage. Solve this problem for the peasant or prove it has no solution. (Note: The peasant is a vegetarian but does not like cabbage and hence can eat neither the goat nor the cabbage to help him solve the problem. And it goes without saying that the wolf is a protected species.)

2. *New World puzzle* There are four people who want to cross a bridge; they all begin on the same side. You have 17 minutes to get them all across to the other side. It is night, and they have one flashlight. A maximum of two people can cross the bridge at one time. Any party that crosses, either one or two people, must have the flashlight with them. The flashlight must be walked back and forth; it cannot be thrown, for example. Person 1 takes 1 minute to cross the bridge, person 2 takes 2 minutes, person 3 takes 5 minutes, and person 4 takes 10 minutes. A pair must walk together at the rate of the slower person's pace. For example, if person 1 and person 4 walk across first, 10 minutes have elapsed when they get to the other side of the bridge. If person 4 returns the flashlight, a total of 20 minutes have passed and you have failed the mission. (Note: According to a rumor on the Internet, interviewers at a well-known software company located near Seattle have given this problem to interviewees.)

3. Which of the following formulas can be considered an algorithm for computing the area of a triangle whose side lengths are given positive numbers a, b, and c?

https://hemanthrajhemu.github.io

a. $S = \sqrt{p(p-a)(p-b)(p-c)}$, where $p = (a+b+c)/2$

b. $S = \frac{1}{2}bc \sin A$, where A is the angle between sides b and c

c. $S = \frac{1}{2}ah_a$, where h_a is the height to base a

4. Write a pseudocode for an algorithm for finding real roots of equation $ax^2 + bx + c = 0$ for arbitrary real coefficients a, b, and c. (You may assume the availability of the square root function $sqrt(x)$.)

5. Describe the standard algorithm for finding the binary representation of a positive decimal integer

a. in English.

b. in a pseudocode.

6. Describe the algorithm used by your favorite ATM machine in dispensing cash. (You may give your description in either English or a pseudocode, whichever you find more convenient.)

7. a. Can the problem of computing the number π be solved exactly?

b. How many instances does this problem have?

c. Look up an algorithm for this problem on the World Wide Web.

8. Give an example of a problem other than computing the greatest common divisor for which you know more than one algorithm. Which of them is simpler? Which is more efficient?

9. Consider the following algorithm for finding the distance between the two closest elements in an array of numbers.

```
ALGORITHM  MinDistance(A[0..n − 1])
    //Input: Array A[0..n − 1] of numbers
    //Output: Minimum distance between two of its elements
    dmin ← ∞
    for i ← 0 to n − 1 do
        for j ← 0 to n − 1 do
            if i ≠ j and |A[i] − A[j]| < dmin
                dmin ← |A[i] − A[j]|
    return dmin
```

Make as many improvements as you can in this algorithmic solution to the problem. (If you need to, you may change the algorithm altogether; if not, improve the implementation given.)

10. One of the most influential books on problem solving, titled *How to Solve It* [Pol57], was written by the Hungarian-American mathematician George Polya (1887–1985). Polya summarized his ideas in a four-point summary. Find

https://hemanthrajhemu.github.io

this summary on the Web or, better yet, in his book, and compare it with the plan outlined in Section 1.2. What do they have in common? How are they different?

1.3 Important Problem Types

In the limitless sea of problems one encounters in computing, there are a few areas that have attracted particular attention from researchers. By and large, interest has been driven either by the problem's practical importance or by some specific characteristics making the problem an interesting research subject; fortunately, these two motivating forces reinforce each other in most cases.

In this section, we take up the most important problem types:

- Sorting
- Searching
- String processing
- Graph problems
- Combinatorial problems
- Geometric problems
- Numerical problems

These problems are used in subsequent chapters of the book to illustrate different algorithm design techniques and methods of algorithm analysis.

Sorting

The ***sorting problem*** asks us to rearrange the items of a given list in ascending order. Of course, for this problem to be meaningful, the nature of the list items must allow such an ordering. (Mathematicians would say that there must exist a relation of total ordering.) As a practical matter, we usually need to sort lists of numbers, characters from an alphabet, character strings, and, most important, records similar to those maintained by schools about their students, libraries about their holdings, and companies about their employees. In the case of records, we need to choose a piece of information to guide sorting. For example, we can choose to sort student records in alphabetical order of names or by student number or by student grade point average. Such a specially chosen piece of information is called a ***key***. Computer scientists often talk about sorting a list of keys even when the list's items are not records but, say, just integers.

Why would we want a sorted list? Well, sorting makes many questions about the list easier to answer. The most important of them is searching: it is why dictionaries, telephone books, class lists, and so on are sorted. You will see other

https://hemanthrajhemu.github.io

examples of the usefulness of list presorting in Section 6.1. In a similar vein, sorting is used as an auxiliary step in several important algorithms in other areas, e.g., geometric algorithms.

By now, computer scientists have discovered dozens of different sorting algorithms. In fact, inventing a new sorting algorithm has been likened to designing the proverbial mousetrap. And I am happy to report that the hunt for a better sorting mousetrap continues. This perseverance is admirable in view of the following facts. On the one hand, there are a few good sorting algorithms that sort an arbitrary array of size n using about $n \log_2 n$ comparisons. On the other hand, no algorithm that sorts by key comparisons (as opposed to, say, comparing small pieces of keys) can do substantially better than that.

There is a reason for this embarrassment of algorithmic riches in the land of sorting. Although some algorithms are indeed better than others, there is no algorithm that would be the best solution in all situations. Some of the algorithms are simple but relatively slow while others are faster but more complex; some work better on randomly ordered inputs while others do better on almost sorted lists; some are suitable only for lists residing in the fast memory while others can be adapted for sorting large files stored on a disk; and so on.

Two properties of sorting algorithms deserve special mention. A sorting algorithm is called ***stable*** if it preserves the relative order of any two equal elements in its input. In other words, if an input list contains two equal elements in positions i and j where $i < j$, then in the sorted list they have to be in positions i' and j', respectively, such that $i' < j'$. This property can be desirable if, for example, we have a list of students sorted alphabetically and we want to sort it according to student GPA: a stable algorithm will yield a list in which students with the same GPA will still be sorted alphabetically. Generally speaking, algorithms that can exchange keys located far apart are not stable but they usually work faster; you will see how this general comment applies to important sorting algorithms later in the book.

The second notable feature of a sorting algorithm is the amount of extra memory the algorithm requires. An algorithm is said to be ***in place*** if it does not require extra memory, except, possibly, for a few memory units. There are important sorting algorithms that are in place and those that are not.

Searching

The ***searching problem*** deals with finding a given value, called a ***search key***, in a given set (or a multiset, which permits several elements to have the same value). There are plenty of searching algorithms to choose from. They range from the straightforward sequential search to a spectacularly efficient but limited binary search and algorithms based on representing the underlying set in a different form more conducive to searching. The latter algorithms are of particular importance for real-life applications because they are indispensable for storing and retrieving information from large databases.

https://hemanthrajhemu.github.io

For searching, too, there is no single algorithm that fits all situations best. Some algorithms work faster than others but require more memory; some are very fast but applicable only to sorted arrays; and so on. Unlike with sorting algorithms, there is no stability problem, but different issues arise. Specifically, in applications where the underlying data may change frequently relative to the number of searches, searching has to be considered in conjunction with two other operations: addition to and deletion from the data set of an item. In such situations, data structures and algorithms should be chosen to strike a balance among the requirements of each operation. Also, organizing very large data sets for efficient searching poses special challenges with important implications for real-life applications.

String Processing

In recent years, the rapid proliferation of applications dealing with nonnumerical data has intensified the interest of researchers and computing practitioners in string-handling algorithms. A ***string*** is a sequence of characters from an alphabet. Strings of particular interest are text strings, which comprise letters, numbers, and special characters; bit strings, which comprise zeros and ones; and gene sequences, which can be modeled by strings of characters from the four-character alphabet {A, C, G, T}. It should be pointed out, however, that string-processing algorithms have been important for computer science for a long time in conjunction with computer languages and compiling issues.

One particular problem—that of searching for a given word in a text—has attracted special attention from researchers. They call it ***string matching***. Several algorithms that exploit the special nature of this type of searching have been invented. We introduce one very simple algorithm in Chapter 3, and discuss two algorithms based on a remarkable idea by R. Boyer and J. Moore in Chapter 7.

Graph Problems

One of the oldest and most interesting areas in algorithmics is graph algorithms. Informally, a ***graph*** can be thought of as a collection of points called vertices, some of which are connected by line segments called edges. (A more formal definition is given in the next section.) Graphs are an interesting subject to study for both theoretical and practical reasons. Graphs can be used for modeling a wide variety of real-life applications, including transportation and communication networks, project scheduling, and games. One interesting recent application is an estimation of the Web's diameter, which is the maximum number of links one needs to follow to reach one Web page from another by the most direct route between them.[2]

2. This number, according to an estimate by a group of researchers at the University of Notre Dame [Alb99], is just 19.

https://hemanthrajhemu.github.io

Basic graph algorithms include graph traversal algorithms (How can one visit all the points in a network?), shortest-path algorithms (What is the best route between two cities?), and topological sorting for graphs with directed edges (Is a set of courses with their prerequisites consistent or self-contradictory?). Fortunately, these algorithms can be considered illustrations of general design techniques; accordingly, you will find them in corresponding chapters of the book.

Some graph problems are computationally very hard; the most well-known examples are the traveling salesman problem and the graph-coloring problem. The ***traveling salesman problem (TSP)*** is the problem of finding the shortest tour through n cities that visits every city exactly once. In addition to obvious applications involving route planning, it arises in such modern applications as circuit board and VLSI chip fabrication, X-ray chrystallography, and genetic engineering. The ***graph-coloring problem*** asks us to assign the smallest number of colors to vertices of a graph so that no two adjacent vertices are the same color. This problem arises in several applications, such as event scheduling: if the events are represented by vertices that are connected by an edge if and only if the corresponding events cannot be scheduled in the same time, a solution to the graph-coloring problem yields an optimal schedule.

Combinatorial Problems

From a more abstract perspective, the traveling salesman problem and the graph-coloring problem are examples of ***combinatorial problems***. These are problems that ask (explicitly or implicitly) to find a combinatorial object—such as a permutation, a combination, or a subset—that satisfies certain constraints and has some desired property (e.g., maximizes a value or minimizes a cost).

Generally speaking, combinatorial problems are the most difficult problems in computing, from both the theoretical and practical standpoints. Their difficulty stems from the following facts. First, the number of combinatorial objects typically grows extremely fast with a problem's size, reaching unimaginable magnitudes even for moderate-sized instances. Second, there are no known algorithms for solving most such problems exactly in an acceptable amount of time. Moreover, most computer scientists believe that such algorithms do not exist. This conjecture has been neither proved nor disapproved, and it remains the most important unresolved issue in theoretical computer science. We discuss this topic in more detail in Section 11.3.

Some combinatorial problems can be solved by efficient algorithms, but they should be considered fortunate exceptions to the rule. The shortest-path problem mentioned earlier is among such exceptions.

Geometric Problems

Geometric algorithms deal with geometric objects such as points, lines, and polygons. Ancient Greeks were very much interested in developing procedures (they did not call them algorithms, of course) for solving a variety of geometric problems,

https://hemanthrajhemu.github.io

including problems of constructing simple geometric shapes—triangles, circles, and so on—with an unmarked ruler and a compass. Then, for about 2000 years, intense interest in geometric algorithms disappeared, to be resurrected in the age of computers—no more rulers and compasses, just bits, bytes, and good old human ingenuity. Of course, today people are interested in geometric algorithms with quite different applications in mind, such as computer graphics, robotics, and tomography.

We will discuss algorithms for only two classic problems of computational geometry: the closest-pair problem and the convex-hull problem. The ***closest-pair problem*** is self-explanatory: given n points in the plane, find the closest pair among them. The ***convex-hull problem*** asks to find the smallest convex polygon that would include all the points of a given set. If you are interested in other geometric algorithms, you will find a wealth of material in specialized monographs (e.g., [ORo98]), or corresponding chapters of textbooks organized around problem types (e.g., [Sed88]).

Numerical Problems

Numerical problems, another large special area of applications, are problems that involve mathematical objects of continuous nature: solving equations and systems of equations, computing definite integrals, evaluating functions, and so on. The majority of such mathematical problems can be solved only approximately. Another principal difficulty stems from the fact that such problems typically require manipulating real numbers, which can be represented in a computer only approximately. Moreover, a large number of arithmetic operations performed on approximately represented numbers can lead to an accumulation of the round-off error to a point where it can drastically distort an output produced by a seemingly sound algorithm.

Many sophisticated algorithms have been developed over the years in this area, and they continue to play a critical role in many scientific and engineering applications. But in the last 25 years or so, the computing industry has shifted its focus to business applications. These new applications require primarily algorithms for information storage, retrieval, transportation through networks, and presentation to users. As a result of this revolutionary change, numerical analysis has lost its formerly dominating position in both industry and computer science programs. Still, it is important for any computer-literate person to have at least a rudimentary idea about numerical algorithms. We discuss several classical numerical algorithms in Sections 6.2, 11.4, and 12.4.

Exercises 1.3

1. Consider the algorithm for the sorting problem that sorts an array by counting, for each of its elements, the number of smaller elements and then uses this information to put the element in its appropriate position in the sorted array:

https://hemanthrajhemu.github.io

```
ALGORITHM   ComparisonCountingSort(A[0..n − 1])
    //Sorts an array by comparison counting
    //Input: Array A[0..n − 1] of orderable values
    //Output: Array S[0..n − 1] of A's elements sorted in nondecreasing order
    for i ← 0 to n − 1 do
        Count[i] ← 0
    for i ← 0 to n − 2 do
        for j ← i + 1 to n − 1 do
            if A[i] < A[j]
                Count[j] ← Count[j] + 1
            else Count[i] ← Count[i] + 1
    for i ← 0 to n − 1 do
        S[Count[i]] ← A[i]
    return S
```

a. Apply this algorithm to sorting the list 60, 35, 81, 98, 14, 47.

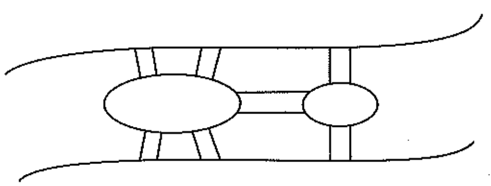
b. Is this algorithm stable?

c. Is it in place?

2. Name the algorithms for the searching problem that you already know. Give a good succinct description of each algorithm in English. (If you know no such algorithms, use this opportunity to design one.)

3. Design a simple algorithm for the string-matching problem.

4. *Königsberg bridges* The Königsberg bridge puzzle is universally accepted as the problem that gave birth to graph theory. It was solved by the great Swiss-born mathematician Leonhard Euler (1707–1783). The problem asked whether one could, in a single stroll, cross all seven bridges of the city of Königsberg exactly once and return to a starting point. Following is a sketch of the river with its two islands and seven bridges:

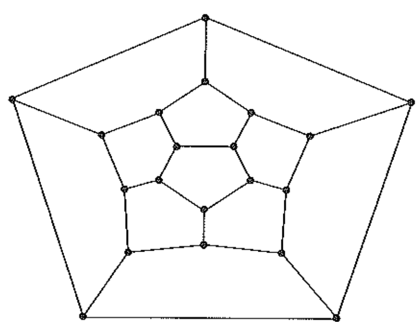


a. State the problem as a graph problem.

b. Does this problem have a solution? If you believe it does, draw such a stroll; if you believe it does not, explain why and indicate the smallest number of new bridges that would be required to make such a stroll possible.

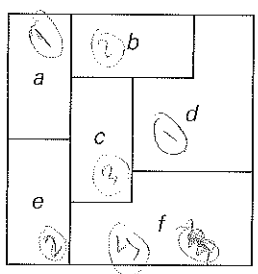
https://hemanthrajhemu.github.io

5. *Icosian Game* A century after Euler's discovery (see Problem 4), another famous puzzle—this one invented by the renown Irish mathematician Sir William Hamilton (1805–1865)—was presented to the world under the name of the Icosian Game. The game was played on a circular wooden board on which the following graph was carved:



Find a ***Hamiltonian circuit***—a path that visits all the graph's vertices exactly once before returning to the starting vertex—for this graph.

6. Consider the following problem: Design an algorithm to determine the best route for a subway passenger to take from one designated station to another in a well-developed subway system similar to those in such cities as Washington, D.C., and London, UK.
   a. The problem's statement is somewhat vague, which is typical of real-life problems. In particular, what reasonable criterion can be used for defining the "best" route?
   b. How would you model this problem by a graph?
7. **a.** Rephrase the traveling salesman problem in combinatorial object terms.
   b. Rephrase the graph-coloring problem in combinatorial object terms.
8. Consider the following map:



https://hemanthrajhemu.github.io

  a. Explain how we can use the graph-coloring problem to color the map so that no two neighboring regions are colored the same.

  b. Use your answer to part (a) to color the map with the smallest number of colors.

9. Design an algorithm for the following problem: Given a set of n points in the Cartesian plane, determine whether all of them lie on the same circumference.

10. Write a program that reads as its inputs the (x, y) coordinates of the endpoints of two line segments P_1Q_1 and P_2Q_2 and determines whether the segments have a common point.

1.4 Fundamental Data Structures

Since the vast majority of algorithms of interest operate on data, particular ways of organizing data play a critical role in the design and analysis of algorithms. A ***data structure*** can be defined as a particular scheme of organizing related data items. The nature of the data items is dictated by a problem at hand; they can range from elementary data types (e.g., integers or characters) to data structures (e.g., a one-dimensional array of one-dimensional arrays is often used for implementing matrices). There are a few data structures that have proved to be particularly important for computer algorithms. Since you are undoubtedly familiar with most if not all of them, just a quick review is provided here.

Linear Data Structures

The two most important elementary data structures are the array and the linked list. A (one-dimensional) ***array*** is a sequence of n items of the same data type that are stored contiguously in computer memory and made accessible by specifying a value of the array's ***index*** (Figure 1.3).

In the majority of cases, the index is an integer either between 0 and $n - 1$ (as shown in Figure 1.3) or between 1 and n. Some computer languages allow an array index to range between any two integer bounds low and $high$, and some even permit nonnumerical indices to specify, for example, data items corresponding to the 12 months of the year by the month names.

Item [0]	*Item* [1]	. . .	*Item* [n–1]

FIGURE 1.3 Array of n elements

https://hemanthrajhemu.github.io

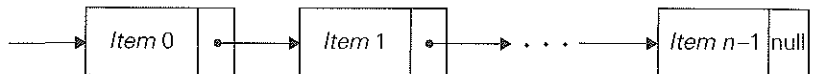


FIGURE 1.4 Singly linked list of n elements

Each and every element of an array can be accessed in the same constant amount of time regardless of where in the array the element in question is located. This feature positively distinguishes arrays from linked lists (see below). It is also assumed that every element of an array occupies the same amount of computer storage.

Arrays are used for implementing a variety of other data structures. Prominent among them is the ***string***, a sequence of characters from an alphabet terminated by a special character indicating the string's end. Strings composed of zeros and ones are called ***binary strings*** or ***bit strings***. Strings are indispensable for processing textual data, defining computer languages and compiling programs written in them, and studying abstract computational models. Operations we usually perform on strings differ from those we typically perform on other arrays (say, arrays of numbers). They include computing the string length, comparing two strings to determine which one precedes the other according to the so-called lexicographic order, i.e., in a dictionary, and concatenating two strings (forming one string from two given strings by appending the second to the end of the first).

A ***linked list*** is a sequence of zero or more elements called ***nodes*** each containing two kinds of information: some data and one or more links called ***pointers*** to other nodes of the linked list. (A special pointer called "null" is used to indicate the absence of a node's successor.) In a ***singly linked list***, each node except the last one contains a single pointer to the next element (Figure 1.4).

To access a particular node of a linked list, we start with the list's first node and traverse the pointer chain until the particular node is reached. Thus, the time needed to access an element of a singly linked list, unlike that of an array, depends on where in the list the element is located. On the positive side, linked lists do not require any preliminary reservation of the computer memory, and insertions and deletions can be made quite efficiently in a linked list by reconnecting a few appropriate pointers.

We can exploit flexibility of the linked list structure in a variety of ways. For example, it is often convenient to start a linked list with a special node called the ***header***. This node often contains information about the linked list such as its current length; it may also contain, in addition to a pointer to the first element, a pointer to the linked list's last element.

Another extension is the structure called the ***doubly linked list***, in which every node, except the first and the last, contains pointers to both its successor and its predecessor (Figure 1.5).

The array and linked list are two principal choices in representing a more abstract data structure called a linear list or simply a list. A ***list*** is a finite sequence

https://hemanthrajhemu.github.io

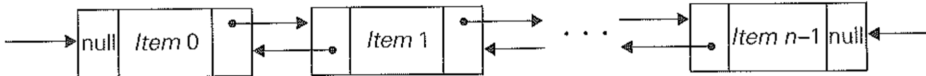


FIGURE 1.5 Doubly linked list of n elements

of data items, i.e., a collection of data items arranged in a certain linear order. The basic operations performed on this data structure are searching for, inserting, and deleting an element.

Two special types of lists, stacks and queues, are particularly important. A ***stack*** is a list in which insertions and deletions can be done only at the end. This end is called the ***top*** because a stack is usually visualized not horizontally but vertically (akin to a stack of plates whose "operations" it mimics very closely). As a result, when elements are added to (pushed onto) a stack and deleted from (popped off) it, the structure operates in the "last-in–first-out" (LIFO) fashion, exactly as the stack of plates does if we can remove only the top plate or add another plate to top of the stack. Stacks have a multitude of applications; in particular, they are indispensable for implementing recursive algorithms.

A ***queue***, on the other hand, is a list from which elements are deleted from one end of the structure, called the ***front*** (this operation is called ***dequeue***), and new elements are added to the other end, called the ***rear*** (this operation is called ***enqueue***). Consequently, a queue operates in the "first-in–first-out" (FIFO) fashion (akin, say, to a queue of customers served by a single teller in a bank). Queues also have many important applications, including several algorithms for graph problems.

Many important applications require selection of an item of the highest priority among a dynamically changing set of candidates. A data structure that seeks to satisfy the needs of such applications is called a priority queue. A ***priority queue*** is a collection of data items from a totally ordered universe (most often, integer or real numbers). The principal operations on a priority queue are finding its largest element, deleting its largest element, and adding a new element. Of course, a priority queue must be implemented so that the last two operations yield another priority queue. Straightforward implementations of this data structure can be based on either an array or a sorted array, but neither of these options yields the most efficient solution possible. A better implementation of a priority queue is based on an ingenious data structure called the ***heap***. We discuss heaps (and an important sorting algorithm based on them) in Section 6.4.

Graphs

As mentioned in the previous section, a graph is informally thought of as a collection of points in the plane called "vertices" or "nodes," some of them connected by line segments called "edges" or "arcs." Formally, a ***graph*** $G = \langle V, E \rangle$ is defined by a pair of two sets: a finite set V of items called ***vertices*** and a set E of pairs

https://hemanthrajhemu.github.io

of these items called ***edges***. If these pairs of vertices are unordered, i.e., a pair of vertices (u, v) is the same as the pair (v, u), we say that the vertices u and v are ***adjacent*** to each other and that they are connected by the ***undirected edge*** (u, v). We call the vertices u and v ***endpoints*** of the edge (u, v) and say that u and v are ***incident*** to this edge; we also say that the edge (u, v) is incedent to its endpoints u and v. A graph G is called ***undirected*** if every edge in it is undirected.

If a pair of vertices (u, v) is not the same as the pair (v, u), we say that the edge (u, v) is ***directed*** from the vertex u, called the edge's ***tail***, to the vertex v, called the edge's ***head***. We also say that the edge (u, v) leaves u and enters v. A graph whose every edge is directed is called ***directed***. Directed graphs are also called ***digraphs***.

It is normally convenient to label vertices of a graph or a digraph with letters, integer numbers, or, if an application calls for it, character strings (Figure 1.6). The graph in Figure 1.6a has six vertices and seven edges:

$$V = \{a, b, c, d, e, f\}, \quad E = \{(a, c), (a, d), (b, c), (b, f), (c, e), (d, e), (e, f)\}.$$

The digraph in Figure 1.6b has six vertices and eight directed edges:

$$V = \{a, b, c, d, e, f\},$$
$$E = \{(a, c), (b, c), (b, f), (c, e), (d, a), (d, e), (e, c), (e, f)\}.$$

Our definition of a graph does not forbid ***loops***, or edges connecting vertices to themselves. Unless explicitly stated otherwise, we will consider graphs without loops. Since our definition disallows multiple edges between the same vertices of an undirected graph, we have the following inequality for the number of edges $|E|$ possible in an undirected graph with $|V|$ vertices and no loops:

$$0 \le |E| \le |V|(|V| - 1)/2.$$

(We get the largest number of edges in a graph if there is an edge connecting each of its $|V|$ vertices with all $|V| - 1$ other vertices. We have to divide product $|V|(|V| - 1)$ by 2, however, because it includes every edge twice.)

A graph with every pair of its vertices connected by an edge is called ***complete***. A standard notation for the complete graph with $|V|$ vertices is $K_{|V|}$. A graph with relatively few possible edges missing is called ***dense***; a graph with few edges

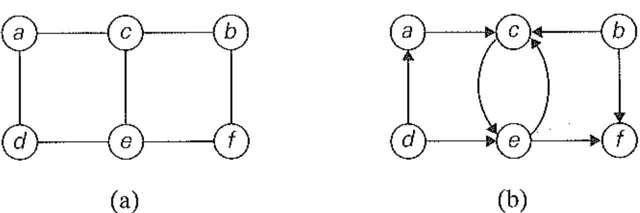


FIGURE 1.6 (a) Undirected graph. (b) Digraph.

https://hemanthrajhemu.github.io

	a	b	c	d	e	f
a	0	0	1	1	0	0
b	0	0	1	0	0	1
c	1	1	0	0	1	0
d	1	0	0	0	1	0
e	0	0	1	1	0	1
f	0	1	0	0	1	0

(a)

a	→	c	→	d		
b	→	c	→	f		
c	→	a	→	b	→	e
d	→	a	→	e		
e	→	c	→	d	→	f
f	→	b	→	e		

(b)

FIGURE 1.7 (a) Adjacency matrix and (b) adjacency lists of the graph in Figure 1.6a

relative to the number of its vertices is called ***sparse***. Whether we are dealing with a dense or sparse graph may influence how we choose to represent the graph and, consequently, the running time of an algorithm being designed or used.

Graph representations Graphs for computer algorithms can be represented in two principal ways: the adjacency matrix and adjacency lists. The ***adjacency matrix*** of a graph with n vertices is an n-by-n boolean matrix with one row and one column for each of the graph's vertices, in which the element in the ith row and the jth column is equal to 1 if there is an edge from the ith vertex to the jth vertex, and equal to 0 if there is no such edge. For example, the adjacency matrix for the graph in Figure 1.6a is given in Figure 1.7a. Note that the adjacency matrix of an undirected graph is always symmetric, i.e., $A[i, j] = A[j, i]$ for every $0 \le i, j \le n - 1$ (why?).

The ***adjacency lists*** of a graph or a digraph is a collection of linked lists, one for each vertex, that contain all the vertices adjacent to the list's vertex (i.e., all the vertices connected to it by an edge). Usually, such lists start with a header identifying a vertex for which the list is compiled. For example, Figure 1.7b represents the graph in Figure 1.6a via its adjacency lists. To put it another way, adjacency lists indicate columns of the adjacency matrix that, for a given vertex, contain 1's.

If a graph is sparse, the adjacency list representation may use less space than the corresponding adjacency matrix despite the extra storage consumed by pointers of the linked lists; the situation is exactly opposite for dense graphs. In general, which of the two representations is more convenient depends on the nature of the problem, on the algorithm used for solving it, and, possibly, on the type of input graph (sparse or dense).

Weighted graphs A ***weighted graph*** (or weighted digraph) is a graph (or digraph) with numbers assigned to its edges. These numbers are called ***weights*** or ***costs***. An interest in such graphs is motivated by numerous real-life applications, such as finding the shortest path between two points in a transportation or communication network or the traveling salesman problem mentioned earlier.

https://hemanthrajhemu.github.io

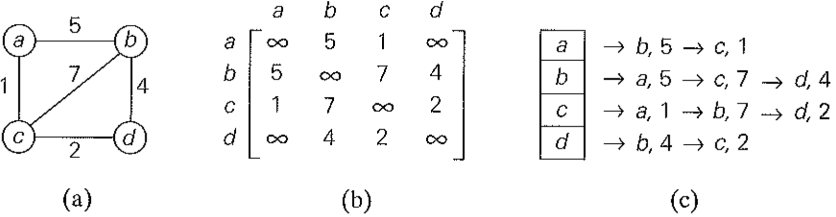


FIGURE 1.8 (a) Weighted graph. (b) Its weight matrix. (c) Its adjacency lists.

Both principal representations of a graph can be easily adopted to accommodate weighted graphs. If a weighted graph is represented by its adjacency matrix, then its element $A[i, j]$ will simply contain the weight of the edge from the ith to the jth vertex if there is such an edge and a special symbol, e.g., ∞, if there is no such edge. Such a matrix is called the ***weight matrix*** or ***cost matrix***. This approach is illustrated in Figure 1.8b. (For some applications, it is more convenient to put 0's on the main diagonal of the adjacency matrix.) Adjacency lists for a weighted graph have to include in their nodes not only the name of an adjacent vertex but also the weight of the corresponding edge (Figure 1.8c).

Paths and cycles Among many interesting properties of graphs, two are important for a great number of applications: ***connectivity*** and ***acyclicity***. Both are based on the notion of a path. A ***path*** from vertex u to vertex v of a graph G can be defined as a sequence of adjacent (connected by an edge) vertices that starts with u and ends with v. If all vertices of a path are distinct, the path is said to be ***simple***. The ***length*** of a path is the total number of vertices in a vertex sequence defining the path minus one, which is the same as the number of edges in the path. For example, a, c, b, f is a simple path of length 3 from a to f in the graph of Figure 1.6a, whereas a, c, e, c, b, f is a path (not simple) of length 5 from a to f.

In the case of a directed graph, we are usually interested in directed paths. A ***directed path*** is a sequence of vertices in which every consecutive pair of the vertices is connected by an edge directed from the vertex listed first to the vertex listed next. For example, a, c, e, f is a directed path from a to f in the graph of Figure 1.6b.

A graph is said to be ***connected*** if for every pair of its vertices u and v there is a path from u to v. Informally, this property means that if we make a model of a connected graph by connecting some balls representing the graph's vertices with strings representing the edges, it will be a single piece. If a graph is not connected, such a model will consist of several connected pieces that are called connected components of the graph. Formally, a ***connected component*** is a maximal (not

https://hemanthrajhemu.github.io

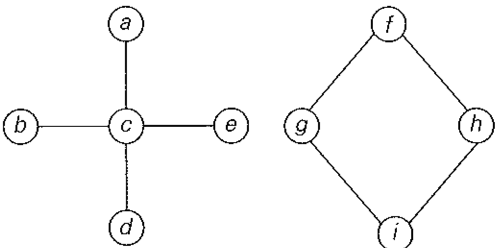


FIGURE 1.9 Graph that is not connected

expandable via an inclusion of an extra vertex) connected subgraph[3] of a given graph. For example, the graphs of Figures 1.6a and 1.8a are connected, while the graph in Figure 1.9 is not because there is no path, for example, from a to f. The graph in Figure 1.9 has two connected components with vertices $\{a, b, c, d, e\}$ and $\{f, g, h, i\}$, respectively.

Graphs with several connected components do happen in real-life applications. A graph representing the Interstate highway system of the United States would be an example (why?).

It is important to know for many applications whether or not a graph under consideration has cycles. A ***cycle*** is a path of a positive length that starts and ends at the same vertex and does not traverse the same edge more than once. For example, f, h, i, g, f is a cycle in the graph of Figure 1.9. A graph with no cycles is said to be ***acyclic***. We discuss acyclic graphs in the next subsection.

Trees

A ***tree*** (more accurately, a ***free tree***) is a connected acyclic graph (Figure 1.10a). A graph that has no cycles but is not necessarily connected is called a ***forest***: each of its connected components is a tree (Figure 1.10b).

Trees have several important properties other graphs do not have. In particular, the number of edges in a tree is always one less than the number of its vertices:

$$|E| = |V| - 1.$$

As the graph of Figure 1.9 demonstrates, this property is necessary but not sufficient for a graph to be a tree. However, for connected graphs it is sufficient and hence provides a convenient way of checking whether a connected graph has a cycle.

3. A ***subgraph*** of a given graph $G = \langle V, E\rangle$ is a graph $G' = \langle V', E'\rangle$ such that $V' \subseteq V$ and $E' \subseteq E$.

https://hemanthrajhemu.github.io

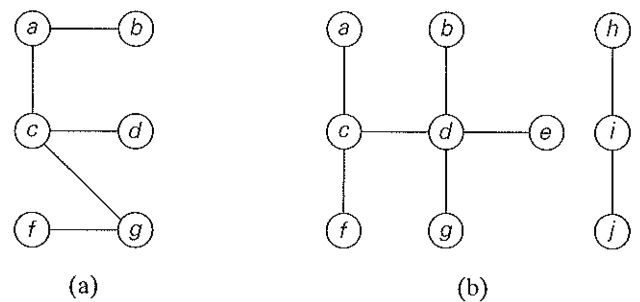


FIGURE 1.10 (a) Tree. (b) Forest.

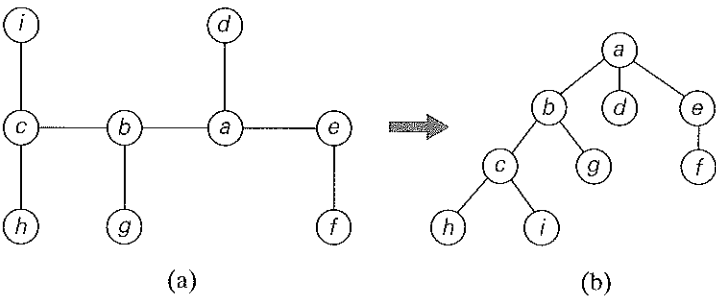


FIGURE 1.11 (a) Free tree. (b) Its transformation into a rooted tree.

Rooted trees Another very important property of trees is the fact that for every two vertices in a tree, there always exists exactly one simple path from one of these vertices to the other. This property makes it possible to select an arbitrary vertex in a free tree and consider it as the ***root*** of the so-called ***rooted tree.*** A rooted tree is usually depicted by placing its root on the top (level 0 of the tree), the vertices adjacent to the root below it (level 1), the vertices two edges apart from the root below that (level 2), and so on. Figure 1.11 presents such a transformation from a free tree to a rooted tree.

Rooted trees play a very important role in computer science, a much more important one than free trees do; in fact, for the sake of brevity, they are often referred to as simply "trees." Obvious applications of trees are for describing hierarchies, from file directories to organizational charts of enterprises. There are many less obvious applications, such as implementing dictionaries (see below), efficient storage of very large data sets (Section 7.4), and data encoding (Section 9.4). As we discuss in Chapter 2, trees also are helpful in analysis of recursive algorithms. To finish this far-from-complete list of tree applications, we should

https://hemanthrajhemu.github.io

mention the so-called ***state-space trees*** that underline two important algorithm design techniques: backtracking and branch-and-bound (Sections 12.1 and 12.2).

For any vertex v in a tree T, all the vertices on the simple path from the root to that vertex are called ***ancestors*** of v. The vertex itself is usually considered its own ancestor; the set of ancestors that excludes the vertex itself is referred to as ***proper ancestors***. If (u, v) is the last edge of the simple path from the root to vertex v (and $u \neq v$), u is said to be the ***parent*** of v and v is called a ***child*** of u; vertices that have the same parent are said to be ***siblings***. A vertex with no children is called a ***leaf***; a vertex with at least one child is called ***parental***. All the vertices for which a vertex v is an ancestor are said to be ***descendants*** of v; the ***proper descendants*** exclude the vertex v itself. All the descendants of a vertex v with all the edges connecting them form the ***subtree*** of T rooted at that vertex. Thus, for the tree of Figure 1.11b, the root of the tree is a; vertices d, g, f, h, and i are leaves, while vertices a, b, e, and c are parental; the parent of b is a; the children of b are c and g; the siblings of b are d and e; the vertices of the subtree rooted at b are $\{b, c, g, h, i\}$.

The ***depth*** of a vertex v is the length of the simple path from the root to v. The ***height*** of a tree is the length of the longest simple path from the root to a leaf. For example, the depth of vertex c in the tree in Figure 1.11b is 2, and the height of the tree is 3. Thus, if we count tree levels top down starting with 0 for the root's level, the depth of a vertex is simply its level in the tree, and the tree's height is the maximum level of its vertices. (You should be alert to the fact that some authors define the height of a tree as the number of levels in it; this makes the height of a tree larger by 1 than the height defined as the length of the longest simple path from the root to a leaf.)

Ordered trees An ***ordered tree*** is a rooted tree in which all the children of each vertex are ordered. It is convenient to assume that in a tree's diagram, all the children are ordered left to right. A ***binary tree*** can be defined as an ordered tree in which every vertex has no more than two children and each child is designated as either a ***left child*** or a ***right child*** of its parent. The subtree with its root at the left (right) child of a vertex is called the ***left (right) subtree*** of that vertex. An example of a binary tree is given in Figure 1.12a.

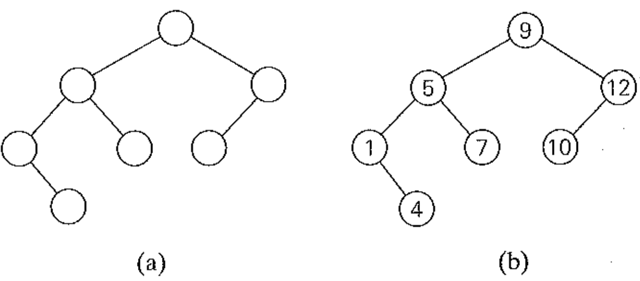


FIGURE 1.12 (a) Binary tree. (b) Binary search tree.

https://hemanthrajhemu.github.io

In Figure 1.12b, some numbers are assigned to vertices of the binary tree in Figure 1.12a. Note that a number assigned to each parental vertex is larger than all the numbers in its left subtree and smaller than all the numbers in its right subtree. Such trees are called ***binary search trees***. Binary trees and binary search trees have a wide variety of applications in computer science; you will encounter some of them throughout the book. In particular, binary search trees can be generalized to more general kinds of search trees called ***multiway search trees***, which are indispensable for efficient storage of very large files on disks.

As you will see later in the book, the efficiency of most important algorithms for binary search trees and their extensions depends on the tree's height. Therefore, the following inequalities for the height h of a binary tree with n nodes are especially important for analysis of such algorithms:

$$\lfloor \log_2 n \rfloor \le h \le n - 1.$$

A binary tree is usually implemented for computing purposes by a collection of nodes corresponding to vertices of the tree. Each node contains some information associated with the vertex (its name or some value assigned to it) and two pointers to the nodes representing the left child and right child of the vertex, respectively. Figure 1.13 illustrates such an implementation for the binary search tree in Figure 1.12b.

A computer representation of an arbitrary ordered tree can be done by simply providing a parental vertex with the number of pointers equal to the number of its children. This representation may prove to be inconvenient if the number of children varies widely among the nodes. We can avoid this inconvenience by using nodes with just two pointers, as we did for binary trees. Here, however, the left pointer will point to the first child of the vertex, while the right pointer will point

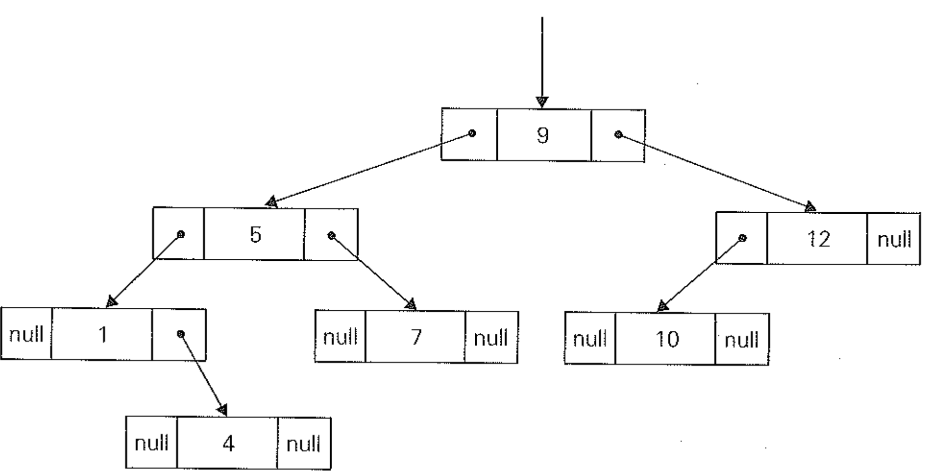


FIGURE 1.13 Standard implementation of the binary search tree in Figure 1.12b

https://hemanthrajhemu.github.io

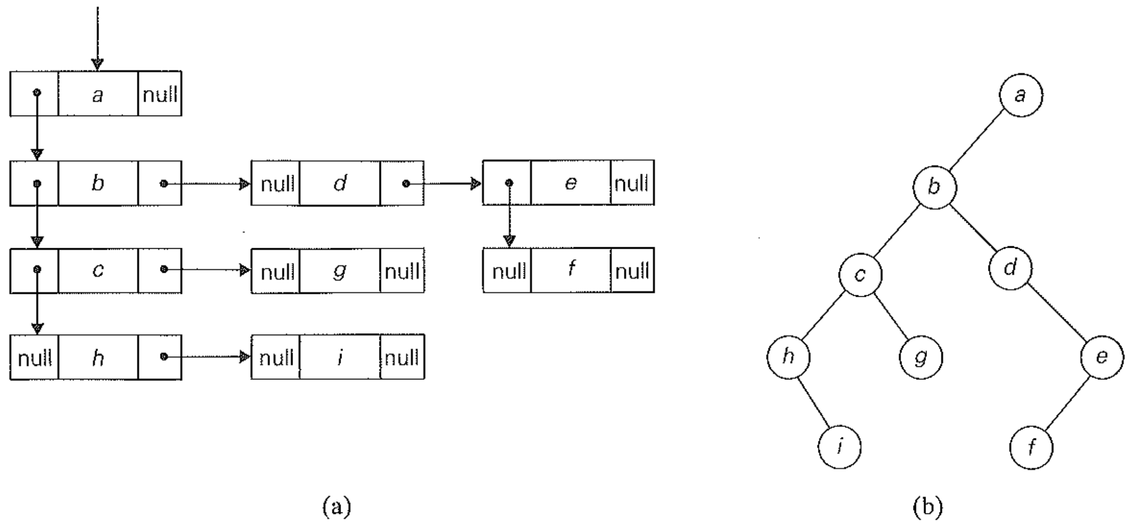


FIGURE 1.14 (a) First child–next sibling representation of the graph in Figure 1.11b. (b) Its binary tree representation.

to its next sibling. Accordingly, this representation is called the ***first child–next sibling representation***. Thus, all the siblings of a vertex are linked (via the nodes' right pointers) in a singly linked list, with the first element of the list pointed to by the left pointer of their parent. Figure 1.14a illustrates this representation for the tree in Figure 1.11b. It is not difficult to see that this representation effectively transforms an ordered tree into a binary tree said to be associated with the ordered tree. We get this representation by "rotating" the pointers about 45 degrees clockwise (see Figure 1.14b).

Sets and Dictionaries

The notion of a set plays a central role in mathematics. A ***set*** can be described as an unordered collection (possibly empty) of distinct items called ***elements*** of the set. A specific set is defined either by an explicit listing of its elements (e.g., $S = \{2, 3, 5, 7\}$) or by specifying a property that all the set's elements and only they must satisfy (e.g., $S = \{n: n \text{ is a prime number and } n < 10\}$). The most important set operations are checking membership of a given item in a given set (whether a given item is among the elements of the set), finding the union of two sets (which set comprises all the elements that belong to either of the two sets or to both of them), and finding the intersection of two sets (which set comprises all the elements that belong to both sets).

Sets can be implemented in computer applications in two ways. The first considers only sets that are subsets of some large set U, called the ***universal set***.

https://hemanthrajhemu.github.io

If set U has n elements, then any subset S of U can be represented by a bit string of size n, called a ***bit vector***, in which the ith element is 1 if and only if the ith element of U is included in set S. Thus, to continue with our example, if $U = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$, then $S = \{2, 3, 5, 7\}$ will be represented by the bit string 011010100. This way of representing sets makes it possible to implement the standard set operations very fast but at the expense of potentially using a large amount of storage.

The second and more common way to represent a set for computing purposes is to use the list structure to indicate the set's elements. (Of course, this option, too, is feasible only for finite sets; fortunately, unlike mathematics, this is the kind of sets most computer applications need.) Note, however, the two principal points of distinction between sets and lists. First, a set cannot contain identical elements; a list can. This requirement for uniqueness is sometimes circumvented by the introduction of a ***multiset*** or a ***bag***, an unordered collection of items that are not necessarily distinct. Second, a set is an unordered collection of items; therefore, changing the order of its elements does not change the set. A list, defined as an ordered collection of items, is exactly the opposite. This is an important theoretical distinction, but fortunately it is not important for many applications. It is also worth mentioning that if a set is represented by a list, depending on the application at hand, it might be worth maintaining the list in a sorted order.

In computing, the operations we need to perform for a set or a multiset most often are searching for a given item, adding a new item, and deleting an item from the collection. A data structure that implements these three operations is called the ***dictionary***. Note the relationship between this data structure and the problem of searching mentioned in Section 1.3; obviously, we are dealing here with searching in a dynamic context. Consequently, an efficient implementation of a dictionary has to strike a compromise between the efficiency of searching and the efficiencies of the other two operations. There are quite a few ways a dictionary can be implemented. They range from an unsophisticated use of arrays (sorted or not) to much more sophisticated techniques such as hashing and balanced search trees, which we discuss later in the book.

A number of applications in computing require a dynamic partition of some n-element set into a collection of disjoints subsets. After being initialized as a collection of n one-element subsets, the collection is subjected to a sequence of intermixed union and search operations. This problem is called the ***set union problem***. We discuss efficient algorithmic solutions to this problem in Section 9.2 in conjunction with one of its most important applications.

You may have noticed that in our review of basic data structures we almost always mentioned specific operations that are typically performed for the structure in question. This intimate relationship between data and operations has been recognized by computer scientists for a long time. It has led them in particular to the idea of an ***abstract data type (ADT)***: a set of abstract objects representing data items with a collection of operations that can be performed on them. As illustrations of this notion, reread, say, our definitions of priority queue and

https://hemanthrajhemu.github.io

dictionary. Although abstract data types could be implemented in older procedural languages such as Pascal (see, e.g., [Aho83]), it is much more convenient to do so in object-oriented languages, such as C++ and Java, that support abstract data types by means of ***classes***.

Exercises 1.4

1. Describe how one can implement each of the following operations on an array so that the time it takes does not depend on the array's size n.

a. Delete the ith element of an array ($1 \le i \le n$).

b. Delete the ith element of a sorted array (the remaining array has to stay sorted, of course).

2. If you have to solve the searching problem for a list of n numbers, how can you take advantage of the fact that the list is known to be sorted? Give separate answers for

a. lists represented as arrays.

b. lists represented as linked lists.

3. a. Show the stack after each operation of the following sequence that starts with the empty stack:

$$push(a), push(b), pop, push(c), push(d), pop$$

b. Show the queue after each operation of the following sequence that starts with the empty queue:

$$enqueue(a), enqueue(b), dequeue, enqueue(c), enqueue(d), dequeue$$

4. a. Let A be the adjacency matrix of an undirected graph. Explain what property of the matrix indicates that

i. the graph is complete.

ii. the graph has a loop, i.e., an edge connecting a vertex to itself.

iii. the graph has an isolated vertex, i.e., a vertex with no edges incident to it.

b. Answer the same questions for the adjacency list representation.

5. Give a detailed description of an algorithm for transforming a free tree into a tree rooted at a given vertex of the free tree.

6. Prove the inequalities that bracket the height of a binary tree with n vertices:

$$\lfloor \log_2 n \rfloor \le h \le n - 1.$$

7. Indicate how the ADT priority queue can be implemented as

https://hemanthrajhemu.github.io

   a. an (unsorted) array.
   b. a sorted array.
   c. a binary search tree.

8. How would you implement a dictionary of a reasonably small size n if you knew that all its elements are distinct (e.g., names of 50 states of the United States)? Specify an implementation of each dictionary operation.

9. For each of the following applications, indicate the most appropriate data structure:
   a. answering telephone calls in the order of their known priorities
   b. sending backlog orders to customers in the order they have been received
   c. implementing a calculator for computing simple arithmetical expressions

10. *Anagram checking* Design an algorithm for checking whether two given words are anagrams, i.e., whether one word can be obtained by permuting the letters of the other. (For example, the words *tea* and *eat* are anagrams.)

SUMMARY

- An *algorithm* is a sequence of nonambiguous instructions for solving a problem in a finite amount of time. An input to an algorithm specifies an *instance* of the problem the algorithm solves.

- Algorithms can be specified in a natural language or a pseudocode; they can also be implemented as computer programs.

- Among several ways to classify algorithms, the two principal alternatives are:
  — to group algorithms according to types of problems they solve;
  — to group algorithms according to underlying design techniques they are based upon.

- The important problem types are sorting, searching, string processing, graph problems, combinatorial problems, geometric problems, and numerical problems.

- Algorithm *design techniques* (or "strategies" or "paradigms") are general approaches to solving problems algorithmically, applicable to a variety of problems from different areas of computing.

- Although designing an algorithm is undoubtedly a creative activity, one can identify a sequence of interrelated actions involved in such a process. They are summarized in Figure 1.2.

- A good algorithm is usually a result of repeated efforts and rework.

https://hemanthrajhemu.github.io

- The same problem can often be solved by several algorithms. For example, three algorithms were given for computing the greatest common divisor of two integers: *Euclid's algorithm*, the consecutive integer checking algorithm, and the middle-school algorithm (enhanced by the *sieve of Eratosthenes* for generating a list of primes).
- Algorithms operate on data. This makes the issue of data structuring critical for efficient algorithmic problem solving. The most important elementary data structures are the *array* and the *linked list*. They are used for representing more abstract data structures such as the *list*, the *stack*, the *queue*, the *graph* (via its *adjacency matrix* or *adjacency lists*), the *binary tree*, and the *set*.
- An abstract collection of objects with several operations that can be performed on them is called an *abstract data type (ADT)*. The *list*, the *stack*, the *queue*, the *priority queue*, and the *dictionary* are important examples of abstract data types. Modern object-oriented languages support implementation of ADTs by means of classes.

https://hemanthrajhemu.github.io

2

Fundamentals of the Analysis of Algorithm Efficiency

I often say that when you can measure what you are speaking about and express it in numbers you know something about it; but when you cannot express it in numbers your knowledge is a meagre and unsatisfactory kind: it may be the beginning of knowledge but you have scarcely, in your thoughts, advanced to the stage of science, whatever the matter may be.

—Lord Kelvin (1824–1907)

Not everything that can be counted counts, and not everything that counts can be counted.

—Albert Einstein (1879–1955)

This chapter is devoted to analysis of algorithms. The *American Heritage Dictionary* defines "analysis" as "the separation of an intellectual or substantial whole into its constituent parts for individual study." Accordingly, each of the principal dimensions of an algorithm pointed out in Section 1.2 is both a legitimate and desirable subject of study. But the term "analysis of algorithms" is usually used in a narrower technical sense to mean an investigation of an algorithm's efficiency with respect to two resources: running time and memory space. This emphasis on efficiency is easy to explain. First, unlike such dimensions as simplicity and generality, efficiency can be studied in precise quantitative terms. Second, one can argue—although this is hardly always the case, given the speed and memory of today's computers—that the efficiency considerations are of primary importance from the practical point of view. In this chapter, we too limit the discussion to an algorithm's efficiency.

https://hemanthrajhemu.github.io

We start with a general framework for analyzing algorithm efficiency in Section 2.1. This section is arguably the most important in the chapter; the fundamental nature of the topic makes it also one of the most important sections in the entire book.

In Section 2.2, we introduce three notations: O ("big oh"), Ω ("big omega"), and Θ ("big theta"). Borrowed from mathematics, these notations have become *the* language for discussing an algorithm's efficiency.

In Section 2.3, we show how the general framework outlined in Section 2.1 can be systematically applied to analyzing the efficiency of nonrecursive algorithms. The main tool of such an analysis is setting up a sum representing the algorithm's running time and then simplifying the sum by using standard sum manipulation techniques.

In Section 2.4, we show how the general framework outlined in Section 2.1 can be systematically applied to analyzing the efficiency of recursive algorithms. Here, the main tool is not a sum but a special kind of equation called a recurrence relation. We explain how such recurrence relations can be set up and then introduce a method for solving them.

Although we illustrate the analysis framework and the methods of its applications by a variety of examples in the first four sections of this chapter, Section 2.5 is devoted to yet another example—that of the Fibonacci numbers. Introduced 800 years ago, this remarkable sequence appears in a variety of applications both within and outside computer science. A discussion of the Fibonacci sequence serves as a natural vehicle for introducing an important class of recurrence relations not solvable by the method of Section 2.4. We also discuss several algorithms for computing the Fibonacci numbers, mostly for the sake of a few general observations about the efficiency of algorithms and methods of analyzing them.

The methods of Sections 2.3 and 2.4 provide a powerful technique for analyzing the efficiency of many algorithms with mathematical clarity and precision, but these methods are far from being foolproof. The last two sections of the chapter deal with two approaches—empirical analysis and algorithm visualization—that complement the pure mathematical techniques of Sections 2.3 and 2.4. Much newer and, hence, less developed than their mathematical counterparts, these approaches promise to play an important role among the tools available for analysis of algorithm efficiency.

2.1 Analysis Framework

In this section, we outline a general framework for analyzing the efficiency of algorithms. To begin with, there are two kinds of efficiency: time efficiency and space efficiency. ***Time efficiency*** indicates how fast an algorithm in question runs; ***space efficiency*** deals with the extra space the algorithm requires. In the early days of electronic computing, both resources—time and space—were at a premium. Half

https://hemanthrajhemu.github.io

a century of relentless technological innovations have improved the computer's speed and memory size by many orders of magnitude. Now the amount of extra space required by an algorithm is typically not of as much concern, with the caveat that there is still, of course, a difference between the fast main memory, the slower secondary memory, and the cache. The time issue has not diminished quite to the same extent, however. In addition, the research experience has shown that for most problems, we can achieve much more spectacular progress in speed than in space. Therefore, following a well-established tradition of algorithm textbooks, we primarily concentrate on time efficiency, but the analytical framework introduced here is applicable to analyzing space efficiency as well.

Measuring an Input's Size

Let us start with the obvious observation that almost all algorithms run longer on larger inputs. For example, it takes longer to sort larger arrays, multiply larger matrices, and so on. Therefore, it is logical to investigate an algorithm's efficiency as a function of some parameter n indicating the algorithm's input size.[1] In most cases, selecting such a parameter is quite straightforward. For example, it will be the size of the list for problems of sorting, searching, finding the list's smallest element, and most other problems dealing with lists. For the problem of evaluating a polynomial $p(x) = a_n x^n + \cdots + a_0$ of degree n, it will be the polynomial's degree or the number of its coefficients, which is larger by one than its degree. You will see from the discussion that such a minor difference is inconsequential for the efficiency analysis.

There are situations, of course, where the choice of a parameter indicating an input size does matter. One such example is computing the product of two n-by-n matrices. There are two natural measures of size for this problem. The first and more frequently used is the matrix order n. But the other natural contender is the total number of elements N in the matrices being multiplied. (The latter is also more general since it is applicable to matrices that are not necessarily square.) Since there is a simple formula relating these two measures, we can easily switch from one to the other, but the answer about an algorithm's efficiency will be qualitatively different depending on which of the two measures we use (see Problem 2 in Exercises 2.1).

The choice of an appropriate size metric can be influenced by operations of the algorithm in question. For example, how should we measure an input's size for a spell-checking algorithm? If the algorithm examines individual characters of its input, then we should measure the size by the number of characters; if it works by processing words, we should count their number in the input.

1. Some algorithms require more than one parameter to indicate the size of their inputs (e.g., the number of vertices and the number of edges for algorithms on graphs represented by adjacency lists).

https://hemanthrajhemu.github.io

We should make a special note about measuring the size of inputs for algorithms involving properties of numbers (e.g., checking whether a given integer n is prime). For such algorithms, computer scientists prefer measuring size by the number b of bits in the n's binary representation:

$$b = \lfloor \log_2 n \rfloor + 1. \tag{2.1}$$

This metric usually gives a better idea about the efficiency of algorithms in question.

Units for Measuring Running Time

The next issue concerns units for measuring an algorithm's running time. Of course, we can simply use some standard unit of time measurement—a second, a millisecond, and so on—to measure the running time of a program implementing the algorithm. There are obvious drawbacks to such an approach, however: dependence on the speed of a particular computer, dependence on the quality of a program implementing the algorithm and of the compiler used in generating the machine code, and the difficulty of clocking the actual running time of the program. Since we are after a measure of an *algorithm*'s efficiency, we would like to have a metric that does not depend on these extraneous factors.

One possible approach is to count the number of times each of the algorithm's operations is executed. This approach is both excessively difficult and, as we shall see, usually unnecessary. The thing to do is to identify the most important operation of the algorithm, called the ***basic operation***, the operation contributing the most to the total running time, and compute the number of times the basic operation is executed.

As a rule, it is not difficult to identify the basic operation of an algorithm: it is usually the most time-consuming operation in the algorithm's innermost loop. For example, most sorting algorithms work by comparing elements (keys) of a list being sorted with each other; for such algorithms, the basic operation is a key comparison. As another example, algorithms for matrix multiplication and polynomial evaluation require two arithmetic operations: multiplication and addition. On most computers, multiplication of two numbers takes longer than addition, making the former an unquestionable choice for the basic operation.[2]

Thus, the established framework for the analysis of an algorithm's time efficiency suggests measuring it by counting the number of times the algorithm's basic operation is executed on inputs of size n. We will find out how to compute such a count for nonrecursive and recursive algorithms in Sections 2.3 and 2.4, respectively.

---

2. On some computers based on the so-called RISC architecture, it is not necessarily the case (see, for example, the timing data provided by Kernighan and Pike [Ker99], pp. 185–186).

https://hemanthrajhemu.github.io

Here is an important application. Let c_{op} be the execution time of an algorithm's basic operation on a particular computer, and let $C(n)$ be the number of times this operation needs to be executed for this algorithm. Then we can estimate the running time $T(n)$ of a program implementing this algorithm on that computer by the formula

$$T(n) \approx c_{op}C(n).$$

Of course, this formula should be used with caution. The count $C(n)$ does not contain any information about operations that are not basic, and, in fact, the count itself is often computed only approximately. Further, the constant c_{op} is also an approximation whose reliability is not always easy to assess. Still, unless n is extremely large or very small, the formula can give a reasonable estimate of the algorithm's running time. It also makes it possible to answer such questions as "How much faster would this algorithm run on a machine that is ten times faster than the one we have?" The answer is, obviously, ten times. Or, assuming that $C(n) = \frac{1}{2}n(n-1)$, how much longer will the algorithm run if we double its input size? The answer is about four times longer. Indeed, for all but very small values of n,

$$C(n) = \frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \approx \frac{1}{2}n^2$$

and therefore

$$\frac{T(2n)}{T(n)} \approx \frac{c_{op}C(2n)}{c_{op}C(n)} \approx \frac{\frac{1}{2}(2n)^2}{\frac{1}{2}n^2} = 4.$$

Note that we were able to answer the last question without actually knowing the value of c_{op}: it was neatly cancelled out in the ratio. Also note that $\frac{1}{2}$, the multiplicative constant in the formula for the count $C(n)$, was also cancelled out. It is for these reasons that the efficiency analysis framework ignores multiplicative constants and concentrates on the count's ***order of growth*** to within a constant multiple for large-size inputs.

Orders of Growth

Why this emphasis on the count's order of growth for large input sizes? A difference in running times on small inputs is not what really distinguishes efficient algorithms from inefficient ones. When we have to compute, for example, the greatest common divisor of two small numbers, it is not immediately clear how much more efficient Euclid's algorithm is compared to the other two algorithms discussed in Section 1.1 or even why we should care which of them is faster and by how much. It is only when we have to find the greatest common divisor of two large numbers that the difference in algorithm efficiencies becomes both clear and important. For large values of n, it is the function's order of growth that counts: just

https://hemanthrajhemu.github.io

TABLE 2.1 Values (some approximate) of several functions important for analysis of algorithms

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

look at Table 2.1, which contains values of a few functions particularly important for analysis of algorithms.

The magnitude of the numbers in Table 2.1 has a profound significance for the analysis of algorithms. The function growing the slowest among these is the logarithmic function. It grows so slowly, in fact, that we should expect a program implementing an algorithm with a logarithmic basic-operation count to run practically instantaneously on inputs of all realistic sizes. Also note that although specific values of such a count depend, of course, on the logarithm's base, the formula

$$\log_a n = \log_a b \log_b n$$

makes it possible to switch from one base to another, leaving the count logarithmic but with a new multiplicative constant. This is why we omit a logarithm's base and write simply $\log n$ in situations where we are interested just in a function's order of growth to within a multiplicative constant.

On the other end of the spectrum are the exponential function 2^n and the factorial function $n!$ Both these functions grow so fast that their values become astronomically large even for rather small values of n. (This is the reason why we did not include their values for $n > 10^2$ in Table 2.1.) For example, it would take about $4 \cdot 10^{10}$ years for a computer making one trillion (10^{12}) operations per second to execute 2^{100} operations. Though this is incomparably faster than it would have taken to execute 100! operations, it is still longer than 4.5 billion ($4.5 \cdot 10^9$) years—the estimated age of the planet Earth. There is a tremendous difference between the orders of growth of the functions 2^n and $n!$, yet both are often referred to as "exponential-growth functions" (or simply "exponential") despite the fact that, strictly speaking, only the former should be referred to as such. The bottom line, which is important to remember, is this:

> Algorithms that require an exponential number of operations are practical for solving only problems of very small sizes.

https://hemanthrajhemu.github.io

Another way to appreciate the qualitative difference among the orders of growth of the functions in Table 2.1 is to consider how they react to, say, a twofold increase in the value of their argument n. The function $\log_2 n$ increases in value by just 1 (because $\log_2 2n = \log_2 2 + \log_2 n = 1 + \log_2 n$); the linear function increases twofold; the "n-log-n" function $n \log_2 n$ increases slightly more than twofold; the quadratic function n^2 and cubic function n^3 increase fourfold and eightfold, respectively (because $(2n)^2 = 4n^2$ and $(2n)^3 = 8n^3$); the value of 2^n gets squared (because $2^{2n} = (2^n)^2$); and $n!$ increases much more than that (yes, even mathematics refuses to cooperate to give a neat answer for $n!$).

Worst-Case, Best-Case, and Average-Case Efficiencies

In the beginning of this section, we established that it is reasonable to measure an algorithm's efficiency as a function of a parameter indicating the size of the algorithm's input. But there are many algorithms for which running time depends not only on an input size but also on the specifics of a particular input. Consider, as an example, sequential search. This is a straightforward algorithm that searches for a given item (some search key K) in a list of n elements by checking successive elements of the list until either a match with the search key is found or the list is exhausted. Here is the algorithm's pseudocode, in which, for simplicity, a list is implemented as an array. (It also assumes that the second condition $A[i] \neq K$ will not be checked if the first one, which checks that the array's index does not exceed its upper bound, fails.)

ALGORITHM *SequentialSearch*$(A[0..n-1], K)$

```
//Searches for a given value in a given array by sequential search
//Input: An array A[0..n − 1] and a search key K
//Output: The index of the first element of A that matches K
//          or −1 if there are no matching elements
i ← 0
while i < n and A[i] ≠ K do
    i ← i + 1
if i < n return i
else return −1
```

Clearly, the running time of this algorithm can be quite different for the same list size n. In the worst case, when there are no matching elements or the first matching element happens to be the last one on the list, the algorithm makes the largest number of key comparisons among all possible inputs of size n: $C_{worst}(n) = n$.

The ***worst-case efficiency*** of an algorithm is its efficiency for the worst-case input of size n, which is an input (or inputs) of size n for which the algorithm runs the longest among all possible inputs of that size. The way to determine

https://hemanthrajhemu.github.io

the worst-case efficiency of an algorithm is, in principle, quite straightforward: we analyze the algorithm to see what kind of inputs yield the largest value of the basic operation's count $C(n)$ among all possible inputs of size n and then compute this worst-case value $C_{worst}(n)$. (For sequential search, the answer was obvious. The methods for handling less trivial situations are explained in subsequent sections of this chapter.) Clearly, the worst-case analysis provides very important information about an algorithm's efficiency by bounding its running time from above. In other words, it guarantees that for any instance of size n, the running time will not exceed $C_{worst}(n)$, its running time on the worst-case inputs.

The ***best-case efficiency*** of an algorithm is its efficiency for the best-case input of size n, which is an input (or inputs) of size n for which the algorithm runs the fastest among all possible inputs of that size. Accordingly, we can analyze the best-case efficiency as follows. First, we determine the kind of inputs for which the count $C(n)$ will be the smallest among all possible inputs of size n. (Note that the best case does not mean the smallest input; it means the input of size n for which the algorithm runs the fastest.) Then we ascertain the value of $C(n)$ on these most convenient inputs. For example, for sequential search, best-case inputs are lists of size n with their first elements equal to a search key; accordingly, $C_{best}(n) = 1$.

The analysis of the best-case efficiency is not nearly as important as that of the worst-case efficiency. But it is not completely useless, either. Though we should not expect to get best-case inputs, we might be able to take advantage of the fact that for some algorithms a good best-case performance extends to some useful types of inputs close to being the best-case ones. For example, there is a sorting algorithm (insertion sort) for which the best-case inputs are already sorted arrays on which the algorithm works very fast. Moreover, this good best-case efficiency deteriorates only slightly for almost sorted arrays. Thus, such an algorithm might well be the method of choice for applications dealing with almost sorted arrays. And, of course, if the best-case efficiency of an algorithm is unsatisfactory, we can immediately discard it without further analysis.

It should be clear from our discussion, however, that neither the worst-case analysis nor its best-case counterpart yields the necessary information about an algorithm's behavior on a "typical" or "random" input. This is the information that the ***average-case efficiency*** seeks to provide. To analyze the algorithm's average-case efficiency, we must make some assumptions about possible inputs of size n.

Let us consider again sequential search. The standard assumptions are that (a) the probability of a successful search is equal to p $(0 \le p \le 1)$ and (b) the probability of the first match occurring in the ith position of the list is the same for every i. Under these assumptions—the validity of which is usually difficult to verify, their reasonableness notwithstanding—we can find the average number of key comparisons $C_{avg}(n)$ as follows. In the case of a successful search, the probability of the first match occurring in the ith position of the list is p/n for every i, and the number of comparisons made by the algorithm in such a situation

https://hemanthrajhemu.github.io

is obviously i. In the case of an unsuccessful search, the number of comparisons is n with the probability of such a search being $(1 - p)$. Therefore,

$$C_{avg}(n) = [1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + \cdots + i \cdot \frac{p}{n} + \cdots + n \cdot \frac{p}{n}] + n \cdot (1 - p)$$

$$= \frac{p}{n}[1 + 2 + \cdots + i + \cdots + n] + n(1 - p)$$

$$= \frac{p}{n} \frac{n(n+1)}{2} + n(1 - p) = \frac{p(n+1)}{2} + n(1 - p).$$

This general formula yields some quite reasonable answers. For example, if $p = 1$ (i.e., the search must be successful), the average number of key comparisons made by sequential search is $(n + 1)/2$; i.e., the algorithm will inspect, on average, about half of the list's elements. If $p = 0$ (i.e., the search must be unsuccessful), the average number of key comparisons will be n because the algorithm will inspect all n elements on all such inputs.

As you can see from this very elementary example, investigation of the average-case efficiency is considerably more difficult than investigation of the worst-case and best-case efficiencies. The direct approach for doing this involves dividing all instances of size n into several classes so that for each instance of the class the number of times the algorithm's basic operation is executed is the same. (What were these classes for sequential search?) Then a probability distribution of inputs is obtained or assumed so that the expected value of the basic operation's count can be found.

The technical implementation of this plan is rarely easy, however, and probabilistic assumptions underlying it in each particular case are usually difficult to verify. Given our quest for simplicity, we will mostly quote known results about average-case efficiency of algorithms under discussion. If you are interested in derivations of these results, consult such books as [Baa00], [Sed96], [KnuI], [KnuII], and [KnuIII].

Does one really need the average-case efficiency information? The answer is unequivocally yes: there are many important algorithms for which the average-case efficiency is much better than the overly pessimistic worst-case efficiency would lead us to believe. So, without the average-case analysis, computer scientists could have missed many important algorithms. Finally, it should be clear from the preceding discussion that the average-case efficiency cannot be obtained by taking the average of the worst-case and the best-case efficiencies. Even though this average does occasionally coincide with the average-case cost, it is not a legitimate way of performing the average-case analysis.

Yet another type of efficiency is called ***amortized efficiency***. It applies not to a single run of an algorithm but rather to a sequence of operations performed on the same data structure. It turns out that in some situations a single operation can be expensive, but the total time for an entire sequence of n such operations is always significantly better than the worst-case efficiency of that single operation

https://hemanthrajhemu.github.io

multiplied by n. So we can "amortize" the high cost of such a worst-case occurrence over the entire sequence in a manner similar to the way a business would amortize the cost of an expensive item over the years of the item's productive life. This sophisticated approach was discovered by the American computer scientist Robert Tarjan, who used it, among other applications, in developing an interesting variation of the classic binary search tree (see [Tar87] for a quite readable nontechnical discussion and [Tar85] for a technical account). We will see an example of the usefulness of amortized efficiency in Section 9.2, when we consider algorithms for finding unions of disjoint sets.

Recapitulation of the Analysis Framework

Before we leave this section, let us summarize the main points of the framework outlined above.

- Both time and space efficiencies are measured as functions of the algorithm's input size.
- Time efficiency is measured by counting the number of times the algorithm's basic operation is executed. Space efficiency is measured by counting the number of extra memory units consumed by the algorithm.
- The efficiencies of some algorithms may differ significantly for inputs of the same size. For such algorithms, we need to distinguish between the worst-case, average-case, and best-case efficiencies.
- The framework's primary interest lies in the order of growth of the algorithm's running time (extra memory units consumed) as its input size goes to infinity.

In the next section, we look at formal means to investigate orders of growth. In Sections 2.3 and 2.4, we discuss particular methods for investigating nonrecursive and recursive algorithms, respectively. It is there that you will see how the analysis framework outlined here can be applied to investigating efficiency of specific algorithms. You will encounter many more examples throughout the rest of the book.

Exercises 2.1

1. For each of the following algorithms, indicate (i) a natural size metric for its inputs; (ii) its basic operation; (iii) whether the basic operation count can be different for inputs of the same size:
   a. computing the sum of n numbers
   b. computing $n!$
   c. finding the largest element in a list of n numbers
   d. Euclid's algorithm

https://hemanthrajhemu.github.io

e. sieve of Eratosthenes

f. pen-and-pencil algorithm for multiplying two n-digit decimal integers

2. a. Consider the definition-based algorithm for adding two n-by-n matrices. What is its basic operation? How many times is it performed as a function of the matrix order n? As a function of the total number of elements in the input matrices?

b. Answer the same questions for the definition-based algorithm for matrix multiplication.

3. Consider a variation of sequential search that scans a list to return the number of occurrences of a given search key in the list. Will its efficiency differ from the efficiency of classic sequential search?

4. a. *Glove selection* There are 22 gloves in a drawer: 5 pairs of red gloves, 4 pairs of yellow, and 2 pairs of green. You select the gloves in the dark and can check them only after a selection has been made. What is the smallest number of gloves you need to select to have at least one matching pair in the best case? in the worst case? (after [Mos01], #18)

b. *Missing socks* Imagine that after washing 5 distinct pairs of socks, you discover that two socks are missing. Of course, you would like to have the largest number of complete pairs remaining. Thus, you are left with 4 complete pairs in the best-case scenario and with 3 complete pairs in the worst case. Assuming that the probability of disappearance for each of the 10 socks is the same, find the probability of the best-case scenario; the probability of the worst-case scenario; the number of pairs you should expect in the average case. (after [Mos01], #48)

5. a. Prove formula (2.1) for the number of bits in the binary representation of a positive decimal integer.

b. What would be the analogous formula for the number of decimal digits?

c. Explain why, within the accepted analysis framework, it does not matter whether we use binary or decimal digits in measuring n's size.

6. Suggest how any sorting algorithm can be augmented in a way to make the best-case count of its key comparisons equal to just $n - 1$ (n is a list's size, of course). Do you think it would be a worthwhile addition to any sorting algorithm?

7. Gaussian elimination, the classic algorithm for solving systems of n linear equations in n unknowns, requires about $\frac{1}{3}n^3$ multiplications, which is the algorithm's basic operation.

a. How much longer should you expect Gaussian elimination to work on a system of 1000 equations versus a system of 500 equations?

b. You are considering buying a computer that is 1000 times faster than the one you currently have. By what factor will the faster computer increase

https://hemanthrajhemu.github.io

the sizes of systems solvable in the same amount of time as on the old computer?

8. For each of the following functions, indicate how much the function's value will change if its argument is increased fourfold.

   a. $\log_2 n$ **b.** $\sqrt{n}$ **c.** n **d.** n^2 **e.** n^3 **f.** 2^n

9. Indicate whether the first function of each of the following pairs has a smaller, same, or larger order of growth (to within a constant multiple) than the second function.

   a. $n(n+1)$ and $2000n^2$ **b.** $100n^2$ and $0.01n^3$
   c. $\log_2 n$ and $\ln n$ **d.** $\log_2^2 n$ and $\log_2 n^2$
   e. 2^{n-1} and 2^n **f.** $(n-1)!$ and $n!$

10. *Invention of chess* According to a well-known legend, the game of chess was invented many centuries ago in northwestern India by a sage named Shashi. When he took his invention to his king, the king liked the game so much that he offered the inventor any reward he wanted. Sashi asked for some grain to be obtained as follows: just a single grain of wheat was to be placed on the first square of the chess board, two on the second, four on the third, eight on the fourth, and so on, until all 64 squares had been filled. What would the ultimate result of this algorithm have been?

2.2 Asymptotic Notations and Basic Efficiency Classes

As pointed out in the previous section, the efficiency analysis framework concentrates on the order of growth of an algorithm's basic operation count as the principal indicator of the algorithm's efficiency. To compare and rank such orders of growth, computer scientists use three notations: O (big oh), Ω (big omega), and Θ (big theta). First, we introduce these notations informally, and then, after several examples, formal definitions are given. In the following discussion, $t(n)$ and $g(n)$ can be any nonnegative functions defined on the set of natural numbers. In the context we are interested in, $t(n)$ will be an algorithm's running time (usually indicated by its basic operation count $C(n)$), and $g(n)$ will be some simple function to compare the count with.

Informal Introduction

Informally, $O(g(n))$ is the set of all functions with a smaller or same order of growth as $g(n)$ (to within a constant multiple, as n goes to infinity). Thus, to give a few examples, the following assertions are all true:

$$n \in O(n^2), \qquad 100n + 5 \in O(n^2), \qquad \frac{1}{2}n(n-1) \in O(n^2).$$

https://hemanthrajhemu.github.io

Indeed, the first two functions are linear and hence have a smaller order of growth than $g(n) = n^2$, while the last one is quadratic and hence has the same order of growth as n^2. On the other hand,

$$n^3 \notin O(n^2), \qquad 0.00001n^3 \notin O(n^2), \qquad n^4 + n + 1 \notin O(n^2).$$

Indeed, the functions n^3 and $0.00001n^3$ are both cubic and hence have a higher order of growth than n^2; and so has the fourth-degree polynomial $n^4 + n + 1$.

The second notation, $\Omega(g(n))$, stands for the set of all functions with a larger or same order of growth as $g(n)$ (to within a constant multiple, as n goes to infinity). For example,

$$n^3 \in \Omega(n^2), \qquad \frac{1}{2}n(n-1) \in \Omega(n^2), \qquad \text{but } 100n + 5 \notin \Omega(n^2).$$

Finally, $\Theta(g(n))$ is the set of all functions that have the same order of growth as $g(n)$ (to within a constant multiple, as n goes to infinity). Thus, every quadratic function $an^2 + bn + c$ with $a > 0$ is in $\Theta(n^2)$, but so are, among infinitely many others, $n^2 + \sin n$ and $n^2 + \log n$. (Can you explain why?)

Hopefully, the preceding informal discussion has made you comfortable with the idea behind the three asymptotic notations. So now come the formal definitions.

O-notation

DEFINITION 1 A function $t(n)$ is said to be in $O(g(n))$, denoted $t(n) \in O(g(n))$, if $t(n)$ is bounded above by some constant multiple of $g(n)$ for all large n, i.e., if there exist some positive constant c and some nonnegative integer n_0 such that

$$t(n) \le cg(n) \quad \text{for all } n \ge n_0.$$

The definition is illustrated in Figure 2.1 where, for the sake of visual clarity, n is extended to be a real number.

As an example, let us formally prove one of the assertions made in the introduction: $100n + 5 \in O(n^2)$. Indeed,

$$100n + 5 \le 100n + n \text{ (for all } n \ge 5) = 101n \le 101n^2.$$

Thus, as values of the constants c and n_0 required by the definition, we can take 101 and 5, respectively.

Note that the definition gives us a lot of freedom in choosing specific values for constants c and n_0. For example, we could also reason that

$$100n + 5 \le 100n + 5n \text{ (for all } n \ge 1) = 105n$$

to complete the proof with $c = 105$ and $n_0 = 1$.

https://hemanthrajhemu.github.io

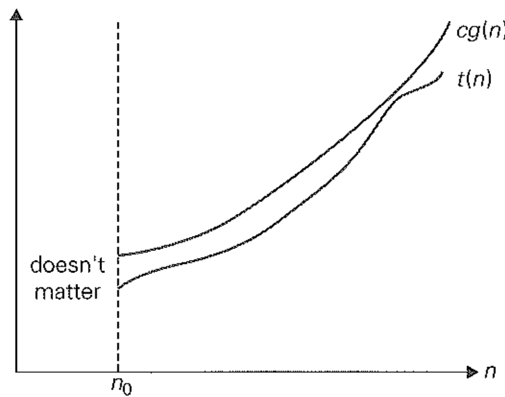


FIGURE 2.1 Big-oh notation: $t(n) \in O(g(n))$

Ω-notation

DEFINITION 2 A function $t(n)$ is said to be in $\Omega(g(n))$, denoted $t(n) \in \Omega(g(n))$, if $t(n)$ is bounded below by some positive constant multiple of $g(n)$ for all large n, i.e., if there exist some positive constant c and some nonnegative integer n_0 such that

$$t(n) \geq cg(n) \quad \text{for all } n \geq n_0.$$

The definition is illustrated in Figure 2.2.

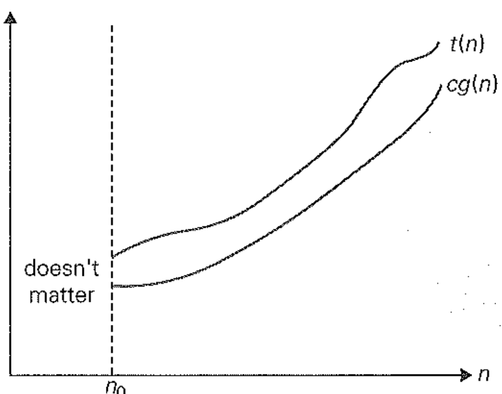


FIGURE 2.2 Big-omega notation: $t(n) \in \Omega(g(n))$

https://hemanthrajhemu.github.io

Here is an example of the formal proof that $n^3 \in \Omega(n^2)$:

$$n^3 \geq n^2 \quad \text{for all } n \geq 0,$$

i.e., we can select $c = 1$ and $n_0 = 0$.

Θ-notation

DEFINITION 3 A function $t(n)$ is said to be in $\Theta(g(n))$, denoted $t(n) \in \Theta(g(n))$, if $t(n)$ is bounded both above and below by some positive constant multiples of $g(n)$ for all large n, i.e., if there exist some positive constant c_1 and c_2 and some nonnegative integer n_0 such that

$$c_2 g(n) \leq t(n) \leq c_1 g(n) \quad \text{for all } n \geq n_0.$$

The definition is illustrated in Figure 2.3.

For example, let us prove that $\frac{1}{2}n(n-1) \in \Theta(n^2)$. First, we prove the right inequality (the upper bound):

$$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \leq \frac{1}{2}n^2 \text{ for all } n \geq 0.$$

Second, we prove the left inequality (the lower bound):

$$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \geq \frac{1}{2}n^2 - \frac{1}{2}n\frac{1}{2}n \text{ (for all } n \geq 2) = \frac{1}{4}n^2.$$

Hence, we can select $c_2 = \frac{1}{4}$, $c_1 = \frac{1}{2}$, and $n_0 = 2$.

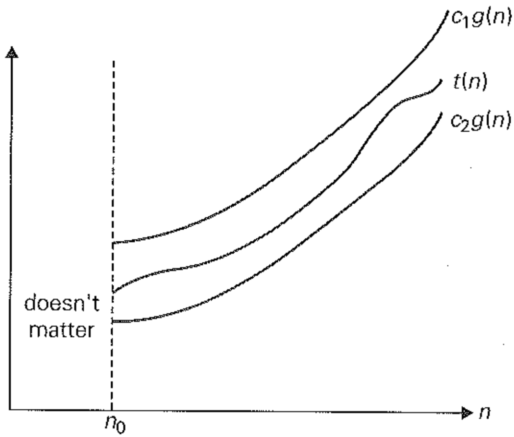


FIGURE 2.3 Big-theta notation: $t(n) \in \Theta(g(n))$

https://hemanthrajhemu.github.io

Useful Property Involving the Asymptotic Notations

Using the formal definitions of the asymptotic notations, we can prove their general properties (see Problem 7 in Exercises 2.2 for a few simple examples). The following property, in particular, is useful in analyzing algorithms that comprise two consecutively executed parts.

THEOREM If $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$, then

$$t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\}).$$

(The analogous assertions are true for the Ω and Θ notations as well.)

PROOF (As you will see, the proof extends to orders of growth the following simple fact about four arbitrary real numbers a_1, b_1, a_2, and b_2: if $a_1 \le b_1$ and $a_2 \le b_2$, then $a_1 + a_2 \le 2\max\{b_1, b_2\}$.) Since $t_1(n) \in O(g_1(n))$, there exist some positive constant c_1 and some nonnegative integer n_1 such that

$$t_1(n) \le c_1 g_1(n) \quad \text{for all } n \ge n_1.$$

Similarly, since $t_2(n) \in O(g_2(n))$,

$$t_2(n) \le c_2 g_2(n) \quad \text{for all } n \ge n_2.$$

Let us denote $c_3 = \max\{c_1, c_2\}$ and consider $n \ge \max\{n_1, n_2\}$ so that we can use both inequalities. Adding the two inequalities above yields the following:

$$\begin{aligned} t_1(n) + t_2(n) &\le c_1 g_1(n) + c_2 g_2(n) \\ &\le c_3 g_1(n) + c_3 g_2(n) = c_3[g_1(n) + g_2(n)] \\ &\le c_3 2\max\{g_1(n), g_2(n)\}. \end{aligned}$$

Hence, $t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$, with the constants c and n_0 required by the O definition being $2c_3 = 2\max\{c_1, c_2\}$ and $\max\{n_1, n_2\}$, respectively. ■

So what does this property imply for an algorithm that comprises two consecutively executed parts? It implies that the algorithm's overall efficiency is determined by the part with a larger order of growth, i.e., its least efficient part:

$$\left.\begin{array}{|c|} \hline t_1(n) \in O(g_1(n)) \\ \hline t_2(n) \in O(g_2(n)) \\ \hline \end{array}\right\} \quad t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\}).$$

For example, we can check whether an array has identical elements by means of the following two-part algorithm: first, sort the array by applying some known sorting algorithm; second, scan the sorted array to check its consecutive elements for equality. If, for example, a sorting algorithm used in the first part makes no more than $\frac{1}{2}n(n-1)$ comparisons (and hence is in $O(n^2)$) while the second part makes no more than $n-1$ comparisons (and hence is in $O(n)$), the efficiency of the entire algorithm will be in $O(\max\{n^2, n\}) = O(n^2)$.

https://hemanthrajhemu.github.io

Using Limits for Comparing Orders of Growth

Though the formal definitions of O, Ω, and Θ are indispensable for proving their abstract properties, they are rarely used for comparing the orders of growth of two specific functions. A much more convenient method for doing so is based on computing the limit of the ratio of two functions in question. Three principal cases may arise:

$$\lim_{n\to\infty}\frac{t(n)}{g(n)}=\begin{cases}0 & \text{implies that } t(n) \text{ has a smaller order of growth than } g(n)\\ c>0 & \text{implies that } t(n) \text{ has the same order of growth as } g(n)\\ \infty & \text{implies that } t(n) \text{ has a larger order of growth than } g(n).^3\end{cases}$$

Note that the first two cases mean that $t(n) \in O(g(n))$, the last two mean that $t(n) \in \Omega(g(n))$, and the second case means that $t(n) \in \Theta(g(n))$.

The limit-based approach is often more convenient than the one based on the definitions because it can take advantage of the powerful calculus techniques developed for computing limits, such as L'Hôpital's rule

$$\lim_{n\to\infty}\frac{t(n)}{g(n)}=\lim_{n\to\infty}\frac{t'(n)}{g'(n)}$$

and Stirling's formula

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \quad \text{for large values of } n.$$

Here are three examples of using the limit-based approach to comparing orders of growth of two functions.

EXAMPLE 1 Compare the orders of growth of $\frac{1}{2}n(n-1)$ and n^2. (This is one of the examples we used at the beginning of this section to illustrate the definitions.)

$$\lim_{n\to\infty}\frac{\frac{1}{2}n(n-1)}{n^2}=\frac{1}{2}\lim_{n\to\infty}\frac{n^2-n}{n^2}=\frac{1}{2}\lim_{n\to\infty}(1-\frac{1}{n})=\frac{1}{2}.$$

Since the limit is equal to a positive constant, the functions have the same order of growth or, symbolically, $\frac{1}{2}n(n-1) \in \Theta(n^2)$. ■

EXAMPLE 2 Compare the orders of growth of $\log_2 n$ and $\sqrt{n}$. (Unlike Example 1, the answer here is not immediately obvious.)

$$\lim_{n\to\infty}\frac{\log_2 n}{\sqrt{n}}=\lim_{n\to\infty}\frac{(\log_2 n)'}{(\sqrt{n})'}=\lim_{n\to\infty}\frac{(\log_2 e)\frac{1}{n}}{\frac{1}{2\sqrt{n}}}=2\log_2 e\lim_{n\to\infty}\frac{\sqrt{n}}{n}=0.$$

3. The fourth case, in which such a limit does not exist, rarely happens in the actual practice of analyzing algorithms. Still, this possibility makes the limit-based approach to comparing orders of growth less general than the one based on the definitions of O, Ω, and Θ.

https://hemanthrajhemu.github.io

Since the limit is equal to zero, $\log_2 n$ has a smaller order of growth than $\sqrt{n}$. (Since $\lim_{n\to\infty} \frac{\log_2 n}{\sqrt{n}} = 0$, we can use the so-called ***little-oh notation***: $\log_2 n \in o(\sqrt{n})$. Unlike the big-oh, the little-oh notation is rarely used in analysis of algorithms.) ■

EXAMPLE 3 Compare the orders of growth of $n!$ and 2^n. (We discussed this issue informally in the previous section.) Taking advantage of Stirling's formula, we get

$$\lim_{n\to\infty} \frac{n!}{2^n} = \lim_{n\to\infty} \frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}{2^n} = \lim_{n\to\infty} \sqrt{2\pi n}\frac{n^n}{2^n e^n} = \lim_{n\to\infty} \sqrt{2\pi n}\left(\frac{n}{2e}\right)^n = \infty.$$

Thus, though 2^n grows very fast, $n!$ grows still faster. We can write symbolically that $n! \in \Omega(2^n)$; note, however, that while big-omega notation does not preclude the possibility that $n!$ and 2^n have the same order of growth, the limit computed here certainly does. ■

Basic Efficiency Classes

Even though the efficiency analysis framework puts together all the functions whose orders of growth differ by a constant multiple, there are still infinitely many such classes. (For example, the exponential functions a^n have different orders of growth for different values of base a.) Therefore, it may come as a surprise that the time efficiencies of a large number of algorithms fall into only a few classes. These classes are listed in Table 2.2 in increasing order of their orders of growth, along with their names and a few comments.

You could raise a concern that classifying algorithms by their asymptotic efficiency would be of little practical use since the values of multiplicative constants are usually left unspecified. This leaves open a possibility of an algorithm in a worse efficiency class running faster than an algorithm in a better efficiency class for inputs of realistic sizes. For example, if the running time of one algorithm is n^3 while the running time of the other is $10^6 n^2$, the cubic algorithm will outperform the quadratic algorithm unless n exceeds 10^6. A few such anomalies are indeed known. For example, there exist algorithms for matrix multiplication with a better asymptotic efficiency than the cubic efficiency of the definition-based algorithm (see Section 4.5). Because of their much larger multiplicative constants, however, the value of these more sophisticated algorithms is mostly theoretical.

Fortunately, multiplicative constants usually do not differ that drastically. As a rule, you should expect an algorithm from a better asymptotic efficiency class to outperform an algorithm from a worse class even for moderately sized inputs. This observation is especially true for an algorithm with a better than exponential running time versus an exponential (or worse) algorithm.

https://hemanthrajhemu.github.io

TABLE 2.2 Basic asymptotic efficiency classes

Class	Name	Comments
1	*constant*	Short of best-case efficiencies, very few reasonable examples can be given since an algorithm's running time typically goes to infinity when its input size grows infinitely large.
$\log n$	*logarithmic*	Typically, a result of cutting a problem's size by a constant factor on each iteration of the algorithm (see Section 5.5). Note that a logarithmic algorithm cannot take into account all its input (or even a fixed fraction of it): any algorithm that does so will have at least linear running time.
n	*linear*	Algorithms that scan a list of size n (e.g., sequential search) belong to this class.
$n \log n$	"*n-log-n*"	Many divide-and-conquer algorithms (see Chapter 4), including mergesort and quicksort in the average case, fall into this category.
n^2	*quadratic*	Typically, characterizes efficiency of algorithms with two embedded loops (see the next section). Elementary sorting algorithms and certain operations on n-by-n matrices are standard examples.
n^3	*cubic*	Typically, characterizes efficiency of algorithms with three embedded loops (see the next section). Several nontrivial algorithms from linear algebra fall into this class.
2^n	*exponential*	Typical for algorithms that generate all subsets of an n-element set. Often, the term "exponential" is used in a broader sense to include this and larger orders of growth as well.
$n!$	*factorial*	Typical for algorithms that generate all permutations of an n-element set.

Exercises 2.2

1. Use the most appropriate notation among O, Θ, and Ω to indicate the time efficiency class of sequential search (see Section 2.1)

a. in the worst case.

b. in the best case.

c. in the average case.

https://hemanthrajhemu.github.io

2. Use the informal definitions of O, Θ, and Ω to determine whether the following assertions are true or false.
   - **a.** $n(n+1)/2 \in O(n^3)$ **b.** $n(n+1)/2 \in O(n^2)$
   - **c.** $n(n+1)/2 \in \Theta(n^3)$ **d.** $n(n+1)/2 \in \Omega(n)$
3. For each of the following functions, indicate the class $\Theta(g(n))$ the function belongs to. (Use the simplest $g(n)$ possible in your answers.) Prove your assertions.
   - **a.** $(n^2+1)^{10}$ **b.** $\sqrt{10n^2+7n+3}$
   - **c.** $2n\lg(n+2)^2 + (n+2)^2 \lg \frac{n}{2}$ **d.** $2^{n+1}+3^{n-1}$
   - **e.** $\lfloor \log_2 n \rfloor$
4. **a.** Table 2.1 contains values of several functions that often arise in analysis of algorithms. These values certainly suggest that the functions

$$\log n,\ n,\ n\log n,\ n^2,\ n^3,\ 2^n,\ n!$$

   are listed in increasing order of their order of growth. Do these values prove this fact with mathematical certainty?

   b. Prove that the functions are indeed listed in increasing order of their order of growth.
5. Order the following functions according to their order of growth (from the lowest to the highest):

$$(n-2)!,\ 5\lg(n+100)^{10},\ 2^{2n},\ 0.001n^4+3n^3+1,\ \ln^2 n,\ \sqrt[3]{n},\ 3^n.$$

6. **a.** Prove that every polynomial of degree k, $p(n) = a_k n^k + a_{k-1}n^{k-1} + \cdots + a_0$, with $a_k > 0$ belongs to $\Theta(n^k)$.

   b. Prove that exponential functions a^n have different orders of growth for different values of base $a > 0$.
7. Prove (by using the definitions of the notations involved) or disprove (by giving a specific counterexample) the following assertions.
   - **a.** If $t(n) \in O(g(n))$, then $g(n) \in \Omega(t(n))$.
   - **b.** $\Theta(\alpha g(n)) = \Theta(g(n))$ where $\alpha > 0$.
   - **c.** $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$.
   - **d.** For any two nonnegative functions $t(n)$ and $g(n)$ defined on the set of nonnegative integers, either $t(n) \in O(g(n))$, or $t(n) \in \Omega(g(n))$, or both.
8. Prove the section's theorem for
   - **a.** Ω-notation.
   - **b.** Θ-notation.
9. We mentioned in this section that one can check whether all elements of an array are distinct by a two-part algorithm based on the array's presorting.

https://hemanthrajhemu.github.io

a. If the presorting is done by an algorithm with the time efficiency in $\Theta(n \log n)$, what will be the time efficiency class of the entire algorithm?

b. If the sorting algorithm used for presorting needs an extra array of size n, what will be the space efficiency class of the entire algorithm?

10. *Door in a wall* You are facing a wall that stretches infinitely in both directions. There is a door in the wall, but you know neither how far away nor in which direction. You can see the door only when you are right next to it. Design an algorithm that enables you to reach the door by walking at most $O(n)$ steps where n is the (unknown to you) number of steps between your initial position and the door. [Par95], #652

2.3 Mathematical Analysis of Nonrecursive Algorithms

In this section, we systematically apply the general framework outlined in Section 2.1 to analyzing the time efficiency of nonrecursive algorithms. Let us start with a very simple example that demonstrates all the principal steps typically taken in analyzing such algorithms.

EXAMPLE 1 Consider the problem of finding the value of the largest element in a list of n numbers. For simplicity, we assume that the list is implemented as an array. The following is a pseudocode of a standard algorithm for solving the problem.

ALGORITHM *MaxElement*($A[0..n-1]$)

```
//Determines the value of the largest element in a given array
//Input: An array A[0..n − 1] of real numbers
//Output: The value of the largest element in A
maxval ← A[0]
for i ← 1 to n − 1 do
    if A[i] > maxval
        maxval ← A[i]
return maxval
```

The obvious measure of an input's size here is the number of elements in the array, i.e., n. The operations that are going to be executed most often are in the algorithm's **for** loop. There are two operations in the loop's body: the comparison $A[i] > maxval$ and the assignment $maxval \leftarrow A[i]$. Which of these two operations should we consider basic? Since the comparison is executed on each repetition of the loop and the assignment is not, we should consider the comparison to be the algorithm's basic operation. (Note that the number of comparisons will be the

https://hemanthrajhemu.github.io

same for all arrays of size n; therefore, in terms of this metric, there is no need to distinguish among the worst, average, and best cases here.)

Let us denote $C(n)$ the number of times this comparison is executed and try to find a formula expressing it as a function of size n. The algorithm makes one comparison on each execution of the loop, which is repeated for each value of the loop's variable i within the bounds 1 and $n-1$ (inclusively). Therefore, we get the following sum for $C(n)$:

$$C(n) = \sum_{i=1}^{n-1} 1.$$

This is an easy sum to compute because it is nothing else but 1 repeated $n-1$ times. Thus,

$$C(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n).$$ ■

Here is a general plan to follow in analyzing nonrecursive algorithms.

General Plan for Analyzing Time Efficiency of Nonrecursive Algorithms

1. Decide on a parameter (or parameters) indicating an input's size.
2. Identify the algorithm's basic operation. (As a rule, it is located in its innermost loop.)
3. Check whether the number of times the basic operation is executed depends only on the size of an input. If it also depends on some additional property, the worst-case, average-case, and, if necessary, best-case efficiencies have to be investigated separately.
4. Set up a sum expressing the number of times the algorithm's basic operation is executed.[4]
5. Using standard formulas and rules of sum manipulation, either find a closed-form formula for the count or, at the very least, establish its order of growth.

Before proceeding with further examples, you may want to review Appendix A, which contains a list of summation formulas and rules that are often useful in analysis of algorithms. In particular, we use especially frequently two basic rules of sum manipulation

$$\sum_{i=l}^{u} ca_i = c \sum_{i=l}^{u} a_i \quad \textbf{(R1)}$$

$$\sum_{i=l}^{u} (a_i \pm b_i) = \sum_{i=l}^{u} a_i \pm \sum_{i=l}^{u} b_i \quad \textbf{(R2)}$$

4. Sometimes, an analysis of a nonrecursive algorithm requires setting up not a sum but a recurrence relation for the number of times its basic operation is executed. Using recurrence relations is much more typical for analyzing recursive algorithms (see Section 2.4).

https://hemanthrajhemu.github.io

and two summation formulas

$$\sum_{i=l}^{u} 1 = u - l + 1 \text{ where } l \le u \text{ are some lower and upper integer limits} \quad \textbf{(S1)}$$

$$\sum_{i=0}^{n} i = \sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} \approx \frac{1}{2}n^2 \in \Theta(n^2). \quad \textbf{(S2)}$$

(Note that the formula $\sum_{i=1}^{n-1} 1 = n - 1$, which we used in Example 1, is a special case of formula (S1) for $l = 1$ and $u = n - 1$.)

EXAMPLE 2 Consider the ***element uniqueness problem***: check whether all the elements in a given array are distinct. This problem can be solved by the following straightforward algorithm.

ALGORITHM *UniqueElements*($A[0..n-1]$)

```
//Determines whether all the elements in a given array are distinct
//Input: An array A[0..n − 1]
//Output: Returns "true" if all the elements in A are distinct
//        and "false" otherwise
for i ← 0 to n − 2 do
    for j ← i + 1 to n − 1 do
        if A[i] = A[j] return false
return true
```

The natural measure of the input's size here is again the number of elements in the array, i.e., n. Since the innermost loop contains a single operation (the comparison of two elements), we should consider it as the algorithm's basic operation. Note, however, that the number of element comparisons will depend not only on n but also on whether there are equal elements in the array and, if there are, which array positions they occupy. We will limit our investigation to the worst case only.

By definition, the worst case input is an array for which the number of element comparisons $C_{worst}(n)$ is the largest among all arrays of size n. An inspection of the innermost loop reveals that there are two kinds of worst-case inputs (inputs for which the algorithm does not exit the loop prematurely): arrays with no equal elements and arrays in which the last two elements are the only pair of equal elements. For such inputs, one comparison is made for each repetition of the innermost loop, i.e., for each value of the loop's variable j between its limits $i + 1$ and $n - 1$; and this is repeated for each value of the outer loop, i.e., for each value of the loop's variable i between its limits 0 and $n - 2$. Accordingly, we get

https://hemanthrajhemu.github.io

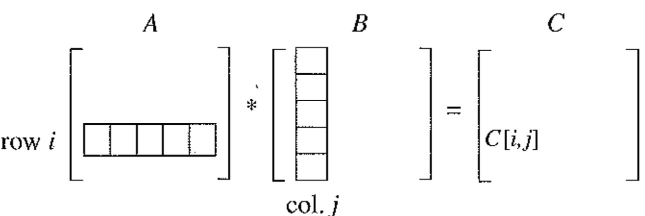
$$C_{worst}(n) = \sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2}[(n-1)-(i+1)+1] = \sum_{i=0}^{n-2}(n-1-i)$$
$$= \sum_{i=0}^{n-2}(n-1) - \sum_{i=0}^{n-2} i = (n-1)\sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2}$$
$$= (n-1)^2 - \frac{(n-2)(n-1)}{2} = \frac{(n-1)n}{2} \approx \frac{1}{2}n^2 \in \Theta(n^2).$$

We also could have computed the sum $\sum_{i=0}^{n-2}(n-1-i)$ faster as follows:

$$\sum_{i=0}^{n-2}(n-1-i) = (n-1) + (n-2) + \cdots + 1 \overset{(S2)}{=} \frac{(n-1)n}{2}.$$

Note that this result was perfectly predictable: in the worst case, the algorithm needs to compare all $n(n-1)/2$ distinct pairs of its n elements. ■

EXAMPLE 3 Given two n-by-n matrices A and B, find the time efficiency of the definition-based algorithm for computing their product $C = AB$. By definition, C is an n-by-n matrix whose elements are computed as the scalar (dot) products of the rows of matrix A and the columns of matrix B:



where $C[i, j] = A[i, 0]B[0, j] + \cdots + A[i, k]B[k, j] + \cdots + A[i, n-1]B[n-1, j]$ for every pair of indices $0 \le i, j \le n-1$.

```
ALGORITHM  MatrixMultiplication(A[0..n − 1, 0..n − 1], B[0..n − 1, 0..n − 1])
    //Multiplies two n-by-n matrices by the definition-based algorithm
    //Input: Two n-by-n matrices A and B
    //Output: Matrix C = AB
    for i ← 0 to n − 1 do
        for j ← 0 to n − 1 do
            C[i, j] ← 0.0
            for k ← 0 to n − 1 do
                C[i, j] ← C[i, j] + A[i, k] * B[k, j]
    return C
```

https://hemanthrajhemu.github.io

We measure an input's size by matrix order n. In the algorithm's innermost loop are two arithmetical operations—multiplication and addition—that, in principle, can compete for designation as the algorithm's basic operation. We consider multiplication as the algorithm's basic operation (see Section 2.1). Note that for this algorithm, we do not have to choose between these two operations because on each repetition of the innermost loop, each of the two is executed exactly once. So by counting one we automatically count the other. Let us set up a sum for the total number of multiplications $M(n)$ executed by the algorithm. (Since this count depends only on the size of the input matrices, we do not have to investigate the worst-case, average-case, and best-case efficiencies separately.)

Obviously, there is just one multiplication executed on each repetition of the algorithm's innermost loop, which is governed by the variable k ranging from the lower bound 0 to the upper bound $n-1$. Therefore, the number of multiplications made for every pair of specific values of variables i and j is

$$\sum_{k=0}^{n-1} 1,$$

and the total number of multiplications $M(n)$ is expressed by the following triple sum:

$$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1.$$

Now we can compute this sum by using formula $(S1)$ and rule $(R1)$ (see above). Starting with the innermost sum $\sum_{k=0}^{n-1} 1$, which is equal to n (why?), we get

$$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1 = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} n = \sum_{i=0}^{n-1} n^2 = n^3.$$

(This example is simple enough so that we could get this result without all the summation machinations. How? The algorithm computes n^2 elements of the product matrix. Each of the product's elements is computed as the scalar (dot) product of an n-element row of the first matrix and an n-element column of the second matrix, which takes n multiplications. So the total number of multiplications is $n \cdot n^2 = n^3$. It is this kind of reasoning we expected you to employ when answering this question in Problem 2 of Exercises 2.1.)

If we now want to estimate the running time of the algorithm on a particular machine, we can do it by the product

$$T(n) \approx c_m M(n) = c_m n^3,$$

where c_m is the time of one multiplication on the machine in question. We would get a more accurate estimate if we took into account the time spent on the additions, too:

$$T(n) \approx c_m M(n) + c_a A(n) = c_m n^3 + c_a n^3 = (c_m + c_a) n^3,$$

https://hemanthrajhemu.github.io

where c_a is the time of one addition. Note that the estimates differ only by their multiplicative constants, not by their order of growth. ■

You should not have the erroneous impression that the plan outlined above always succeeds in analyzing a nonrecursive algorithm. An irregular change in a loop's variable, a sum too complicated to analyze, and the difficulties intrinsic to the average-case analysis are just some of the obstacles that can prove to be insurmountable. These caveats notwithstanding, the plan does work for many simple nonrecursive algorithms, as you will see throughout the subsequent chapters of the book.

As a last example, let us consider an algorithm in which the loop's variable changes in a different manner from that of the previous examples.

EXAMPLE 4 The following algorithm finds the number of binary digits in the binary representation of a positive decimal integer.

```
ALGORITHM Binary(n)
    //Input: A positive decimal integer n
    //Output: The number of binary digits in n's binary representation
    count ← 1
    while n > 1 do
        count ← count + 1
        n ← ⌊n/2⌋
    return count
```

First, notice that the most frequently executed operation here is not inside the **while** loop but rather the comparison $n > 1$ that determines whether the loop's body will be executed. Since the number of times the comparison will be executed is larger than the number of repetitions of the loop's body by exactly 1, the choice is not that important.

A more significant feature of this example is the fact that the loop's variable takes on only a few values between its lower and upper limits; therefore we have to use an alternative way of computing the number of times the loop is executed. Since the value of n is about halved on each repetition of the loop, the answer should be about $\log_2 n$. The exact formula for the number of times the comparison $n > 1$ will be executed is actually $\lfloor \log_2 n \rfloor + 1$—the number of bits in the binary representation of n according to formula (2.1). We could also get this answer by applying the analysis technique based on recurrence relations; we discuss this technique in the next section because it is more pertinent to the analysis of recursive algorithms. ■

https://hemanthrajhemu.github.io

Exercises 2.3

1. Compute the following sums.
   a. $1+3+5+7+\cdots+999$
   b. $2+4+8+16+\cdots+1024$
   c. $\sum_{i=3}^{n+1} 1$ **d.** $\sum_{i=3}^{n+1} i$ **e.** $\sum_{i=0}^{n-1} i(i+1)$
   f. $\sum_{j=1}^{n} 3^{j+1}$ **g.** $\sum_{i=1}^{n}\sum_{j=1}^{n} ij$ **h.** $\sum_{i=1}^{n} 1/i(i+1)$
2. Find the order of growth of the following sums.
   a. $\sum_{i=0}^{n-1}(i^2+1)^2$ **b.** $\sum_{i=2}^{n-1} \lg i^2$
   c. $\sum_{i=1}^{n}(i+1)2^{i-1}$ **d.** $\sum_{i=0}^{n-1}\sum_{j=0}^{i-1}(i+j)$
   Use the $\Theta(g(n))$ notation with the simplest function $g(n)$ possible.
3. The sample variance of n measurements $x_1, \ldots, x_n$ can be computed as
$$\frac{\sum_{i=1}^{n}(x_i-\bar{x})^2}{n-1} \text{ where } \bar{x}=\frac{\sum_{i=1}^{n} x_i}{n}$$
   or
$$\frac{\sum_{i=1}^{n} x_i^2-(\sum_{i=1}^{n} x_i)^2/n}{n-1}.$$
   Find and compare the number of divisions, multiplications, and additions/subtractions (additions and subtractions are usually bunched together) that are required for computing the variance according to each of these formulas.
4. Consider the following algorithm.

```
ALGORITHM Mystery(n)
    //Input: A nonnegative integer n
    S ← 0
    for i ← 1 to n do
        S ← S + i * i
    return S
```

   a. What does this algorithm compute?
   b. What is its basic operation?
   c. How many times is the basic operation executed?
   d. What is the efficiency class of this algorithm?
   e. Suggest an improvement or a better algorithm altogether and indicate its efficiency class. If you cannot do it, try to prove that in fact it cannot be done.

https://hemanthrajhemu.github.io

5. Consider the following algorithm.

ALGORITHM *Secret*($A[0..n-1]$)
  //Input: An array $A[0..n-1]$ of n real numbers
  $minval \leftarrow A[0]$; $maxval \leftarrow A[0]$
  for $i \leftarrow 1$ **to** $n-1$ **do**
    if $A[i] < minval$
      $minval \leftarrow A[i]$
    if $A[i] > maxval$
      $maxval \leftarrow A[i]$
  return $maxval - minval$

Answer questions (a)–(e) of Problem 4 about this algorithm.

6. Consider the following algorithm.

ALGORITHM *Enigma*($A[0..n-1, 0..n-1]$)
  //Input: A matrix $A[0..n-1, 0..n-1]$ of real numbers
  for $i \leftarrow 0$ **to** $n-2$ **do**
    for $j \leftarrow i+1$ **to** $n-1$ **do**
      if $A[i, j] \neq A[j, i]$
        return false
  return true

Answer questions (a)–(e) of Problem 4 about this algorithm.

7. Improve the implementation of the matrix multiplication algorithm (see Example 3) by reducing the number of additions made by the algorithm. What effect will this change have on the algorithm's efficiency?

8. Determine the asymptotic order of growth for the total number of times all the doors are toggled in the locker doors puzzle (Problem 11 in Exercises 1.1).

9. Prove the formula

$$\sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2}$$

either by mathematical induction or by following the insight of a 10-year old schoolboy named Karl Friedrich Gauss (1777–1855), who grew up to become one of the greatest mathematicians of all times.

10. Consider the following version of an important algorithm that we will study later in the book.

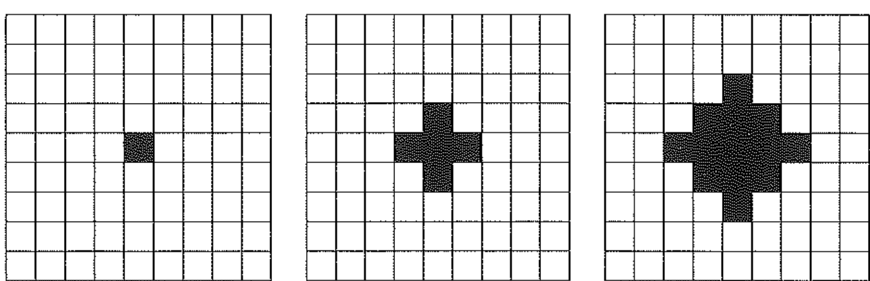
https://hemanthrajhemu.github.io

```
ALGORITHM GE(A[0..n − 1, 0..n])
    //Input: An n-by-n + 1 matrix A[0..n − 1, 0..n] of real numbers
    for i ← 0 to n − 2 do
        for j ← i + 1 to n − 1 do
            for k ← i to n do
                A[j, k] ← A[j, k] − A[i, k] * A[j, i] / A[i, i]
```

a. Find the time efficiency class of this algorithm.

b. What glaring inefficiency does this pseudocode contain and how can it be eliminated to speed the algorithm up?

11. *von Neumann's neighborhood* How many one-by-one squares are generated by the algorithm that starts with a single square and on each of its n iterations adds new squares all round the outside? [Gar99], p. 88. (In the parlance of cellular automata theory, the answer is the number of cells in the von Newmann neighborhood of range n.) The results for $n = 0$, 1, and 2 are illustrated below.



2.4 Mathematical Analysis of Recursive Algorithms

In this section, we will see how to apply the general framework for analysis of algorithms to recursive algorithms. We start with an example often used to introduce novices to the idea of a recursive algorithm.

EXAMPLE 1 Compute the factorial function $F(n) = n!$ for an arbitrary nonnegative integer n. Since

$$n! = 1 \cdot \ldots \cdot (n-1) \cdot n = (n-1)! \cdot n \quad \text{for } n \geq 1$$

and $0! = 1$ by definition, we can compute $F(n) = F(n-1) \cdot n$ with the following recursive algorithm.

https://hemanthrajhemu.github.io

ALGORITHM $F(n)$

//Computes $n!$ recursively
//Input: A nonnegative integer n
//Output: The value of $n!$
if $n = 0$ **return** 1
else return $F(n-1) * n$

For simplicity, we consider n itself as an indicator of this algorithm's input size (rather than the number of bits in its binary expansion). The basic operation of the algorithm is multiplication,[5] whose number of executions we denote $M(n)$. Since the function $F(n)$ is computed according to the formula

$$F(n) = F(n-1) \cdot n \quad \text{for } n > 0,$$

the number of multiplications $M(n)$ needed to compute it must satisfy the equality

$$M(n) = \underset{\substack{\text{to compute} \\ F(n-1)}}{M(n-1)} + \underset{\substack{\text{to multiply} \\ F(n-1) \text{ by } n}}{1} \quad \text{for } n > 0.$$

Indeed, $M(n-1)$ multiplications are spent to compute $F(n-1)$, and one more multiplication is needed to multiply the result by n.

The last equation defines the sequence $M(n)$ that we need to find. This equation defines $M(n)$ not explicitly, i.e., as a function of n, but implicitly as a function of its value at another point, namely $n-1$. Such equations are called ***recurrence relations*** or, for brevity, ***recurrences***. Recurrence relations play an important role not only in analysis of algorithms but also in some areas of applied mathematics. They are usually studied in detail in courses on discrete mathematics or discrete structures; a very brief tutorial on them is provided in Appendix B. Our goal now is to solve the recurrence relation $M(n) = M(n-1) + 1$, i.e., to find an explicit formula for $M(n)$ in terms of n only.

Note, however, that there is not one but infinitely many sequences that satisfy this recurrence. (Can you give examples of, say, two of them?) To determine a solution uniquely, we need an ***initial condition*** that tells us the value with which the sequence starts. We can obtain this value by inspecting the condition that makes the algorithm stop its recursive calls:

if $n = 0$ **return** 1.

This tells us two things. First, since the calls stop when $n = 0$, the smallest value of n for which this algorithm is executed and hence $M(n)$ defined is 0. Second, by inspecting the pseudocode's exiting line, we can see that when $n = 0$, the algorithm performs no multiplications. Thus, the initial condition we are after is

5. Alternatively, we could count the number of times the comparison $n = 0$ is executed, which is the same as counting the total number of calls made by the algorithm (see Problem 2 in Exercises 2.4).

https://hemanthrajhemu.github.io

$$M(0) = 0.$$

the calls stop when $n = 0$ —— no multiplications when $n = 0$

Thus, we succeed in setting up the recurrence relation and initial condition for the algorithm's number of multiplications $M(n)$:

$$M(n) = M(n-1) + 1 \quad \text{for } n > 0, \qquad (2.2)$$
$$M(0) = 0.$$

Before we embark on a discussion of how to solve this recurrence, let us pause to reiterate an important point. We are dealing here with two recursively defined functions. The first is the factorial function $F(n)$ itself; it is defined by the recurrence

$$F(n) = F(n-1) \cdot n \quad \text{for every } n > 0,$$
$$F(0) = 1.$$

The second is the number of multiplications $M(n)$ needed to compute $F(n)$ by the recursive algorithm whose pseudocode was given at the beginning of the section. As we just showed, $M(n)$ is defined by recurrence (2.2). And it is recurrence (2.2) that we need to solve now.

Though it is not difficult to "guess" the solution (what sequence starts with 0 when $n = 0$ and increases by 1 at each step?), it will be more useful to arrive at it in a systematic fashion. Among several techniques available for solving recurrence relations, we use what can be called the ***method of backward substitutions***. The method's idea (and the reason for the name) is immediately clear from the way it applies to solving our particular recurrence:

$$\begin{aligned} M(n) &= M(n-1) + 1 && \text{substitute } M(n-1) = M(n-2) + 1 \\ &= [M(n-2) + 1] + 1 = M(n-2) + 2 && \text{substitute } M(n-2) = M(n-3) + 1 \\ &= [M(n-3) + 1] + 2 = M(n-3) + 3. \end{aligned}$$

After inspecting the first three lines, we see an emerging pattern, which makes it possible to predict not only the next line (what would it be?) but also a general formula for the pattern: $M(n) = M(n-i) + i$. Strictly speaking, the correctness of this formula should be proved by mathematical induction, but it is easier to get the solution as follows and then verify its correctness.

What remains to be done is to take advantage of the initial condition given. Since it is specified for $n = 0$, we have to substitute $i = n$ in the pattern's formula to get the ultimate result of our backward substitutions:

$$M(n) = M(n-1) + 1 = \cdots = M(n-i) + i = \cdots = M(n-n) + n = n.$$

You should not be disappointed after exerting so much effort to get this "obvious" answer. The benefits of the method illustrated in this simple example will become clear very soon, when we have to solve more difficult recurrences.

https://hemanthrajhemu.github.io

Also note that the simple iterative algorithm that accumulates the product of n consecutive integers requires the same number of multiplications, and it does so without the overhead of time and space used for maintaining the recursion's stack.

The issue of time efficiency is actually not that important for the problem of computing $n!$, however. As we saw in Section 2.1, the function's values get so large so fast that we can realistically compute its values only for very small n's. Again, we use this example just as a simple and convenient vehicle to introduce the standard approach to analyzing recursive algorithms. ■

Generalizing our experience with investigating the recursive algorithm for computing $n!$, we can now outline a general plan for investigating recursive algorithms.

General Plan for Analyzing Time Efficiency of Recursive Algorithms

1. Decide on a parameter (or parameters) indicating an input's size.
2. Identify the algorithm's basic operation.
3. Check whether the number of times the basic operation is executed can vary on different inputs of the same size; if it can, the worst-case, average-case, and best-case efficiencies must be investigated separately.
4. Set up a recurrence relation, with an appropriate initial condition, for the number of times the basic operation is executed.
5. Solve the recurrence or at least ascertain the order of growth of its solution.

EXAMPLE 2 As our next example, we consider another educational workhorse of recursive algorithms: the ***Tower of Hanoi*** puzzle. In this puzzle, we (or mythical monks, if you do not like to move disks) have n disks of different sizes and three pegs. Initially, all the disks are on the first peg in order of size, the largest on the bottom and the smallest on top. The goal is to move all the disks to the third peg, using the second one as an auxiliary, if necessary. We can move only one disk at a time, and it is forbidden to place a larger disk on top of a smaller one.

The problem has an elegant recursive solution that is illustrated in Figure 2.4. To move $n > 1$ disks from peg 1 to peg 3 (with peg 2 as auxiliary), we first move recursively $n - 1$ disks from peg 1 to peg 2 (with peg 3 as auxiliary), then move the largest disk directly from peg 1 to peg 3, and, finally, move recursively $n - 1$ disks from peg 2 to peg 3 (using peg 1 as auxiliary). Of course, if $n = 1$, we can simply move the single disk directly from the source peg to the destination peg.

Let us apply the general plan to the Tower of Hanoi problem. The number of disks n is the obvious choice for the input's size indicator, and so is moving one disk as the algorithm's basic operation. Clearly, the number of moves $M(n)$ depends on n only, and we get the following recurrence equation for it:

$$M(n) = M(n-1) + 1 + M(n-1) \quad \text{for } n > 1.$$

With the obvious initial condition $M(1) = 1$, we have the following recurrence relation for the number of moves $M(n)$:

https://hemanthrajhemu.github.io

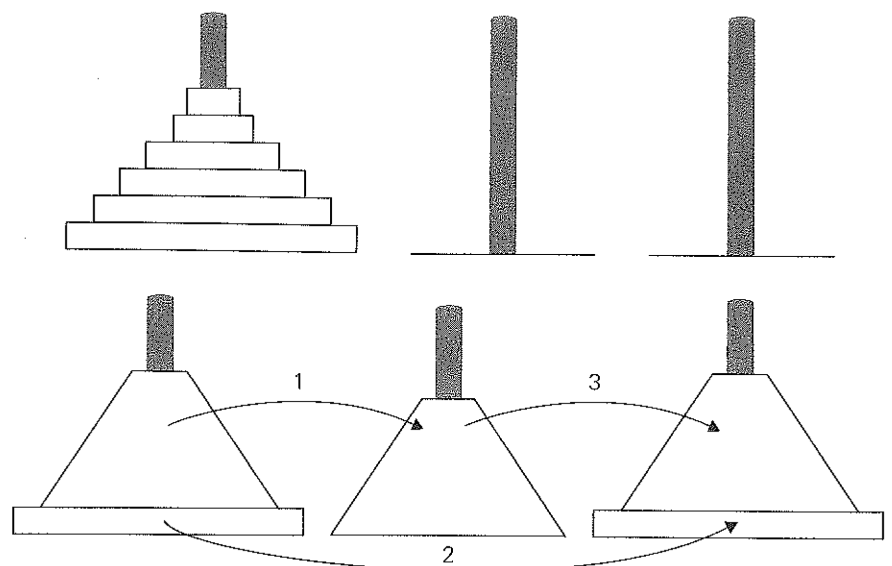


FIGURE 2.4 Recursive solution to the Tower of Hanoi puzzle

$$M(n) = 2M(n-1) + 1 \quad \text{for } n > 1, \tag{2.3}$$
$$M(1) = 1.$$

We solve this recurrence by the same method of backward substitutions:

$$\begin{aligned} M(n) &= 2M(n-1) + 1 && \text{sub. } M(n-1) = 2M(n-2) + 1 \\ &= 2[2M(n-2) + 1] + 1 = 2^2M(n-2) + 2 + 1 && \text{sub. } M(n-2) = 2M(n-3) + 1 \\ &= 2^2[2M(n-3) + 1] + 2 + 1 = 2^3M(n-3) + 2^2 + 2 + 1. \end{aligned}$$

The pattern of the first three sums on the left suggests that the next one will be $2^4M(n-4) + 2^3 + 2^2 + 2 + 1$ and, generally, after i substitutions, we get

$$M(n) = 2^iM(n-i) + 2^{i-1} + 2^{i-2} + \cdots + 2 + 1 = 2^iM(n-i) + 2^i - 1.$$

Since the initial condition is specified for $n = 1$, which is achieved for $i = n - 1$, we get the following formula for the solution to recurrence (2.3):

$$\begin{aligned} M(n) &= 2^{n-1}M(n-(n-1)) + 2^{n-1} - 1 \\ &= 2^{n-1}M(1) + 2^{n-1} - 1 = 2^{n-1} + 2^{n-1} - 1 = 2^n - 1. \end{aligned}$$

Thus, we have an exponential algorithm, which will run for an unimaginably long time even for moderate values of n (see Problem 5 in Exercises 2.4). This is not due to the fact that this particular algorithm is poor; in fact, it is not difficult to prove that this is the most efficient algorithm possible for this problem. It is

https://hemanthrajhemu.github.io

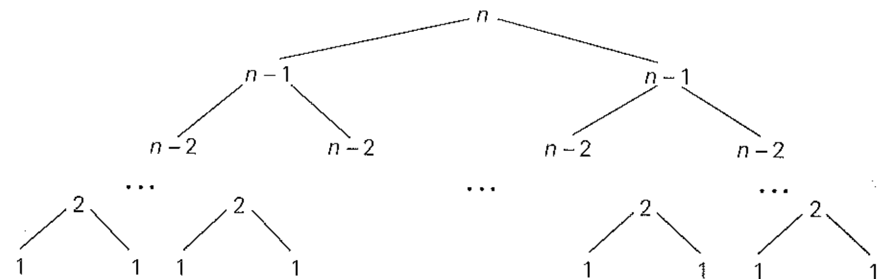


FIGURE 2.5 Tree of recursive calls made by the recursive algorithm for the Tower of Hanoi puzzle

the problem's intrinsic difficulty that makes it so computationally hard. Still, this example makes an important general point:

> One should be careful with recursive algorithms because their succinctness may mask their inefficiency.

When a recursive algorithm makes more than a single call to itself, it is useful for analysis purposes to construct a tree of its recursive calls. In this tree, nodes correspond to recursive calls, and we can label them with the value of the parameter (or, more generally, parameters) of the calls. For the Tower of Hanoi example, the tree is given in Figure 2.5. By counting the number of nodes in the tree, we can get the total number of calls made by the Tower of Hanoi algorithm:

$$C(n) = \sum_{l=0}^{n-1} 2^l \quad \text{(where } l \text{ is the level in the tree above)} = 2^n - 1.$$

The number agrees, as it should, with the move count obtained earlier. ■

EXAMPLE 3 As our next example, we investigate a recursive version of the algorithm discussed at the end of Section 2.3.

ALGORITHM *BinRec*(n)

```
//Input: A positive decimal integer n
//Output: The number of binary digits in n's binary representation
if n = 1 return 1
else return BinRec(⌊n/2⌋) + 1
```

Let us set up a recurrence and an initial condition for the number of additions $A(n)$ made by the algorithm. The number of additions made in computing

https://hemanthrajhemu.github.io

BinRec($\lfloor n/2 \rfloor$) is $A(\lfloor n/2 \rfloor)$, plus one more addition is made by the algorithm to increase the returned value by 1. This leads to the recurrence

$$A(n) = A(\lfloor n/2 \rfloor) + 1 \quad \text{for } n > 1. \tag{2.4}$$

Since the recursive calls end when n is equal to 1 and there are no additions made then, the initial condition is

$$A(1) = 0.$$

The presence of $\lfloor n/2 \rfloor$ in the function's argument makes the method of backward substitutions stumble on values of n that are not powers of 2. Therefore, the standard approach to solving such a recurrence is to solve it only for $n = 2^k$ and then take advantage of the theorem called the ***smoothness rule*** (see Appendix B), which claims that under very broad assumptions the order of growth observed for $n = 2^k$ gives a correct answer about the order of growth for all values of n. (Alternatively, after obtaining a solution for powers of 2, we can sometimes fine-tune this solution to get a formula valid for an arbitrary n.) So let us apply this recipe to our recurrence, which for $n = 2^k$ takes the form

$$A(2^k) = A(2^{k-1}) + 1 \quad \text{for } k > 0,$$
$$A(2^0) = 0.$$

Now backward substitutions encounter no problems:

$$\begin{aligned}
A(2^k) &= A(2^{k-1}) + 1 && \text{substitute } A(2^{k-1}) = A(2^{k-2}) + 1 \\
&= [A(2^{k-2}) + 1] + 1 = A(2^{k-2}) + 2 && \text{substitute } A(2^{k-2}) = A(2^{k-3}) + 1 \\
&= [A(2^{k-3}) + 1] + 2 = A(2^{k-3}) + 3 && \ldots \\
&\ldots \\
&= A(2^{k-i}) + i \\
&\ldots \\
&= A(2^{k-k}) + k.
\end{aligned}$$

Thus, we end up with

$$A(2^k) = A(1) + k = k$$

or, after returning to the original variable $n = 2^k$ and hence $k = \log_2 n$,

$$A(n) = \log_2 n \in \Theta(\log n).$$

In fact, we can prove (Problem 6 in Exercises 2.4) that the exact solution for an arbitrary value of n is given by just a slightly more refined formula $A(n) = \lfloor \log_2 n \rfloor$. ■

This section provides an introduction to analysis of recursive algorithms. These techniques will be used throughout the book and expanded further as necessary. In the next section, we discuss the Fibonacci numbers; their analysis

https://hemanthrajhemu.github.io

involves more difficult recurrence relations to be solved by a method different from backward substitutions.

Exercises 2.4

1. Solve the following recurrence relations.
   a. $x(n) = x(n-1) + 5$ for $n > 1$, $x(1) = 0$
   b. $x(n) = 3x(n-1)$ for $n > 1$, $x(1) = 4$
   c. $x(n) = x(n-1) + n$ for $n > 0$, $x(0) = 0$
   d. $x(n) = x(n/2) + n$ for $n > 1$, $x(1) = 1$ (solve for $n = 2^k$)
   e. $x(n) = x(n/3) + 1$ for $n > 1$, $x(1) = 1$ (solve for $n = 3^k$)
2. Set up and solve a recurrence relation for the number of calls made by $F(n)$, the recursive algorithm for computing $n!$.
3. Consider the following recursive algorithm for computing the sum of the first n cubes: $S(n) = 1^3 + 2^3 + \cdots + n^3$.

   ALGORITHM $S(n)$
   //Input: A positive integer n
   //Output: The sum of the first n cubes
   if $n = 1$ **return** 1
   else return $S(n-1) + n * n * n$

   a. Set up and solve a recurrence relation for the number of times the algorithm's basic operation is executed.
   b. How does this algorithm compare with the straightforward nonrecursive algorithm for computing this sum?
4. Consider the following recursive algorithm.

   ALGORITHM $Q(n)$
   //Input: A positive integer n
   if $n = 1$ **return** 1
   else return $Q(n-1) + 2 * n - 1$

   a. Set up a recurrence relation for this function's values and solve it to determine what this algorithm computes.
   b. Set up a recurrence relation for the number of multiplications made by this algorithm and solve it.
   c. Set up a recurrence relation for the number of additions/subtractions made by this algorithm and solve it.

https://hemanthrajhemu.github.io

5. a. *Tower of Hanoi* In the original version of the Tower of Hanoi puzzle, as it was published by Edouard Lucas, a French mathematician, in the 1890s, the world will end after 64 disks have been moved from a mystical Tower of Brahma. Estimate the number of years it will take if monks could move one disk per minute. (Assume that monks do not eat, sleep, or die.)

b. How many moves are made by the ith largest disk ($1 \leq i \leq n$) in this algorithm?

c. Design a nonrecursive algorithm for the Tower of Hanoi puzzle.

6. a. Prove that the exact number of additions made by the recursive algorithm *BinRec*(n) for an arbitrary positive decimal integer n is $\lfloor \log_2 n \rfloor$.

b. Set up a recurrence relation for the number of additions made by the nonrecursive version of this algorithm (see Section 2.3, Example 4) and solve it.

7. a. Design a recursive algorithm for computing 2^n for any nonnegative integer n that is based on the formula: $2^n = 2^{n-1} + 2^{n-1}$.

b. Set up a recurrence relation for the number of additions made by the algorithm and solve it.

c. Draw a tree of recursive calls for this algorithm and count the number of calls made by the algorithm.

d. Is it a good algorithm for solving this problem?

8. Consider the following recursive algorithm.

```
ALGORITHM  Min1(A[0..n − 1])
    //Input: An array A[0..n − 1] of real numbers
    if n = 1 return A[0]
    else temp ← Min1(A[0..n − 2])
        if temp ≤ A[n − 1] return temp
        else return A[n − 1]
```

a. What does this algorithm compute?

b. Set up a recurrence relation for the algorithm's basic operation count and solve it.

9. Consider another algorithm for solving the problem of Exercise 8, which recursively divides an array into two halves: call *Min2*($A[0..n-1]$) where

```
ALGORITHM  Min2(A[l..r])
    if l = r return A[l]
    else temp1 ← Min2(A[l..⌊(l + r)/2⌋])
        temp2 ← Min2(A[⌊(l + r)/2⌋+1..r])
        if temp1 ≤ temp2 return temp1
        else return temp2
```

https://hemanthrajhemu.github.io

a. Set up a recurrence relation for the algorithm's basic operation count and solve it.

b. Which of the algorithms *Min1* or *Min2* is faster? Can you suggest an algorithm for the problem they solve that would be more efficient than both of them?

10. The determinant of an n-by-n matrix

$$A = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ a_{21} & & a_{2n} \\ \vdots & & \\ a_{n1} & & a_{nn} \end{bmatrix},$$

denoted det A, can be defined as a_{11} for $n = 1$ and, for $n > 1$, by the recursive formula

$$\det A = \sum_{j=1}^{n} s_j a_{1j} \det A_j,$$

where s_j is +1 if j is odd and -1 if j is even, a_{1j} is the element in row 1 and column j, and A_j is the $(n-1)$-by-$(n-1)$ matrix obtained from matrix A by deleting its row 1 and column j.

a. Set up a recurrence relation for the number of multiplications made by the algorithm implementing this recursive definition.

b. Without solving the recurrence, what can you say about the solution's order of growth as compared to $n!$?

11. *von Newmann's neighborhood revisited* Find the number of cells in the von Newmann neighborhood of range n (see Problem 11 in Exercises 2.3) by setting up and solving a recurrence relation.

2.5 Example: Fibonacci Numbers

In this section, we consider the ***Fibonacci numbers***, a famous sequence

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots \tag{2.5}$$

that can be defined by the simple recurrence

$$F(n) = F(n-1) + F(n-2) \quad \text{for } n > 1 \tag{2.6}$$

and two initial conditions

$$F(0) = 0, \quad F(1) = 1. \tag{2.7}$$

The Fibonacci numbers were introduced by Leonardo Fibonacci in 1202 as a solution to a problem about the size of a rabbit population. Many more examples

https://hemanthrajhemu.github.io

of Fibonacci-like numbers have since been discovered in the natural world, and they have even been used in predicting the prices of stocks and commodities. There are some interesting applications of the Fibonacci numbers in computer science as well. For example, worst-case inputs for Euclid's algorithm happen to be consecutive elements of the Fibonacci sequence. Our discussion goals are quite limited here, however. First, we find an explicit formula for the nth Fibonacci number $F(n)$, and then we briefly discuss algorithms for computing it.

Explicit Formula for the *n*th Fibonacci Number

If we try to apply the method of backward substitutions to solve recurrence (2.6), we will fail to get an easily discernible pattern. Instead, let us take advantage of a theorem that describes solutions to a ***homogeneous second-order linear recurrence with constant coefficients***

$$ax(n) + bx(n-1) + cx(n-2) = 0, \qquad \textbf{(2.8)}$$

where a, b, and c are some fixed real numbers ($a \neq 0$) called the coefficients of the recurrence and $x(n)$ is the generic term of an unknown sequence to be found. According to this theorem—see Theorem 1 in Appendix B—recurrence (2.8) has an infinite number of solutions that can be obtained by one of the three formulas. Which of the three formulas applies to a particular case depends on the number of real roots of the quadratic equation with the same coefficients as recurrence (2.8):

$$ar^2 + br + c = 0. \qquad \textbf{(2.9)}$$

Quite logically, equation (2.9) is called the ***characteristic equation*** for recurrence (2.8).

Let us apply this theorem to the case of the Fibonacci numbers. To do so, recurrence (2.6) needs to be rewritten as

$$F(n) - F(n-1) - F(n-2) = 0. \qquad \textbf{(2.10)}$$

Its characteristic equation is

$$r^2 - r - 1 = 0,$$

with the roots

$$r_{1,2} = \frac{1 \pm \sqrt{1 - 4(-1)}}{2} = \frac{1 \pm \sqrt{5}}{2}.$$

Since this characteristic equation has two distinct real roots, we have to use the formula indicated in Case 1 of Theorem 1:

$$F(n) = \alpha \left(\frac{1+\sqrt{5}}{2}\right)^n + \beta \left(\frac{1-\sqrt{5}}{2}\right)^n.$$

https://hemanthrajhemu.github.io

So far, we have ignored initial conditions (2.7). Now we take advantage of them to find specific values of parameters α and β. We do this by substituting 0 and 1—the values of n for which the initial conditions are given—into the last formula and equating the results to 0 and 1 (the values of $F(0)$ and $F(1)$ according to (2.7)), respectively:

$$F(0) = \alpha\left(\tfrac{1+\sqrt{5}}{2}\right)^0 + \beta\left(\tfrac{1-\sqrt{5}}{2}\right)^0 = 0$$

$$F(1) = \alpha\left(\tfrac{1+\sqrt{5}}{2}\right)^1 + \beta\left(\tfrac{1-\sqrt{5}}{2}\right)^1 = 1.$$

After some standard algebraic simplifications, we get the following system of two linear equations in two unknowns α and β:

$$\begin{aligned} \alpha \quad + \quad \beta &= 0 \\ \left(\tfrac{1+\sqrt{5}}{2}\right)\alpha + \left(\tfrac{1-\sqrt{5}}{2}\right)\beta &= 1. \end{aligned}$$

Solving the system (e.g., by substituting $\beta = -\alpha$ into the second equation and solving the equation obtained for α), we get the values $\alpha = 1/\sqrt{5}$ and $\beta = -1/\sqrt{5}$ for the unknowns. Thus,

$$F(n) = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n = \frac{1}{\sqrt{5}}(\phi^n - \hat{\phi}^n), \qquad \textbf{(2.11)}$$

where $\phi = (1+\sqrt{5})/2 \approx 1.61803$ and $\hat{\phi} = -1/\phi \approx -0.61803$.[6] It is hard to believe that formula (2.11), which includes arbitrary integer powers of irrational numbers, yields nothing else but all the elements of Fibonacci sequence (2.5), but it does!

One of the benefits of formula (2.11) is that it immediately implies that $F(n)$ grows exponentially (remember Fibonacci's rabbits?), i.e., $F(n) \in \Theta(\phi^n)$. This follows from the observation that $\hat{\phi}$ is between -1 and 0, and, hence, $\hat{\phi}^n$ gets infinitely small as n goes to infinity. In fact, one can prove that the impact of the second term $\frac{1}{\sqrt{5}}\hat{\phi}^n$ on the value of $F(n)$ can be obtained by rounding off the value of the first term to the nearest integer. In other words, for every nonnegative integer n,

$$F(n) = \frac{1}{\sqrt{5}}\phi^n \text{ rounded to the nearest integer.} \qquad \textbf{(2.12)}$$

Algorithms for Computing Fibonacci Numbers

Though the Fibonacci numbers have many fascinating properties, we limit our discussion to a few remarks about algorithms for computing them. Actually, the

6. Constant ϕ is known as the ***golden ratio***. Since antiquity, it has been considered the most pleasing ratio of a rectangle's two sides to the human eye and might have been consciously used by ancient architects and sculptors.

https://hemanthrajhemu.github.io

sequence grows so fast that it is the size of the numbers rather than a time-efficient method for computing them that should be of primary concern here. Also, for the sake of simplicity, we consider such operations as additions and multiplications at unit cost in the algorithms that follow. Since the Fibonacci numbers grow infinitely large (and grow rapidly), a more detailed analysis than the one offered here is warranted. These caveats notwithstanding, the algorithms we outline and their analysis are useful examples for a student of the design and analysis of algorithms.

To begin with, we can use recurrence (2.6) and initial condition (2.7) for the obvious recursive algorithm for computing $F(n)$.

ALGORITHM $F(n)$

```
//Computes the nth Fibonacci number recursively by using its definition
//Input: A nonnegative integer n
//Output: The nth Fibonacci number
if n ≤ 1 return n
else return F(n − 1) + F(n − 2)
```

Before embarking on its formal analysis, can you tell whether this is an efficient algorithm? Well, we need to do a formal analysis anyway. The algorithm's basic operation is clearly addition, so let $A(n)$ be the number of additions performed by the algorithm in computing $F(n)$. Then the numbers of additions needed for computing $F(n-1)$ and $F(n-2)$ are $A(n-1)$ and $A(n-2)$, respectively, and the algorithm needs one more addition to compute their sum. Thus, we get the following recurrence for $A(n)$:

$$A(n) = A(n-1) + A(n-2) + 1 \quad \text{for } n > 1, \qquad \textbf{(2.13)}$$
$$A(0) = 0, \;\; A(1) = 0.$$

The recurrence $A(n) - A(n-1) - A(n-2) = 1$ is quite similar to recurrence (2.10), but its right-hand side is not equal to zero. Such recurrences are called ***inhomogeneous recurrences***. There are general techniques for solving inhomogeneous recurrences (see Appendix B or any textbook on discrete mathematics), but for this particular recurrence, a special trick leads to a faster solution. We can reduce our inhomogeneous recurrence to a homogeneous one by rewriting it as

$$[A(n) + 1] - [A(n-1) + 1] - [A(n-2) + 1] = 0$$

and substituting $B(n) = A(n) + 1$:

$$B(n) - B(n-1) - B(n-2) = 0$$
$$B(0) = 1, \quad B(1) = 1.$$

This homogeneous recurrence can be solved exactly in the same manner as recurrence (2.10) was solved to find an explicit formula for $F(n)$. But it can actually be avoided by noting that $B(n)$ is, in fact, the same recurrence as $F(n)$ except that it starts with two ones and thus runs one step ahead of $F(n)$. So $B(n) = F(n+1)$,

https://hemanthrajhemu.github.io

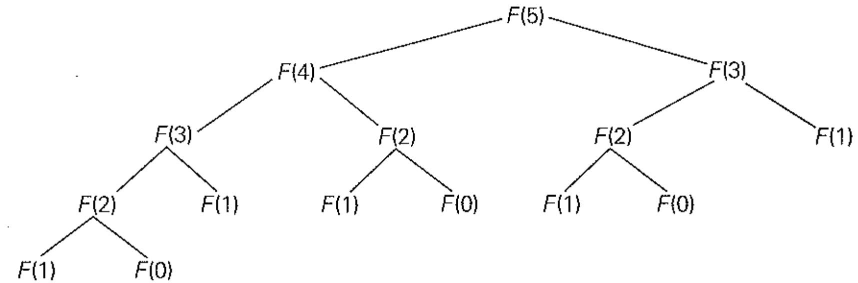


FIGURE 2.6 Tree of recursive calls for computing the 5th Fibonacci number by the definition-based algorithm

and

$$A(n) = B(n) - 1 = F(n+1) - 1 = \frac{1}{\sqrt{5}}(\phi^{n+1} - \hat{\phi}^{n+1}) - 1.$$

Hence, $A(n) \in \Theta(\phi^n)$ and, if we measure the size of n by the number of bits $b = \lfloor \log_2 n \rfloor + 1$ in its binary representation, the efficiency class will be even worse, namely doubly exponential.

The poor efficiency class of the algorithm could be anticipated by the nature of recurrence (2.13). Indeed, it contains two recursive calls with the sizes of smaller instances only slightly smaller than size n. (Have you encountered such a situation before?) We can also see the reason behind the algorithm's inefficiency by looking at a recursive tree of calls tracing the algorithm's execution. An example of such a tree for $n = 5$ is given in Figure 2.6. Note that the same values of the function are being evaluated again and again, which is clearly extremely inefficient.

We can obtain a much faster algorithm by simply computing the successive elements of the Fibonacci sequence iteratively, as is done in the following algorithm.

ALGORITHM *Fib*(n)

```
//Computes the nth Fibonacci number iteratively by using its definition
//Input: A nonnegative integer n
//Output: The nth Fibonacci number
F[0] ← 0; F[1] ← 1
for i ← 2 to n do
    F[i] ← F[i − 1] + F[i − 2]
return F[n]
```

This algorithm clearly makes $n - 1$ additions. Hence, it is linear as a function of n and "only" exponential as a function of the number of bits b in n's binary representation. Note that using an extra array for storing all the preced-

https://hemanthrajhemu.github.io

ing elements of the Fibonacci sequence can be avoided: storing just two values is necessary to accomplish the task (see Problem 6 in Exercises 2.5).

The third alternative for computing the nth Fibonacci number lies in using formula (2.12). The efficiency of the algorithm will obviously be determined by the efficiency of an exponentiation algorithm used for computing ϕ^n. If it is done by simply multiplying ϕ by itself $n-1$ times, the algorithm will be in $\Theta(n) = \Theta(2^b)$. There are faster algorithms for the exponentiation problem. For example, we discuss $\Theta(\log n) = \Theta(b)$ algorithms for this problem in Chapters 5 and 6. Note also that special care should be exercised in implementing this approach to computing the nth Fibonacci number. Since all its intermediate results are irrational numbers, we would have to make sure that their approximations in the computer are accurate enough so that the final round-off yields a correct result.

Finally, there exists a $\Theta(\log n)$ algorithm for computing the nth Fibonacci number that manipulates only integers. It is based on the equality

$$\begin{bmatrix} F(n-1) & F(n) \\ F(n) & F(n+1) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n \quad \text{for } n \geq 1$$

and an efficient way of computing matrix powers.

Exercises 2.5

1. Find a Web site dedicated to applications of the Fibonacci numbers and study it.
2. Check by direct substitutions that the function $\frac{1}{\sqrt{5}}(\phi^n - \hat{\phi}^n)$ indeed satisfies recurrence (2.6) for every $n > 1$ and initial conditions (2.7) for $n = 0$ and 1.
3. The maximum values of the Java primitive types `int` and `long` are $2^{31} - 1$ and $2^{63} - 1$, respectively. Find the smallest n for which the nth Fibonacci number is not going to fit in a memory allocated for

   a. the type `int`. **b.** the type `long`.

4. *Climbing stairs* Find the number of different ways to climb an n-stair staircase if each step is either one or two stairs. (For example, a 3-stair staircase can be climbed three ways: 1-1-1, 1-2, and 2-1.) [Tuc80], p. 112
5. Consider the recursive definition-based algorithm for computing the nth Fibonacci number $F(n)$. Let $C(n)$ and $Z(n)$ be the number of times $F(1)$ and $F(0)$, respectively, are computed. Prove that

   a. $C(n) = F(n)$ **b.** $Z(n) = F(n-1)$.
6. Improve algorithm *Fib* so that it requires only $\Theta(1)$ space.
7. Prove the equality

$$\begin{bmatrix} F(n-1) & F(n) \\ F(n) & F(n+1) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n \quad \text{for } n \geq 1.$$

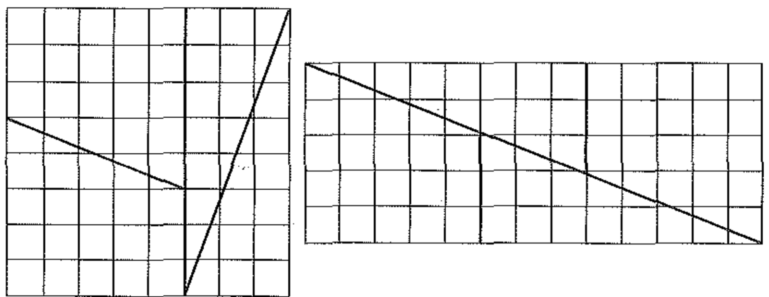
https://hemanthrajhemu.github.io

8. How many modulo divisions are made by Euclid's algorithm on two consecutive Fibonacci numbers $F(n)$ and $F(n-1)$ as the algorithm's input?

9. a. Prove ***Cassini's identity***:

$$F(n+1)F(n-1) - [F(n)]^2 = (-1)^n \quad \text{for } n \geq 1.$$

b. *Disappearing square* Consider the following paradox, which is based on Cassini's identity. Take an 8-by-8 chessboard (more generally, any $F(n)$-by-$F(n)$ board divided into $[F(n)]^2$ squares). Cut it into two trapezoids and two triangles as shown in the left portion of the figure below. Then reassemble it as shown in the right portion of the figure. The area of the left rectangle is $8 \times 8 = 64$ squares, while the area of the right rectangle is $13 \times 5 = 65$ squares. Explain the paradox.



10. In the language of your choice, implement two algorithms for computing the last five digits of the nth Fibonacci number that are based on (a) the recursive definition-based algorithm $F(n)$; (b) the iterative definition-based algorithm $Fib(n)$. Perform an experiment to find the largest value of n for which your programs run under 1 minute on your computer.

2.6 Empirical Analysis of Algorithms

In Sections 2.3 and 2.4, we saw how algorithms, both nonrecursive and recursive, can be analyzed mathematically. Though these techniques can be applied successfully to many simple algorithms, the power of mathematics, even when enhanced with more advanced techniques (see [Sed96], [Pur85], [Gra94], and [Gre82]), is far from limitless. In fact, even some seemingly simple algorithms have proved to be very difficult to analyze with mathematical precision and certainty. As we pointed out in Section 2.2, this is especially true for average-case analysis.

https://hemanthrajhemu.github.io

The principal alternative to the mathematical analysis of an algorithm's efficiency is its empirical analysis. This approach implies the steps spelled out in the following plan.

General Plan for Empirical Analysis of Algorithm Time Efficiency

1. Understand the experiment's purpose.
2. Decide on the efficiency metric M to be measured and the measurement unit (an operation's count vs. a time unit).
3. Decide on characteristics of the input sample (its range, size, and so on).
4. Prepare a program implementing the algorithm (or algorithms) for the experimentation.
5. Generate a sample of inputs.
6. Run the algorithm (or algorithms) on the sample's inputs and record the data observed.
7. Analyze the data obtained.

Let us discuss these steps one at a time. There are several different goals one can pursue in analyzing algorithms empirically. They include checking the accuracy of a theoretical assertion about the algorithm's efficiency, comparing the efficiency of several algorithms for solving the same problem or different implementations of the same algorithm, developing a hypothesis about the algorithm's efficiency class, and ascertaining the efficiency of the program implementing the algorithm on a particular machine. Obviously, an experiment's design should depend on the question the experimenter seeks to answer.

In particular, the experiment's goal should influence, if not dictate, how the algorithm's efficiency is to be measured. The first alternative is to insert a counter (or counters) into a program implementing the algorithm to count the number of times the algorithm's basic operation is executed. This is usually a straightforward operation; you should only be mindful of the possibility that the basic operation is located in several places in the program and that all its executions need to be accounted for. As straightforward as this task usually is, you should always test the modified program to ensure that it works correctly, in terms of both the problem it solves and the counts it yields.

The second alternative is to time the program implementing the algorithm in question. The easiest way to do this is to use a system's command, such as the `time` command in UNIX. Alternatively, we can measure the running time of a code fragment by asking for the system time right before the fragment's start (t_{start}) and just after its completion (t_{finish}), and then computing the difference between the two ($t_{finish} - t_{start}$).[7] In C and C++, you can use the function `clock`

7. If the system time is given in units called "ticks," the difference should be divided by a constant indicating the number of ticks per time unit.

https://hemanthrajhemu.github.io

for this purpose; in Java, the method `currentTimeMillis()` in the `System` class is available.

It is important to keep several facts in mind, however. First, a system's time is typically not very accurate, and you might get somewhat different results on repeated runs of the same program on the same inputs. An obvious remedy is to make several such measurements and then take their average (or the median) as the sample's observation point. Second, given the high speed of modern computers, the running time may fail to register at all and be reported as zero. The standard trick to overcome this obstacle is to run the program in an extra loop many times, measure the total running time, and then divide it by the number of the loop's repetitions. Third, on a computer running under a time-sharing system (such as UNIX), the reported time may include the time spent by the CPU on other programs, which obviously defeats the purpose of the experiment. Therefore, you should take care to ask the system for the time devoted specifically to execution of your program. (In UNIX, this time is called the "user time," and it is automatically provided by the `time` command.)

Thus, measuring the physical running time has several disadvantages, both fundamental (dependence on a particular machine being the most important of them) and technical, not shared by counting the executions of a basic operation. On the other hand, the physical running time provides very specific information about an algorithm's performance in a particular computing environment, which can be of more importance to the experimenter than, say, the algorithm's asymptotic efficiency class. In addition, measuring time spent on different segments of a program can pinpoint a bottleneck in the program's performance that can be missed by an abstract deliberation about the algorithm's basic operation. Getting such data—called ***profiling***—is an important resource in the empirical analysis of an algorithm's running time; the data in question can usually be obtained from the system tools available in most computing environments.

Whether you decide to measure the efficiency by basic operation counting or by time clocking, you will need to decide on a sample of inputs for the experiment. Often, the goal is to use a sample representing a "typical" input; so the challenge is to understand what a "typical" input is. For some classes of algorithms—e.g., algorithms for the traveling salesman problem discussed later in the book—researchers have developed a set of instances they use for benchmarking. But much more often than not, an input sample has to be developed by the experimenter. Typically, you will have to make decisions about the sample size (it is sensible to start with a relatively small sample and increase it later if necessary), the range of input sizes in your sample (typically neither trivially small nor excessively large), and a procedure for generating inputs in the range chosen. The instance sizes can either adhere to some pattern (e.g., 1000, 2000, 3000, . . . , 10,000 or 500, 1000, 2000, 4000, . . . , 128000) or be generated randomly within the range chosen.

The principal advantage of size changing according to a pattern is that its impact is easier to analyze. For example, if a sample's sizes are generated by doubling, we can compute the ratios $M(2n)/M(n)$ of the observed metric M (the count or

https://hemanthrajhemu.github.io

the time) and see whether the ratios exhibit a behavior typical of algorithms in one of the basic efficiency classes (see Section 2.2). The major disadvantage of nonrandom sizes is the possibility that the algorithm under investigation exhibits atypical behavior on the sample chosen. For example, if all the sizes in a sample are even and an algorithm under investigation runs much more slowly on odd-size inputs, the empirical results will be quite misleading.

Another important issue concerning sizes in an experiment's sample is whether several instances of the same size should be included. If you expect the observed metric to vary considerably on instances of the same size, it is probably wise to include several instances for every size in the sample. (There are well-developed methods in statistics to help the experimenter make such decisions; you will find no shortage of books on this subject.) Of course, if several instances of the same size are included in the sample, the averages or medians of the observed values for each size should be computed and investigated instead of or in addition to individual sample points.

Much more often than not, an empirical analysis of an algorithm's efficiency requires generating random numbers. Even if we decide to use a pattern for input sizes, we typically want instances themselves generated randomly. Generating random numbers on a digital computer is known to present a difficult problem because, in principle, the problem can be solved only approximately. This is the reason computer scientists prefer to call such numbers ***pseudorandom***. As a practical matter, the easiest and most natural way of getting such numbers is to take advantage of a random number generator available in computer language libraries. Typically, its output will be a value of a (pseudo)random variable uniformly distributed in the interval between 0 and 1. If a different (pseudo)random variable is desired, an appropriate transformation needs to be made. For example, if x is a continuous random variable uniformly distributed on the interval $0 \le x < 1$, the variable $y = l + \lfloor x(r - l) \rfloor$ will be uniformly distributed among the integer values between integers l and $r - 1$ $(l < r)$.

Alternatively, you can implement one of several known algorithms for generating (pseudo)random numbers. The most widely used and thoroughly studied of such algorithms is the ***linear congruential method***.

ALGORITHM *Random(n, m, seed, a, b)*

//Generates a sequence of n pseudorandom numbers according to the linear
//congruential method
//Input: A positive integer n and positive integer parameters $m, seed, a, b$
//Output: A sequence $r_1, \ldots, r_n$ of n pseudorandom integers uniformly
// distributed among integer values between 0 and $m - 1$
//Note: Pseudorandom numbers between 0 and 1 can be obtained
// by treating the integers generated as digits after the decimal point
$r_0 \leftarrow seed$
for $i \leftarrow 1$ **to** n **do**
  $r_i \leftarrow (a * r_{i-1} + b) \bmod m$

https://hemanthrajhemu.github.io

The simplicity of the algorithm's pseudocode is misleading because the devil lies in the details of choosing the algorithm's parameters. Here is a partial list of recommendations based on the results of a sophisticated mathematical analysis (see [KnuII], pp. 184–185, for details): *seed* may be chosen arbitrarily and is often set to the current date and time; m should be large and may be conveniently taken as 2^w, where w is the computer's word size; a should be selected as an integer between $0.01m$ and $0.99m$ with no particular pattern in its digits but such that $a \bmod 8 = 5$; the value of b can be chosen as 1.

The empirical data obtained as the result of an experiment need to be recorded and then presented for an analysis. Data can be presented numerically in a table or graphically in a ***scatterplot***, that is by points in a Cartesian coordinate system. It is a good idea to use both these options whenever it is feasible because both methods have their unique strengths and weaknesses.

The principal advantage of tabulated data lies in the opportunity to manipulate it easily. For example, we can compute the ratios $M(n)/g(n)$ where $g(n)$ is a candidate to represent the efficiency class of the algorithm in question. If the algorithm is indeed in $\Theta(g(n))$, most likely these ratios will converge to some positive constant as n gets large. (Note that careless novices sometimes assume that this constant must be 1, which is, of course, incorrect according to the definition of $\Theta(g(n))$.) Or we can compute the ratios $M(2n)/M(n)$ and see how the running time reacts to doubling of its input size. As we discussed in Section 2.2, such ratios should change only slightly for logarithmic algorithms and most likely converge to 2, 4, and 8 for linear, quadratic, and cubic algorithms, respectively—to name the most obvious and convenient cases.

On the other hand, the form of a scatterplot may also help in ascertaining the algorithm's probable efficiency class. For a logarithmic algorithm, the scatterplot will have a concave shape (Figure 2.7a); this fact distinguishes it from all the other basic efficiency classes. For a linear algorithm, the points will tend to aggregate around a straight line or, more generally, to be contained between two straight lines (Figure 2.7b). Scatterplots of functions in $\Theta(n \lg n)$ and $\Theta(n^2)$ will have a convex shape (Figure 2.7c), making them difficult to differentiate. A scatterplot of a cubic algorithm will also have a convex shape, but it will show a much more rapid increase in the metric's values. An exponential algorithm will most probably require a logarithmic scale for the vertical axis, in which the values of $\log_a M(n)$ rather than those of $M(n)$ are plotted. (The commonly used logarithm base is 2 or 10.) In such a coordinate system, a scatterplot of a truly exponential algorithm should resemble a linear function because $M(n) \approx ca^n$ implies $\log_b M(n) \approx \log_b c + n \log_b a$ and vise versa.

One of the possible applications of the empirical analysis is to predict the algorithm's performance on an instance not included in the experiment sample. For example, if we observe that the ratios $M(n)/g(n)$ are close to some constant c for the sample instances, we can approximate $M(n)$ by the product $cg(n)$ for other instances, too. Though this approach is sensible, it should be used with caution, especially for values of n outside the sample range. (Mathematicians call such

https://hemanthrajhemu.github.io

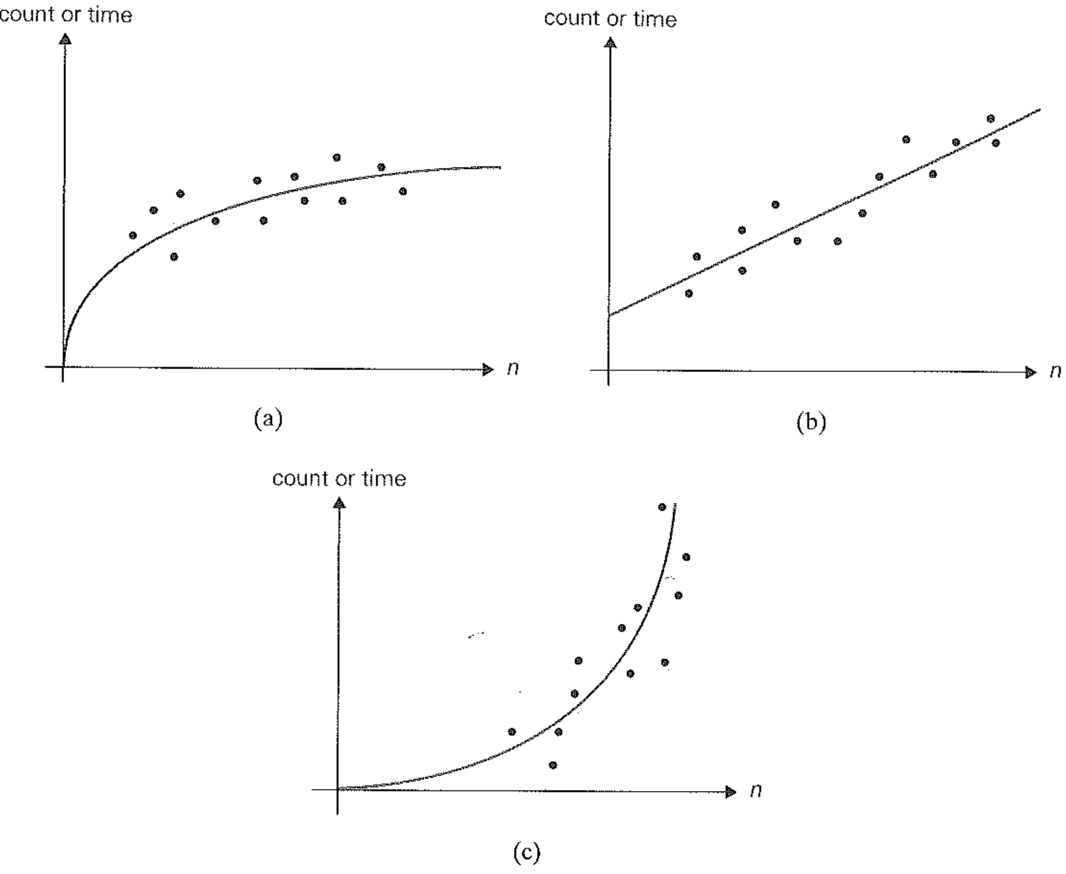


FIGURE 2.7 Typical scatterplots: (a) logarithmic; (b) linear; (c) one of the convex functions

predictions ***extrapolation***, as opposed to ***interpolation***, which deals with values within the sample range.) Of course, we can also try unleashing the standard techniques of statistical data analysis and prediction. Note, however, that the majority of such techniques are based on specific probabilistic assumptions that may or may not be valid for the experimental data in question.

It seems appropriate to end this section by pointing out the basic differences between mathematical and empirical analyses of algorithms. The principal strength of the mathematical analysis is its independence of specific inputs; its principal weakness is its limited applicability, especially for investigating the average-case efficiency. The principal strength of the empirical analysis lies in its applicability to any algorithm, but its results can depend on the particular sample of instances and the computer used in the experiment.

https://hemanthrajhemu.github.io

Exercises 2.6

1. Consider the following well-known sorting algorithm (we study it more closely later in the book) with a counter inserted to count the number of key comparisons.

```
ALGORITHM SortAnalysis(A[0..n − 1])
    //Input: An array A[0..n − 1] of n orderable elements
    //Output: The total number of key comparisons made
    count ← 0
    for i ← 1 to n − 1 do
        v ← A[i]
        j ← i − 1
        while j ≥ 0 and A[j] > v do
            count ← count + 1
            A[j + 1] ← A[j]
            j ← j − 1
        A[j + 1] ← v
    return count
```

   Is the comparison counter inserted in the right place? If you believe it is, prove it; if you believe it is not, make an appropriate correction.

2. **a.** Run the program of Problem 1, with a properly inserted counter (or counters) for the number of key comparisons, on 20 random arrays of sizes 1000, 1500, 2000, 2500, . . . , 9000, 9500.

   b. Analyze the data obtained to form a hypothesis about the algorithm's average-case efficiency.

   c. Estimate the number of key comparisons one should expect for a randomly generated array of size 10,000 sorted by the same algorithm.

3. Repeat Problem 2 by measuring the program's running time in milliseconds.

4. Hypothesize a likely efficiency class of an algorithm based on the following empirical observations of its basic operation's count:

size	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
count	11,966	24,303	39,992	53,010	67,272	78,692	91,274	113,063	129,799	140,538

5. What scale transformation will make a logarithmic scatterplot look like a linear one?

6. How can we distinguish a scatterplot for an algorithm in $\Theta(\lg \lg n)$ from a scatterplot for an algorithm in $\Theta(\lg n)$?

https://hemanthrajhemu.github.io

7. **a.** Find empirically the largest number of divisions made by Euclid's algorithm for computing gcd(m, n) for $1 \le n \le m \le 100$.
   b. For each positive integer k, find empirically the smallest pair of integers $1 \le n \le m \le 100$ for which Euclid's algorithm needs to make k divisions in order to find gcd(m, n).
8. The average-case efficiency of Euclid's algorithm on inputs of size n can be measured by the average number of divisions $D_{avg}(n)$ made by the algorithm in computing gcd(n, 1), gcd(n, 2), ..., gcd(n, n). For example,

$$D_{avg}(5) = \frac{1}{5}(1 + 2 + 3 + 2 + 1) = 1.8.$$

   Produce a scatterplot of $D_{avg}(n)$ and indicate the algorithm's likely average-case efficiency class.
9. Run an experiment to ascertain the efficiency class of the sieve of Eratosthenes (see Section 1.1).
10. Run a timing experiment for the three algorithms for computing gcd(m, n) presented in Section 1.1.

2.7 Algorithm Visualization

In addition to the mathematical and empirical analyses of algorithms, there is yet a third way to study algorithms. It is called ***algorithm visualization*** and can be defined as the use of images to convey some useful information about algorithms. That information can be a visual illustration of an algorithm's operation, of its performance on different kinds of inputs, or of its execution speed versus that of other algorithms for the same problem. To accomplish this goal, an algorithm visualization uses graphic elements (points, line segments, two- or three-dimensional bars, and so on) to represent some "interesting events" in the algorithm's operation.

There are two principal variations of algorithm visualization:

- static algorithm visualization
- dynamic algorithm visualization, also called ***algorithm animation***

Static algorithm visualization shows an algorithm's progress through a series of still images. Algorithm animation, on the other hand, shows a continuous, movie-like presentation of an algorithm's operations. Animation is an arguably more sophisticated option, and it is, of course, much more difficult to implement.

Early efforts in the area of algorithm visualization go back to the 1970s. The watershed event happened in 1981 with the appearance of a 30-minute color sound film titled *Sorting Out Sorting*. The algorithm visualization classic was produced at the University of Toronto by Ronald Baecker with the assistance of D. Sherman [Bae81, Bae98]. It contained visualizations of nine well-known sorting algorithms

https://hemanthrajhemu.github.io

(more than half of them are discussed later in the book) and provided quite a convincing demonstration of their relative speeds.

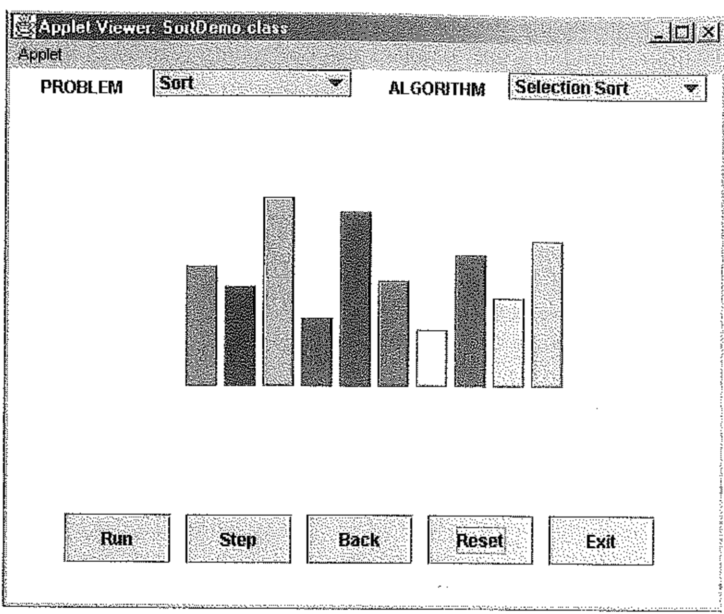
The success of *Sorting Out Sorting* made sorting algorithms a perennial favorite for algorithm animation. Indeed, the sorting problem lends itself quite naturally to visual presentation via vertical or horizontal bars or sticks of different heights or lengths, which are rearranged according to their sizes (Figure 2.8). This presentation is convenient, however, only for illustrating actions of a typical sorting algorithm on small inputs. For larger files, *Sorting Out Sorting* used the ingenious idea of presenting data by a scatterplot of points on a coordinate plane, with the first coordinate representing an item's position in the file and the second one representing the item's value; with such a representation, the process of sorting looks like a transformation of a "random" scatterplot of points into the points along a frame's diagonal (Figure 2.9). In addition, most sorting algorithms work by comparing and exchanging two given items at a time—an event that can be animated relatively easily.

Since the appearance of *Sorting Out Sorting*, a great number of algorithm animations have been created. They range in scope from one particular algorithm to a group of algorithms for the same problem (e.g., sorting) or the same application area (e.g., geometric algorithms) to general-purpose animation systems. The most widely known of the general-purpose systems include BALSA [Bro84], TANGO [Sta90], and ZEUS [Bro91]; a comparative review of their features, along with those of nine other packages, can be found in [Pri93]. A good general-purpose animation system should allow a user to not only watch and interact with existing animations of a wide variety of algorithms; it should also provide facilities for creating new animations. Experience has shown that creating such systems is a difficult but not impossible task.

The appearance of Java and the World Wide Web has given a new impetus to algorithm animation. You are advised to start an exploration with an up-to-date site containing a collection of links to sites devoted to algorithm animation. Since the Web world is notorious for its instability, no specific Web addresses appear here; a search for the phrase "algorithm animation" or "algorithm visualization" with a good search engine should do the trick. While you peruse and evaluate different algorithm animations, you may want to keep in mind the "ten commandments of algorithm animations." This list of desirable features of an animation's user interface was suggested by Peter Gloor [Glo98], who was a principal developer of Animated Algorithms, another well-known algorithm visualization system:

1. Be consistent.
2. Be interactive.
3. Be clear and concise.
4. Be forgiving to the user.
5. Adapt to the knowledge level of the user.

https://hemanthrajhemu.github.io



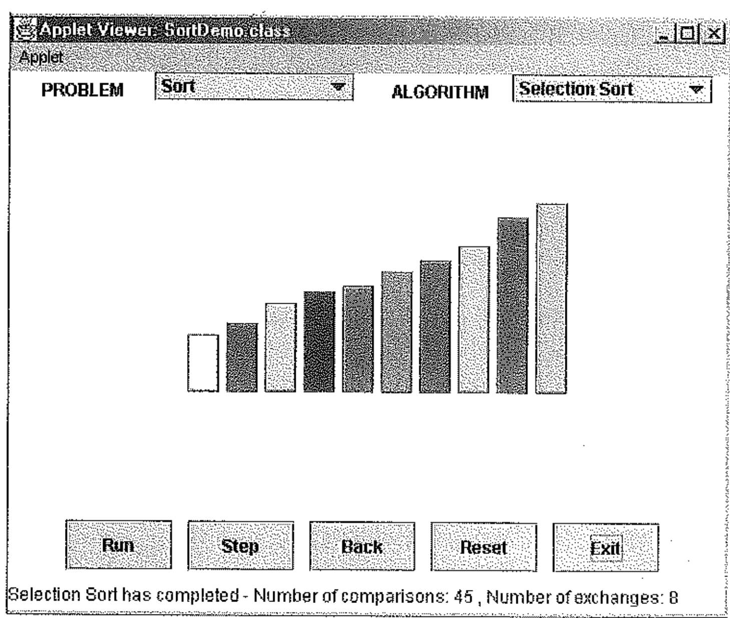
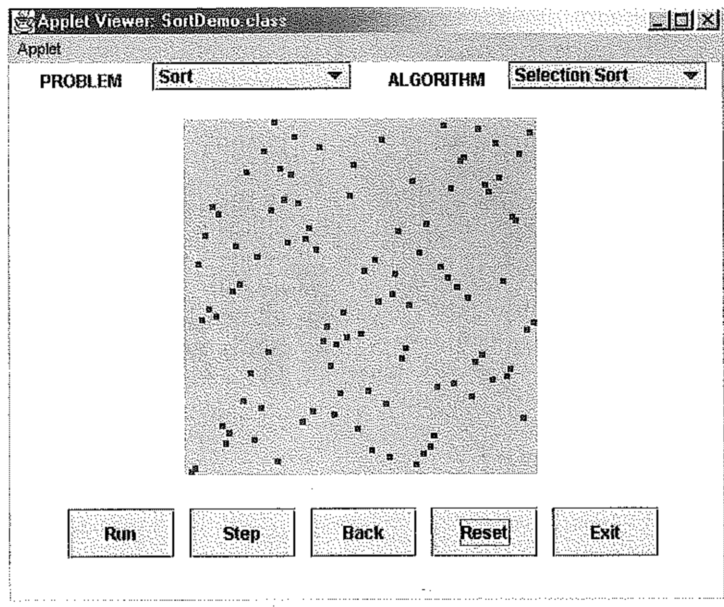


FIGURE 2.8 Initial and final screens of a typical visualization of a sorting algorithm using the bar representation

https://hemanthrajhemu.github.io



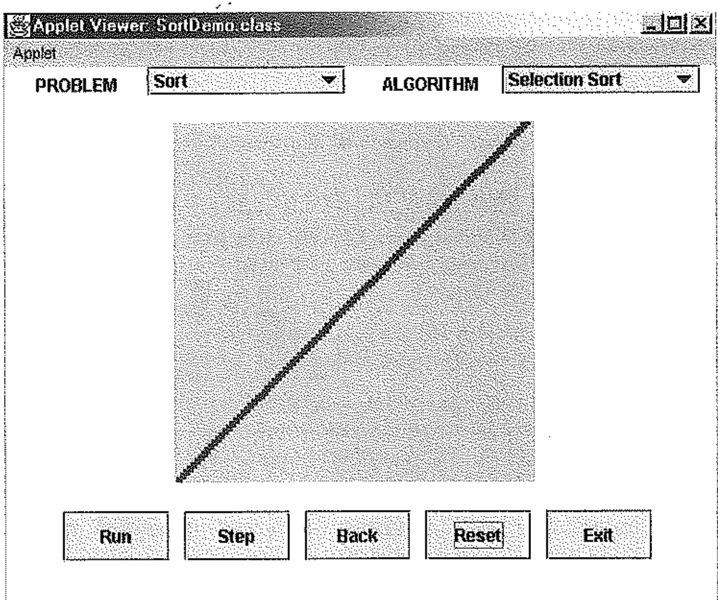


FIGURE 2.9 Initial and final screens of a typical visualization of a sorting algorithm using the scatterplot representation

https://hemanthrajhemu.github.io

6. Emphasize the visual component.
7. Keep the user interested.
8. Incorporate both symbolic and iconic representations.
9. Include algorithm's analysis (run statistics) and comparisons with other algorithms for the same problem.
10. Include execution history.

There are two principal applications of algorithm visualization: research and education. The application to education seeks to help students learning algorithms. Potential benefits for researchers are based on expectations that algorithm visualization may help uncover some unknown features of algorithms. For example, one researcher used a visualization of the recursive Tower of Hanoi algorithm in which odd- and even-numbered disks were two different colors. He noticed that two disks of the same color never came in direct contact during the algorithm's execution. This observation helped him in developing a better nonrecursive version of the classic algorithm.

Although some successes in both education and research applications have been reported, they are not as impressive as one might expect. Experience has shown that creating sophisticated software systems is not going to be enough. A deeper understanding of human perception of images will be required before the true potential of algorithm visualization is fulfilled.

SUMMARY

- There are two kinds of algorithm efficiency: time efficiency and space efficiency. *Time efficiency* indicates how fast the algorithm runs; *space efficiency* deals with the extra space it requires.

- An algorithm's time efficiency is principally measured as a function of its input size by counting the number of times its basic operation is executed. A *basic operation* is the operation that contributes most toward running time. Typically, it is the most time-consuming operation in the algorithm's innermost loop.

- For some algorithms, the running time may differ considerably for inputs of the same size, leading to *worst-case* efficiency, *average-case* efficiency, and *best-case* efficiency.

- The established framework for analyzing an algorithm's time efficiency is primarily grounded in the order of growth of the algorithm's running time as its input size goes to infinity.

- The notations O, Ω, and Θ are used to indicate and compare the asymptotic orders of growth of functions expressing algorithm efficiencies.

https://hemanthrajhemu.github.io

- The efficiencies of a large number of algorithms fall into the following few classes: *constant*, *logarithmic*, *linear*, *"n-log-n,"* *quadratic*, *cubic*, and *exponential*.
- The main tool for analyzing the time efficiency of a nonrecursive algorithm is to set up a sum expressing the number of executions of its basic operation and ascertain the sum's order of growth.
- The main tool for analyzing the time efficiency of a recursive algorithm is to set up a recurrence relation expressing the number of executions of its basic operation and ascertain the solution's order of growth.
- Succinctness of a recursive algorithm may mask its inefficiency.
- The *Fibonacci numbers* are an important sequence of integers in which every element is equal to the sum of its two immediate predecessors. There are several algorithms for computing the Fibonacci numbers with drastically different efficiencies.
- Empirical analysis of an algorithm is performed by running a program implementing the algorithm on a sample of inputs and analyzing the data observed (the basic operation's count or physical running time). This often involves generating pseudorandom numbers. The applicability to any algorithm is the principal strength of this approach; the dependence of results on the particular computer and instance sample is its main weakness.
- *Algorithm visualization* is the use of images to convey useful information about algorithms. The two principal variations of algorithm visualization are static algorithm visualization and dynamic algorithm visualization (also called *algorithm animation*).

https://hemanthrajhemu.github.io