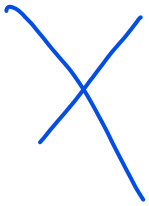
Chapter 1

(Introduction)



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 just 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

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, investigation of these three algorithms leads to some general observations about several important properties of algorithms in general.

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., [Sed11]) 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.

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 the 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.
- The range of inputs for which an algorithm works has to be specified carefully.
- The same algorithm can be represented in several different ways.
- There may exist several algorithms for solving the same problem.

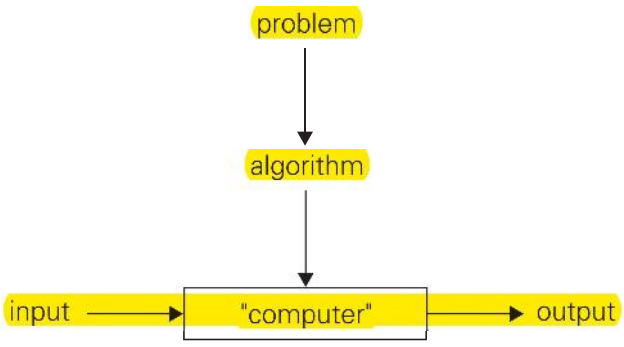


FIGURE 1.1 The notion of the algorithm.

- 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 6 in this section's exercises.)

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 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 integer of the pair gets smaller with 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 (why?). Hence, the value of the second integer 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 set 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

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

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 math teacher did not explain how to obtain such a list. 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 $n > 1$. 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 its first iteration, the algorithm 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. 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.

What is the largest number p whose multiples can still remain on the list to make further iterations of the algorithm necessary? 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 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 $p \cdot p \leq n$ as the loop continuation condition there.

ALGORITHM *Sieve*(n)

//Implements the sieve of Eratosthenes
//Input: A positive integer $n > 1$
//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.

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 from an algorithm's description.
   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. Design an algorithm to find all the common elements in two sorted lists of numbers. For example, for the lists 2, 5, 5, 5 and 2, 2, 3, 5, 5, 7, the output should be 2, 5, 5. What is the maximum number of comparisons your algorithm makes if the lengths of the two given lists are m and n, respectively?

6. **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 $\gcd(m, n)$.

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

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

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

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

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

    b. *Euclid's game* (see [Bog]) starts with two unequal positive integers 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?

11. 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 $mx + ny = d$.

    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 to find integer solutions to the Diophantine equation $ax + by = c$ with any set of integer coefficients a, b, and c.

12. *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, $i = 1, 2, \ldots, n$, you toggle the door of every ith locker: if the door is closed, you open it; if it is open, you close it. 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 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 set of instances the algorithm needs to handle. (As an example, recall the variations in the set 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; otherwise, you will run the risk of unnecessary rework.

Ascertaining the Capabilities of the 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

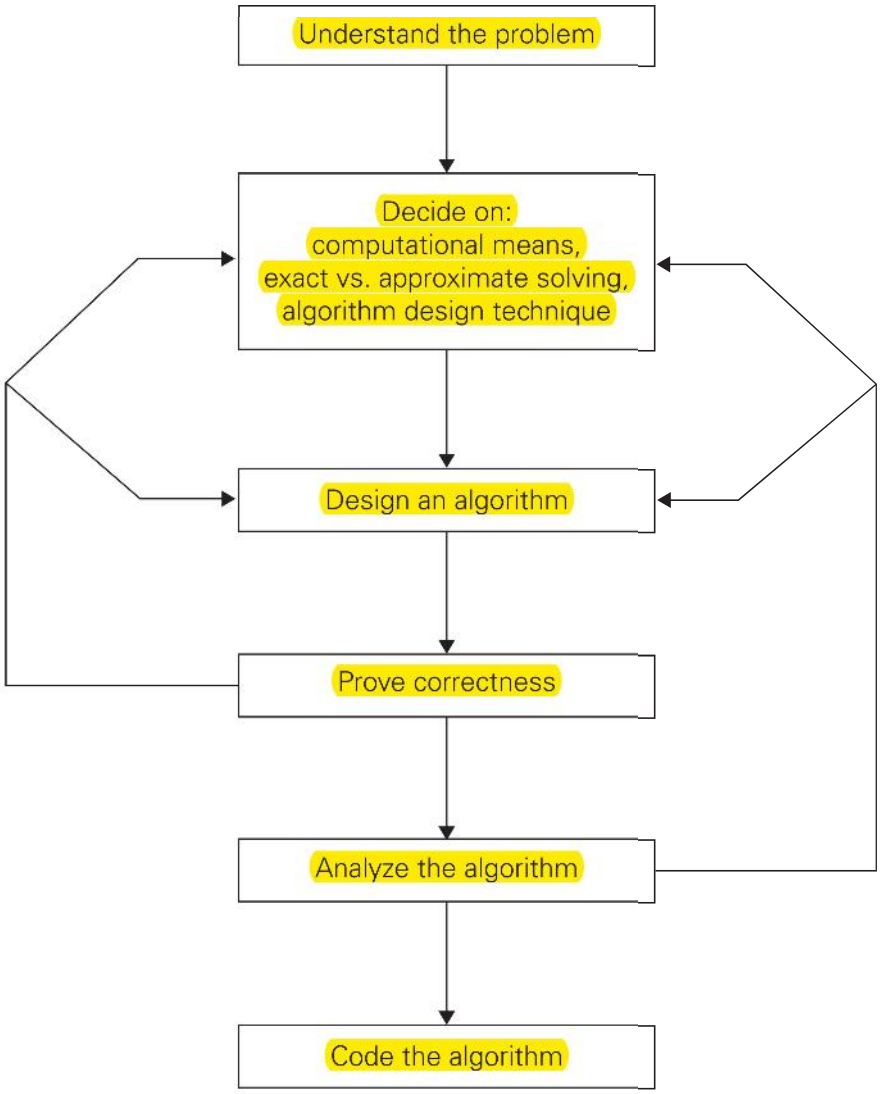


FIGURE 1.2 Algorithm design and analysis process.

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 the "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, or have to process huge volumes of data, or deal with applications where the 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.

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.

Designing an Algorithm and Data Structures

While the algorithm design techniques do provide a powerful set of general approaches to algorithmic problem solving, designing an algorithm for a particular problem may still be a challenging task. Some design techniques can be simply inapplicable to the problem in question. Sometimes, several techniques need to be combined, and there are algorithms that are hard to pinpoint as applications of the known design techniques. Even when a particular design technique is applicable, getting an algorithm often requires a nontrivial ingenuity on the part of the algorithm designer. With practice, both tasks—choosing among the general techniques and applying them—get easier, but they are rarely easy.

Of course, one should pay close attention to choosing data structures appropriate for the operations performed by the algorithm. For example, the sieve of Eratosthenes introduced in Section 1.1 would run longer if we used a linked list instead of an array in its implementation (why?). Also note that some of the algorithm design techniques discussed 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.

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, Euclid's algorithm is described 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.

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.

Pseudocode is a mixture of a natural language and programming language-like constructs. Pseudocode is usually more precise than 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 with a need 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 "←" 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—be it in a natural language or 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, the correctness of Euclid's algorithm for computing the greatest common divisor stems from the correctness of the equality $\gcd(m, n) = \gcd(n, m \bmod n)$ (which, in turn, needs a proof; see Problem 7 in Exercises 1.1), the simple observation that the second integer gets smaller on every iteration of the algorithm, and the fact that the algorithm stops when the second integer 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 show that an algorithm is incorrect, you need just one instance of its input for which the algorithm fails.

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***, indicating how fast the algorithm runs, and ***space efficiency***, indicating how much extra memory it uses. 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 set 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.

As to the set of inputs, your main concern should be designing an algorithm that can handle a set 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 belong to the specified sets and hence require no verification. When implementing algorithms as programs to be used in actual applications, you should provide such verifications.

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.

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 the writings of both Jon Bentley and Antoine de Saint-Exupéry.

A working program provides an additional opportunity in allowing an empirical analysis of the underlying algorithm. Such an 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 a project's schedule or the impatience 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? 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 integer 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 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.

Chapter 2

(Fundamentals of the Analysis of Algorithm Efficiency)

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 a practical point of view. In this chapter, we too will limit the discussion to an algorithm's efficiency.

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 the efficiency of algorithms.

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 summation 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. Discovered 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 The Analysis Framework

In this section, we outline a general framework for analyzing the efficiency of algorithms. We already mentioned in Section 1.2 that there are two kinds of efficiency: time efficiency and space efficiency. ***Time efficiency***, also called ***time complexity***, indicates how fast an algorithm in question runs. ***Space efficiency***, also called ***space complexity***, refers to the amount of memory units required by the algorithm in addition to the space needed for its input and output. In the early days of electronic computing, both resources—time and space—were at a premium. Half a century

size. In such situations, it is preferable to measure 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, or 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 mathematical problems typically involve some or all of the four arithmetical operations: addition, subtraction, multiplication, and division. Of the four, the most time-consuming operation is division, followed by multiplication and then addition and subtraction, with the last two usually considered together.[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.

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

2. On some computers, multiplication does not take longer than addition/subtraction (see, for example, the timing data provided by Kernighan and Pike in [Ker99, pp. 185–186]).

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 10 times faster than the one we have?" The answer is, obviously, 10 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 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

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}		

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 a 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.

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 linearithmic function $n \log_2 n$ increases slightly more than twofold; the quadratic function n^2 and cubic function n^3 increase fourfold and

9. For each of the following pairs of functions, indicate whether the first function of each of the following pairs has a lower, same, or higher 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*

a. According to a well-known legend, the game of chess was invented many centuries ago in northwestern India by a certain sage. When he took his invention to his king, the king liked the game so much that he offered the inventor any reward he wanted. The inventor 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 chessboard, two on the second, four on the third, eight on the fourth, and so on, until all 64 squares had been filled. If it took just 1 second to count each grain, how long would it take to count all the grain due to him?

b. How long would it take if instead of doubling the number of grains for each square of the chessboard, the inventor asked for adding two grains?

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 lower 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:

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

Indeed, the first two functions are linear and hence have a lower 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 higher 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, this informal introduction has made you comfortable with the idea behind the three asymptotic notations. So now come the formal definitions.

O-notation

DEFINITION 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 $n_0 = 1$.

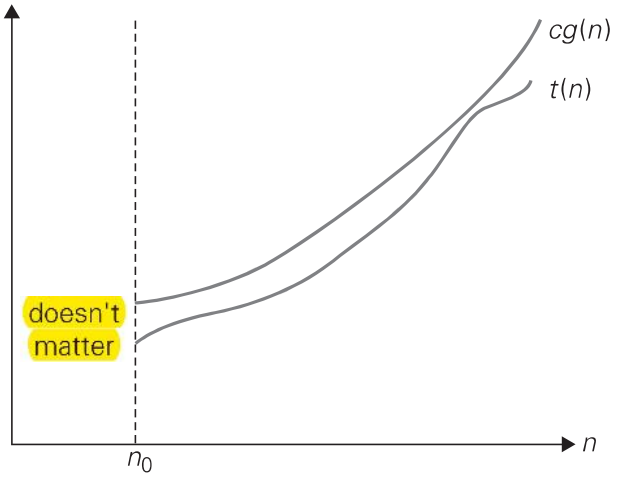


FIGURE 2.1 Big-oh notation: $t(n) \in O(g(n))$.

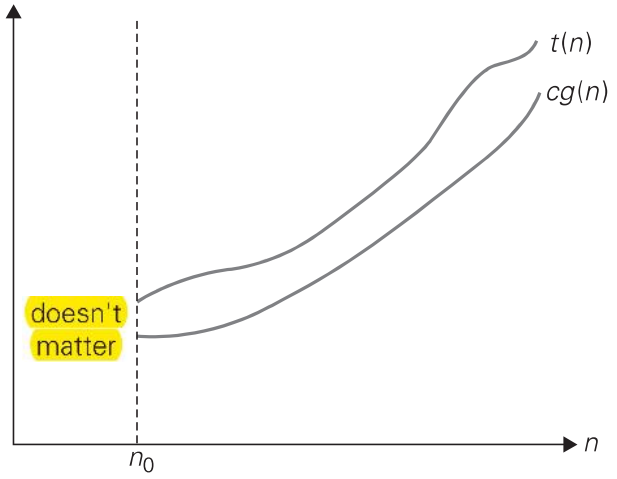


FIGURE 2.2 Big-omega notation: $t(n) \in \Omega(g(n))$.

Ω-notation

DEFINITION 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.

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 $n_0 = 0$.

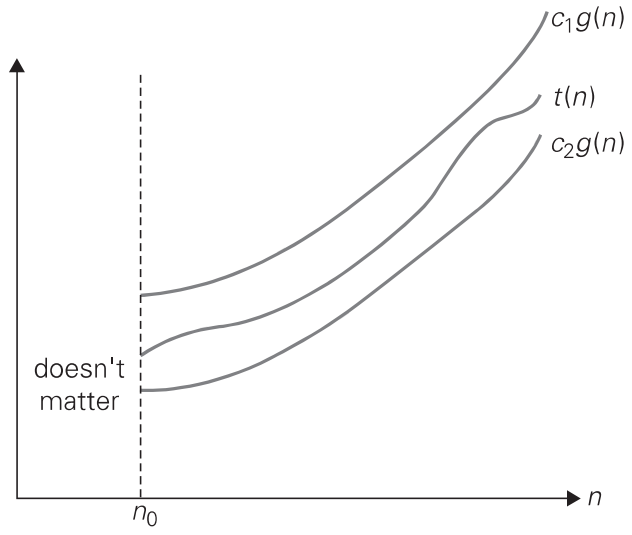


FIGURE 2.3 Big-theta notation: $t(n) \in \Theta(g(n))$.

Θ-notation

DEFINITION 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 constants c_1 and c_2 and some nonnegative integer n_0 such that

$$c_2g(n) \le t(n) \le c_1g(n) \quad \text{for all } n \ge n_0.$$

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):

$$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \le \frac{1}{2}n^2 \quad \text{for all } n \ge 0.$$

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

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

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

Useful Property Involving the Asymptotic Notations

Using the formal definitions of the asymptotic notations, we can prove their general properties (see Problem 7 in this section's exercises 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 The proof extends to orders of growth the following simple fact about four arbitrary real numbers a_1, b_1, a_2, 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 them 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 higher 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 equal elements by 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 $O(\max\{n^2, n\}) = O(n^2)$.

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:

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

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.)

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

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.) ■

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.

EXAMPLE 3 Compare the orders of growth of $n!$ and 2^n. (We discussed this informally in Section 2.1.) 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 the 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 the 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. 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.

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.
2. Use the informal definitions of O, Θ, and Ω to determine whether the following assertions are true or false.

11. *Lighter or heavier?* You have $n > 2$ identical-looking coins and a two-pan balance scale with no weights. One of the coins is a fake, but you do not know whether it is lighter or heavier than the genuine coins, which all weigh the same. Design a $\Theta(1)$ algorithm to determine whether the fake coin is lighter or heavier than the others.

12. *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]

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 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 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 $n-1$, inclusive. 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 other than 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 the 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 the 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

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

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

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).

and two summation formulas

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

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

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 of n elements 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 n, the number of elements in the array. 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 depends 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 variable j between its limits $i + 1$ and $n - 1$; this is repeated for each value of the outer loop, i.e., for each value of the loop variable i between its limits 0 and $n - 2$. Accordingly, we get

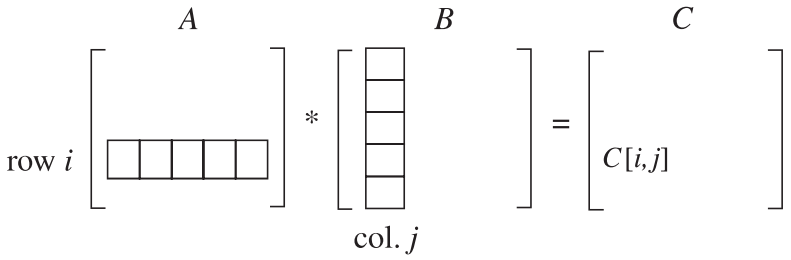
$$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i)$$
$$= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i = (n-1) \sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2}$$
$$= (n-1)^2 - \frac{(n-2)(n-1)}{2} = \frac{(n-1)n}{2} \approx \frac{1}{2} n^2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

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

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

where the last equality is obtained by applying summation formula (S2). 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 $n \times n$ matrices A and B, find the time efficiency of the definition-based algorithm for computing their product $C = AB$. By definition, C is an $n \times n$ 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 square matrices of order n by the definition-based algorithm
//Input: Two n × 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
```

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. Use the $\Theta(g(n))$ notation with the simplest function $g(n)$ possible.
   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. $\sum_{i=0}^{n-1} \sum_{j=0}^{i-1} (i+j)$

3. The sample variance of n measurements $x_1, \ldots, x_n$ can be computed as either

$$\frac{\sum_{i=1}^{n} (x_i - \bar{x})^2}{n-1} \quad \text{where } \bar{x} = \frac{\sum_{i=1}^{n} x_i}{n}$$

   or

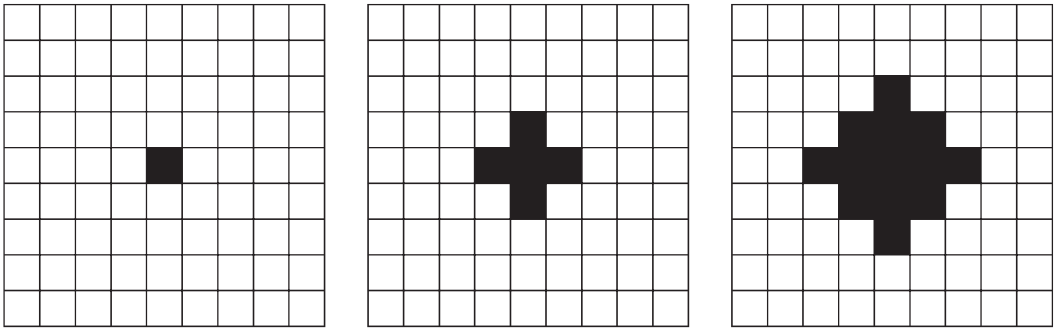
$$\frac{\sum_{i=1}^{n} x_i^2 - (\sum_{i=1}^{n} x_i)^2/n}{n-1}.$$

   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.



$n = 0$ $n = 1$ $n = 2$

13. *Page numbering* Find the total number of decimal digits needed for numbering pages in a book of 1000 pages. Assume that the pages are numbered consecutively starting with 1.

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.

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

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

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 this section's exercises).

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

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

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. Therefore, the initial condition we are after is

$$M(0) = 0.$$

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

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

$$\begin{aligned} M(n) &= M(n-1) + 1 \quad \text{for } n > 0, \\ M(0) &= 0. \end{aligned} \tag{2.2}$$

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

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

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 here (what sequence starts with 0 when $n = 0$ and increases by 1 on each step?), it will be more useful to arrive at it in a systematic fashion. From the 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 to 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. 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 exact values of $n!$ 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 the 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 that can slide onto any of 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, which 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 simply move the single disk directly from the source peg to the destination peg.

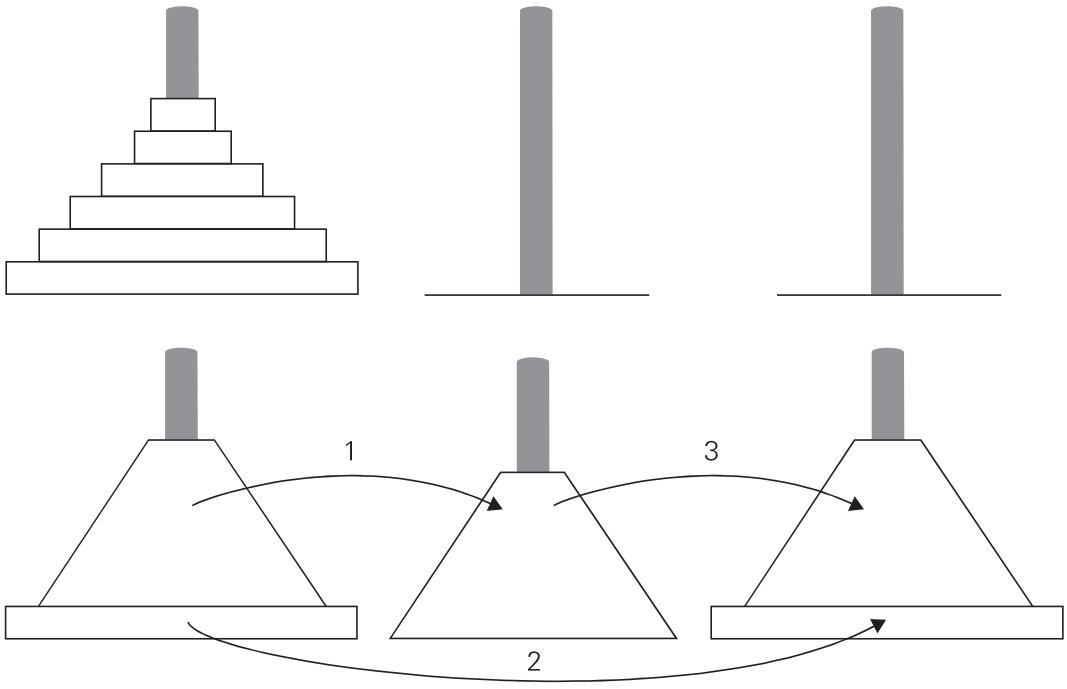


FIGURE 2.4 Recursive solution to the Tower of Hanoi puzzle.

Let us apply the general plan outlined above 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 $M(n)$:

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

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

$$M(n) = 2M(n-1) + 1 \qquad \text{sub. } M(n-1) = 2M(n-2) + 1$$
$$= 2[2M(n-2) + 1] + 1 = 2^2 M(n-2) + 2 + 1 \quad \text{sub. } M(n-2) = 2M(n-3) + 1$$
$$= 2^2[2M(n-3) + 1] + 2 + 1 = 2^3 M(n-3) + 2^2 + 2 + 1.$$

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

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

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):

$$M(n) = 2^{n-1} M(n-(n-1)) + 2^{n-1} - 1$$
$$= 2^{n-1} M(1) + 2^{n-1} - 1 = 2^{n-1} + 2^{n-1} - 1 = 2^n - 1.$$

Thus, we have an exponential algorithm, which will run for an unimaginably long time even for moderate values of n (see Problem 5 in this section's exercises). 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 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 can be 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:

$$C(n) = \sum_{l=0}^{n-1} 2^l \text{ (where } l \text{ is the level in the tree in Figure 2.5)} = 2^n - 1.$$

Chapter 3

(Brute Force and Exhaustive Search)

3

Brute Force and Exhaustive Search

Science is as far removed from brute force as this sword from a crowbar.
—Edward Lytton (1803–1873), *Leila*, Book II, Chapter I

Doing a thing well is often a waste of time.
—Robert Byrne, a master pool and billiards player and a writer

After introducing the framework and methods for algorithm analysis in the preceding chapter, we are ready to embark on a discussion of algorithm design strategies. Each of the next eight chapters is devoted to a particular design strategy. The subject of this chapter is brute force and its important special case, exhaustive search. Brute force can be described as follows:

> ***Brute force*** is a straightforward approach to solving a problem, usually directly based on the problem statement and definitions of the concepts involved.

The "force" implied by the strategy's definition is that of a computer and not that of one's intellect. "Just do it!" would be another way to describe the prescription of the brute-force approach. And often, the brute-force strategy is indeed the one that is easiest to apply.

As an example, consider the exponentiation problem: compute a^n for a nonzero number a and a nonnegative integer n. Although this problem might seem trivial, it provides a useful vehicle for illustrating several algorithm design strategies, including the brute force. (Also note that computing a^n mod m for some large integers is a principal component of a leading encryption algorithm.) By the definition of exponentiation,

$$a^n = \underbrace{a * \cdots * a}_{n \text{ times}}.$$

This suggests simply computing a^n by multiplying 1 by a n times.

We have already encountered at least two brute-force algorithms in the book: the consecutive integer checking algorithm for computing gcd(m, n) in Section 1.1 and the definition-based algorithm for matrix multiplication in Section 2.3. Many other examples are given later in this chapter. (Can you identify a few algorithms you already know as being based on the brute-force approach?)

Though rarely a source of clever or efficient algorithms, the brute-force approach should not be overlooked as an important algorithm design strategy. First, unlike some of the other strategies, brute force is applicable to a very wide variety of problems. In fact, it seems to be the only general approach for which it is more difficult to point out problems it *cannot* tackle. Second, for some important problems—e.g., sorting, searching, matrix multiplication, string matching—the brute-force approach yields reasonable algorithms of at least some practical value with no limitation on instance size. Third, the expense of designing a more efficient algorithm may be unjustifiable if only a few instances of a problem need to be solved and a brute-force algorithm can solve those instances with acceptable speed. Fourth, even if too inefficient in general, a brute-force algorithm can still be useful for solving small-size instances of a problem. Finally, a brute-force algorithm can serve an important theoretical or educational purpose as a yardstick with which to judge more efficient alternatives for solving a problem.

3.1 Selection Sort and Bubble Sort

In this section, we consider the application of the brute-force approach to the problem of sorting: given a list of n orderable items (e.g., numbers, characters from some alphabet, character strings), rearrange them in nondecreasing order. As we mentioned in Section 1.3, dozens of algorithms have been developed for solving this very important problem. You might have learned several of them in the past. If you have, try to forget them for the time being and look at the problem afresh.

Now, after your mind is unburdened of previous knowledge of sorting algorithms, ask yourself a question: "What would be the most straightforward method for solving the sorting problem?" Reasonable people may disagree on the answer to this question. The two algorithms discussed here—selection sort and bubble sort—seem to be the two prime candidates.

Selection Sort

We start selection sort by scanning the entire given list to find its smallest element and exchange it with the first element, putting the smallest element in its final position in the sorted list. Then we scan the list, starting with the second element, to find the smallest among the last $n - 1$ elements and exchange it with the second element, putting the second smallest element in its final position. Generally, on the

ith pass through the list, which we number from 0 to $n-2$, the algorithm searches for the smallest item among the last $n-i$ elements and swaps it with A_i:

$$\underbrace{A_0 \le A_1 \le \cdots \le A_{i-1}}_{\text{in their final positions}} \;\Big|\; \underbrace{A_i, \ldots, A_{min}, \ldots, A_{n-1}}_{\text{the last } n-i \text{ elements}}$$

After $n-1$ passes, the list is sorted.

Here is pseudocode of this algorithm, which, for simplicity, assumes that the list is implemented as an array:

ALGORITHM *SelectionSort*$(A[0..n-1])$
```
//Sorts a given array by selection sort
//Input: An array A[0..n − 1] of orderable elements
//Output: Array A[0..n − 1] sorted in nondecreasing order
for i ← 0 to n − 2 do
    min ← i
    for j ← i + 1 to n − 1 do
        if A[j] < A[min]  min ← j
    swap A[i] and A[min]
```

As an example, the action of the algorithm on the list 89, 45, 68, 90, 29, 34, 17 is illustrated in Figure 3.1.

The analysis of selection sort is straightforward. The input size is given by the number of elements n; the basic operation is the key comparison $A[j] < A[min]$. The number of times it is executed depends only on the array size and is given by the following sum:

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i).$$

| 89	45	68	90	29	34	**17**
17 |	45	68	90	**29**	34	89
17	29 |	68	90	45	**34**	89
17	29	34 |	90	**45**	68	89
17	29	34	45 |	90	**68**	89
17	29	34	45	68 |	90	**89**
17	29	34	45	68	89 |	90

FIGURE 3.1 Example of sorting with selection sort. Each line corresponds to one iteration of the algorithm, i.e., a pass through the list's tail to the right of the vertical bar; an element in bold indicates the smallest element found. Elements to the left of the vertical bar are in their final positions and are not considered in this and subsequent iterations.

Since we have already encountered the last sum in analyzing the algorithm of Example 2 in Section 2.3, you should be able to compute it now on your own. Whether you compute this sum by distributing the summation symbol or by immediately getting the sum of decreasing integers, the answer, of course, must be the same:

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n-1-i) = \frac{(n-1)n}{2}.$$

Thus, selection sort is a $\Theta(n^2)$ algorithm on all inputs. Note, however, that the number of key swaps is only $\Theta(n)$, or, more precisely, $n-1$ (one for each repetition of the i loop). This property distinguishes selection sort positively from many other sorting algorithms.

Bubble Sort

Another brute-force application to the sorting problem is to compare adjacent elements of the list and exchange them if they are out of order. By doing it repeatedly, we end up "bubbling up" the largest element to the last position on the list. The next pass bubbles up the second largest element, and so on, until after $n-1$ passes the list is sorted. Pass i $(0 \le i \le n-2)$ of bubble sort can be represented by the following diagram:

$$A_0, \ldots, A_j \overset{?}{\leftrightarrow} A_{j+1}, \ldots, A_{n-i-1} \mid \underbrace{A_{n-i} \le \cdots \le A_{n-1}}_{\text{in their final positions}}$$

Here is pseudocode of this algorithm.

ALGORITHM *BubbleSort*$(A[0..n-1])$
//Sorts a given array by bubble sort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
for $i \leftarrow 0$ **to** $n-2$ **do**
  for $j \leftarrow 0$ **to** $n-2-i$ **do**
    if $A[j+1] < A[j]$ swap $A[j]$ and $A[j+1]$

The action of the algorithm on the list 89, 45, 68, 90, 29, 34, 17 is illustrated as an example in Figure 3.2.

The number of key comparisons for the bubble-sort version given above is the same for all arrays of size n; it is obtained by a sum that is almost identical to the sum for selection sort:

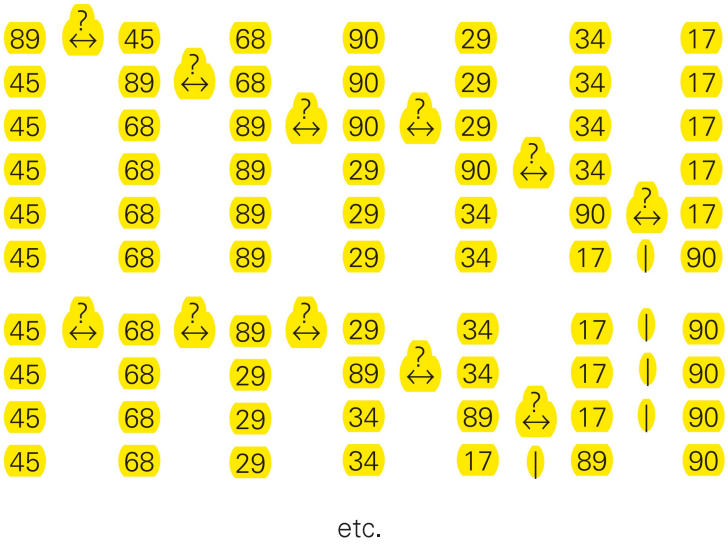


FIGURE 3.2 First two passes of bubble sort on the list 89, 45, 68, 90, 29, 34, 17. A new line is shown after a swap of two elements is done. The elements to the right of the vertical bar are in their final positions and are not considered in subsequent iterations of the algorithm.

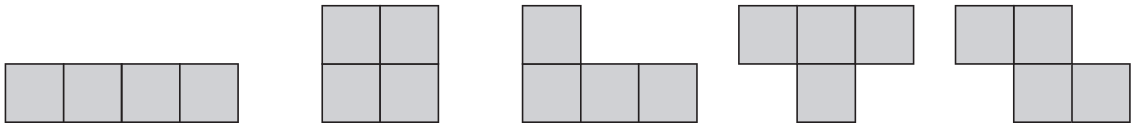
$$C(n) = \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1 = \sum_{i=0}^{n-2} [(n-2-i) - 0 + 1]$$
$$= \sum_{i=0}^{n-2} (n-1-i) = \frac{(n-1)n}{2} \in \Theta(n^2).$$

The number of key swaps, however, depends on the input. In the worst case of decreasing arrays, it is the same as the number of key comparisons:

$$S_{worst}(n) = C(n) = \frac{(n-1)n}{2} \in \Theta(n^2).$$

As is often the case with an application of the brute-force strategy, the first version of an algorithm obtained can often be improved upon with a modest amount of effort. Specifically, we can improve the crude version of bubble sort given above by exploiting the following observation: if a pass through the list makes no exchanges, the list has been sorted and we can stop the algorithm (Problem 12a in this section's exercises). Though the new version runs faster on some inputs, it is still in $\Theta(n^2)$ in the worst and average cases. In fact, even among elementary sorting methods, bubble sort is an inferior choice, and if it were not for its catchy name, you would probably have never heard of it. However, the general lesson you just learned is important and worth repeating:

> A first application of the brute-force approach often results in an algorithm that can be improved with a modest amount of effort.



Is it possible to tile—i.e., cover exactly without overlaps—an 8×8 chessboard with

a. straight tetrominoes? **b.** square tetrominoes?

c. L-tetrominoes? **d.** T-tetrominoes?

e. Z-tetrominoes?

7. *A stack of fake coins* There are n stacks of n identical-looking coins. All of the coins in one of these stacks are counterfeit, while all the coins in the other stacks are genuine. Every genuine coin weighs 10 grams; every fake weighs 11 grams. You have an analytical scale that can determine the exact weight of any number of coins.

a. Devise a brute-force algorithm to identify the stack with the fake coins and determine its worst-case efficiency class.

b. What is the minimum number of weighings needed to identify the stack with the fake coins?

8. Sort the list E, X, A, M, P, L, E in alphabetical order by selection sort.

9. Is selection sort stable? (The definition of a stable sorting algorithm was given in Section 1.3.)

10. Is it possible to implement selection sort for linked lists with the same $\Theta(n^2)$ efficiency as the array version?

11. Sort the list E, X, A, M, P, L, E in alphabetical order by bubble sort.

12. **a.** Prove that if bubble sort makes no exchanges on its pass through a list, the list is sorted and the algorithm can be stopped.

b. Write pseudocode of the method that incorporates this improvement.

c. Prove that the worst-case efficiency of the improved version is quadratic.

13. Is bubble sort stable?

14. *Alternating disks* You have a row of $2n$ disks of two colors, n dark and n light. They alternate: dark, light, dark, light, and so on. You want to get all the dark disks to the right-hand end, and all the light disks to the left-hand end. The only moves you are allowed to make are those that interchange the positions of two neighboring disks.

Design an algorithm for solving this puzzle and determine the number of moves it takes. [Gar99]

3.2 Sequential Search and Brute-Force String Matching

We saw in the previous section two applications of the brute-force approach to the sorting porblem. Here we discuss two applications of this strategy to the problem of searching. The first deals with the canonical problem of searching for an item of a given value in a given list. The second is different in that it deals with the string-matching problem.

Sequential Search

We have already encountered a brute-force algorithm for the general searching problem: it is called sequential search (see Section 2.1). To repeat, the algorithm simply compares successive elements of a given list with a given search key until either a match is encountered (successful search) or the list is exhausted without finding a match (unsuccessful search). A simple extra trick is often employed in implementing sequential search: if we append the search key to the end of the list, the search for the key will have to be successful, and therefore we can eliminate the end of list check altogether. Here is pseudocode of this enhanced version.

ALGORITHM *SequentialSearch2*$(A[0..n], K)$

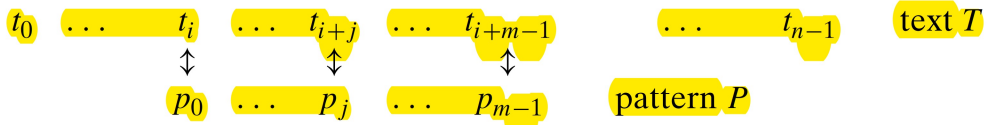
```
//Implements sequential search with a search key as a sentinel
//Input: An array A of n elements and a search key K
//Output: The index of the first element in A[0..n − 1] whose value is
//        equal to K or −1 if no such element is found
A[n] ← K
i ← 0
while A[i] ≠ K do
        i ← i + 1
if i < n return i
else return −1
```

Another straightforward improvement can be incorporated in sequential search if a given list is known to be sorted: searching in such a list can be stopped as soon as an element greater than or equal to the search key is encountered.

Sequential search provides an excellent illustration of the brute-force approach, with its characteristic strength (simplicity) and weakness (inferior efficiency). The efficiency results obtained in Section 2.1 for the standard version of sequential search change for the enhanced version only very slightly, so that the algorithm remains linear in both the worst and average cases. We discuss later in the book several searching algorithms with a better time efficiency.

Brute-Force String Matching

Recall the string-matching problem introduced in Section 1.3: given a string of n characters called the ***text*** and a string of m characters ($m \le n$) called the ***pattern***, find a substring of the text that matches the pattern. To put it more precisely, we want to find i—the index of the leftmost character of the first matching substring in the text—such that $t_i = p_0, \ldots, t_{i+j} = p_j, \ldots, t_{i+m-1} = p_{m-1}$:



If matches other than the first one need to be found, a string-matching algorithm can simply continue working until the entire text is exhausted.

A brute-force algorithm for the string-matching problem is quite obvious: align the pattern against the first m characters of the text and start matching the corresponding pairs of characters from left to right until either all the m pairs of the characters match (then the algorithm can stop) or a mismatching pair is encountered. In the latter case, shift the pattern one position to the right and resume the character comparisons, starting again with the first character of the pattern and its counterpart in the text. Note that the last position in the text that can still be a beginning of a matching substring is $n - m$ (provided the text positions are indexed from 0 to $n - 1$). Beyond that position, there are not enough characters to match the entire pattern; hence, the algorithm need not make any comparisons there.

ALGORITHM *BruteForceStringMatch*$(T[0..n-1], P[0..m-1])$

```
//Implements brute-force string matching
//Input: An array T[0..n − 1] of n characters representing a text and
//       an array P[0..m − 1] of m characters representing a pattern
//Output: The index of the first character in the text that starts a
//        matching substring or −1 if the search is unsuccessful
for i ← 0 to n − m do
    j ← 0
    while j < m and P[j] = T[i + j] do
        j ← j + 1
    if j = m return i
return −1
```

An operation of the algorithm is illustrated in Figure 3.3. Note that for this example, the algorithm shifts the pattern almost always after a single character comparison. The worst case is much worse: the algorithm may have to make all m comparisons before shifting the pattern, and this can happen for each of the $n - m + 1$ tries. (Problem 6 in this section's exercises asks you to give a specific example of such a situation.) Thus, in the worst case, the algorithm makes

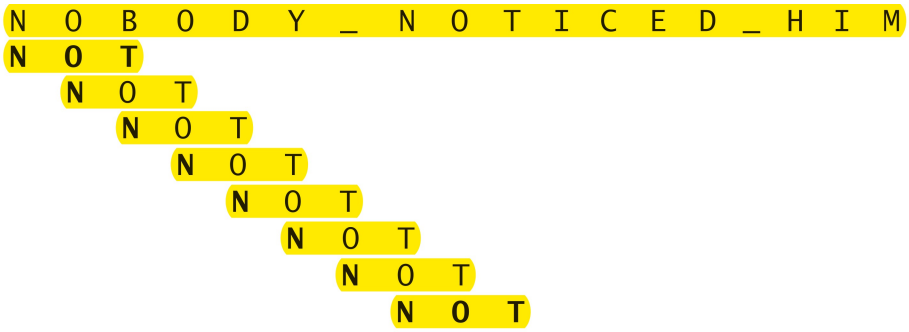


FIGURE 3.3 Example of brute-force string matching. The pattern's characters that are compared with their text counterparts are in bold type.

$m(n - m + 1)$ character comparisons, which puts it in the $O(nm)$ class. For a typical word search in a natural language text, however, we should expect that most shifts would happen after very few comparisons (check the example again). Therefore, the average-case efficiency should be considerably better than the worst-case efficiency. Indeed it is: for searching in random texts, it has been shown to be linear, i.e., $\Theta(n)$. There are several more sophisticated and more efficient algorithms for string searching. The most widely known of them—by R. Boyer and J. Moore—is outlined in Section 7.2 along with its simplification suggested by R. Horspool.

Exercises 3.2

1. Find the number of comparisons made by the sentinel version of sequential search
   a. in the worst case.
   b. in the average case if the probability of a successful search is p $(0 \le p \le 1)$.
2. As shown in Section 2.1, the average number of key comparisons made by sequential search (without a sentinel, under standard assumptions about its inputs) is given by the formula

$$C_{avg}(n) = \frac{p(n+1)}{2} + n(1-p),$$

   where p is the probability of a successful search. Determine, for a fixed n, the values of p $(0 \le p \le 1)$ for which this formula yields the maximum value of $C_{avg}(n)$ and the minimum value of $C_{avg}(n)$.

3. *Gadget testing* A firm wants to determine the highest floor of its n-story headquarters from which a gadget can fall without breaking. The firm has two identical gadgets to experiment with. If one of them gets broken, it cannot be repaired, and the experiment will have to be completed with the remaining gadget. Design an algorithm in the best efficiency class you can to solve this problem.

4. Determine the number of character comparisons made by the brute-force algorithm in searching for the pattern GANDHI in the text

   THERE_IS_MORE_TO_LIFE_THAN_INCREASING_ITS_SPEED

   Assume that the length of the text—it is 47 characters long—is known before the search starts.

5. How many comparisons (both successful and unsuccessful) will be made by the brute-force algorithm in searching for each of the following patterns in the binary text of one thousand zeros?

   a. 00001 **b.** 10000 **c.** 01010

6. Give an example of a text of length n and a pattern of length m that constitutes a worst-case input for the brute-force string-matching algorithm. Exactly how many character comparisons will be made for such input?

7. In solving the string-matching problem, would there be any advantage in comparing pattern and text characters right-to-left instead of left-to-right?

8. Consider the problem of counting, in a given text, the number of substrings that start with an A and end with a B. For example, there are four such substrings in CABAAXBYA.

   a. Design a brute-force algorithm for this problem and determine its efficiency class.

   b. Design a more efficient algorithm for this problem. [Gin04]

9. Write a visualization program for the brute-force string-matching algorithm.

10. *Word Find* A popular diversion in the United States, "word find" (or "word search") puzzles ask the player to find each of a given set of words in a square table filled with single letters. A word can read horizontally (left or right), vertically (up or down), or along a 45 degree diagonal (in any of the four directions) formed by consecutively adjacent cells of the table; it may wrap around the table's boundaries, but it must read in the same direction with no zigzagging. The same cell of the table may be used in different words, but, in a given word, the same cell may be used no more than once. Write a computer program for solving this puzzle.

11. *Battleship game* Write a program based on a version of brute-force pattern matching for playing the game Battleship on the computer. The rules of the game are as follows. There are two opponents in the game (in this case, a human player and the computer). The game is played on two identical boards (10×10 tables of squares) on which each opponent places his or her ships, not seen by the opponent. Each player has five ships, each of which occupies a certain number of squares on the board: a destroyer (two squares), a submarine (three squares), a cruiser (three squares), a battleship (four squares), and an aircraft carrier (five squares). Each ship is placed either horizontally or vertically, with no two ships touching each other. The game is played by the opponents taking turns "shooting" at each other's ships. The

result of every shot is displayed as either a hit or a miss. In case of a hit, the player gets to go again and keeps playing until missing. The goal is to sink all the opponent's ships before the opponent succeeds in doing it first. To sink a ship, all squares occupied by the ship must be hit.

3.3 Closest-Pair and Convex-Hull Problems by Brute Force

In this section, we consider a straightforward approach to two well-known problems dealing with a finite set of points in the plane. These problems, aside from their theoretical interest, arise in two important applied areas: computational geometry and operations research.

Closest-Pair Problem

The closest-pair problem calls for finding the two closest points in a set of n points. It is the simplest of a variety of problems in computational geometry that deals with proximity of points in the plane or higher-dimensional spaces. Points in question can represent such physical objects as airplanes or post offices as well as database records, statistical samples, DNA sequences, and so on. An air-traffic controller might be interested in two closest planes as the most probable collision candidates. A regional postal service manager might need a solution to the closest-pair problem to find candidate post-office locations to be closed.

One of the important applications of the closest-pair problem is cluster analysis in statistics. Based on n data points, hierarchical cluster analysis seeks to organize them in a hierarchy of clusters based on some similarity metric. For numerical data, this metric is usually the Euclidean distance; for text and other nonnumerical data, metrics such as the Hamming distance (see Problem 5 in this section's exercises) are used. A bottom-up algorithm begins with each element as a separate cluster and merges them into successively larger clusters by combining the closest pair of clusters.

For simplicity, we consider the two-dimensional case of the closest-pair problem. We assume that the points in question are specified in a standard fashion by their (x, y) Cartesian coordinates and that the distance between two points $p_i(x_i, y_i)$ and $p_j(x_j, y_j)$ is the standard Euclidean distance

$$d(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}.$$

The brute-force approach to solving this problem leads to the following obvious algorithm: compute the distance between each pair of distinct points and find a pair with the smallest distance. Of course, we do not want to compute the distance between the same pair of points twice. To avoid doing so, we consider only the pairs of points (p_i, p_j) for which $i < j$.

Pseudocode below computes the distance between the two closest points; getting the closest points themselves requires just a trivial modification.

ALGORITHM *BruteForceClosestPair*(P)

```
//Finds distance between two closest points in the plane by brute force
//Input: A list P of n (n ≥ 2) points p1(x1, y1), . . . , pn(xn, yn)
//Output: The distance between the closest pair of points
d ← ∞
for i ← 1 to n − 1 do
    for j ← i + 1 to n do
        d ← min(d, sqrt((xi − xj)^2 + (yi − yj)^2)) //sqrt is square root
return d
```

The basic operation of the algorithm is computing the square root. In the age of electronic calculators with a square-root button, one might be led to believe that computing the square root is as simple an operation as, say, addition or multiplication. Of course, it is not. For starters, even for most integers, square roots are irrational numbers that therefore can be found only approximately. Moreover, computing such approximations is not a trivial matter. But, in fact, computing square roots in the loop can be avoided! (Can you think how?) The trick is to realize that we can simply ignore the square-root function and compare the values $(x_i - x_j)^2 + (y_i - y_j)^2$ themselves. We can do this because the smaller a number of which we take the square root, the smaller its square root, or, as mathematicians say, the square-root function is strictly increasing.

Then the basic operation of the algorithm will be squaring a number. The number of times it will be executed can be computed as follows:

$$C(n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 2 = 2 \sum_{i=1}^{n-1} (n-i)$$
$$= 2[(n-1) + (n-2) + \cdots + 1] = (n-1)n \in \Theta(n^2).$$

Of course, speeding up the innermost loop of the algorithm could only decrease the algorithm's running time by a constant factor (see Problem 1 in this section's exercises), but it cannot improve its asymptotic efficiency class. In Chapter 5, we discuss a linearithmic algorithm for this problem, which is based on a more sophisticated design technique.

Convex-Hull Problem

On to the other problem—that of computing the convex hull. Finding the convex hull for a given set of points in the plane or a higher dimensional space is one of the most important—some people believe the most important—problems in computational geometry. This prominence is due to a variety of applications in which

this problem needs to be solved, either by itself or as a part of a larger task. Several such applications are based on the fact that convex hulls provide convenient approximations of object shapes and data sets given. For example, in computer animation, replacing objects by their convex hulls speeds up collision detection; the same idea is used in path planning for Mars mission rovers. Convex hulls are used in computing accessibility maps produced from satellite images by Geographic Information Systems. They are also used for detecting outliers by some statistical techniques. An efficient algorithm for computing a diameter of a set of points, which is the largest distance between two of the points, needs the set's convex hull to find the largest distance between two of its extreme points (see below). Finally, convex hulls are important for solving many optimization problems, because their extreme points provide a limited set of solution candidates.

We start with a definition of a convex set.

DEFINITION A set of points (finite or infinite) in the plane is called ***convex*** if for any two points p and q in the set, the entire line segment with the endpoints at p and q belongs to the set.

All the sets depicted in Figure 3.4a are convex, and so are a straight line, a triangle, a rectangle, and, more generally, any convex polygon,[1] a circle, and the entire plane. On the other hand, the sets depicted in Figure 3.4b, any finite set of two or more distinct points, the boundary of any convex polygon, and a circumference are examples of sets that are not convex.

Now we are ready for the notion of the convex hull. Intuitively, the convex hull of a set of n points in the plane is the smallest convex polygon that contains all of them either inside or on its boundary. If this formulation does not fire up your enthusiasm, consider the problem as one of barricading n sleeping tigers by a fence of the shortest length. This interpretation is due to D. Harel [Har92]; it is somewhat lively, however, because the fenceposts have to be erected right at the spots where some of the tigers sleep! There is another, much tamer interpretation of this notion. Imagine that the points in question are represented by nails driven into a large sheet of plywood representing the plane. Take a rubber band and stretch it to include all the nails, then let it snap into place. The convex hull is the area bounded by the snapped rubber band (Figure 3.5).

A formal definition of the convex hull that is applicable to arbitrary sets, including sets of points that happen to lie on the same line, follows.

DEFINITION The ***convex hull*** of a set S of points is the smallest convex set containing S. (The "smallest" requirement means that the convex hull of S must be a subset of any convex set containing S.)

If S is convex, its convex hull is obviously S itself. If S is a set of two points, its convex hull is the line segment connecting these points. If S is a set of three

1. By "a triangle, a rectangle, and, more generally, any convex polygon," we mean here a region, i.e., the set of points both inside and on the boundary of the shape in question.

b. a square

c. the boundary of a square

d. a straight line

9. Design a linear-time algorithm to determine two extreme points of the convex hull of a given set of $n > 1$ points in the plane.

10. What modification needs to be made in the brute-force algorithm for the convex-hull problem to handle more than two points on the same straight line?

11. Write a program implementing the brute-force algorithm for the convex-hull problem.

12. Consider the following small instance of the linear programming problem:

$$\begin{aligned} \text{maximize} \quad & 3x + 5y \\ \text{subject to} \quad & x + \ \ y \le 4 \\ & x + 3y \le 6 \\ & x \ge 0,\ y \ge 0. \end{aligned}$$

a. Sketch, in the Cartesian plane, the problem's ***feasible region***, defined as the set of points satisfying all the problem's constraints.

b. Identify the region's extreme points.

c. Solve this optimization problem by using the following theorem: A linear programming problem with a nonempty bounded feasible region always has a solution, which can be found at one of the extreme points of its feasible region.

3.4 Exhaustive Search

Many important problems require finding an element with a special property in a domain that grows exponentially (or faster) with an instance size. Typically, such problems arise in situations that involve—explicitly or implicitly—combinatorial objects such as permutations, combinations, and subsets of a given set. Many such problems are optimization problems: they ask to find an element that maximizes or minimizes some desired characteristic such as a path length or an assignment cost.

Exhaustive search is simply a brute-force approach to combinatorial problems. It suggests generating each and every element of the problem domain, selecting those of them that satisfy all the constraints, and then finding a desired element (e.g., the one that optimizes some objective function). Note that although the idea of exhaustive search is quite straightforward, its implementation typically requires an algorithm for generating certain combinatorial objects. We delay a discussion of such algorithms until the next chapter and assume here that they exist.

We illustrate exhaustive search by applying it to three important problems: the traveling salesman problem, the knapsack problem, and the assignment problem.

Traveling Salesman Problem

The ***traveling salesman problem (TSP)*** has been intriguing researchers for the last 150 years by its seemingly simple formulation, important applications, and interesting connections to other combinatorial problems. In layman's terms, the problem asks to find the shortest tour through a given set of n cities that visits each city exactly once before returning to the city where it started. The problem can be conveniently modeled by a weighted graph, with the graph's vertices representing the cities and the edge weights specifying the distances. Then the problem can be stated as the problem of finding the shortest ***Hamiltonian circuit*** of the graph. (A Hamiltonian circuit is defined as a cycle that passes through all the vertices of the graph exactly once. It is named after the Irish mathematician Sir William Rowan Hamilton (1805–1865), who became interested in such cycles as an application of his algebraic discoveries.)

It is easy to see that a Hamiltonian circuit can also be defined as a sequence of $n + 1$ adjacent vertices $v_{i_0}, v_{i_1}, \ldots, v_{i_{n-1}}, v_{i_0}$, where the first vertex of the sequence is the same as the last one and all the other $n - 1$ vertices are distinct. Further, we can assume, with no loss of generality, that all circuits start and end at one particular vertex (they are cycles after all, are they not?). Thus, we can get all the tours by generating all the permutations of $n - 1$ intermediate cities, compute the tour lengths, and find the shortest among them. Figure 3.7 presents a small instance of the problem and its solution by this method.

An inspection of Figure 3.7 reveals three pairs of tours that differ only by their direction. Hence, we could cut the number of vertex permutations by half. We could, for example, choose any two intermediate vertices, say, b and c, and then consider only permutations in which b precedes c. (This trick implicitly defines a tour's direction.)

This improvement cannot brighten the efficiency picture much, however. The total number of permutations needed is still $\frac{1}{2}(n-1)!$, which makes the exhaustive-search approach impractical for all but very small values of n. On the other hand, if you always see your glass as half-full, you can claim that cutting the work by half is nothing to sneeze at, even if you solve a small instance of the problem, especially by hand. Also note that had we not limited our investigation to the circuits starting at the same vertex, the number of permutations would have been even larger, by a factor of n.

Knapsack Problem

Here is another well-known problem in algorithmics. Given n items of known weights $w_1, w_2, \ldots, w_n$ and values $v_1, v_2, \ldots, v_n$ and a knapsack of capacity W, find the most valuable subset of the items that fit into the knapsack. If you do not like the idea of putting yourself in the shoes of a thief who wants to steal the most

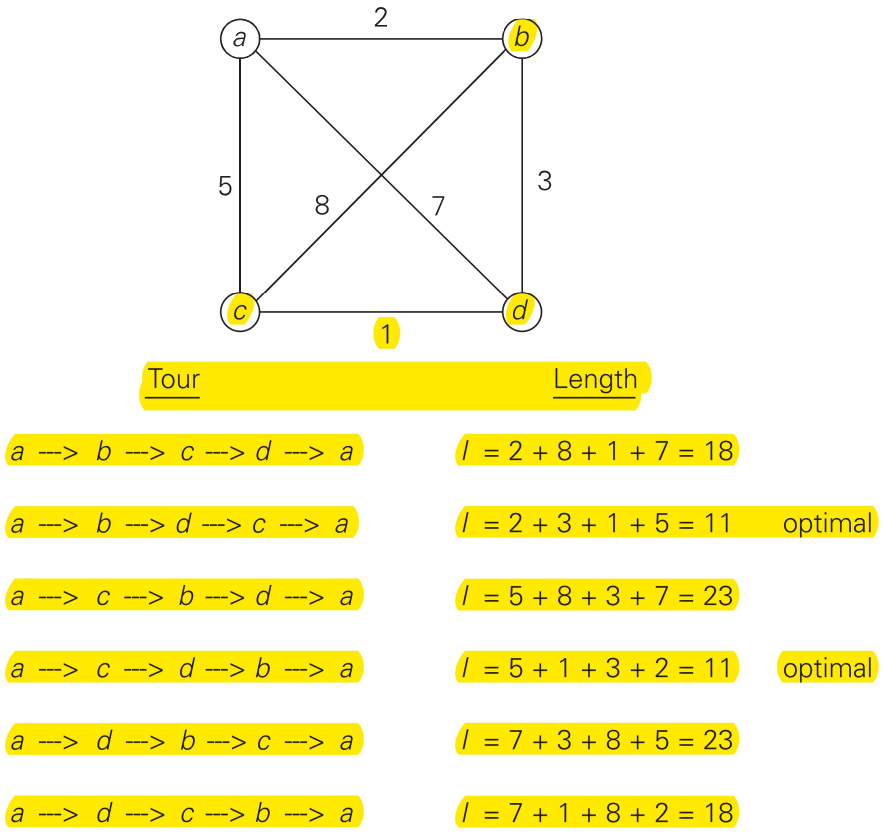


FIGURE 3.7 Solution to a small instance of the traveling salesman problem by exhaustive search.

valuable loot that fits into his knapsack, think about a transport plane that has to deliver the most valuable set of items to a remote location without exceeding the plane's capacity. Figure 3.8a presents a small instance of the knapsack problem.

The exhaustive-search approach to this problem leads to generating all the subsets of the set of n items given, computing the total weight of each subset in order to identify feasible subsets (i.e., the ones with the total weight not exceeding the knapsack capacity), and finding a subset of the largest value among them. As an example, the solution to the instance of Figure 3.8a is given in Figure 3.8b. Since the number of subsets of an n-element set is 2^n, the exhaustive search leads to a $\Omega(2^n)$ algorithm, no matter how efficiently individual subsets are generated.

Thus, for both the traveling salesman and knapsack problems considered above, exhaustive search leads to algorithms that are extremely inefficient on every input. In fact, these two problems are the best-known examples of so-called ***NP-hard problems***. No polynomial-time algorithm is known for any *NP*-hard problem. Moreover, most computer scientists believe that such algorithms do not exist, although this very important conjecture has never been proven. More-sophisticated approaches—backtracking and branch-and-bound (see Sections 12.1 and 12.2)—enable us to solve some but not all instances of these and

Chapter 4

(Decrease-and-Conquer)

4

Decrease-and-Conquer

Plutarch says that Sertorius, in order to teach his soldiers that perseverance and wit are better than brute force, had two horses brought before them, and set two men to pull out their tails. One of the men was a burly Hercules, who tugged and tugged, but all to no purpose; the other was a sharp, weasel-faced tailor, who plucked one hair at a time, amidst roars of laughter, and soon left the tail quite bare.

—E. Cobham Brewer, *Dictionary of Phrase and Fable*, 1898

The ***decrease-and-conquer*** technique is based on exploiting the relationship between a solution to a given instance of a problem and a solution to its smaller instance. Once such a relationship is established, it can be exploited either top down or bottom up. The former leads naturally to a recursive implementation, although, as one can see from several examples in this chapter, an ultimate implementation may well be nonrecursive. The bottom-up variation is usually implemented iteratively, starting with a solution to the smallest instance of the problem; it is called sometimes the ***incremental approach***.

There are three major variations of decrease-and-conquer:

- decrease by a constant
- decrease by a constant factor
- variable size decrease

In the ***decrease-by-a-constant*** variation, the size of an instance is reduced by the same constant on each iteration of the algorithm. Typically, this constant is equal to one (Figure 4.1), although other constant size reductions do happen occasionally.

Consider, as an example, the exponentiation problem of computing a^n where $a \neq 0$ and n is a nonnegative integer. The relationship between a solution to an instance of size n and an instance of size $n-1$ is obtained by the obvious formula $a^n = a^{n-1} \cdot a$. So the function $f(n) = a^n$ can be computed either "top down" by using its recursive definition

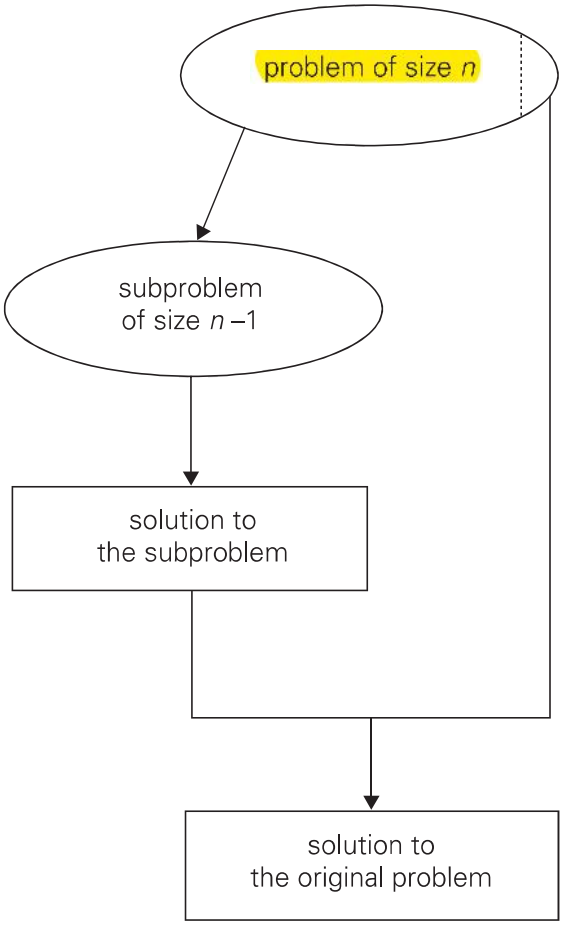


FIGURE 4.1 Decrease-(by one)-and-conquer technique.

$$f(n) = \begin{cases} f(n-1) \cdot a & \text{if } n > 0, \\ 1 & \text{if } n = 0, \end{cases} \tag{4.1}$$

or "bottom up" by multiplying 1 by a n times. (Yes, it is the same as the brute-force algorithm, but we have come to it by a different thought process.) More interesting examples of decrease-by-one algorithms appear in Sections 4.1–4.3.

The ***decrease-by-a-constant-factor*** technique suggests reducing a problem instance by the same constant factor on each iteration of the algorithm. In most applications, this constant factor is equal to two. (Can you give an example of such an algorithm?) The decrease-by-half idea is illustrated in Figure 4.2.

For an example, let us revisit the exponentiation problem. If the instance of size n is to compute a^n, the instance of half its size is to compute $a^{n/2}$, with the obvious relationship between the two: $a^n = (a^{n/2})^2$. But since we consider here instances with integer exponents only, the former does not work for odd n. If n is odd, we have to compute a^{n-1} by using the rule for even-valued exponents and then multiply the result by a. To summarize, we have the following formula:

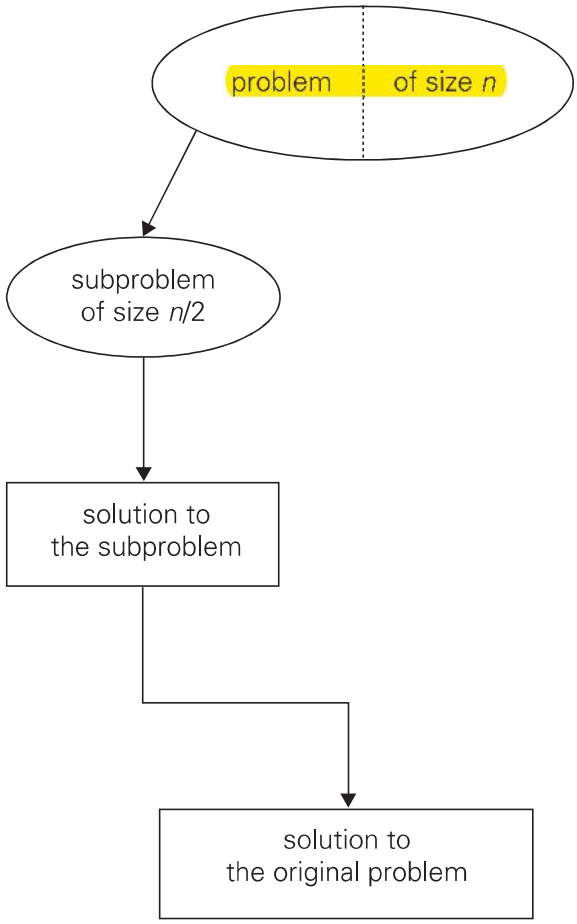


FIGURE 4.2 Decrease-(by half)-and-conquer technique.

$$a^n = \begin{cases} (a^{n/2})^2 & \text{if } n \text{ is even and positive,} \\ (a^{(n-1)/2})^2 \cdot a & \text{if } n \text{ is odd,} \\ 1 & \text{if } n = 0. \end{cases} \tag{4.2}$$

If we compute a^n recursively according to formula (4.2) and measure the algorithm's efficiency by the number of multiplications, we should expect the algorithm to be in $\Theta(\log n)$ because, on each iteration, the size is reduced by about a half at the expense of one or two multiplications.

A few other examples of decrease-by-a-constant-factor algorithms are given in Section 4.4 and its exercises. Such algorithms are so efficient, however, that there are few examples of this kind.

Finally, in the ***variable-size-decrease*** variety of decrease-and-conquer, the size-reduction pattern varies from one iteration of an algorithm to another. Euclid's algorithm for computing the greatest common divisor provides a good example of such a situation. Recall that this algorithm is based on the formula

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

Though the value of the second argument is always smaller on the right-hand side than on the left-hand side, it decreases neither by a constant nor by a constant factor. A few other examples of such algorithms appear in Section 4.5.

4.1 Insertion Sort

In this section, we consider an application of the decrease-by-one technique to sorting an array $A[0..n-1]$. Following the technique's idea, we assume that the smaller problem of sorting the array $A[0..n-2]$ has already been solved to give us a sorted array of size $n-1$: $A[0] \le \cdots \le A[n-2]$. How can we take advantage of this solution to the smaller problem to get a solution to the original problem by taking into account the element $A[n-1]$? Obviously, all we need is to find an appropriate position for $A[n-1]$ among the sorted elements and insert it there. This is usually done by scanning the sorted subarray from right to left until the first element smaller than or equal to $A[n-1]$ is encountered to insert $A[n-1]$ right after that element. The resulting algorithm is called ***straight insertion sort*** or simply ***insertion sort***.

Though insertion sort is clearly based on a recursive idea, it is more efficient to implement this algorithm bottom up, i.e., iteratively. As shown in Figure 4.3, starting with $A[1]$ and ending with $A[n-1]$, $A[i]$ is inserted in its appropriate place among the first i elements of the array that have been already sorted (but, unlike selection sort, are generally not in their final positions).

Here is pseudocode of this algorithm.

ALGORITHM *InsertionSort*$(A[0..n-1])$
//Sorts a given array by insertion sort
//Input: An array $A[0..n-1]$ of n orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
for $i \leftarrow 1$ **to** $n-1$ **do**
    $v \leftarrow A[i]$
    $j \leftarrow i-1$
    while $j \ge 0$ **and** $A[j] > v$ **do**
        $A[j+1] \leftarrow A[j]$
        $j \leftarrow j-1$
    $A[j+1] \leftarrow v$

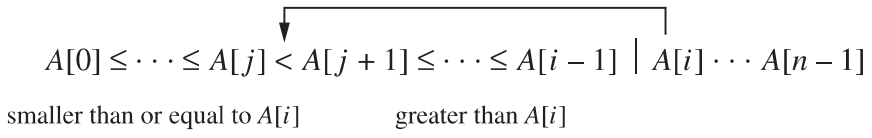


FIGURE 4.3 Iteration of insertion sort: $A[i]$ is inserted in its proper position among the preceding elements previously sorted.

89 | **45** 68 90 29 34 17
45 89 | **68** 90 29 34 17
45 68 89 | **90** 29 34 17
45 68 89 90 | **29** 34 17
29 45 68 89 90 | **34** 17
29 34 45 68 89 90 | **17**
17 29 34 45 68 89 90

FIGURE 4.4 Example of sorting with insertion sort. A vertical bar separates the sorted part of the array from the remaining elements; the element being inserted is in bold.

The operation of the algorithm is illustrated in Figure 4.4.

The basic operation of the algorithm is the key comparison $A[j] > v$. (Why not $j \geq 0$? Because it is almost certainly faster than the former in an actual computer implementation. Moreover, it is not germane to the algorithm: a better implementation with a sentinel—see Problem 8 in this section's exercises—eliminates it altogether.)

The number of key comparisons in this algorithm obviously depends on the nature of the input. In the worst case, $A[j] > v$ is executed the largest number of times, i.e., for every $j = i - 1, \ldots, 0$. Since $v = A[i]$, it happens if and only if $A[j] > A[i]$ for $j = i - 1, \ldots, 0$. (Note that we are using the fact that on the ith iteration of insertion sort all the elements preceding $A[i]$ are the first i elements in the input, albeit in the sorted order.) Thus, for the worst-case input, we get $A[0] > A[1]$ (for $i = 1$), $A[1] > A[2]$ (for $i = 2$), $\ldots$, $A[n-2] > A[n-1]$ (for $i = n - 1$). In other words, the worst-case input is an array of strictly decreasing values. The number of key comparisons for such an input is

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

Thus, in the worst case, insertion sort makes exactly the same number of comparisons as selection sort (see Section 3.1).

In the best case, the comparison $A[j] > v$ is executed only once on every iteration of the outer loop. It happens if and only if $A[i-1] \leq A[i]$ for every $i = 1, \ldots, n-1$, i.e., if the input array is already sorted in nondecreasing order. (Though it "makes sense" that the best case of an algorithm happens when the problem is already solved, it is not always the case, as you are going to see in our discussion of quicksort in Chapter 5.) Thus, for sorted arrays, the number of key comparisons is

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

This very good performance in the best case of sorted arrays is not very useful by itself, because we cannot expect such convenient inputs. However, almost-sorted files do arise in a variety of applications, and insertion sort preserves its excellent performance on such inputs.

A rigorous analysis of the algorithm's average-case efficiency is based on investigating the number of element pairs that are out of order (see Problem 11 in this section's exercises). It shows that on randomly ordered arrays, insertion sort makes on average half as many comparisons as on decreasing arrays, i.e.,

$$C_{avg}(n) \approx \frac{n^2}{4} \in \Theta(n^2).$$

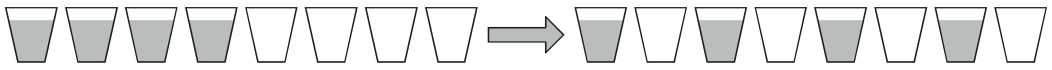
This twice-as-fast average-case performance coupled with an excellent efficiency on almost-sorted arrays makes insertion sort stand out among its principal competitors among elementary sorting algorithms, selection sort and bubble sort. In addition, its extension named ***shellsort***, after its inventor D. L. Shell [She59], gives us an even better algorithm for sorting moderately large files (see Problem 12 in this section's exercises).

Exercises 4.1

1. *Ferrying soldiers* A detachment of n soldiers must cross a wide and deep river with no bridge in sight. They notice two 12-year-old boys playing in a rowboat by the shore. The boat is so tiny, however, that it can only hold two boys or one soldier. How can the soldiers get across the river and leave the boys in joint possession of the boat? How many times need the boat pass from shore to shore?

2. *Alternating glasses*

a. There are $2n$ glasses standing next to each other in a row, the first n of them filled with a soda drink and the remaining n glasses empty. Make the glasses alternate in a filled-empty-filled-empty pattern in the minimum number of glass moves. [Gar78]



b. Solve the same problem if $2n$ glasses—n with a drink and n empty—are initially in a random order.

3. *Marking cells* Design an algorithm for the following task. For any even n, mark n cells on an infinite sheet of graph paper so that each marked cell has an odd number of marked neighbors. Two cells are considered neighbors if they are next to each other either horizontally or vertically but not diagonally. The marked cells must form a contiguous region, i.e., a region in which there is a path between any pair of marked cells that goes through a sequence of marked neighbors. [Kor05]

4. Design a decrease-by-one algorithm for generating the power set of a set of n elements. (The power set of a set S is the set of all the subsets of S, including the empty set and S itself.)

5. Consider the following algorithm to check connectivity of a graph defined by its adjacency matrix.

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

```
//Input: Adjacency matrix A[0..n − 1, 0..n − 1]) of an undirected graph G
//Output: 1 (true) if G is connected and 0 (false) if it is not
if n = 1 return 1   //one-vertex graph is connected by definition
else
    if not Connected(A[0..n − 2, 0..n − 2]) return 0
    else for j ← 0 to n − 2 do
            if A[n − 1, j] return 1
        return 0
```

Does this algorithm work correctly for every undirected graph with $n > 0$ vertices? If you answer yes, indicate the algorithm's efficiency class in the worst case; if you answer no, explain why.

6. *Team ordering* You have the results of a completed round-robin tournament in which n teams played each other once. Each game ended either with a victory for one of the teams or with a tie. Design an algorithm that lists the teams in a sequence so that every team did not lose the game with the team listed immediately after it. What is the time efficiency class of your algorithm?

7. Apply insertion sort to sort the list E, X, A, M, P, L, E in alphabetical order.

8. **a.** What sentinel should be put before the first element of an array being sorted in order to avoid checking the in-bound condition $j \geq 0$ on each iteration of the inner loop of insertion sort?

   b. Is the sentinel version in the same efficiency class as the original version?

9. Is it possible to implement insertion sort for sorting linked lists? Will it have the same $O(n^2)$ time efficiency as the array version?

10. Compare the text's implementation of insertion sort with the following version.

ALGORITHM *InsertSort2*$(A[0..n-1])$

```
for i ← 1 to n − 1 do
    j ← i − 1
    while j ≥ 0 and A[j] > A[j + 1] do
        swap(A[j], A[j + 1])
        j ← j − 1
```

What is the time efficiency of this algorithm? How is it compared to that of the version given in Section 4.1?

11. Let $A[0..n-1]$ be an array of n sortable elements. (For simplicity, you may assume that all the elements are distinct.) A pair $(A[i], A[j])$ is called an ***inversion*** if $i < j$ and $A[i] > A[j]$.

a. What arrays of size n have the largest number of inversions and what is this number? Answer the same questions for the smallest number of inversions.

b. Show that the average-case number of key comparisons in insertion sort is given by the formula

$$C_{avg}(n) \approx \frac{n^2}{4}.$$

12. Shellsort (more accurately Shell's sort) is an important sorting algorithm that works by applying insertion sort to each of several interleaving sublists of a given list. On each pass through the list, the sublists in question are formed by stepping through the list with an increment h_i taken from some predefined decreasing sequence of step sizes, $h_1 > \cdots > h_i > \cdots > 1$, which must end with 1. (The algorithm works for any such sequence, though some sequences are known to yield a better efficiency than others. For example, the sequence 1, 4, 13, 40, 121, . . . , used, of course, in reverse, is known to be among the best for this purpose.)

a. Apply shellsort to the list

$$S, H, E, L, L, S, O, R, T, I, S, U, S, E, F, U, L$$

b. Is shellsort a stable sorting algorithm?

c. Implement shellsort, straight insertion sort, selection sort, and bubble sort in the language of your choice and compare their performance on random arrays of sizes 10^n for $n = 2, 3, 4, 5$, and 6 as well as on increasing and decreasing arrays of these sizes.

4.2 Topological Sorting

In this section, we discuss an important problem for directed graphs, with a variety of applications involving prerequisite-restricted tasks. Before we pose this problem, though, let us review a few basic facts about directed graphs themselves. A ***directed graph***, or ***digraph*** for short, is a graph with directions specified for all its edges (Figure 4.5a is an example). The adjacency matrix and adjacency lists are still two principal means of representing a digraph. There are only two notable differences between undirected and directed graphs in representing them: (1) the adjacency matrix of a directed graph does not have to be symmetric; (2) an edge in a directed graph has just one (not two) corresponding nodes in the digraph's adjacency lists.

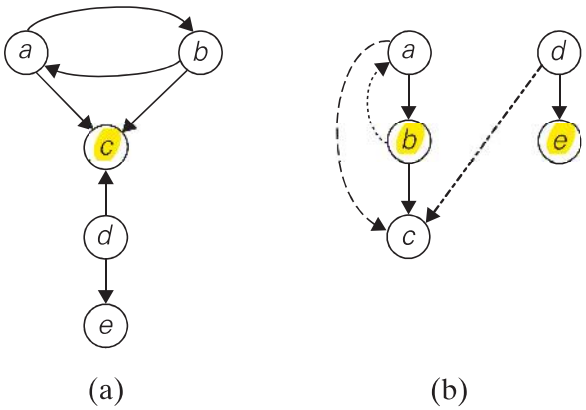


FIGURE 4.5 (a) Digraph. (b) DFS forest of the digraph for the DFS traversal started at a.

Depth-first search and breadth-first search are principal traversal algorithms for traversing digraphs as well, but the structure of corresponding forests can be more complex than for undirected graphs. Thus, even for the simple example of Figure 4.5a, the depth-first search forest (Figure 4.5b) exhibits all four types of edges possible in a DFS forest of a directed graph: ***tree edges*** (ab, bc, de), ***back edges*** (ba) from vertices to their ancestors, ***forward edges*** (ac) from vertices to their descendants in the tree other than their children, and ***cross edges*** (dc), which are none of the aforementioned types.

Note that a back edge in a DFS forest of a directed graph can connect a vertex to its parent. Whether or not it is the case, the presence of a back edge indicates that the digraph has a directed cycle. A ***directed cycle*** in a digraph is a sequence of three or more of its vertices that starts and ends with the same vertex and in which every vertex is connected to its immediate predecessor by an edge directed from the predecessor to the successor. For example, a, b, a is a directed cycle in the digraph in Figure 4.5a. Conversely, if a DFS forest of a digraph has no back edges, the digraph is a ***dag***, an acronym for ***directed acyclic graph***.

Edge directions lead to new questions about digraphs that are either meaningless or trivial for undirected graphs. In this section, we discuss one such question. As a motivating example, consider a set of five required courses {C1, C2, C3, C4, C5} a part-time student has to take in some degree program. The courses can be taken in any order as long as the following course prerequisites are met: C1 and C2 have no prerequisites, C3 requires C1 and C2, C4 requires C3, and C5 requires C3 and C4. The student can take only one course per term. In which order should the student take the courses?

The situation can be modeled by a digraph in which vertices represent courses and directed edges indicate prerequisite requirements (Figure 4.6). In terms of this digraph, the question is whether we can list its vertices in such an order that for every edge in the graph, the vertex where the edge starts is listed before the vertex where the edge ends. (Can you find such an ordering of this digraph's vertices?) This problem is called ***topological sorting***. It can be posed for an

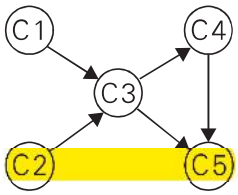


FIGURE 4.6 Digraph representing the prerequisite structure of five courses.

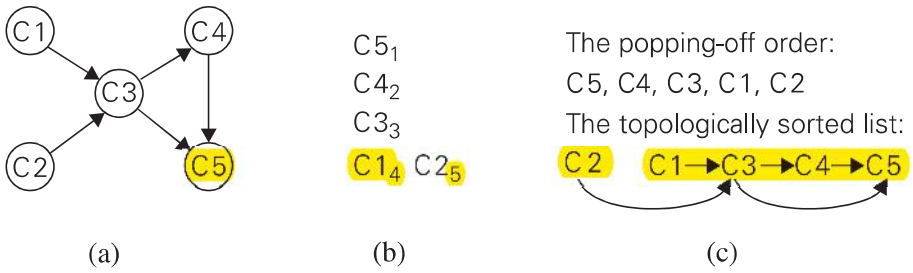


FIGURE 4.7 (a) Digraph for which the topological sorting problem needs to be solved. (b) DFS traversal stack with the subscript numbers indicating the popping-off order. (c) Solution to the problem.

arbitrary digraph, but it is easy to see that the problem cannot have a solution if a digraph has a directed cycle. Thus, for topological sorting to be possible, a digraph in question must be a dag. It turns out that being a dag is not only necessary but also sufficient for topological sorting to be possible; i.e., if a digraph has no directed cycles, the topological sorting problem for it has a solution. Moreover, there are two efficient algorithms that both verify whether a digraph is a dag and, if it is, produce an ordering of vertices that solves the topological sorting problem.

The first algorithm is a simple application of depth-first search: perform a DFS traversal and note the order in which vertices become dead-ends (i.e., popped off the traversal stack). Reversing this order yields a solution to the topological sorting problem, provided, of course, no back edge has been encountered during the traversal. If a back edge has been encountered, the digraph is not a dag, and topological sorting of its vertices is impossible.

Why does the algorithm work? When a vertex v is popped off a DFS stack, no vertex u with an edge from u to v can be among the vertices popped off before v. (Otherwise, (u, v) would have been a back edge.) Hence, any such vertex u will be listed after v in the popped-off order list, and before v in the reversed list.

Figure 4.7 illustrates an application of this algorithm to the digraph in Figure 4.6. Note that in Figure 4.7c, we have drawn the edges of the digraph, and they all point from left to right as the problem's statement requires. It is a convenient way to check visually the correctness of a solution to an instance of the topological sorting problem.

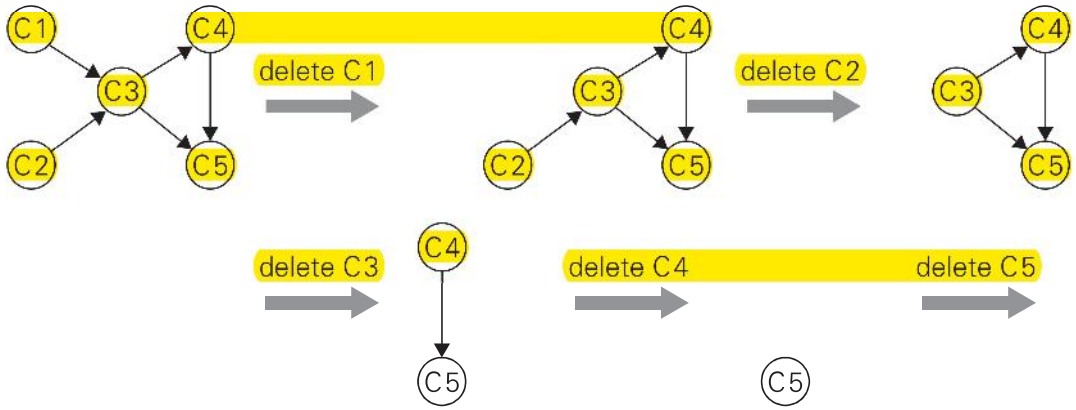


FIGURE 4.8 Illustration of the source-removal algorithm for the topological sorting problem. On each iteration, a vertex with no incoming edges is deleted from the digraph.

The second algorithm is based on a direct implementation of the decrease-(by one)-and-conquer technique: repeatedly, identify in a remaining digraph a ***source***, which is a vertex with no incoming edges, and delete it along with all the edges outgoing from it. (If there are several sources, break the tie arbitrarily. If there are none, stop because the problem cannot be solved—see Problem 6a in this section's exercises.) The order in which the vertices are deleted yields a solution to the topological sorting problem. The application of this algorithm to the same digraph representing the five courses is given in Figure 4.8.

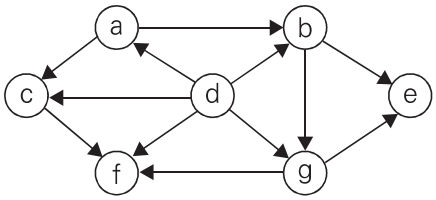
Note that the solution obtained by the source-removal algorithm is different from the one obtained by the DFS-based algorithm. Both of them are correct, of course; the topological sorting problem may have several alternative solutions.

The tiny size of the example we used might create a wrong impression about the topological sorting problem. But imagine a large project—e.g., in construction, research, or software development—that involves a multitude of interrelated tasks with known prerequisites. The first thing to do in such a situation is to make sure that the set of given prerequisites is not contradictory. The convenient way of doing this is to solve the topological sorting problem for the project's digraph. Only then can one start thinking about scheduling tasks to, say, minimize the total completion time of the project. This would require, of course, other algorithms that you can find in general books on operations research or in special ones on CPM (Critical Path Method) and PERT (Program Evaluation and Review Technique) methodologies.

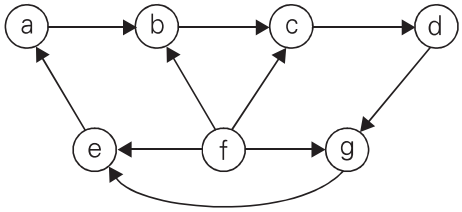
As to applications of topological sorting in computer science, they include instruction scheduling in program compilation, cell evaluation ordering in spreadsheet formulas, and resolving symbol dependencies in linkers.

Exercises 4.2

1. Apply the DFS-based algorithm to solve the topological sorting problem for the following digraphs:



(a)



(b)

2. **a.** Prove that the topological sorting problem has a solution if and only if it is a dag.
   b. For a digraph with n vertices, what is the largest number of distinct solutions the topological sorting problem can have?
3. **a.** What is the time efficiency of the DFS-based algorithm for topological sorting?
   b. How can one modify the DFS-based algorithm to avoid reversing the vertex ordering generated by DFS?
4. Can one use the order in which vertices are pushed onto the DFS stack (instead of the order they are popped off it) to solve the topological sorting problem?
5. Apply the source-removal algorithm to the digraphs of Problem 1 above.
6. **a.** Prove that a nonempty dag must have at least one source.
   b. How would you find a source (or determine that such a vertex does not exist) in a digraph represented by its adjacency matrix? What is the time efficiency of this operation?
   c. How would you find a source (or determine that such a vertex does not exist) in a digraph represented by its adjacency lists? What is the time efficiency of this operation?
7. Can you implement the source-removal algorithm for a digraph represented by its adjacency lists so that its running time is in $O(|V| + |E|)$?
8. Implement the two topological sorting algorithms in the language of your choice. Run an experiment to compare their running times.
9. A digraph is called ***strongly connected*** if for any pair of two distinct vertices u and v there exists a directed path from u to v and a directed path from v to u. In general, a digraph's vertices can be partitioned into disjoint maximal subsets of vertices that are mutually accessible via directed paths; these subsets are called ***strongly connected components*** of the digraph. There are two DFS-

4.4 Decrease-by-a-Constant-Factor Algorithms

You may recall from the introduction to this chapter that decrease-by-a-constant-factor is the second major variety of decrease-and-conquer. As an example of an algorithm based on this technique, we mentioned there exponentiation by squaring defined by formula (4.2). In this section, you will find a few other examples of such algorithms.. The most important and well-known of them is binary search. Decrease-by-a-constant-factor algorithms usually run in logarithmic time, and, being very efficient, do not happen often; a reduction by a factor other than two is especially rare.

Binary Search

Binary search is a remarkably efficient algorithm for searching in a sorted array. It works by comparing a search key K with the array's middle element $A[m]$. If they match, the algorithm stops; otherwise, the same operation is repeated recursively for the first half of the array if $K < A[m]$, and for the second half if $K > A[m]$:

$$\underbrace{A[0] \ldots A[m-1]}_{\text{search here if } K<A[m]} \quad \begin{matrix} K \\ \updownarrow \\ A[m] \end{matrix} \quad \underbrace{A[m+1] \ldots A[n-1]}_{\text{search here if } K>A[m]}.$$

As an example, let us apply binary search to searching for $K = 70$ in the array

3	14	27	31	39	42	55	70	74	81	85	93	98

The iterations of the algorithm are given in the following table:

index	0	1	2	3	4	5	6	7	8	9	10	11	12
value	3	14	27	31	39	42	55	70	74	81	85	93	98
iteration 1	l						m						r
iteration 2								l		m			r
iteration 3								l,m	r				

Though binary search is clearly based on a recursive idea, it can be easily implemented as a nonrecursive algorithm, too. Here is pseudocode of this nonrecursive version.

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

```
//Implements nonrecursive binary search
//Input: An array A[0..n − 1] sorted in ascending order and
//       a search key K
//Output: An index of the array's element that is equal to K
//       or −1 if there is no such element
l ← 0;    r ← n − 1
while l ≤ r do
    m ← ⌊(l + r)/2⌋
    if K = A[m] return m
    else if K < A[m] r ← m − 1
    else l ← m + 1
return −1
```

The standard way to analyze the efficiency of binary search is to count the number of times the search key is compared with an element of the array. Moreover, for the sake of simplicity, we will count the so-called three-way comparisons. This assumes that after one comparison of K with $A[m]$, the algorithm can determine whether K is smaller, equal to, or larger than $A[m]$.

How many such comparisons does the algorithm make on an array of n elements? The answer obviously depends not only on n but also on the specifics of a particular instance of the problem. Let us find the number of key comparisons in the worst case $C_{worst}(n)$. The worst-case inputs include all arrays that do not contain a given search key, as well as some successful searches. Since after one comparison the algorithm faces the same situation but for an array half the size, we get the following recurrence relation for $C_{worst}(n)$:

$$C_{worst}(n) = C_{worst}(\lfloor n/2 \rfloor) + 1 \quad \text{for } n > 1, \quad C_{worst}(1) = 1. \tag{4.3}$$

(Stop and convince yourself that $n/2$ must be, indeed, rounded down and that the initial condition must be written as specified.)

We already encountered recurrence (4.3), with a different initial condition, in Section 2.4 (see recurrence (2.4) and its solution there for $n = 2^k$). For the initial condition $C_{worst}(1) = 1$, we obtain

$$C_{worst}(2^k) = k + 1 = \log_2 n + 1. \tag{4.4}$$

Further, similarly to the case of recurrence (2.4) (Problem 7 in Exercises 2.4), the solution given by formula (4.4) for $n = 2^k$ can be tweaked to get a solution valid for an arbitrary positive integer n:

$$C_{worst}(n) = \lfloor \log_2 n \rfloor + 1 = \lceil \log_2 (n + 1) \rceil. \tag{4.5}$$

Formula (4.5) deserves attention. First, it implies that the worst-case time efficiency of binary search is in $\Theta(\log n)$. Second, it is the answer we should have

fully expected: since the algorithm simply reduces the size of the remaining array by about half on each iteration, the number of such iterations needed to reduce the initial size n to the final size 1 has to be about $\log_2 n$. Third, to reiterate the point made in Section 2.1, the logarithmic function grows so slowly that its values remain small even for very large values of n. In particular, according to formula (4.5), it will take no more than $\lceil \log_2(10^3 + 1) \rceil = 10$ three-way comparisons to find an element of a given value (or establish that there is no such element) in any sorted array of one thousand elements, and it will take no more than $\lceil \log_2(10^6 + 1) \rceil = 20$ comparisons to do it for any sorted array of size one million!

What can we say about the average-case efficiency of binary search? A sophisticated analysis shows that the average number of key comparisons made by binary search is only slightly smaller than that in the worst case:

$$C_{avg}(n) \approx \log_2 n.$$

(More accurate formulas for the average number of comparisons in a successful and an unsuccessful search are $C_{avg}^{yes}(n) \approx \log_2 n - 1$ and $C_{avg}^{no}(n) \approx \log_2(n + 1)$, respectively.)

Though binary search is an optimal searching algorithm if we restrict our operations only to comparisons between keys (see Section 11.2), there are searching algorithms (see interpolation search in Section 4.5 and hashing in Section 7.3) with a better average-case time efficiency, and one of them (hashing) does not even require the array to be sorted! These algorithms do require some special calculations in addition to key comparisons, however. Finally, the idea behind binary search has several applications beyond searching (see, e.g., [Ben00]). In addition, it can be applied to solving nonlinear equations in one unknown; we discuss this continuous analogue of binary search, called the method of bisection, in Section 12.4.

Fake-Coin Problem

Of several versions of the fake-coin identification problem, we consider here the one that best illustrates the decrease-by-a-constant-factor strategy. Among n identical-looking coins, one is fake. With a balance scale, we can compare any two sets of coins. That is, by tipping to the left, to the right, or staying even, the balance scale will tell whether the sets weigh the same or which of the sets is heavier than the other but not by how much. The problem is to design an efficient algorithm for detecting the fake coin. An easier version of the problem—the one we discuss here—assumes that the fake coin is known to be, say, lighter than the genuine one.[1]

The most natural idea for solving this problem is to divide n coins into two piles of $\lfloor n/2 \rfloor$ coins each, leaving one extra coin aside if n is odd, and put the two

1. A much more challenging version assumes no additional information about the relative weights of the fake and genuine coins or even the presence of the fake coin among n given coins. We pursue this more difficult version in the exercises for Section 11.2.

Chapter 5

(Divide-and-Conquer)

5

Divide-and-Conquer

Whatever man prays for, he prays for a miracle. Every prayer reduces itself to this—Great God, grant that twice two be not four.

—Ivan Turgenev (1818–1883), Russian novelist and short-story writer

Divide-and-conquer is probably the best-known general algorithm design technique. Though its fame may have something to do with its catchy name, it is well deserved: quite a few very efficient algorithms are specific implementations of this general strategy. Divide-and-conquer algorithms work according to the following general plan:

1. A problem is divided into several subproblems of the same type, ideally of about equal size.
2. The subproblems are solved (typically recursively, though sometimes a different algorithm is employed, especially when subproblems become small enough).
3. If necessary, the solutions to the subproblems are combined to get a solution to the original problem.

The divide-and-conquer technique is diagrammed in Figure 5.1, which depicts the case of dividing a problem into two smaller subproblems, by far the most widely occurring case (at least for divide-and-conquer algorithms designed to be executed on a single-processor computer).

As an example, let us consider the problem of computing the sum of n numbers $a_0, \ldots, a_{n-1}$. If $n > 1$, we can divide the problem into two instances of the same problem: to compute the sum of the first $\lfloor n/2 \rfloor$ numbers and to compute the sum of the remaining $\lceil n/2 \rceil$ numbers. (Of course, if $n = 1$, we simply return a_0 as the answer.) Once each of these two sums is computed by applying the same method recursively, we can add their values to get the sum in question:

$$a_0 + \cdots + a_{n-1} = (a_0 + \cdots + a_{\lfloor n/2 \rfloor - 1}) + (a_{\lfloor n/2 \rfloor} + \cdots + a_{n-1}).$$

Is this an efficient way to compute the sum of n numbers? A moment of reflection (why could it be more efficient than the brute-force summation?), a

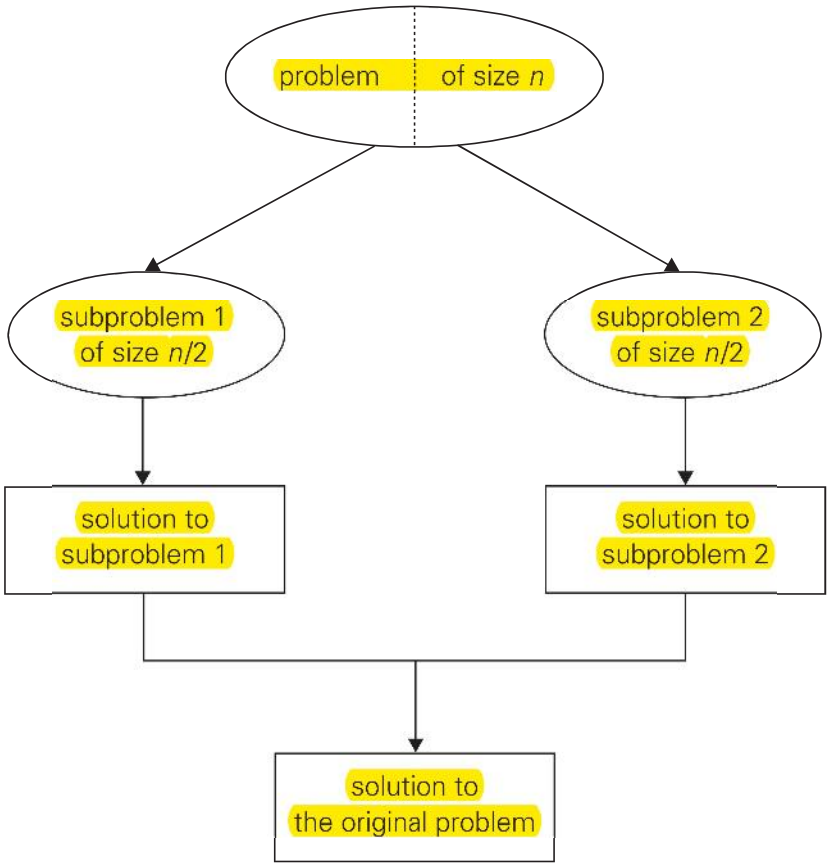


FIGURE 5.1 Divide-and-conquer technique (typical case).

small example of summing, say, four numbers by this algorithm, a formal analysis (which follows), and common sense (we do not normally compute sums this way, do we?) all lead to a negative answer to this question.[1]

Thus, not every divide-and-conquer algorithm is necessarily more efficient than even a brute-force solution. But often our prayers to the Goddess of Algorithmics—see the chapter's epigraph—*are* answered, and the time spent on executing the divide-and-conquer plan turns out to be significantly smaller than solving a problem by a different method. In fact, the divide-and-conquer approach yields some of the most important and efficient algorithms in computer science. We discuss a few classic examples of such algorithms in this chapter. Though we consider only sequential algorithms here, it is worth keeping in mind that the divide-and-conquer technique is ideally suited for parallel computations, in which each subproblem can be solved simultaneously by its own processor.

1. Actually, the divide-and-conquer algorithm, called the ***pairwise summation***, may substantially reduce the accumulated round-off error of the sum of numbers that can be represented only approximately in a digital computer [Hig93].

As mentioned above, in the most typical case of divide-and-conquer a problem's instance of size n is divided into two instances of size $n/2$. More generally, an instance of size n can be divided into b instances of size n/b, with a of them needing to be solved. (Here, a and b are constants; $a \geq 1$ and $b > 1$.) Assuming that size n is a power of b to simplify our analysis, we get the following recurrence for the running time $T(n)$:

$$T(n) = aT(n/b) + f(n), \tag{5.1}$$

where $f(n)$ is a function that accounts for the time spent on dividing an instance of size n into instances of size n/b and combining their solutions. (For the sum example above, $a = b = 2$ and $f(n) = 1$.) Recurrence (5.1) is called the ***general divide-and-conquer recurrence***. Obviously, the order of growth of its solution $T(n)$ depends on the values of the constants a and b and the order of growth of the function $f(n)$. The efficiency analysis of many divide-and-conquer algorithms is greatly simplified by the following theorem (see Appendix B).

Master Theorem If $f(n) \in \Theta(n^d)$ where $d \geq 0$ in recurrence (5.1), then

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d, \\ \Theta(n^d \log n) & \text{if } a = b^d, \\ \Theta(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

Analogous results hold for the O and Ω notations, too.

For example, the recurrence for the number of additions $A(n)$ made by the divide-and-conquer sum-computation algorithm (see above) on inputs of size $n = 2^k$ is

$$A(n) = 2A(n/2) + 1.$$

Thus, for this example, $a = 2$, $b = 2$, and $d = 0$; hence, since $a > b^d$,

$$A(n) \in \Theta(n^{\log_b a}) = \Theta(n^{\log_2 2}) = \Theta(n).$$

Note that we were able to find the solution's efficiency class without going through the drudgery of solving the recurrence. But, of course, this approach can only establish a solution's order of growth to within an unknown multiplicative constant, whereas solving a recurrence equation with a specific initial condition yields an exact answer (at least for n's that are powers of b).

It is also worth pointing out that if $a = 1$, recurrence (5.1) covers decrease-by-a-constant-factor algorithms discussed in the previous chapter. In fact, some people consider such algorithms as binary search degenerate cases of divide-and-conquer, where just one of two subproblems of half the size needs to be solved. It is better not to do this and consider decrease-by-a-constant-factor and divide-and-conquer as different design paradigms.

5.1 Mergesort

Mergesort is a perfect example of a successful application of the divide-and-conquer technique. It sorts a given array $A[0..n-1]$ by dividing it into two halves $A[0..\lfloor n/2 \rfloor - 1]$ and $A[\lfloor n/2 \rfloor..n-1]$, sorting each of them recursively, and then merging the two smaller sorted arrays into a single sorted one.

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

```
//Sorts array A[0..n − 1] by recursive mergesort
//Input: An array A[0..n − 1] of orderable elements
//Output: Array A[0..n − 1] sorted in nondecreasing order
if n > 1
    copy A[0..⌊n/2⌋ − 1] to B[0..⌊n/2⌋ − 1]
    copy A[⌊n/2⌋..n − 1] to C[0..⌈n/2⌉ − 1]
    Mergesort(B[0..⌊n/2⌋ − 1])
    Mergesort(C[0..⌈n/2⌉ − 1])
    Merge(B, C, A)    //see below
```

The ***merging*** of two sorted arrays can be done as follows. Two pointers (array indices) are initialized to point to the first elements of the arrays being merged. The elements pointed to are compared, and the smaller of them is added to a new array being constructed; after that, the index of the smaller element is incremented to point to its immediate successor in the array it was copied from. This operation is repeated until one of the two given arrays is exhausted, and then the remaining elements of the other array are copied to the end of the new array.

ALGORITHM *Merge*$(B[0..p-1], C[0..q-1], A[0..p+q-1])$

```
//Merges two sorted arrays into one sorted array
//Input: Arrays B[0..p − 1] and C[0..q − 1] both sorted
//Output: Sorted array A[0..p + q − 1] of the elements of B and C
i ← 0; j ← 0; k ← 0
while i < p and j < q do
    if B[i] ≤ C[j]
        A[k] ← B[i]; i ← i + 1
    else A[k] ← C[j]; j ← j + 1
    k ← k + 1
if i = p
    copy C[j..q − 1] to A[k..p + q − 1]
else copy B[i..p − 1] to A[k..p + q − 1]
```

The operation of the algorithm on the list 8, 3, 2, 9, 7, 1, 5, 4 is illustrated in Figure 5.2.

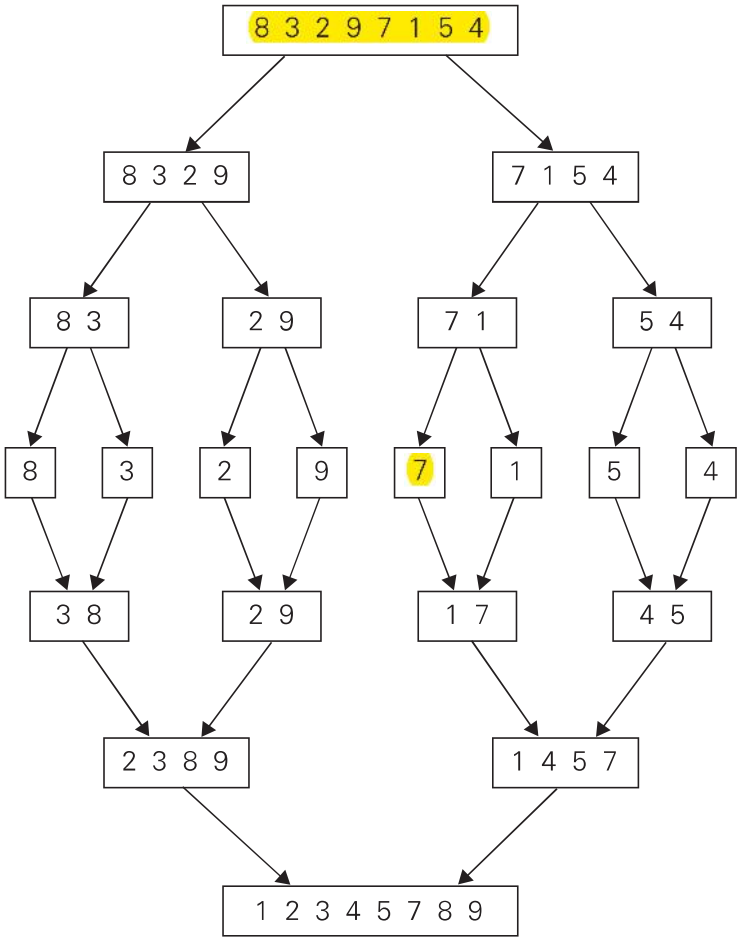


FIGURE 5.2 Example of mergesort operation.

How efficient is mergesort? Assuming for simplicity that n is a power of 2, the recurrence relation for the number of key comparisons $C(n)$ is

$$C(n) = 2C(n/2) + C_{merge}(n) \quad \text{for } n > 1, \quad C(1) = 0.$$

Let us analyze $C_{merge}(n)$, the number of key comparisons performed during the merging stage. At each step, exactly one comparison is made, after which the total number of elements in the two arrays still needing to be processed is reduced by 1. In the worst case, neither of the two arrays becomes empty before the other one contains just one element (e.g., smaller elements may come from the alternating arrays). Therefore, for the worst case, $C_{merge}(n) = n - 1$, and we have the recurrence

$$C_{worst}(n) = 2C_{worst}(n/2) + n - 1 \quad \text{for } n > 1, \quad C_{worst}(1) = 0.$$

Hence, according to the Master Theorem, $C_{worst}(n) \in \Theta(n \log n)$ (why?). In fact, it is easy to find the exact solution to the worst-case recurrence for $n = 2^k$:

$$C_{worst}(n) = n \log_2 n - n + 1.$$

The number of key comparisons made by mergesort in the worst case comes very close to the theoretical minimum[2] that any general comparison-based sorting algorithm can have. For large n, the number of comparisons made by this algorithm in the average case turns out to be about $0.25n$ less (see [Gon91, p. 173]) and hence is also in $\Theta(n \log n)$. A noteworthy advantage of mergesort over quicksort and heapsort—the two important advanced sorting algorithms to be discussed later—is its stability (see Problem 7 in this section's exercises). The principal shortcoming of mergesort is the linear amount of extra storage the algorithm requires. Though merging can be done in-place, the resulting algorithm is quite complicated and of theoretical interest only.

There are two main ideas leading to several variations of mergesort. First, the algorithm can be implemented bottom up by merging pairs of the array's elements, then merging the sorted pairs, and so on. (If n is not a power of 2, only slight bookkeeping complications arise.) This avoids the time and space overhead of using a stack to handle recursive calls. Second, we can divide a list to be sorted in more than two parts, sort each recursively, and then merge them together. This scheme, which is particularly useful for sorting files residing on secondary memory devices, is called ***multiway mergesort***.

Exercises 5.1

1. **a.** Write pseudocode for a divide-and-conquer algorithm for finding the position of the largest element in an array of n numbers.
   b. What will be your algorithm's output for arrays with several elements of the largest value?
   c. Set up and solve a recurrence relation for the number of key comparisons made by your algorithm.
   d. How does this algorithm compare with the brute-force algorithm for this problem?
2. **a.** Write pseudocode for a divide-and-conquer algorithm for finding values of both the largest and smallest elements in an array of n numbers.
   b. Set up and solve (for $n = 2^k$) a recurrence relation for the number of key comparisons made by your algorithm.
   c. How does this algorithm compare with the brute-force algorithm for this problem?
3. **a.** Write pseudocode for a divide-and-conquer algorithm for the exponentiation problem of computing a^n where n is a positive integer.
   b. Set up and solve a recurrence relation for the number of multiplications made by this algorithm.

2. As we shall see in Section 11.2, this theoretical minimum is $\lceil \log_2 n! \rceil \approx \lceil n \log_2 n - 1.44n \rceil$.

c. How does this algorithm compare with the brute-force algorithm for this problem?

4. As mentioned in Chapter 2, logarithm bases are irrelevant in most contexts arising in analyzing an algorithm's efficiency class. Is this true for both assertions of the Master Theorem that include logarithms?

5. Find the order of growth for solutions of the following recurrences.

a. $T(n) = 4T(n/2) + n$, $T(1) = 1$

b. $T(n) = 4T(n/2) + n^2$, $T(1) = 1$

c. $T(n) = 4T(n/2) + n^3$, $T(1) = 1$

6. Apply mergesort to sort the list E, X, A, M, P, L, E in alphabetical order.

7. Is mergesort a stable sorting algorithm?

8. a. Solve the recurrence relation for the number of key comparisons made by mergesort in the worst case. You may assume that $n = 2^k$.

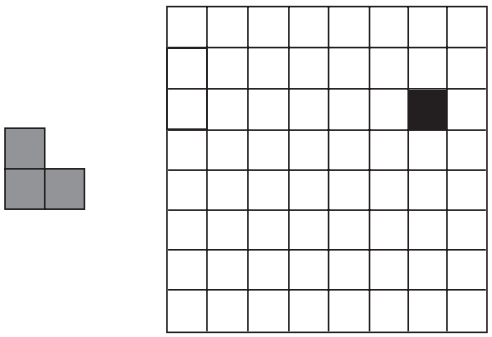
b. Set up a recurrence relation for the number of key comparisons made by mergesort on best-case inputs and solve it for $n = 2^k$.

c. Set up a recurrence relation for the number of key moves made by the version of mergesort given in Section 5.1. Does taking the number of key moves into account change the algorithm's efficiency class?

9. Let $A[0..n-1]$ be an array of n real numbers. A pair $(A[i], A[j])$ is said to be an ***inversion*** if these numbers are out of order, i.e., $i < j$ but $A[i] > A[j]$. Design an $O(n \log n)$ algorithm for counting the number of inversions.

10. Implement the bottom-up version of mergesort in the language of your choice.

11. *Tromino puzzle* A tromino (more accurately, a right tromino) is an L-shaped tile formed by three 1×1 squares. The problem is to cover any $2^n \times 2^n$ chessboard with a missing square with trominoes. Trominoes can be oriented in an arbitrary way, but they should cover all the squares of the board except the missing one exactly and with no overlaps. [Gol94]



Design a divide-and-conquer algorithm for this problem.

5.2 Quicksort

Quicksort is the other important sorting algorithm that is based on the divide-and-conquer approach. Unlike mergesort, which divides its input elements according to their position in the array, quicksort divides them according to their value. We already encountered this idea of an array partition in Section 4.5, where we discussed the selection problem. A partition is an arrangement of the array's elements so that all the elements to the left of some element $A[s]$ are less than or equal to $A[s]$, and all the elements to the right of $A[s]$ are greater than or equal to it:

$$\underbrace{A[0] \dots A[s-1]}_{\text{all are} \le A[s]} \quad A[s] \quad \underbrace{A[s+1] \dots A[n-1]}_{\text{all are} \ge A[s]}$$

Obviously, after a partition is achieved, $A[s]$ will be in its final position in the sorted array, and we can continue sorting the two subarrays to the left and to the right of $A[s]$ independently (e.g., by the same method). Note the difference with mergesort: there, the division of the problem into two subproblems is immediate and the entire work happens in combining their solutions; here, the entire work happens in the division stage, with no work required to combine the solutions to the subproblems.

Here is pseudocode of quicksort: call $Quicksort(A[0..n-1])$ where

ALGORITHM $Quicksort(A[l..r])$

```
//Sorts a subarray by quicksort
//Input: Subarray of array A[0..n − 1], defined by its left and right
//         indices l and r
//Output: Subarray A[l..r] sorted in nondecreasing order
if l < r
    s ←Partition(A[l..r])  //s is a split position
    Quicksort(A[l..s − 1])
    Quicksort(A[s + 1..r])
```

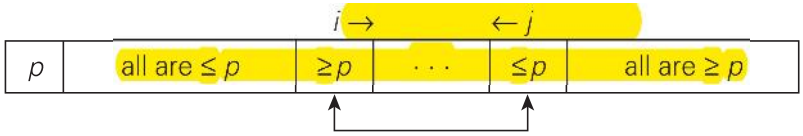
As a partition algorithm, we can certainly use the Lomuto partition discussed in Section 4.5. Alternatively, we can partition $A[0..n-1]$ and, more generally, its subarray $A[l..r]$ $(0 \le l < r \le n-1)$ by the more sophisticated method suggested by C.A.R. Hoare, the prominent British computer scientist who invented quicksort.[3]

3. C.A.R. Hoare, at age 26, invented his algorithm in 1960 while trying to sort words for a machine translation project from Russian to English. Says Hoare, "My first thought on how to do this was bubblesort and, by an amazing stroke of luck, my second thought was Quicksort." It is hard to disagree with his overall assessment: "I have been very lucky. What a wonderful way to start a career in Computing, by discovering a new sorting algorithm!" [Hoa96]. Twenty years later, he received the Turing Award for "fundamental contributions to the definition and design of programming languages"; in 1980, he was also knighted for services to education and computer science.

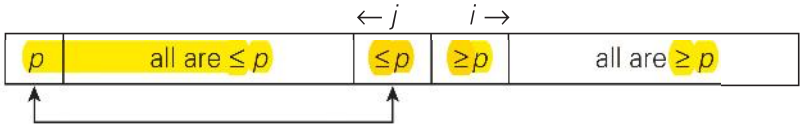
As before, we start by selecting a pivot—an element with respect to whose value we are going to divide the subarray. There are several different strategies for selecting a pivot; we will return to this issue when we analyze the algorithm's efficiency. For now, we use the simplest strategy of selecting the subarray's first element: $p = A[l]$.

Unlike the Lomuto algorithm, we will now scan the subarray from both ends, comparing the subarray's elements to the pivot. The left-to-right scan, denoted below by index pointer i, starts with the second element. Since we want elements smaller than the pivot to be in the left part of the subarray, this scan skips over elements that are smaller than the pivot and stops upon encountering the first element greater than or equal to the pivot. The right-to-left scan, denoted below by index pointer j, starts with the last element of the subarray. Since we want elements larger than the pivot to be in the right part of the subarray, this scan skips over elements that are larger than the pivot and stops on encountering the first element smaller than or equal to the pivot. (Why is it worth stopping the scans after encountering an element equal to the pivot? Because doing this tends to yield more even splits for arrays with a lot of duplicates, which makes the algorithm run faster. For example, if we did otherwise for an array of n equal elements, we would have gotten a split into subarrays of sizes $n - 1$ and 0, reducing the problem size just by 1 after scanning the entire array.)

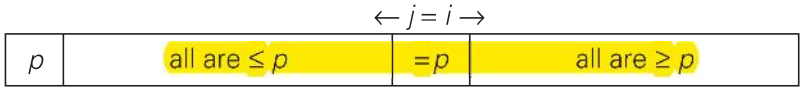
After both scans stop, three situations may arise, depending on whether or not the scanning indices have crossed. If scanning indices i and j have not crossed, i.e., $i < j$, we simply exchange $A[i]$ and $A[j]$ and resume the scans by incrementing i and decrementing j, respectively:



If the scanning indices have crossed over, i.e., $i > j$, we will have partitioned the subarray after exchanging the pivot with $A[j]$:



Finally, if the scanning indices stop while pointing to the same element, i.e., $i = j$, the value they are pointing to must be equal to p (why?). Thus, we have the subarray partitioned, with the split position $s = i = j$:



We can combine the last case with the case of crossed-over indices ($i > j$) by exchanging the pivot with $A[j]$ whenever $i \geq j$.

Here is pseudocode implementing this partitioning procedure.

ALGORITHM *HoarePartition*$(A[l..r])$

```
//Partitions a subarray by Hoare's algorithm, using the first element
//        as a pivot
//Input: Subarray of array A[0..n − 1], defined by its left and right
//        indices l and r (l < r)
//Output: Partition of A[l..r], with the split position returned as
//        this function's value
p ← A[l]
i ← l; j ← r + 1
repeat
    repeat i ← i + 1 until A[i] ≥ p
    repeat j ← j − 1 until A[j] ≤ p
    swap(A[i], A[j])
until i ≥ j
swap(A[i], A[j])   //undo last swap when i ≥ j
swap(A[l], A[j])
return j
```

Note that index i can go out of the subarray's bounds in this pseudocode. Rather than checking for this possibility every time index i is incremented, we can append to array $A[0..n-1]$ a "sentinel" that would prevent index i from advancing beyond position n. Note that the more sophisticated method of pivot selection mentioned at the end of the section makes such a sentinel unnecessary.

An example of sorting an array by quicksort is given in Figure 5.3.

We start our discussion of quicksort's efficiency by noting that the number of key comparisons made before a partition is achieved is $n + 1$ if the scanning indices cross over and n if they coincide (why?). If all the splits happen in the middle of corresponding subarrays, we will have the best case. The number of key comparisons in the best case satisfies the recurrence

$$C_{best}(n) = 2C_{best}(n/2) + n \quad \text{for } n > 1, \quad C_{best}(1) = 0.$$

According to the Master Theorem, $C_{best}(n) \in \Theta(n \log_2 n)$; solving it exactly for $n = 2^k$ yields $C_{best}(n) = n \log_2 n$.

In the worst case, all the splits will be skewed to the extreme: one of the two subarrays will be empty, and the size of the other will be just 1 less than the size of the subarray being partitioned. This unfortunate situation will happen, in particular, for increasing arrays, i.e., for inputs for which the problem is already solved! Indeed, if $A[0..n-1]$ is a strictly increasing array and we use $A[0]$ as the pivot, the left-to-right scan will stop on $A[1]$ while the right-to-left scan will go all the way to reach $A[0]$, indicating the split at position 0:

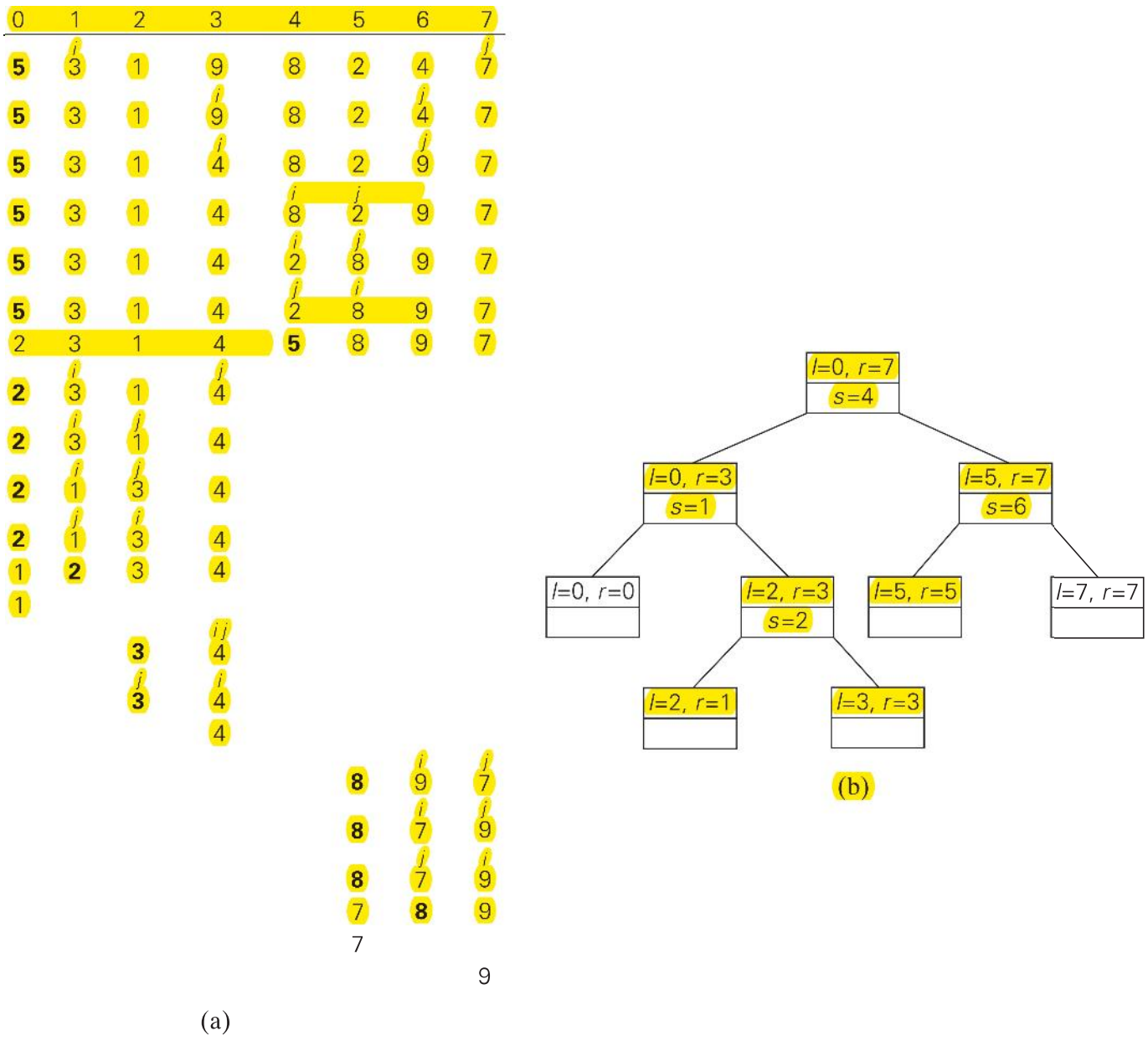
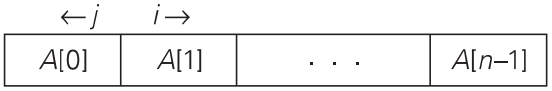


FIGURE 5.3 Example of quicksort operation. (a) Array's transformations with pivots shown in bold. (b) Tree of recursive calls to *Quicksort* with input values l and r of subarray bounds and split position s of a partition obtained.



So, after making $n + 1$ comparisons to get to this partition and exchanging the pivot $A[0]$ with itself, the algorithm will be left with the strictly increasing array $A[1..n-1]$ to sort. This sorting of strictly increasing arrays of diminishing sizes will

the size of this stack can be made to be in $O(\log n)$ by always sorting first the smaller of two subarrays obtained by partitioning, it is worse than the $O(1)$ space efficiency of heapsort. Although more sophisticated ways of choosing a pivot make the quadratic running time of the worst case very unlikely, they do not eliminate it completely. And even the performance on randomly ordered arrays is known to be sensitive not only to implementation details of the algorithm but also to both computer architecture and data type. Still, the January/February 2000 issue of *Computing in Science & Engineering,* a joint publication of the American Institute of Physics and the IEEE Computer Society, selected quicksort as one of the 10 algorithms "with the greatest influence on the development and practice of science and engineering in the 20th century."

Exercises 5.2

1. Apply quicksort to sort the list E, X, A, M, P, L, E in alphabetical order. Draw the tree of the recursive calls made.

2. For the partitioning procedure outlined in this section:

a. Prove that if the scanning indices stop while pointing to the same element, i.e., $i = j$, the value they are pointing to must be equal to p.

b. Prove that when the scanning indices stop, j cannot point to an element more than one position to the left of the one pointed to by i.

3. Give an example showing that quicksort is not a stable sorting algorithm.

4. Give an example of an array of n elements for which the sentinel mentioned in the text is actually needed. What should be its value? Also explain why a single sentinel suffices for any input.

5. For the version of quicksort given in this section:

a. Are arrays made up of all equal elements the worst-case input, the best-case input, or neither?

b. Are strictly decreasing arrays the worst-case input, the best-case input, or neither?

6. a. For quicksort with the median-of-three pivot selection, are strictly increasing arrays the worst-case input, the best-case input, or neither?

b. Answer the same question for strictly decreasing arrays.

7. a. Estimate how many times faster quicksort will sort an array of one million random numbers than insertion sort.

b. True or false: For every $n > 1$, there are n-element arrays that are sorted faster by insertion sort than by quicksort?

8. Design an algorithm to rearrange elements of a given array of n real numbers so that all its negative elements precede all its positive elements. Your algorithm should be both time efficient and space efficient.

Chapter 6

(Transform-and-Conquer)

6

Transform-and-Conquer

That's the secret to life . . . replace one worry with another.

—Charles M. Schulz (1922–2000), American cartoonist,
the creator of *Peanuts*

This chapter deals with a group of design methods that are based on the idea of transformation. We call this general technique ***transform-and-conquer*** because these methods work as two-stage procedures. First, in the transformation stage, the problem's instance is modified to be, for one reason or another, more amenable to solution. Then, in the second or conquering stage, it is solved.

There are three major variations of this idea that differ by what we transform a given instance to (Figure 6.1):

- Transformation to a simpler or more convenient instance of the same problem—we call it ***instance simplification***.
- Transformation to a different representation of the same instance—we call it ***representation change***.
- Transformation to an instance of a different problem for which an algorithm is already available—we call it ***problem reduction***.

In the first three sections of this chapter, we encounter examples of the instance-simplification variety. Section 6.1 deals with the simple but fruitful idea of presorting. Many algorithmic problems are easier to solve if their input is sorted. Of course, the benefits of sorting should more than compensate for the

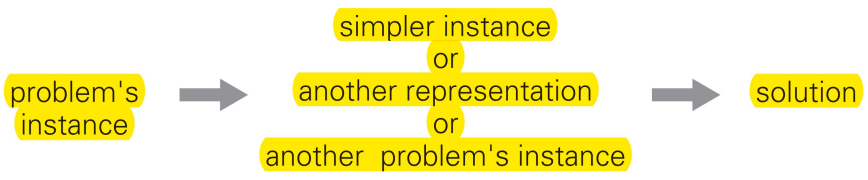


FIGURE 6.1 Transform-and-conquer strategy.

11. *Anagram detection*

a. Design an efficient algorithm for finding all sets of anagrams in a large file such as a dictionary of English words [Ben00]. For example, *eat, ate,* and *tea* belong to one such set.

b. Write a program implementing the algorithm.

6.2 Gaussian Elimination

You are certainly familiar with systems of two linear equations in two unknowns:

$$\begin{aligned} a_{11}x + a_{12}y &= b_1 \\ a_{21}x + a_{22}y &= b_2. \end{aligned}$$

Recall that unless the coefficients of one equation are proportional to the coefficients of the other, the system has a unique solution. The standard method for finding this solution is to use either equation to express one of the variables as a function of the other and then substitute the result into the other equation, yielding a linear equation whose solution is then used to find the value of the second variable.

In many applications, we need to solve a system of n equations in n unknowns:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n \end{aligned}$$

where n is a large number. Theoretically, we can solve such a system by generalizing the substitution method for solving systems of two linear equations (what general design technique would such a method be based upon?); however, the resulting algorithm would be extremely cumbersome.

Fortunately, there is a much more elegant algorithm for solving systems of linear equations called ***Gaussian elimination.***[2] The idea of Gaussian elimination is to transform a system of n linear equations in n unknowns to an equivalent system (i.e., a system with the same solution as the original one) with an upper-triangular coefficient matrix, a matrix with all zeros below its main diagonal:

2. The method is named after Carl Friedrich Gauss (1777–1855), who—like other giants in the history of mathematics such as Isaac Newton and Leonhard Euler—made numerous fundamental contributions to both theoretical and computational mathematics. The method was known to the Chinese 1800 years before the Europeans rediscovered it.

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n \end{aligned} \quad \Longrightarrow \quad \begin{aligned} a'_{11}x_1 + a'_{12}x_2 + \cdots + a'_{1n}x_n &= b'_1 \\ a'_{22}x_2 + \cdots + a'_{2n}x_n &= b'_2 \\ &\vdots \\ a'_{nn}x_n &= b'_n. \end{aligned}$$

In matrix notations, we can write this as

$$Ax = b \quad \Longrightarrow \quad A'x = b',$$

where

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & & & \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix}, \quad b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}, \quad A' = \begin{bmatrix} a'_{11} & a'_{12} & \dots & a'_{1n} \\ 0 & a'_{22} & \dots & a'_{2n} \\ \vdots & & & \\ 0 & 0 & \dots & a'_{nn} \end{bmatrix}, \quad b = \begin{bmatrix} b'_1 \\ b'_2 \\ \vdots \\ b'_n \end{bmatrix}.$$

(We added primes to the matrix elements and right-hand sides of the new system to stress the point that their values differ from their counterparts in the original system.)

Why is the system with the upper-triangular coefficient matrix better than a system with an arbitrary coefficient matrix? Because we can easily solve the system with an upper-triangular coefficient matrix by back substitutions as follows. First, we can immediately find the value of x_n from the last equation; then we can substitute this value into the next to last equation to get x_{n-1}, and so on, until we substitute the known values of the last $n-1$ variables into the first equation, from which we find the value of x_1.

So how can we get from a system with an arbitrary coefficient matrix A to an equivalent system with an upper-triangular coefficient matrix A'? We can do that through a series of the so-called ***elementary operations***:

- exchanging two equations of the system
- replacing an equation with its nonzero multiple
- replacing an equation with a sum or difference of this equation and some multiple of another equation

Since no elementary operation can change a solution to a system, any system that is obtained through a series of such operations will have the same solution as the original one.

Let us see how we can get to a system with an upper-triangular matrix. First, we use a_{11} as a ***pivot*** to make all x_1 coefficients zeros in the equations below the first one. Specifically, we replace the second equation with the difference between it and the first equation multiplied by a_{21}/a_{11} to get an equation with a zero coefficient for x_1. Doing the same for the third, fourth, and finally nth equation—with the multiples a_{31}/a_{11}, a_{41}/a_{11}, $\dots$, a_{n1}/a_{11} of the first equation, respectively—makes all the coefficients of x_1 below the first equation zero. Then we get rid of all the coefficients of x_2 by subtracting an appropriate multiple of the second equation from each of the equations below the second one. Repeating this

elimination for each of the first $n-1$ variables ultimately yields a system with an upper-triangular coefficient matrix.

Before we look at an example of Gaussian elimination, let us note that we can operate with just a system's coefficient matrix augmented, as its $(n+1)$st column, with the equations' right-hand side values. In other words, we need to write explicitly neither the variable names nor the plus and equality signs.

EXAMPLE 1 Solve the system by Gaussian elimination.

$$\begin{aligned} 2x_1 - x_2 + x_3 &= 1 \\ 4x_1 + x_2 - x_3 &= 5 \\ x_1 + x_2 + x_3 &= 0. \end{aligned}$$

$$\begin{bmatrix} 2 & -1 & 1 & 1 \\ 4 & 1 & -1 & 5 \\ 1 & 1 & 1 & 0 \end{bmatrix} \begin{matrix} \\ \text{row } 2 - \frac{4}{2} \text{ row } 1 \\ \text{row } 3 - \frac{1}{2} \text{ row } 1 \end{matrix}$$

$$\begin{bmatrix} 2 & -1 & 1 & 1 \\ 0 & 3 & -3 & 3 \\ 0 & \frac{3}{2} & \frac{1}{2} & -\frac{1}{2} \end{bmatrix} \quad \text{row } 3 - \frac{1}{2} \text{ row } 2$$

$$\begin{bmatrix} 2 & -1 & 1 & 1 \\ 0 & 3 & -3 & 3 \\ 0 & 0 & 2 & -2 \end{bmatrix}$$

Now we can obtain the solution by back substitutions:

$$x_3 = (-2)/2 = -1, \quad x_2 = (3 - (-3)x_3)/3 = 0, \quad \text{and} \quad x_1 = (1 - x_3 - (-1)x_2)/2 = 1.$$

■

Here is pseudocode of the first stage, called ***forward elimination***, of the algorithm.

ALGORITHM *ForwardElimination*$(A[1..n, 1..n], b[1..n])$

```
//Applies Gaussian elimination to matrix A of a system's coefficients,
//augmented with vector b of the system's right-hand side values
//Input: Matrix A[1..n, 1..n] and column-vector b[1..n]
//Output: An equivalent upper-triangular matrix in place of A with the
//corresponding right-hand side values in the (n + 1)st column
for i ← 1 to n do A[i, n + 1] ← b[i]    //augments the matrix
for i ← 1 to n − 1 do
    for j ← i + 1 to n do
        for k ← i to n + 1 do
            A[j, k] ← A[j, k] − A[i, k] ∗ A[j, i] / A[i, i]
```

not equal to zero. Moreover, this solution can be found by the formulas called ***Cramer's rule***,

$$x_1 = \frac{\det A_1}{\det A}, \ldots, x_j = \frac{\det A_j}{\det A}, \ldots, x_n = \frac{\det A_n}{\det A},$$

where $\det A_j$ is the determinant of the matrix obtained by replacing the jth column of A by the column b. You are asked to investigate in the exercises whether using Cramer's rule is a good algorithm for solving systems of linear equations.

Exercises 6.2

1. Solve the following system by Gaussian elimination:

$$\begin{aligned} x_1 + x_2 + x_3 &= 2 \\ 2x_1 + x_2 + x_3 &= 3 \\ x_1 - x_2 + 3x_3 &= 8. \end{aligned}$$

2. a. Solve the system of the previous question by the LU decomposition method.

b. From the standpoint of general algorithm design techniques, how would you classify the LU decomposition method?

3. Solve the system of Problem 1 by computing the inverse of its coefficient matrix and then multiplying it by the vector on the right-hand side.

4. Would it be correct to get the efficiency class of the forward elimination stage of Gaussian elimination as follows?

$$\begin{aligned} C(n) &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \sum_{k=i}^{n+1} 1 = \sum_{i=1}^{n-1} (n+2-i)(n-i) \\ &= \sum_{i=1}^{n-1} [(n+2)n - i(2n+2) + i^2] \\ &= \sum_{i=1}^{n-1} (n+2)n - \sum_{i=1}^{n-1} (2n+2)i + \sum_{i=1}^{n-1} i^2. \end{aligned}$$

Since $s_1(n) = \sum_{i=1}^{n-1} (n+2)n \in \Theta(n^3)$, $s_2(n) = \sum_{i=1}^{n-1} (2n+2)i \in \Theta(n^3)$, and $s_3(n) = \sum_{i=1}^{n-1} i^2 \in \Theta(n^3)$, $s_1(n) - s_2(n) + s_3(n) \in \Theta(n^3)$.

5. Write pseudocode for the back-substitution stage of Gaussian elimination and show that its running time is in $\Theta(n^2)$.

6. Assuming that division of two numbers takes three times longer than their multiplication, estimate how much faster *BetterForwardElimination* is than *ForwardElimination*. (Of course, you should also assume that a compiler is not going to eliminate the inefficiency in *ForwardElimination*.)

a. randomly generated files of integers in the range $[1..n]$.

b. increasing files of integers $1, 2, \ldots, n$.

c. decreasing files of integers $n, n-1, \ldots, 1$.

12. *Spaghetti sort* Imagine a handful of uncooked spaghetti, individual rods whose lengths represent numbers that need to be sorted.

a. Outline a "spaghetti sort"—a sorting algorithm that takes advantage of this unorthodox representation.

b. What does this example of computer science folklore (see [Dew93]) have to do with the topic of this chapter in general and heapsort in particular?

6.5 Horner's Rule and Binary Exponentiation

In this section, we discuss the problem of computing the value of a polynomial

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0 \qquad \textbf{(6.1)}$$

at a given point x and its important special case of computing x^n. Polynomials constitute the most important class of functions because they possess a wealth of good properties on the one hand and can be used for approximating other types of functions on the other. The problem of manipulating polynomials efficiently has been important for several centuries; new discoveries were still being made the last 50 years. By far the most important of them was the ***fast Fourier transform (FFT)***. The practical importance of this remarkable algorithm, which is based on representing a polynomial by its values at specially chosen points, was such that some people consider it one of the most important algorithmic discoveries of all time. Because of its relative complexity, we do not discuss the FFT algorithm in this book. An interested reader will find a wealth of literature on the subject, including reasonably accessible treatments in such textbooks as [Kle06] and [Cor09].

Horner's Rule

Horner's rule is an old but very elegant and efficient algorithm for evaluating a polynomial. It is named after the British mathematician W. G. Horner, who published it in the early 19th century. But according to Knuth [KnuII, p. 486], the method was used by Isaac Newton 150 years before Horner. You will appreciate this method much more if you first design an algorithm for the polynomial evaluation problem by yourself and investigate its efficiency (see Problems 1 and 2 in this section's exercises).

Horner's rule is a good example of the representation-change technique since it is based on representing $p(x)$ by a formula different from (6.1). This new formula is obtained from (6.1) by successively taking x as a common factor in the remaining polynomials of diminishing degrees:

$$p(x) = (\cdots(a_n x + a_{n-1})x + \cdots)x + a_0. \qquad \textbf{(6.2)}$$

For example, for the polynomial $p(x) = 2x^4 - x^3 + 3x^2 + x - 5$, we get

$$\begin{aligned} p(x) &= 2x^4 - x^3 + 3x^2 + x - 5 \\ &= x(2x^3 - x^2 + 3x + 1) - 5 \\ &= x(x(2x^2 - x + 3) + 1) - 5 \\ &= x(x(x(2x - 1) + 3) + 1) - 5. \end{aligned} \tag{6.3}$$

It is in formula (6.2) that we will substitute a value of x at which the polynomial needs to be evaluated. It is hard to believe that this is a way to an efficient algorithm, but the unpleasant appearance of formula (6.2) is just that, an appearance. As we shall see, there is no need to go explicitly through the transformation leading to it: all we need is an original list of the polynomial's coefficients.

The pen-and-pencil calculation can be conveniently organized with a two-row table. The first row contains the polynomial's coefficients (including all the coefficients equal to zero, if any) listed from the highest a_n to the lowest a_0. Except for its first entry, which is a_n, the second row is filled left to right as follows: the next entry is computed as the x's value times the last entry in the second row plus the next coefficient from the first row. The final entry computed in this fashion is the value being sought.

EXAMPLE 1 Evaluate $p(x) = 2x^4 - x^3 + 3x^2 + x - 5$ at $x = 3$.

coefficients	2	-1	3	1	-5
$x = 3$	2	$3 \cdot 2 + (-1) = 5$	$3 \cdot 5 + 3 = 18$	$3 \cdot 18 + 1 = 55$	$3 \cdot 55 + (-5) = 160$

Thus, $p(3) = 160$. (On comparing the table's entries with formula (6.3), you will see that $3 \cdot 2 + (-1) = 5$ is the value of $2x - 1$ at $x = 3$, $3 \cdot 5 + 3 = 18$ is the value of $x(2x - 1) + 3$ at $x = 3$, $3 \cdot 18 + 1 = 55$ is the value of $x(x(2x - 1) + 3) + 1$ at $x = 3$, and, finally, $3 \cdot 55 + (-5) = 160$ is the value of $x(x(x(2x - 1) + 3) + 1) - 5 = p(x)$ at $x = 3$.) ■

Pseudocode of this algorithm is the shortest one imaginable for a nontrivial algorithm:

ALGORITHM *Horner*$(P[0..n], x)$

```
//Evaluates a polynomial at a given point by Horner's rule
//Input: An array P[0..n] of coefficients of a polynomial of degree n,
//       stored from the lowest to the highest and a number x
//Output: The value of the polynomial at x
p ← P[n]
for i ← n − 1 downto 0 do
    p ← x ∗ p + P[i]
return p
```

The number of multiplications and the number of additions are given by the same sum:

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

To appreciate how efficient Horner's rule is, consider only the first term of a polynomial of degree n: $a_n x^n$. Just computing this single term by the brute-force algorithm would require n multiplications, whereas Horner's rule computes, in addition to this term, $n - 1$ other terms, and it still uses the same number of multiplications! It is not surprising that Horner's rule is an optimal algorithm for polynomial evaluation without preprocessing the polynomial's coefficients. But it took scientists 150 years after Horner's publication to come to the realization that such a question was worth investigating.

Horner's rule also has some useful byproducts. The intermediate numbers generated by the algorithm in the process of evaluating $p(x)$ at some point x_0 turn out to be the coefficients of the quotient of the division of $p(x)$ by $x - x_0$, and the final result, in addition to being $p(x_0)$, is equal to the remainder of this division. Thus, according to Example 1, the quotient and the remainder of the division of $2x^4 - x^3 + 3x^2 + x - 5$ by $x - 3$ are $2x^3 + 5x^2 + 18x + 55$ and 160, respectively. This division algorithm, known as ***synthetic division***, is more convenient than so-called long division.

Binary Exponentiation

The amazing efficiency of Horner's rule fades if the method is applied to computing a^n, which is the value of x^n at $x = a$. In fact, it degenerates to the brute-force multiplication of a by itself, with wasteful additions of zeros in between. Since computing a^n (actually, $a^n \bmod m$) is an essential operation in several important primality-testing and encryption methods, we consider now two algorithms for computing a^n that are based on the representation-change idea. They both exploit the binary representation of exponent n, but one of them processes this binary string left to right, whereas the second does it right to left.

Let

$$n = b_I \dots b_i \dots b_0$$

be the bit string representing a positive integer n in the binary number system. This means that the value of n can be computed as the value of the polynomial

$$p(x) = b_I x^I + \cdots + b_i x^i + \cdots + b_0 \qquad \textbf{(6.4)}$$

at $x = 2$. For example, if $n = 13$, its binary representation is 1101 and

$$13 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0.$$

Let us now compute the value of this polynomial by applying Horner's rule and see what the method's operations imply for computing the power

$$a^n = a^{p(2)} = a^{b_I 2^I + \cdots + b_i 2^i + \cdots + b_0}.$$

Chapter 7

(Space and Time Trade-Offs)

7

Space and Time Trade-Offs

Things which matter most must never be at the mercy of things which matter less.

—Johann Wolfgang von Göethe (1749–1832)

Space and time trade-offs in algorithm design are a well-known issue for both theoreticians and practitioners of computing. Consider, as an example, the problem of computing values of a function at many points in its domain. If it is time that is at a premium, we can precompute the function's values and store them in a table. This is exactly what human computers had to do before the advent of electronic computers, in the process burdening libraries with thick volumes of mathematical tables. Though such tables have lost much of their appeal with the widespread use of electronic computers, the underlying idea has proven to be quite useful in the development of several important algorithms for other problems. In somewhat more general terms, the idea is to preprocess the problem's input, in whole or in part, and store the additional information obtained to accelerate solving the problem afterward. We call this approach ***input enhancement***[1] and discuss the following algorithms based on it:

- counting methods for sorting (Section 7.1)
- Boyer-Moore algorithm for string matching and its simplified version suggested by Horspool (Section 7.2)

The other type of technique that exploits space-for-time trade-offs simply uses extra space to facilitate faster and/or more flexible access to the data. We call this approach ***prestructuring***. This name highlights two facets of this variation of the space-for-time trade-off: some processing is done before a problem in question

1. The standard terms used synonymously for this technique are ***preprocessing*** and ***preconditioning***. Confusingly, these terms can also be applied to methods that use the idea of preprocessing but do not use extra space (see Chapter 6). Thus, in order to avoid confusion, we use "input enhancement" as a special name for the space-for-time trade-off technique being discussed here.

4. Is the distribution-counting algorithm stable?

5. Design a one-line algorithm for sorting any array of size n whose values are n distinct integers from 1 to n.

6. The ***ancestry problem*** asks to determine whether a vertex u is an ancestor of vertex v in a given binary (or, more generally, rooted ordered) tree of n vertices. Design a $O(n)$ input-enhancement algorithm that provides sufficient information to solve this problem for any pair of the tree's vertices in constant time.

7. The following technique, known as ***virtual initialization***, provides a time-efficient way to initialize just some elements of a given array $A[0..n-1]$ so that for each of its elements, we can say in constant time whether it has been initialized and, if it has been, with which value. This is done by utilizing a variable $counter$ for the number of initialized elements in A and two auxiliary arrays of the same size, say $B[0..n-1]$ and $C[0..n-1]$, defined as follows. $B[0], \ldots, B[counter-1]$ contain the indices of the elements of A that were initialized: $B[0]$ contains the index of the element initialized first, $B[1]$ contains the index of the element initialized second, etc. Furthermore, if $A[i]$ was the kth element ($0 \le k \le counter-1$) to be initialized, $C[i]$ contains k.

   a. Sketch the state of arrays $A[0..7]$, $B[0..7]$, and $C[0..7]$ after the three assignments

   $$A[3] \leftarrow x; \qquad A[7] \leftarrow z; \qquad A[1] \leftarrow y.$$

   b. In general, how can we check with this scheme whether $A[i]$ has been initialized and, if it has been, with which value?

8. *Least distance sorting* There are 10 Egyptian stone statues standing in a row in an art gallery hall. A new curator wants to move them so that the statues are ordered by their height. How should this be done to minimize the total distance that the statues are moved? You may assume for simplicity that all the statues have different heights. [Azi10]

9. **a.** Write a program for multiplying two sparse matrices, a $p \times q$ matrix A and a $q \times r$ matrix B.

   b. Write a program for multiplying two sparse polynomials $p(x)$ and $q(x)$ of degrees m and n, respectively.

10. Is it a good idea to write a program that plays the classic game of tic-tac-toe with the human user by storing all possible positions on the game's 3×3 board along with the best move for each of them?

7.2 Input Enhancement in String Matching

In this section, we see how the technique of input enhancement can be applied to the problem of string matching. Recall that the problem of string matching

requires finding an occurrence of a given string of m characters called the ***pattern*** in a longer string of n characters called the ***text***. We discussed the brute-force algorithm for this problem in Section 3.2: it simply matches corresponding pairs of characters in the pattern and the text left to right and, if a mismatch occurs, shifts the pattern one position to the right for the next trial. Since the maximum number of such trials is $n - m + 1$ and, in the worst case, m comparisons need to be made on each of them, the worst-case efficiency of the brute-force algorithm is in the $O(nm)$ class. On average, however, we should expect just a few comparisons before a pattern's shift, and for random natural-language texts, the average-case efficiency indeed turns out to be in $O(n + m)$.

Several faster algorithms have been discovered. Most of them exploit the input-enhancement idea: preprocess the pattern to get some information about it, store this information in a table, and then use this information during an actual search for the pattern in a given text. This is exactly the idea behind the two best-known algorithms of this type: the Knuth-Morris-Pratt algorithm [Knu77] and the Boyer-Moore algorithm [Boy77].

The principal difference between these two algorithms lies in the way they compare characters of a pattern with their counterparts in a text: the Knuth-Morris-Pratt algorithm does it left to right, whereas the Boyer-Moore algorithm does it right to left. Since the latter idea leads to simpler algorithms, it is the only one that we will pursue here. (Note that the Boyer-Moore algorithm starts by aligning the pattern against the beginning characters of the text; if the first trial fails, it shifts the pattern to the right. It is comparisons within a trial that the algorithm does right to left, starting with the last character in the pattern.)

Although the underlying idea of the Boyer-Moore algorithm is simple, its actual implementation in a working method is less so. Therefore, we start our discussion with a simplified version of the Boyer-Moore algorithm suggested by R. Horspool [Hor80]. In addition to being simpler, Horspool's algorithm is not necessarily less efficient than the Boyer-Moore algorithm on random strings.

Horspool's Algorithm

Consider, as an example, searching for the pattern BARBER in some text:

$$\begin{array}{ccccccc} s_0 & \ldots & & & c & \ldots & s_{n-1} \\ & & \mathtt{B\ A\ R\ B\ E\ R} & & & & \end{array}$$

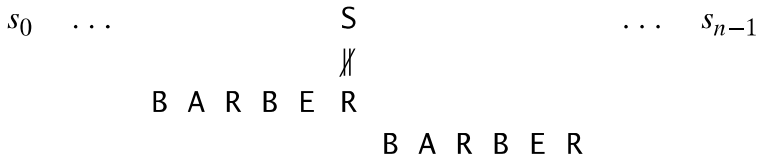
Starting with the last R of the pattern and moving right to left, we compare the corresponding pairs of characters in the pattern and the text. If all the pattern's characters match successfully, a matching substring is found. Then the search can be either stopped altogether or continued if another occurrence of the same pattern is desired.

If a mismatch occurs, we need to shift the pattern to the right. Clearly, we would like to make as large a shift as possible without risking the possibility of missing a matching substring in the text. Horspool's algorithm determines the size

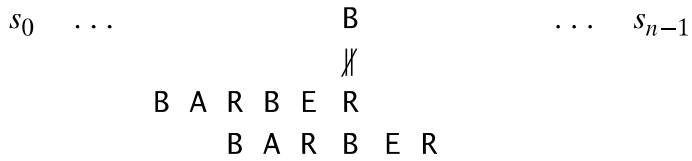
of such a shift by looking at the character c of the text that is aligned against the last character of the pattern. This is the case even if character c itself matches its counterpart in the pattern.

In general, the following four possibilities can occur.

Case 1 If there are no c's in the pattern—e.g., c is letter S in our example—we can safely shift the pattern by its entire length (if we shift less, some character of the pattern would be aligned against the text's character c that is known not to be in the pattern):



Case 2 If there are occurrences of character c in the pattern but it is not the last one there—e.g., c is letter B in our example—the shift should align the rightmost occurrence of c in the pattern with the c in the text:



Case 3 If c happens to be the last character in the pattern but there are no c's among its other $m - 1$ characters—e.g., c is letter R in our example—the situation is similar to that of Case 1 and the pattern should be shifted by the entire pattern's length m:

Case 4 Finally, if c happens to be the last character in the pattern and there are other c's among its first $m - 1$ characters—e.g., c is letter R in our example—the situation is similar to that of Case 2 and the rightmost occurrence of c among the first $m - 1$ characters in the pattern should be aligned with the text's c:

These examples clearly demonstrate that right-to-left character comparisons can lead to farther shifts of the pattern than the shifts by only one position

always made by the brute-force algorithm. However, if such an algorithm had to check all the characters of the pattern on every trial, it would lose much of this superiority. Fortunately, the idea of input enhancement makes repetitive comparisons unnecessary. We can precompute shift sizes and store them in a table. The table will be indexed by all possible characters that can be encountered in a text, including, for natural language texts, the space, punctuation symbols, and other special characters. (Note that no other information about the text in which eventual searching will be done is required.) The table's entries will indicate the shift sizes computed by the formula

$$t(c) = \begin{cases} \text{the pattern's length } m, \\ \text{if } c \text{ is not among the first } m-1 \text{ characters of the pattern;} \\ \\ \text{the distance from the rightmost } c \text{ among the first } m-1 \text{ characters} \\ \text{of the pattern to its last character, otherwise.} \end{cases} \quad \textbf{(7.1)}$$

For example, for the pattern BARBER, all the table's entries will be equal to 6, except for the entries for E, B, R, and A, which will be 1, 2, 3, and 4, respectively.

Here is a simple algorithm for computing the shift table entries. Initialize all the entries to the pattern's length m and scan the pattern left to right repeating the following step $m-1$ times: for the jth character of the pattern ($0 \le j \le m-2$), overwrite its entry in the table with $m-1-j$, which is the character's distance to the last character of the pattern. Note that since the algorithm scans the pattern from left to right, the last overwrite will happen for the character's rightmost occurrence—exactly as we would like it to be.

ALGORITHM *ShiftTable*$(P[0..m-1])$

```
//Fills the shift table used by Horspool's and Boyer-Moore algorithms
//Input: Pattern P[0..m − 1] and an alphabet of possible characters
//Output: Table[0..size − 1] indexed by the alphabet's characters and
//        filled with shift sizes computed by formula (7.1)
for i ← 0 to size − 1 do Table[i] ← m
for j ← 0 to m − 2 do Table[P[j]] ← m − 1 − j
return Table
```

Now, we can summarize the algorithm as follows:

Horspool's algorithm

Step 1 For a given pattern of length m and the alphabet used in both the pattern and text, construct the shift table as described above.

Step 2 Align the pattern against the beginning of the text.

Step 3 Repeat the following until either a matching substring is found or the pattern reaches beyond the last character of the text. Starting with the last character in the pattern, compare the corresponding characters in the pattern and text until either all m characters are matched (then

stop) or a mismatching pair is encountered. In the latter case, retrieve the entry $t(c)$ from the c's column of the shift table where c is the text's character currently aligned against the last character of the pattern, and shift the pattern by $t(c)$ characters to the right along the text.

Here is pseudocode of Horspool's algorithm.

ALGORITHM *HorspoolMatching*$(P[0..m-1], T[0..n-1])$
```
//Implements Horspool's algorithm for string matching
//Input: Pattern P[0..m − 1] and text T[0..n − 1]
//Output: The index of the left end of the first matching substring
//        or −1 if there are no matches
ShiftTable(P[0..m − 1])      //generate Table of shifts
i ← m − 1                    //position of the pattern's right end
while i ≤ n − 1 do
    k ← 0                    //number of matched characters
    while k ≤ m − 1 and P[m − 1 − k] = T[i − k] do
        k ← k + 1
    if k = m
        return i − m + 1
    else i ← i + Table[T[i]]
return −1
```

EXAMPLE As an example of a complete application of Horspool's algorithm, consider searching for the pattern BARBER in a text that comprises English letters and spaces (denoted by underscores). The shift table, as we mentioned, is filled as follows:

character c	A	B	C	D	E	F	. . .	R	. . .	Z	_
shift $t(c)$	4	2	6	6	1	6	6	3	6	6	6

The actual search in a particular text proceeds as follows:

```
J I M _ S A W _ M E _ I N _ A _ B A R B E R S H O P
B A R B E R             B A R B E R
        B A R B E R             B A R B E R
          B A R B E R                   B A R B E R
```

■

A simple example can demonstrate that the worst-case efficiency of Horspool's algorithm is in $O(nm)$ (Problem 4 in this section's exercises). But for random texts, it is in $\Theta(n)$, and, although in the same efficiency class, Horspool's algorithm is obviously faster on average than the brute-force algorithm. In fact, as mentioned, it is often at least as efficient as its more sophisticated predecessor discovered by R. Boyer and J. Moore.

Boyer-Moore Algorithm

Now we outline the Boyer-Moore algorithm itself. If the first comparison of the rightmost character in the pattern with the corresponding character c in the text fails, the algorithm does exactly the same thing as Horspool's algorithm. Namely, it shifts the pattern to the right by the number of characters retrieved from the table precomputed as explained earlier.

The two algorithms act differently, however, after some positive number k $(0 < k < m)$ of the pattern's characters are matched successfully before a mismatch is encountered:

s_0	$\ldots$	c	s_{i-k+1}	$\ldots$	s_i	$\ldots$	s_{n-1}	text
		$\not\|$	$\|$		$\|$			
	p_0 $\ldots$	p_{m-k-1}	p_{m-k}	$\ldots$	p_{m-1}			pattern

In this situation, the Boyer-Moore algorithm determines the shift size by considering two quantities. The first one is guided by the text's character c that caused a mismatch with its counterpart in the pattern. Accordingly, it is called the ***bad-symbol shift***. The reasoning behind this shift is the reasoning we used in Horspool's algorithm. If c is not in the pattern, we shift the pattern to just pass this c in the text. Conveniently, the size of this shift can be computed by the formula $t_1(c) - k$ where $t_1(c)$ is the entry in the precomputed table used by Horspool's algorithm (see above) and k is the number of matched characters:

s_0	$\ldots$	c	s_{i-k+1}	$\ldots$	s_i	$\ldots$	s_{n-1}	text
		$\not\|$	$\|$		$\|$			
	p_0 $\ldots$	p_{m-k-1}	p_{m-k}	$\ldots$	p_{m-1}			pattern
			p_0	$\ldots$		p_{m-1}		

For example, if we search for the pattern BARBER in some text and match the last two characters before failing on letter S in the text, we can shift the pattern by $t_1(\mathsf{S}) - 2 = 6 - 2 = 4$ positions:

```
s0 ...    S E R        ... sn-1
          ✗ | |
      B A R B E R
            B A R B E R
```

The same formula can also be used when the mismatching character c of the text occurs in the pattern, provided $t_1(c) - k > 0$. For example, if we search for the pattern BARBER in some text and match the last two characters before failing on letter A, we can shift the pattern by $t_1(\mathsf{A}) - 2 = 4 - 2 = 2$ positions:

```
s0 ...    A E R        ... sn-1
          ✗ | |
    B A R B E R
      B A R B E R
```

If $t_1(c) - k \leq 0$, we obviously do not want to shift the pattern by 0 or a negative number of positions. Rather, we can fall back on the brute-force thinking and simply shift the pattern by one position to the right.

To summarize, the bad-symbol shift d_1 is computed by the Boyer-Moore algorithm either as $t_1(c) - k$ if this quantity is positive and as 1 if it is negative or zero. This can be expressed by the following compact formula:

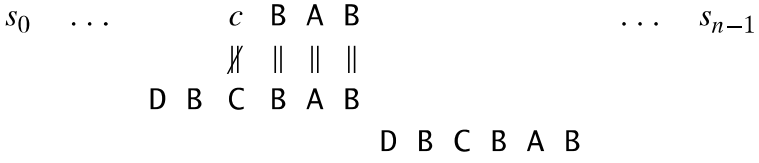
$$d_1 = \max\{t_1(c) - k, 1\}. \quad \textbf{(7.2)}$$

The second type of shift is guided by a successful match of the last $k > 0$ characters of the pattern. We refer to the ending portion of the pattern as its suffix of size k and denote it $suff(k)$. Accordingly, we call this type of shift the ***good-suffix shift***. We now apply the reasoning that guided us in filling the bad-symbol shift table, which was based on a single alphabet character c, to the pattern's suffixes of sizes $1, \ldots, m-1$ to fill in the good-suffix shift table.

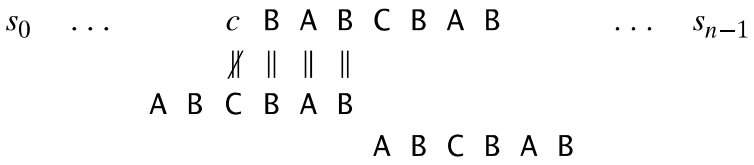
Let us first consider the case when there is another occurrence of $suff(k)$ in the pattern or, to be more accurate, there is another occurrence of $suff(k)$ not preceded by the same character as in its rightmost occurrence. (It would be useless to shift the pattern to match another occurrence of $suff(k)$ preceded by the same character because this would simply repeat a failed trial.) In this case, we can shift the pattern by the distance d_2 between such a second rightmost occurrence (not preceded by the same character as in the rightmost occurrence) of $suff(k)$ and its rightmost occurrence. For example, for the pattern ABCBAB, these distances for $k = 1$ and 2 will be 2 and 4, respectively:

k	pattern	d_2
1	ABC$\overline{\text{B}}$A$\underline{\text{B}}$	2
2	$\overline{\text{AB}}$CB$\underline{\text{AB}}$	4

What is to be done if there is no other occurrence of $suff(k)$ not preceded by the same character as in its rightmost occurrence? In most cases, we can shift the pattern by its entire length m. For example, for the pattern DBCBAB and $k = 3$, we can shift the pattern by its entire length of 6 characters:



Unfortunately, shifting the pattern by its entire length when there is no other occurrence of $suff(k)$ not preceded by the same character as in its rightmost occurrence is not always correct. For example, for the pattern ABCBAB and $k = 3$, shifting by 6 could miss a matching substring that starts with the text's AB aligned with the last two characters of the pattern:



Note that the shift by 6 is correct for the pattern DBCBAB but not for ABCBAB, because the latter pattern has the same substring AB as its prefix (beginning part of the pattern) and as its suffix (ending part of the pattern). To avoid such an erroneous shift based on a suffix of size k, for which there is no other occurrence in the pattern not preceded by the same character as in its rightmost occurrence, we need to find the longest prefix of size $l < k$ that matches the suffix of the same size l. If such a prefix exists, the shift size d_2 is computed as the distance between this prefix and the corresponding suffix; otherwise, d_2 is set to the pattern's length m. As an example, here is the complete list of the d_2 values—the good-suffix table of the Boyer-Moore algorithm—for the pattern ABCBAB:

k	pattern	d_2
1	ABCBAB	2
2	ABCBAB	4
3	ABCBAB	4
4	ABCBAB	4
5	ABCBAB	4

Now we are prepared to summarize the Boyer-Moore algorithm in its entirety.

The Boyer-Moore algorithm

Step 1 For a given pattern and the alphabet used in both the pattern and the text, construct the bad-symbol shift table as described earlier.

Step 2 Using the pattern, construct the good-suffix shift table as described earlier.

Step 3 Align the pattern against the beginning of the text.

Step 4 Repeat the following step until either a matching substring is found or the pattern reaches beyond the last character of the text. Starting with the last character in the pattern, compare the corresponding characters in the pattern and the text until either all m character pairs are matched (then stop) or a mismatching pair is encountered after $k \geq 0$ character pairs are matched successfully. In the latter case, retrieve the entry $t_1(c)$ from the c's column of the bad-symbol table where c is the text's mismatched character. If $k > 0$, also retrieve the corresponding d_2 entry from the good-suffix table. Shift the pattern to the right by the

number of positions computed by the formula

$$d = \begin{cases} d_1 & \text{if } k = 0, \\ \max\{d_1, d_2\} & \text{if } k > 0, \end{cases} \qquad \textbf{(7.3)}$$

where $d_1 = \max\{t_1(c) - k, 1\}$.

Shifting by the maximum of the two available shifts when $k > 0$ is quite logical. The two shifts are based on the observations—the first one about a text's mismatched character, and the second one about a matched group of the pattern's rightmost characters—that imply that shifting by less than d_1 and d_2 characters, respectively, cannot lead to aligning the pattern with a matching substring in the text. Since we are interested in shifting the pattern as far as possible without missing a possible matching substring, we take the maximum of these two numbers.

EXAMPLE As a complete example, let us consider searching for the pattern BAOBAB in a text made of English letters and spaces. The bad-symbol table looks as follows:

c	A	B	C	D	. . .	O	. . .	Z	_
$t_1(c)$	1	2	6	6	6	3	6	6	6

The good-suffix table is filled as follows:

k	pattern	d_2
1	BAO$\overline{\text{B}}$AB	2
2	$\overline{\text{B}}$AOBAB	5
3	$\overline{\text{B}}$AO$\overline{\text{BAB}}$	5
4	$\overline{\text{B}}$A$\overline{\text{OBAB}}$	5
5	$\overline{\text{B}}$$\overline{\text{AOBAB}}$	5

The actual search for this pattern in the text given in Figure 7.3 proceeds as follows. After the last B of the pattern fails to match its counterpart K in the text, the algorithm retrieves $t_1(\text{K}) = 6$ from the bad-symbol table and shifts the pattern by $d_1 = \max\{t_1(\text{K}) - 0, 1\} = 6$ positions to the right. The new try successfully matches two pairs of characters. After the failure of the third comparison on the space character in the text, the algorithm retrieves $t_1(\_) = 6$ from the bad-symbol table and $d_2 = 5$ from the good-suffix table to shift the pattern by $\max\{d_1, d_2\} = \max\{6 - 2, 5\} = 5$. Note that on this iteration it is the good-suffix rule that leads to a farther shift of the pattern.

The next try successfully matches just one pair of B's. After the failure of the next comparison on the space character in the text, the algorithm retrieves $t_1(\_) = 6$ from the bad-symbol table and $d_2 = 2$ from the good-suffix table to shift

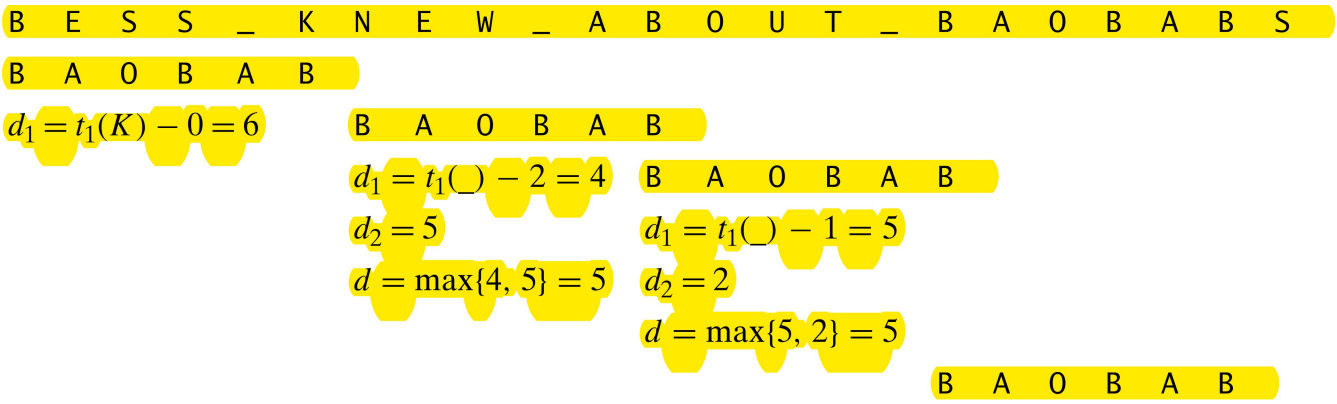


FIGURE 7.3 Example of string matching with the Boyer-Moore algorithm.

the pattern by $\max\{d_1, d_2\} = \max\{6 - 1, 2\} = 5$. Note that on this iteration it is the bad-symbol rule that leads to a farther shift of the pattern. The next try finds a matching substring in the text after successfully matching all six characters of the pattern with their counterparts in the text. ■

When searching for the first occurrence of the pattern, the worst-case efficiency of the Boyer-Moore algorithm is known to be linear. Though this algorithm runs very fast, especially on large alphabets (relative to the length of the pattern), many people prefer its simplified versions, such as Horspool's algorithm, when dealing with natural-language–like strings.

Exercises 7.2

1. Apply Horspool's algorithm to search for the pattern BAOBAB in the text

BESS_KNEW_ABOUT_BAOBABS

2. Consider the problem of searching for genes in DNA sequences using Horspool's algorithm. A DNA sequence is represented by a text on the alphabet {A, C, G, T}, and the gene or gene segment is the pattern.

a. Construct the shift table for the following gene segment of your chromosome 10:

TCCTATTCTT

b. Apply Horspool's algorithm to locate the above pattern in the following DNA sequence:

TTATAGATCTCGTATTCTTTTATAGATCTCCTATTCTT

3. How many character comparisons will be made by Horspool's algorithm in searching for each of the following patterns in the binary text of 1000 zeros?

a. 00001 **b.** 10000 **c.** 01010

4. For searching in a text of length n for a pattern of length m $(n \geq m)$ with Horspool's algorithm, give an example of

a. worst-case input. **b.** best-case input.

5. Is it possible for Horspool's algorithm to make more character comparisons than the brute-force algorithm would make in searching for the same pattern in the same text?

6. If Horspool's algorithm discovers a matching substring, how large a shift should it make to search for a next possible match?

7. How many character comparisons will the Boyer-Moore algorithm make in searching for each of the following patterns in the binary text of 1000 zeros?

a. 00001 **b.** 10000 **c.** 01010

8. a. Would the Boyer-Moore algorithm work correctly with just the bad-symbol table to guide pattern shifts?

b. Would the Boyer-Moore algorithm work correctly with just the good-suffix table to guide pattern shifts?

9. a. If the last characters of a pattern and its counterpart in the text do match, does Horspool's algorithm have to check other characters right to left, or can it check them left to right too?

b. Answer the same question for the Boyer-Moore algorithm.

10. Implement Horspool's algorithm, the Boyer-Moore algorithm, and the brute-force algorithm of Section 3.2 in the language of your choice and run an experiment to compare their efficiencies for matching

a. random binary patterns in random binary texts.

b. random natural-language patterns in natural-language texts.

11. You are given two strings S and T, each n characters long. You have to establish whether one of them is a right cyclic shift of the other. For example, PLEA is a right cyclic shift of LEAP, and vice versa. (Formally, T is a right cyclic shift of S if T can be obtained by concatenating the $(n - i)$-character suffix of S and the i-character prefix of S for some $1 \leq i \leq n$.)

a. Design a space-efficient algorithm for the task. Indicate the space and time efficiencies of your algorithm.

b. Design a time-efficient algorithm for the task. Indicate the time and space efficiencies of your algorithm.

Chapter 8

(Dynamic Programming)

8

Dynamic Programming

An idea, like a ghost . . . must be spoken to a little before it will explain itself.

—Charles Dickens (1812–1870)

Dynamic programming is an algorithm design technique with a rather interesting history. It was invented by a prominent U.S. mathematician, Richard Bellman, in the 1950s as a general method for optimizing multistage decision processes. Thus, the word "programming" in the name of this technique stands for "planning" and does not refer to computer programming. After proving its worth as an important tool of applied mathematics, dynamic programming has eventually come to be considered, at least in computer science circles, as a general algorithm design technique that does not have to be limited to special types of optimization problems. It is from this point of view that we will consider this technique here.

Dynamic programming is a technique for solving problems with overlapping subproblems. Typically, these subproblems arise from a recurrence relating a given problem's solution to solutions of its smaller subproblems. Rather than solving overlapping subproblems again and again, dynamic programming suggests solving each of the smaller subproblems only once and recording the results in a table from which a solution to the original problem can then be obtained.

This technique can be illustrated by revisiting the Fibonacci numbers discussed in Section 2.5. (If you have not read that section, you will be able to follow the discussion anyway. But it is a beautiful topic, so if you feel a temptation to read it, do succumb to it.) The Fibonacci numbers are the elements of the sequence

$$0, \quad 1, \quad 1, \quad 2, \quad 3, \quad 5, \quad 8, \quad 13, \quad 21, \quad 34, \ldots,$$

which can be defined by the simple recurrence

$$F(n) = F(n-1) + F(n-2) \quad \text{for } n > 1 \tag{8.1}$$

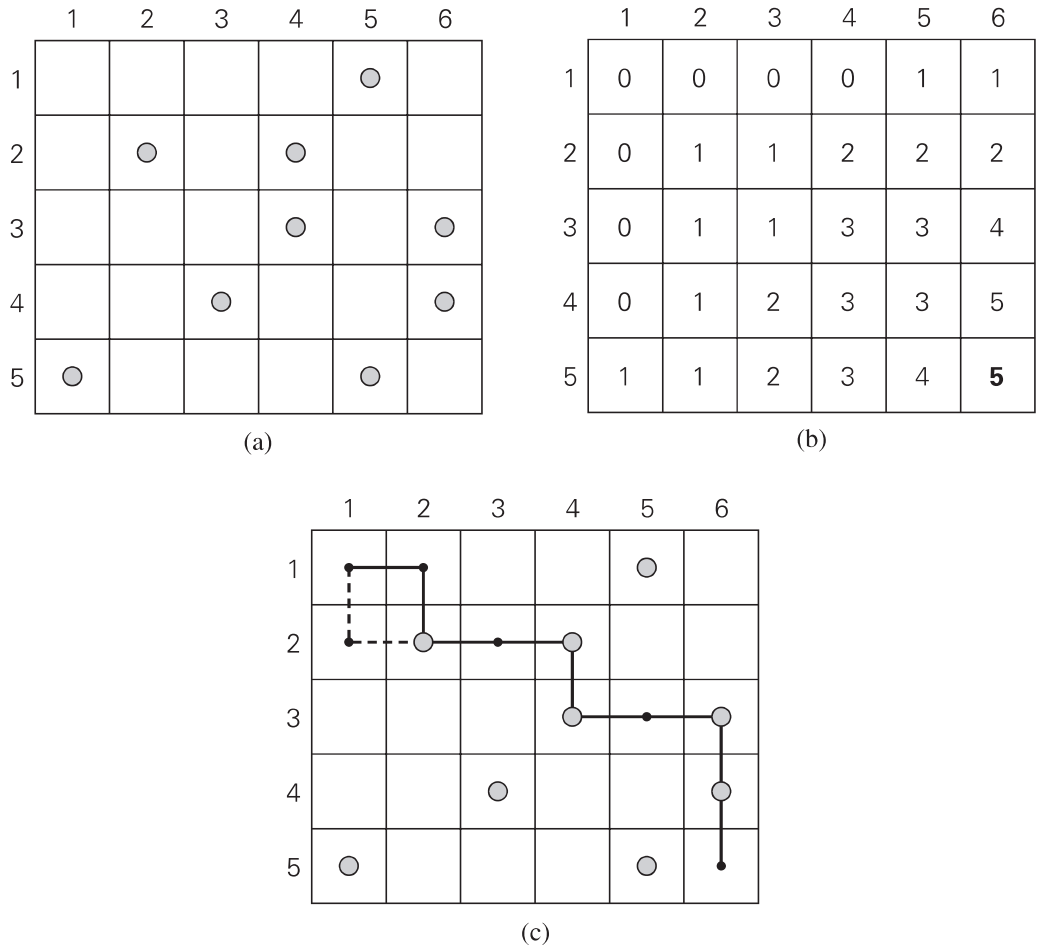


FIGURE 8.3 (a) Coins to collect. (b) Dynamic programming algorithm results. (c) Two paths to collect 5 coins, the maximum number of coins possible.

Exercises 8.1

1. What does dynamic programming have in common with divide-and-conquer? What is a principal difference between them?

2. Solve the instance 5, 1, 2, 10, 6 of the coin-row problem.

3. **a.** Show that the time efficiency of solving the coin-row problem by straightforward application of recurrence (8.3) is exponential.

b. Show that the time efficiency of solving the coin-row problem by exhaustive search is at least exponential.

4. Apply the dynamic programming algorithm to find all the solutions to the change-making problem for the denominations 1, 3, 5 and the amount $n = 9$.

a. Give an example of three matrices for which the number of multiplications in $(A_1 \cdot A_2) \cdot A_3$ and $A_1 \cdot (A_2 \cdot A_3)$ differ at least by a factor of 1000.

b. How many different ways are there to compute the product of n matrices?

c. Design a dynamic programming algorithm for finding an optimal order of multiplying n matrices.

8.4 Warshall's and Floyd's Algorithms

In this section, we look at two well-known algorithms: Warshall's algorithm for computing the transitive closure of a directed graph and Floyd's algorithm for the all-pairs shortest-paths problem. These algorithms are based on essentially the same idea: exploit a relationship between a problem and its simpler rather than smaller version. Warshall and Floyd published their algorithms without mentioning dynamic programming. Nevertheless, the algorithms certainly have a dynamic programming flavor and have come to be considered applications of this technique.

Warshall's Algorithm

Recall that the adjacency matrix $A = \{a_{ij}\}$ of a directed graph is the boolean matrix that has 1 in its ith row and jth column if and only if there is a directed edge from the ith vertex to the jth vertex. We may also be interested in a matrix containing the information about the existence of directed paths of arbitrary lengths between vertices of a given graph. Such a matrix, called the transitive closure of the digraph, would allow us to determine in constant time whether the jth vertex is reachable from the ith vertex.

Here are a few application examples. When a value in a spreadsheet cell is changed, the spreadsheet software must know all the other cells affected by the change. If the spreadsheet is modeled by a digraph whose vertices represent the spreadsheet cells and edges indicate cell dependencies, the transitive closure will provide such information. In software engineering, transitive closure can be used for investigating data flow and control flow dependencies as well as for inheritance testing of object-oriented software. In electronic engineering, it is used for redundancy identification and test generation for digital circuits.

DEFINITION The ***transitive closure*** of a directed graph with n vertices can be defined as the $n \times n$ boolean matrix $T = \{t_{ij}\}$, in which the element in the ith row and the jth column is 1 if there exists a nontrivial path (i.e., directed path of a positive length) from the ith vertex to the jth vertex; otherwise, t_{ij} is 0.

An example of a digraph, its adjacency matrix, and its transitive closure is given in Figure 8.11.

We can generate the transitive closure of a digraph with the help of depth-first search or breadth-first search. Performing either traversal starting at the ith

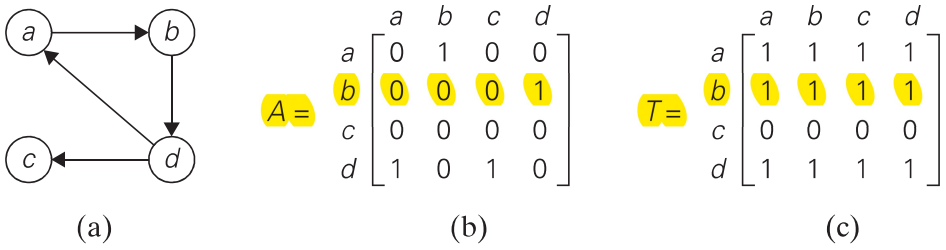


FIGURE 8.11 (a) Digraph. (b) Its adjacency matrix. (c) Its transitive closure.

vertex gives the information about the vertices reachable from it and hence the columns that contain 1's in the ith row of the transitive closure. Thus, doing such a traversal for every vertex as a starting point yields the transitive closure in its entirety.

Since this method traverses the same digraph several times, we should hope that a better algorithm can be found. Indeed, such an algorithm exists. It is called ***Warshall's algorithm*** after Stephen Warshall, who discovered it [War62]. It is convenient to assume that the digraph's vertices and hence the rows and columns of the adjacency matrix are numbered from 1 to n. Warshall's algorithm constructs the transitive closure through a series of $n \times n$ boolean matrices:

$$R^{(0)}, \ldots, R^{(k-1)}, R^{(k)}, \ldots R^{(n)}. \tag{8.9}$$

Each of these matrices provides certain information about directed paths in the digraph. Specifically, the element $r_{ij}^{(k)}$ in the ith row and jth column of matrix $R^{(k)}$ $(i, j = 1, 2, \ldots, n, k = 0, 1, \ldots, n)$ is equal to 1 if and only if there exists a directed path of a positive length from the ith vertex to the jth vertex with each intermediate vertex, if any, numbered not higher than k. Thus, the series starts with $R^{(0)}$, which does not allow any intermediate vertices in its paths; hence, $R^{(0)}$ is nothing other than the adjacency matrix of the digraph. (Recall that the adjacency matrix contains the information about one-edge paths, i.e., paths with no intermediate vertices.) $R^{(1)}$ contains the information about paths that can use the first vertex as intermediate; thus, with more freedom, so to speak, it may contain more 1's than $R^{(0)}$. In general, each subsequent matrix in series (8.9) has one more vertex to use as intermediate for its paths than its predecessor and hence may, but does not have to, contain more 1's. The last matrix in the series, $R^{(n)}$, reflects paths that can use all n vertices of the digraph as intermediate and hence is nothing other than the digraph's transitive closure.

The central point of the algorithm is that we can compute all the elements of each matrix $R^{(k)}$ from its immediate predecessor $R^{(k-1)}$ in series (8.9). Let $r_{ij}^{(k)}$, the element in the ith row and jth column of matrix $R^{(k)}$, be equal to 1. This means that there exists a path from the ith vertex v_i to the jth vertex v_j with each intermediate vertex numbered not higher than k:

$$v_i, \text{ a list of intermediate vertices each numbered not higher than } k, v_j. \tag{8.10}$$

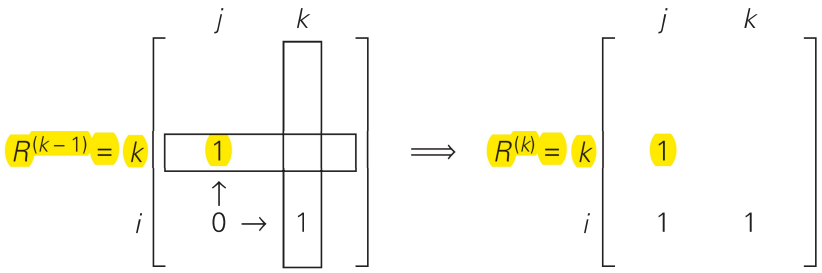


FIGURE 8.12 Rule for changing zeros in Warshall's algorithm.

Two situations regarding this path are possible. In the first, the list of its intermediate vertices does not contain the kth vertex. Then this path from v_i to v_j has intermediate vertices numbered not higher than $k-1$, and therefore $r_{ij}^{(k-1)}$ is equal to 1 as well. The second possibility is that path (8.10) does contain the kth vertex v_k among the intermediate vertices. Without loss of generality, we may assume that v_k occurs only once in that list. (If it is not the case, we can create a new path from v_i to v_j with this property by simply eliminating all the vertices between the first and last occurrences of v_k in it.) With this caveat, path (8.10) can be rewritten as follows:

$$v_i, \text{ vertices numbered} \leq k-1, v_k, \text{ vertices numbered} \leq k-1, v_j.$$

The first part of this representation means that there exists a path from v_i to v_k with each intermediate vertex numbered not higher than $k-1$ (hence, $r_{ik}^{(k-1)} = 1$), and the second part means that there exists a path from v_k to v_j with each intermediate vertex numbered not higher than $k-1$ (hence, $r_{kj}^{(k-1)} = 1$).

What we have just proved is that if $r_{ij}^{(k)} = 1$, then either $r_{ij}^{(k-1)} = 1$ or both $r_{ik}^{(k-1)} = 1$ and $r_{kj}^{(k-1)} = 1$. It is easy to see that the converse of this assertion is also true. Thus, we have the following formula for generating the elements of matrix $R^{(k)}$ from the elements of matrix $R^{(k-1)}$:

$$r_{ij}^{(k)} = r_{ij}^{(k-1)} \text{ or } \left(r_{ik}^{(k-1)} \text{ and } r_{kj}^{(k-1)}\right). \tag{8.11}$$

Formula (8.11) is at the heart of Warshall's algorithm. This formula implies the following rule for generating elements of matrix $R^{(k)}$ from elements of matrix $R^{(k-1)}$, which is particularly convenient for applying Warshall's algorithm by hand:

- If an element r_{ij} is 1 in $R^{(k-1)}$, it remains 1 in $R^{(k)}$.
- If an element r_{ij} is 0 in $R^{(k-1)}$, it has to be changed to 1 in $R^{(k)}$ if and only if the element in its row i and column k and the element in its column j and row k are both 1's in $R^{(k-1)}$. This rule is illustrated in Figure 8.12.

As an example, the application of Warshall's algorithm to the digraph in Figure 8.11 is shown in Figure 8.13.

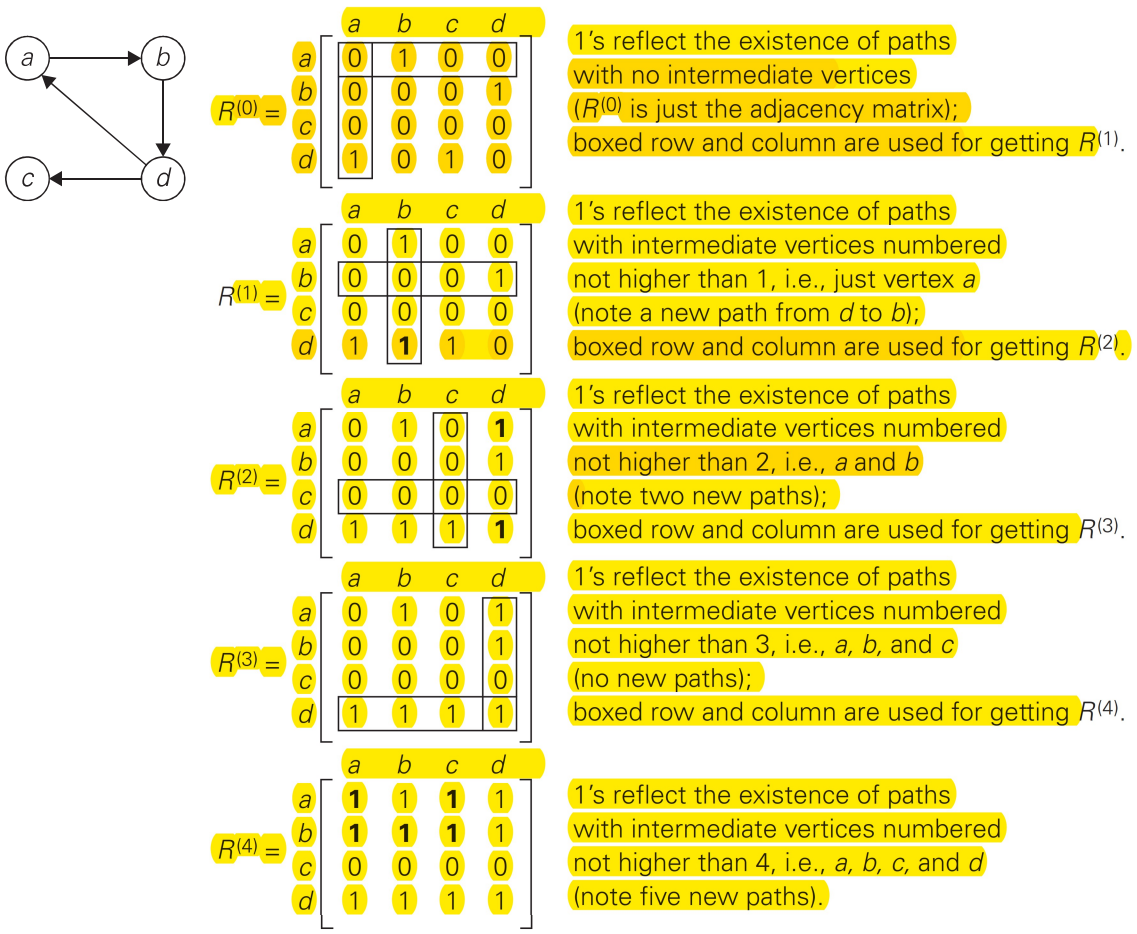


FIGURE 8.13 Application of Warshall's algorithm to the digraph shown. New 1's are in bold.

Here is pseudocode of Warshall's algorithm.

ALGORITHM *Warshall*$(A[1..n, 1..n])$
//Implements Warshall's algorithm for computing the transitive closure
//Input: The adjacency matrix A of a digraph with n vertices
//Output: The transitive closure of the digraph
$R^{(0)} \leftarrow A$
for $k \leftarrow 1$ **to** n **do**
  for $i \leftarrow 1$ **to** n **do**
    for $j \leftarrow 1$ **to** n **do**
      $R^{(k)}[i, j] \leftarrow R^{(k-1)}[i, j]$ **or** $(R^{(k-1)}[i, k]$ **and** $R^{(k-1)}[k, j])$
return $R^{(n)}$

Several observations need to be made about Warshall's algorithm. First, it is remarkably succinct, is it not? Still, its time efficiency is only $\Theta(n^3)$. In fact, for sparse graphs represented by their adjacency lists, the traversal-based algorithm

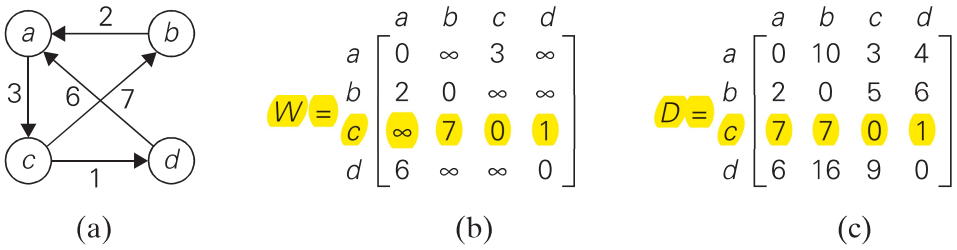


FIGURE 8.14 (a) Digraph. (b) Its weight matrix. (c) Its distance matrix.

mentioned at the beginning of this section has a better asymptotic efficiency than Warshall's algorithm (why?). We can speed up the above implementation of Warshall's algorithm for some inputs by restructuring its innermost loop (see Problem 4 in this section's exercises). Another way to make the algorithm run faster is to treat matrix rows as bit strings and employ the bitwise *or* operation available in most modern computer languages.

As to the space efficiency of Warshall's algorithm, the situation is similar to that of computing a Fibonacci number and some other dynamic programming algorithms. Although we used separate matrices for recording intermediate results of the algorithm, this is, in fact, unnecessary. Problem 3 in this section's exercises asks you to find a way of avoiding this wasteful use of the computer memory. Finally, we shall see below how the underlying idea of Warshall's algorithm can be applied to the more general problem of finding lengths of shortest paths in weighted graphs.

Floyd's Algorithm for the All-Pairs Shortest-Paths Problem

Given a weighted connected graph (undirected or directed), the ***all-pairs shortest-paths problem*** asks to find the distances—i.e., the lengths of the shortest paths—from each vertex to all other vertices. This is one of several variations of the problem involving shortest paths in graphs. Because of its important applications to communications, transportation networks, and operations research, it has been thoroughly studied over the years. Among recent applications of the all-pairs shortest-path problem is precomputing distances for motion planning in computer games.

It is convenient to record the lengths of shortest paths in an $n \times n$ matrix D called the ***distance matrix***: the element d_{ij} in the ith row and the jth column of this matrix indicates the length of the shortest path from the ith vertex to the jth vertex. For an example, see Figure 8.14.

We can generate the distance matrix with an algorithm that is very similar to Warshall's algorithm. It is called ***Floyd's algorithm*** after its co-inventor Robert W. Floyd.[1] It is applicable to both undirected and directed weighted graphs provided

1. Floyd explicitly referenced Warshall's paper in presenting his algorithm [Flo62]. Three years earlier, Bernard Roy published essentially the same algorithm in the proceedings of the French Academy of Sciences [Roy59].

that they do not contain a cycle of a negative length. (The distance between any two vertices in such a cycle can be made arbitrarily small by repeating the cycle enough times.) The algorithm can be enhanced to find not only the lengths of the shortest paths for all vertex pairs but also the shortest paths themselves (Problem 10 in this section's exercises).

Floyd's algorithm computes the distance matrix of a weighted graph with n vertices through a series of $n \times n$ matrices:

$$D^{(0)}, \ldots, D^{(k-1)}, D^{(k)}, \ldots, D^{(n)}. \quad \textbf{(8.12)}$$

Each of these matrices contains the lengths of shortest paths with certain constraints on the paths considered for the matrix in question. Specifically, the element $d_{ij}^{(k)}$ in the ith row and the jth column of matrix $D^{(k)}$ ($i, j = 1, 2, \ldots, n$, $k = 0, 1, \ldots, n$) is equal to the length of the shortest path among all paths from the ith vertex to the jth vertex with each intermediate vertex, if any, numbered not higher than k. In particular, the series starts with $D^{(0)}$, which does not allow any intermediate vertices in its paths; hence, $D^{(0)}$ is simply the weight matrix of the graph. The last matrix in the series, $D^{(n)}$, contains the lengths of the shortest paths among all paths that can use all n vertices as intermediate and hence is nothing other than the distance matrix being sought.

As in Warshall's algorithm, we can compute all the elements of each matrix $D^{(k)}$ from its immediate predecessor $D^{(k-1)}$ in series (8.12). Let $d_{ij}^{(k)}$ be the element in the ith row and the jth column of matrix $D^{(k)}$. This means that $d_{ij}^{(k)}$ is equal to the length of the shortest path among all paths from the ith vertex v_i to the jth vertex v_j with their intermediate vertices numbered not higher than k:

$$v_i, \text{ a list of intermediate vertices each numbered not higher than } k, v_j. \quad \textbf{(8.13)}$$

We can partition all such paths into two disjoint subsets: those that do not use the kth vertex v_k as intermediate and those that do. Since the paths of the first subset have their intermediate vertices numbered not higher than $k-1$, the shortest of them is, by definition of our matrices, of length $d_{ij}^{(k-1)}$.

What is the length of the shortest path in the second subset? If the graph does not contain a cycle of a negative length, we can limit our attention only to the paths in the second subset that use vertex v_k as their intermediate vertex exactly once (because visiting v_k more than once can only increase the path's length). All such paths have the following form:

$$v_i, \text{ vertices numbered} \le k-1, v_k, \text{ vertices numbered} \le k-1, v_j.$$

In other words, each of the paths is made up of a path from v_i to v_k with each intermediate vertex numbered not higher than $k-1$ and a path from v_k to v_j with each intermediate vertex numbered not higher than $k-1$. The situation is depicted symbolically in Figure 8.15.

Since the length of the shortest path from v_i to v_k among the paths that use intermediate vertices numbered not higher than $k-1$ is equal to $d_{ik}^{(k-1)}$ and the length of the shortest path from v_k to v_j among the paths that use intermediate

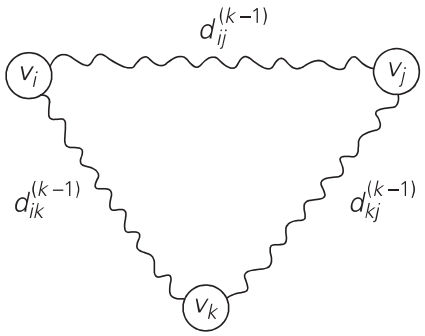


FIGURE 8.15 Underlying idea of Floyd's algorithm.

vertices numbered not higher than $k-1$ is equal to $d_{kj}^{(k-1)}$, the length of the shortest path among the paths that use the kth vertex is equal to $d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$. Taking into account the lengths of the shortest paths in both subsets leads to the following recurrence:

$$d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)},\ d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\} \quad \text{for } k \geq 1, \quad d_{ij}^{(0)} = w_{ij}. \tag{8.14}$$

To put it another way, the element in row i and column j of the current distance matrix $D^{(k-1)}$ is replaced by the sum of the elements in the same row i and the column k and in the same column j and the row k if and only if the latter sum is smaller than its current value.

The application of Floyd's algorithm to the graph in Figure 8.14 is illustrated in Figure 8.16.

Here is pseudocode of Floyd's algorithm. It takes advantage of the fact that the next matrix in sequence (8.12) can be written over its predecessor.

ALGORITHM *Floyd*$(W[1..n, 1..n])$

```
//Implements Floyd's algorithm for the all-pairs shortest-paths problem
//Input: The weight matrix W of a graph with no negative-length cycle
//Output: The distance matrix of the shortest paths' lengths
D ← W //is not necessary if W can be overwritten
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            D[i, j] ← min{D[i, j], D[i, k] + D[k, j]}
return D
```

Obviously, the time efficiency of Floyd's algorithm is cubic—as is the time efficiency of Warshall's algorithm. In the next chapter, we examine Dijkstra's algorithm—another method for finding shortest paths.

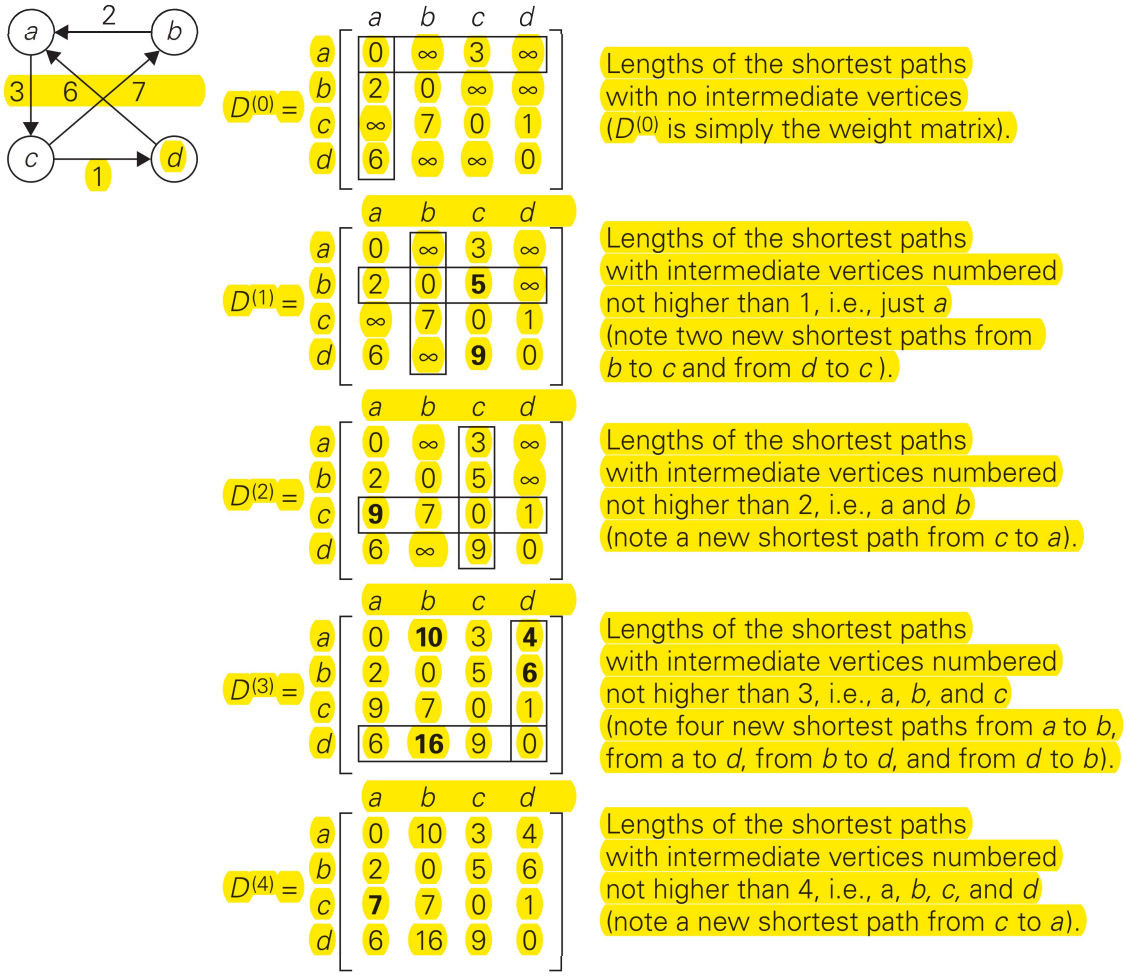


FIGURE 8.16 Application of Floyd's algorithm to the digraph shown. Updated elements are shown in bold.

Exercises 8.4

1. Apply Warshall's algorithm to find the transitive closure of the digraph defined by the following adjacency matrix:

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

2. a. Prove that the time efficiency of Warshall's algorithm is cubic.

b. Explain why the time efficiency class of Warshall's algorithm is inferior to that of the traversal-based algorithm for sparse graphs represented by their adjacency lists.

3. Explain how to implement Warshall's algorithm without using extra memory for storing elements of the algorithm's intermediate matrices.
4. Explain how to restructure the innermost loop of the algorithm *Warshall* to make it run faster at least on some inputs.
5. Rewrite pseudocode of Warshall's algorithm assuming that the matrix rows are represented by bit strings on which the bitwise *or* operation can be performed.
6. **a.** Explain how Warshall's algorithm can be used to determine whether a given digraph is a dag (directed acyclic graph). Is it a good algorithm for this problem?
   b. Is it a good idea to apply Warshall's algorithm to find the transitive closure of an undirected graph?
7. Solve the all-pairs shortest-path problem for the digraph with the following weight matrix:

$$\begin{bmatrix} 0 & 2 & \infty & 1 & 8 \\ 6 & 0 & 3 & 2 & \infty \\ \infty & \infty & 0 & 4 & \infty \\ \infty & \infty & 2 & 0 & 3 \\ 3 & \infty & \infty & \infty & 0 \end{bmatrix}$$

8. Prove that the next matrix in sequence (8.12) of Floyd's algorithm can be written over its predecessor.
9. Give an example of a graph or a digraph with negative weights for which Floyd's algorithm does not yield the correct result.
10. Enhance Floyd's algorithm so that shortest paths themselves, not just their lengths, can be found.
11. *Jack Straws* In the game of Jack Straws, a number of plastic or wooden "straws" are dumped on the table and players try to remove them one by one without disturbing the other straws. Here, we are only concerned with whether various pairs of straws are connected by a path of touching straws. Given a list of the endpoints for $n > 1$ straws (as if they were dumped on a large piece of graph paper), determine all the pairs of straws that are connected. Note that touching is connecting, but also that two straws can be connected indirectly via other connected straws. [1994 East-Central Regionals of the ACM International Collegiate Programming Contest]

SUMMARY

- *Dynamic programming* is a technique for solving problems with overlapping subproblems. Typically, these subproblems arise from a recurrence relating a solution to a given problem with solutions to its smaller subproblems of the

Chapter 9

(Greedy Technique)

9

Greedy Technique

Greed, for lack of a better word, is good! Greed is right! Greed works!

—Michael Douglas, US actor in the role of Gordon Gecko,
in the film *Wall Street*, 1987

Let us revisit the ***change-making problem*** faced, at least subconsciously, by millions of cashiers all over the world: give change for a specific amount n with the least number of coins of the denominations $d_1 > d_2 > \cdots > d_m$ used in that locale. (Here, unlike Section 8.1, we assume that the denominations are ordered in decreasing order.) For example, the widely used coin denominations in the United States are $d_1 = 25$ (quarter), $d_2 = 10$ (dime), $d_3 = 5$ (nickel), and $d_4 = 1$ (penny). How would you give change with coins of these denominations of, say, 48 cents? If you came up with the answer 1 quarter, 2 dimes, and 3 pennies, you followed—consciously or not—a logical strategy of making a sequence of best choices among the currently available alternatives. Indeed, in the first step, you could have given one coin of any of the four denominations. "Greedy" thinking leads to giving one quarter because it reduces the remaining amount the most, namely, to 23 cents. In the second step, you had the same coins at your disposal, but you could not give a quarter, because it would have violated the problem's constraints. So your best selection in this step was one dime, reducing the remaining amount to 13 cents. Giving one more dime left you with 3 cents to be given with three pennies.

Is this solution to the instance of the change-making problem optimal? Yes, it is. In fact, one can prove that the greedy algorithm yields an optimal solution for every positive integer amount with these coin denominations. At the same time, it is easy to give an example of coin denominations that do not yield an optimal solution for some amounts—e.g., $d_1 = 25$, $d_2 = 10$, $d_3 = 1$ and $n = 30$.

The approach applied in the opening paragraph to the change-making problem is called ***greedy***. Computer scientists consider it a general design technique despite the fact that it is applicable to optimization problems only. The greedy approach suggests constructing a solution through a sequence of steps, each expanding a partially constructed solution obtained so far, until a complete solution

to the problem is reached. On each step—and this is the central point of this technique—the choice made must be:

- *feasible*, i.e., it has to satisfy the problem's constraints
- *locally optimal*, i.e., it has to be the best local choice among all feasible choices available on that step
- *irrevocable*, i.e., once made, it cannot be changed on subsequent steps of the algorithm

These requirements explain the technique's name: on each step, it suggests a "greedy" grab of the best alternative available in the hope that a sequence of locally optimal choices will yield a (globally) optimal solution to the entire problem. We refrain from a philosophical discussion of whether greed is good or bad. (If you have not seen the movie from which the chapter's epigraph is taken, its hero did not end up well.) From our algorithmic perspective, the question is whether such a greedy strategy works or not. As we shall see, there are problems for which a sequence of locally optimal choices does yield an optimal solution for every instance of the problem in question. However, there are others for which this is not the case; for such problems, a greedy algorithm can still be of value if we are interested in or have to be satisfied with an approximate solution.

In the first two sections of the chapter, we discuss two classic algorithms for the minimum spanning tree problem: Prim's algorithm and Kruskal's algorithm. What is remarkable about these algorithms is the fact that they solve the same problem by applying the greedy approach in two different ways, and both of them always yield an optimal solution. In Section 9.3, we introduce another classic algorithm—Dijkstra's algorithm for the shortest-path problem in a weighted graph. Section 9.4 is devoted to Huffman trees and their principal application, Huffman codes—an important data compression method that can be interpreted as an application of the greedy technique. Finally, a few examples of approximation algorithms based on the greedy approach are discussed in Section 12.3.

As a rule, greedy algorithms are both intuitively appealing and simple. Given an optimization problem, it is usually easy to figure out how to proceed in a greedy manner, possibly after considering a few small instances of the problem. What is usually more difficult is to prove that a greedy algorithm yields an optimal solution (when it does). One of the common ways to do this is illustrated by the proof given in Section 9.1: using mathematical induction, we show that a partially constructed solution obtained by the greedy algorithm on each iteration can be extended to an optimal solution to the problem.

The second way to prove optimality of a greedy algorithm is to show that on each step it does at least as well as any other algorithm could in advancing toward the problem's goal. Consider, as an example, the following problem: find the minimum number of moves needed for a chess knight to go from one corner of a 100×100 board to the diagonally opposite corner. (The knight's moves are L-shaped jumps: two squares horizontally or vertically followed by one square in

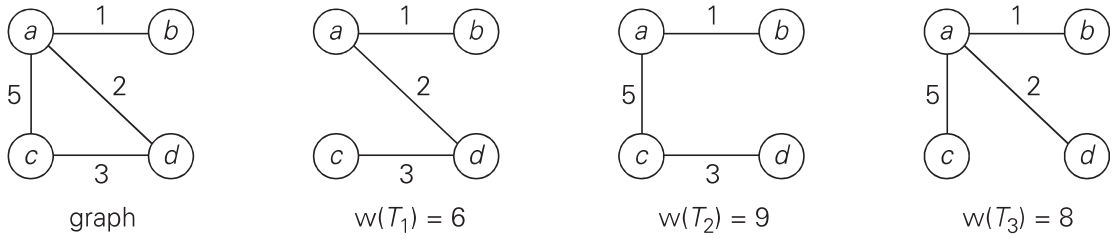


FIGURE 9.2 Graph and its spanning trees, with T_1 being the minimum spanning tree.

9.1 Prim's Algorithm

The following problem arises naturally in many practical situations: given n points, connect them in the cheapest possible way so that there will be a path between every pair of points. It has direct applications to the design of all kinds of networks—including communication, computer, transportation, and electrical—by providing the cheapest way to achieve connectivity. It identifies clusters of points in data sets. It has been used for classification purposes in archeology, biology, sociology, and other sciences. It is also helpful for constructing approximate solutions to more difficult problems such the traveling salesman problem (see Section 12.3).

We can represent the points given by vertices of a graph, possible connections by the graph's edges, and the connection costs by the edge weights. Then the question can be posed as the minimum spanning tree problem, defined formally as follows.

DEFINITION A ***spanning tree*** of an undirected connected graph is its connected acyclic subgraph (i.e., a tree) that contains all the vertices of the graph. If such a graph has weights assigned to its edges, a ***minimum spanning tree*** is its spanning tree of the smallest weight, where the ***weight*** of a tree is defined as the sum of the weights on all its edges. The ***minimum spanning tree problem*** is the problem of finding a minimum spanning tree for a given weighted connected graph.

Figure 9.2 presents a simple example illustrating these notions.

If we were to try constructing a minimum spanning tree by exhaustive search, we would face two serious obstacles. First, the number of spanning trees grows exponentially with the graph size (at least for dense graphs). Second, generating all spanning trees for a given graph is not easy; in fact, it is more difficult than finding a *minimum* spanning tree for a weighted graph by using one of several efficient algorithms available for this problem. In this section, we outline ***Prim's algorithm***, which goes back to at least 1957[1] [Pri57].

1. Robert Prim rediscovered the algorithm published 27 years earlier by the Czech mathematician Vojtěch Jarník in a Czech journal.

Prim's algorithm constructs a minimum spanning tree through a sequence of expanding subtrees. The initial subtree in such a sequence consists of a single vertex selected arbitrarily from the set V of the graph's vertices. On each iteration, the algorithm expands the current tree in the greedy manner by simply attaching to it the nearest vertex not in that tree. (By the nearest vertex, we mean a vertex not in the tree connected to a vertex in the tree by an edge of the smallest weight. Ties can be broken arbitrarily.) The algorithm stops after all the graph's vertices have been included in the tree being constructed. Since the algorithm expands a tree by exactly one vertex on each of its iterations, the total number of such iterations is $n - 1$, where n is the number of vertices in the graph. The tree generated by the algorithm is obtained as the set of edges used for the tree expansions.

Here is pseudocode of this algorithm.

ALGORITHM *Prim(G)*

```
//Prim's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph G = ⟨V, E⟩
//Output: E_T, the set of edges composing a minimum spanning tree of G
V_T ← {v_0}  //the set of tree vertices can be initialized with any vertex
E_T ← ∅
for i ← 1 to |V| − 1 do
    find a minimum-weight edge e* = (v*, u*) among all the edges (v, u)
    such that v is in V_T and u is in V − V_T
    V_T ← V_T ∪ {u*}
    E_T ← E_T ∪ {e*}
return E_T
```

The nature of Prim's algorithm makes it necessary to provide each vertex not in the current tree with the information about the shortest edge connecting the vertex to a tree vertex. We can provide such information by attaching two labels to a vertex: the name of the nearest tree vertex and the length (the weight) of the corresponding edge. Vertices that are not adjacent to any of the tree vertices can be given the ∞ label indicating their "infinite" distance to the tree vertices and a null label for the name of the nearest tree vertex. (Alternatively, we can split the vertices that are not in the tree into two sets, the "fringe" and the "unseen." The fringe contains only the vertices that are not in the tree but are adjacent to at least one tree vertex. These are the candidates from which the next tree vertex is selected. The unseen vertices are all the other vertices of the graph, called "unseen" because they are yet to be affected by the algorithm.) With such labels, finding the next vertex to be added to the current tree $T = \langle V_T, E_T \rangle$ becomes a simple task of finding a vertex with the smallest distance label in the set $V - V_T$. Ties can be broken arbitrarily.

After we have identified a vertex u^* to be added to the tree, we need to perform two operations:

- Move u^* from the set $V - V_T$ to the set of tree vertices V_T.
- For each remaining vertex u in $V - V_T$ that is connected to u^* by a shorter edge than the u's current distance label, update its labels by u^* and the weight of the edge between u^* and u, respectively.[2]

Figure 9.3 demonstrates the application of Prim's algorithm to a specific graph.

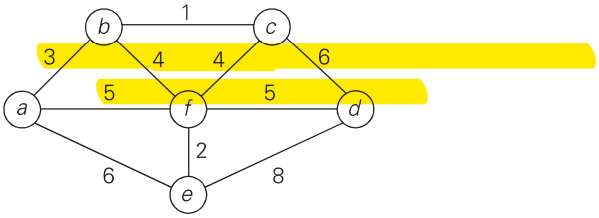
Does Prim's algorithm always yield a minimum spanning tree? The answer to this question is yes. Let us prove by induction that each of the subtrees T_i, $i = 0, \ldots, n-1$, generated by Prim's algorithm is a part (i.e., a subgraph) of some minimum spanning tree. (This immediately implies, of course, that the last tree in the sequence, T_{n-1}, is a minimum spanning tree itself because it contains all n vertices of the graph.) The basis of the induction is trivial, since T_0 consists of a single vertex and hence must be a part of any minimum spanning tree. For the inductive step, let us assume that T_{i-1} is part of some minimum spanning tree T. We need to prove that T_i, generated from T_{i-1} by Prim's algorithm, is also a part of a minimum spanning tree. We prove this by contradiction by assuming that no minimum spanning tree of the graph can contain T_i. Let $e_i = (v, u)$ be the minimum weight edge from a vertex in T_{i-1} to a vertex not in T_{i-1} used by Prim's algorithm to expand T_{i-1} to T_i. By our assumption, e_i cannot belong to any minimum spanning tree, including T. Therefore, if we add e_i to T, a cycle must be formed (Figure 9.4).

In addition to edge $e_i = (v, u)$, this cycle must contain another edge (v', u') connecting a vertex $v' \in T_{i-1}$ to a vertex u' that is not in T_{i-1}. (It is possible that v' coincides with v or u' coincides with u but not both.) If we now delete the edge (v', u') from this cycle, we will obtain another spanning tree of the entire graph whose weight is less than or equal to the weight of T since the weight of e_i is less than or equal to the weight of (v', u'). Hence, this spanning tree is a minimum spanning tree, which contradicts the assumption that no minimum spanning tree contains T_i. This completes the correctness proof of Prim's algorithm.

How efficient is Prim's algorithm? The answer depends on the data structures chosen for the graph itself and for the priority queue of the set $V - V_T$ whose vertex priorities are the distances to the nearest tree vertices. (You may want to take another look at the example in Figure 9.3 to see that the set $V - V_T$ indeed operates as a priority queue.) In particular, if a graph is represented by its weight matrix and the priority queue is implemented as an unordered array, the algorithm's running time will be in $\Theta(|V|^2)$. Indeed, on each of the $|V| - 1$ iterations, the array implementing the priority queue is traversed to find and delete the minimum and then to update, if necessary, the priorities of the remaining vertices.

We can also implement the priority queue as a ***min-heap***. A min-heap is a mirror image of the heap structure discussed in Section 6.4. (In fact, it can be implemented by constructing a heap after negating all the key values given.) Namely, a min-heap is a complete binary tree in which every element is less than or equal

2. If the implementation with the fringe/unseen split is pursued, all the unseen vertices adjacent to u^* must also be moved to the fringe.



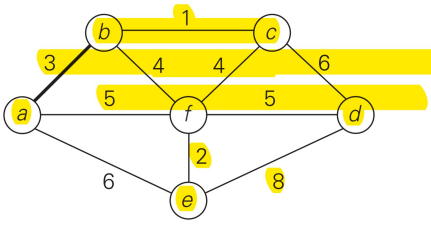
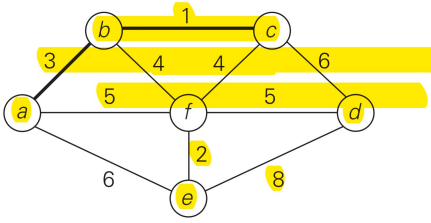
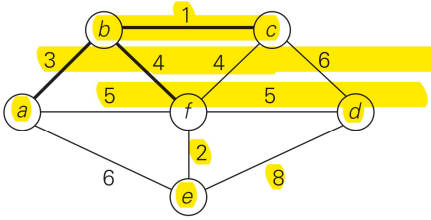
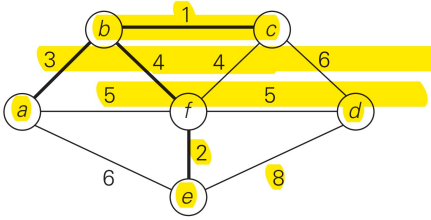
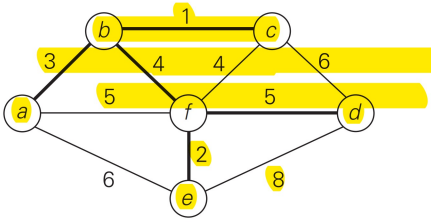
Tree vertices	Remaining vertices	Illustration
a(−, −)	**b(a, 3)** c(−, ∞) d(−, ∞) e(a, 6) f(a, 5)	
b(a, 3)	**c(b, 1)** d(−, ∞) e(a, 6) f(b, 4)	
c(b, 1)	d(c, 6) e(a, 6) **f(b, 4)**	
f(b, 4)	d(f, 5) **e(f, 2)**	
e(f, 2)	**d(f, 5)**	
d(f, 5)		

FIGURE 9.3 Application of Prim's algorithm. The parenthesized labels of a vertex in the middle column indicate the nearest tree vertex and edge weight; selected vertices and edges are shown in bold.

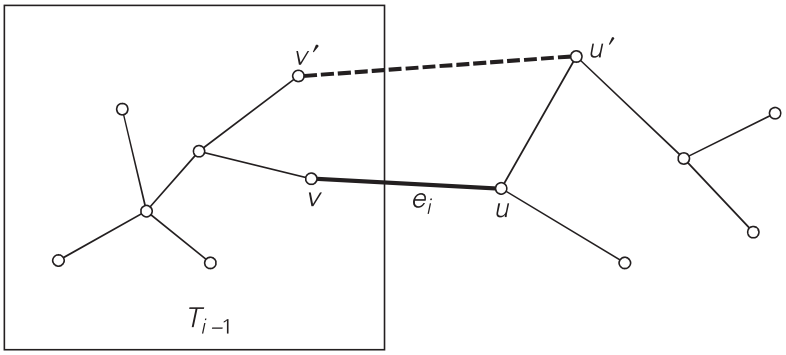


FIGURE 9.4 Correctness proof of Prim's algorithm.

to its children. All the principal properties of heaps remain valid for min-heaps, with some obvious modifications. For example, the root of a min-heap contains the smallest rather than the largest element. Deletion of the smallest element from and insertion of a new element into a min-heap of size n are $O(\log n)$ operations, and so is the operation of changing an element's priority (see Problem 15 in this section's exercises).

If a graph is represented by its adjacency lists and the priority queue is implemented as a min-heap, the running time of the algorithm is in $O(|E| \log |V|)$. This is because the algorithm performs $|V| - 1$ deletions of the smallest element and makes $|E|$ verifications and, possibly, changes of an element's priority in a min-heap of size not exceeding $|V|$. Each of these operations, as noted earlier, is a $O(\log |V|)$ operation. Hence, the running time of this implementation of Prim's algorithm is in

$$(|V| - 1 + |E|)O(\log |V|) = O(|E| \log |V|)$$

because, in a connected graph, $|V| - 1 \leq |E|$.

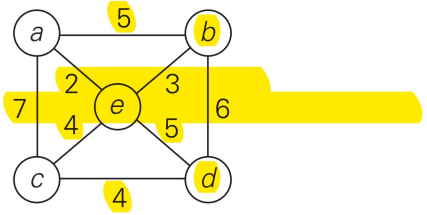
In the next section, you will find another greedy algorithm for the minimum spanning tree problem, which is "greedy" in a manner different from that of Prim's algorithm.

Exercises 9.1

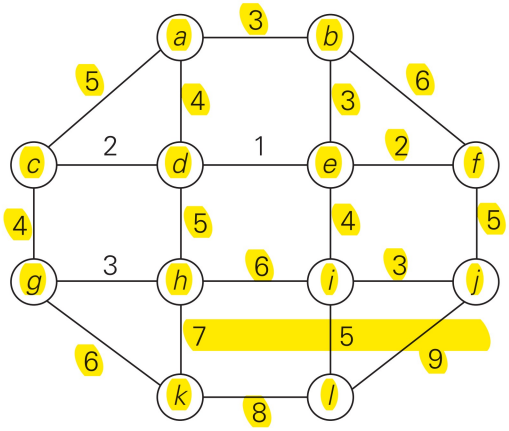
1. Write pseudocode of the greedy algorithm for the change-making problem, with an amount n and coin denominations $d_1 > d_2 > \cdots > d_m$ as its input. What is the time efficiency class of your algorithm?
2. Design a greedy algorithm for the assignment problem (see Section 3.4). Does your greedy algorithm always yield an optimal solution?
3. *Job scheduling* Consider the problem of scheduling n jobs of known durations $t_1, t_2, \ldots, t_n$ for execution by a single processor. The jobs can be executed in any order, one job at a time. You want to find a schedule that minimizes

b. weights can be put on both cups of the scale.

9. a. Apply Prim's algorithm to the following graph. Include in the priority queue all the vertices not already in the tree.



b. Apply Prim's algorithm to the following graph. Include in the priority queue only the fringe vertices (the vertices not in the current tree which are adjacent to at least one tree vertex).



10. The notion of a minimum spanning tree is applicable to a connected weighted graph. Do we have to check a graph's connectivity before applying Prim's algorithm, or can the algorithm do it by itself?

11. Does Prim's algorithm always work correctly on graphs with negative edge weights?

12. Let T be a minimum spanning tree of graph G obtained by Prim's algorithm. Let G_{new} be a graph obtained by adding to G a new vertex and some edges, with weights, connecting the new vertex to some vertices in G. Can we construct a minimum spanning tree of G_{new} by adding one of the new edges to T? If you answer yes, explain how; if you answer no, explain why not.

13. How can one use Prim's algorithm to find a spanning tree of a connected graph with no weights on its edges? Is it a good algorithm for this problem?

14. Prove that any weighted connected graph with distinct weights has exactly one minimum spanning tree.

15. Outline an efficient algorithm for changing an element's value in a min-heap. What is the time efficiency of your algorithm?

9.2 Kruskal's Algorithm

In the previous section, we considered the greedy algorithm that "grows" a minimum spanning tree through a greedy inclusion of the nearest vertex to the vertices already in the tree. Remarkably, there is another greedy algorithm for the minimum spanning tree problem that also always yields an optimal solution. It is named ***Kruskal's algorithm*** after Joseph Kruskal, who discovered this algorithm when he was a second-year graduate student [Kru56]. Kruskal's algorithm looks at a minimum spanning tree of a weighted connected graph $G = \langle V, E \rangle$ as an acyclic subgraph with $|V| - 1$ edges for which the sum of the edge weights is the smallest. (It is not difficult to prove that such a subgraph must be a tree.) Consequently, the algorithm constructs a minimum spanning tree as an expanding sequence of subgraphs that are always acyclic but are not necessarily connected on the intermediate stages of the algorithm.

The algorithm begins by sorting the graph's edges in nondecreasing order of their weights. Then, starting with the empty subgraph, it scans this sorted list, adding the next edge on the list to the current subgraph if such an inclusion does not create a cycle and simply skipping the edge otherwise.

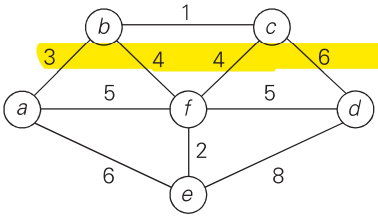
ALGORITHM *Kruskal(G)*

```
//Kruskal's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph G = ⟨V, E⟩
//Output: E_T, the set of edges composing a minimum spanning tree of G
sort E in nondecreasing order of the edge weights w(e_{i_1}) ≤ ··· ≤ w(e_{i_|E|})
E_T ← ∅;  ecounter ← 0        //initialize the set of tree edges and its size
k ← 0                         //initialize the number of processed edges
while ecounter < |V| − 1 do
    k ← k + 1
    if E_T ∪ {e_{i_k}} is acyclic
        E_T ← E_T ∪ {e_{i_k}};  ecounter ← ecounter + 1
return E_T
```

The correctness of Kruskal's algorithm can be proved by repeating the essential steps of the proof of Prim's algorithm given in the previous section. The fact that E_T is actually a tree in Prim's algorithm but generally just an acyclic subgraph in Kruskal's algorithm turns out to be an obstacle that can be overcome.

Figure 9.5 demonstrates the application of Kruskal's algorithm to the same graph we used for illustrating Prim's algorithm in Section 9.1. As you trace the algorithm's operations, note the disconnectedness of some of the intermediate subgraphs.

Applying Prim's and Kruskal's algorithms to the same small graph by hand may create the impression that the latter is simpler than the former. This impression is wrong because, on each of its iterations, Kruskal's algorithm has to check whether the addition of the next edge to the edges already selected would create a



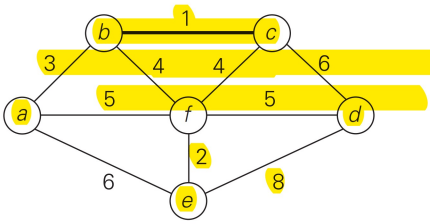
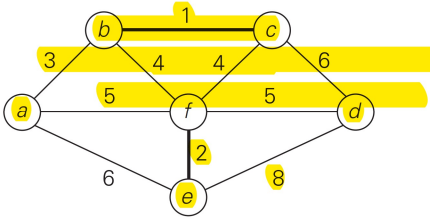
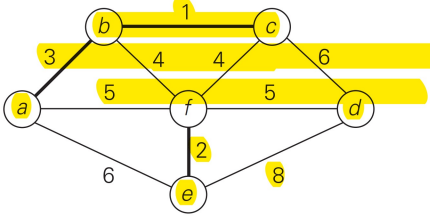
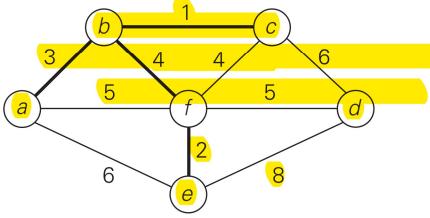
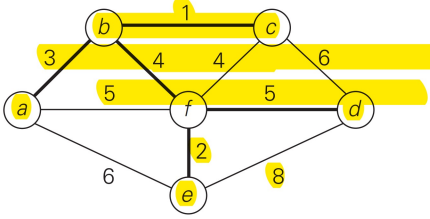
Tree edges	Sorted list of edges	Illustration
	bc ef ab bf cf af df ae cd de **1** 2 3 4 4 5 5 6 6 8	
bc 1	bc **ef** ab bf cf af df ae cd de 1 **2** 3 4 4 5 5 6 6 8	
ef 2	bc ef **ab** bf cf af df ae cd de 1 2 **3** 4 4 5 5 6 6 8	
ab 3	bc ef ab **bf** cf af df ae cd de 1 2 3 **4** 4 5 5 6 6 8	
bf 4	bc ef ab bf cf af **df** ae cd de 1 2 3 4 4 5 **5** 6 6 8	
df 5		

FIGURE 9.5 Application of Kruskal's algorithm. Selected edges are shown in bold.

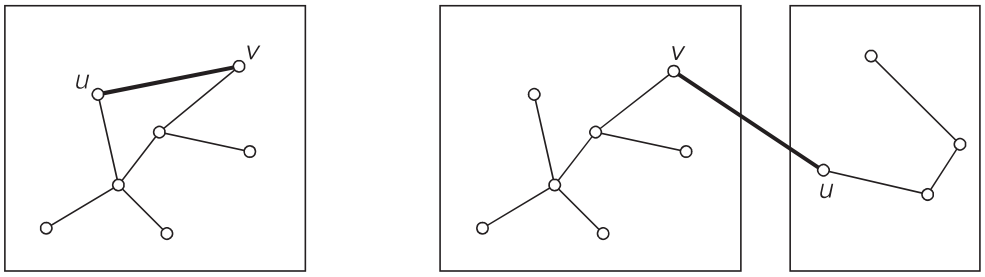


FIGURE 9.6 New edge connecting two vertices may (a) or may not (b) create a cycle.

cycle. It is not difficult to see that a new cycle is created if and only if the new edge connects two vertices already connected by a path, i.e., if and only if the two vertices belong to the same connected component (Figure 9.6). Note also that each connected component of a subgraph generated by Kruskal's algorithm is a tree because it has no cycles.

In view of these observations, it is convenient to use a slightly different interpretation of Kruskal's algorithm. We can consider the algorithm's operations as a progression through a series of forests containing *all* the vertices of a given graph and *some* of its edges. The initial forest consists of $|V|$ trivial trees, each comprising a single vertex of the graph. The final forest consists of a single tree, which is a minimum spanning tree of the graph. On each iteration, the algorithm takes the next edge (u, v) from the sorted list of the graph's edges, finds the trees containing the vertices u and v, and, if these trees are not the same, unites them in a larger tree by adding the edge (u, v).

Fortunately, there are efficient algorithms for doing so, including the crucial check for whether two vertices belong to the same tree. They are called ***union-find*** algorithms. We discuss them in the following subsection. With an efficient union-find algorithm, the running time of Kruskal's algorithm will be dominated by the time needed for sorting the edge weights of a given graph. Hence, with an efficient sorting algorithm, the time efficiency of Kruskal's algorithm will be in $O(|E| \log |E|)$.

Disjoint Subsets and Union-Find Algorithms

Kruskal's algorithm is one of a number of applications that require a dynamic partition of some n element set S into a collection of disjoint subsets $S_1, S_2, \ldots, S_k$. After being initialized as a collection of n one-element subsets, each containing a different element of S, the collection is subjected to a sequence of intermixed union and find operations. (Note that the number of union operations in any such sequence must be bounded above by $n - 1$ because each union increases a subset's size at least by 1 and there are only n elements in the entire set S.) Thus, we are

3. Give a counterexample that shows that Dijkstra's algorithm may not work for a weighted connected graph with negative weights.
4. Let T be a tree constructed by Dijkstra's algorithm in the process of solving the single-source shortest-paths problem for a weighted connected graph G.
   a. True or false: T is a spanning tree of G?
   b. True or false: T is a minimum spanning tree of G?
5. Write pseudocode for a simpler version of Dijkstra's algorithm that finds only the distances (i.e., the lengths of shortest paths but not shortest paths themselves) from a given vertex to all other vertices of a graph represented by its weight matrix.
6. Prove the correctness of Dijkstra's algorithm for graphs with positive weights.
7. Design a linear-time algorithm for solving the single-source shortest-paths problem for dags (directed acyclic graphs) represented by their adjacency lists.
8. Explain how the minimum-sum descent problem (Problem 8 in Exercises 8.1) can be solved by Dijkstra's algorithm.
9. *Shortest-path modeling* Assume you have a model of a weighted connected graph made of balls (representing the vertices) connected by strings of appropriate lengths (representing the edges).
   a. Describe how you can solve the single-pair shortest-path problem with this model.
   b. Describe how you can solve the single-source shortest-paths problem with this model.
10. Revisit the exercise from Section 1.3 about determining the best route for a subway passenger to take from one designated station to another in a well-developed subway system like those in Washington, DC, or London, UK. Write a program for this task.

9.4 Huffman Trees and Codes

Suppose we have to encode a text that comprises symbols from some n-symbol alphabet by assigning to each of the text's symbols some sequence of bits called the ***codeword***. For example, we can use a ***fixed-length encoding*** that assigns to each symbol a bit string of the same length m ($m \geq \log_2 n$). This is exactly what the standard ASCII code does. One way of getting a coding scheme that yields a shorter bit string on the average is based on the old idea of assigning shorter codewords to more frequent symbols and longer codewords to less frequent symbols. This idea was used, in particular, in the telegraph code invented in the mid-19th century by Samuel Morse. In that code, frequent letters such as e ($\cdot$) and a ($\cdot -$) are assigned short sequences of dots and dashes while infrequent letters such as q ($- - \cdot -$) and z ($- - \cdot \cdot$) have longer ones.

Variable-length encoding, which assigns codewords of different lengths to different symbols, introduces a problem that fixed-length encoding does not have. Namely, how can we tell how many bits of an encoded text represent the first (or, more generally, the ith) symbol? To avoid this complication, we can limit ourselves to the so-called ***prefix-free*** (or simply ***prefix***) ***codes***. In a prefix code, no codeword is a prefix of a codeword of another symbol. Hence, with such an encoding, we can simply scan a bit string until we get the first group of bits that is a codeword for some symbol, replace these bits by this symbol, and repeat this operation until the bit string's end is reached.

If we want to create a binary prefix code for some alphabet, it is natural to associate the alphabet's symbols with leaves of a binary tree in which all the left edges are labeled by 0 and all the right edges are labeled by 1. The codeword of a symbol can then be obtained by recording the labels on the simple path from the root to the symbol's leaf. Since there is no simple path to a leaf that continues to another leaf, no codeword can be a prefix of another codeword; hence, any such tree yields a prefix code.

Among the many trees that can be constructed in this manner for a given alphabet with known frequencies of the symbol occurrences, how can we construct a tree that would assign shorter bit strings to high-frequency symbols and longer ones to low-frequency symbols? It can be done by the following greedy algorithm, invented by David Huffman while he was a graduate student at MIT [Huf52].

Huffman's algorithm

Step 1 Initialize n one-node trees and label them with the symbols of the alphabet given. Record the frequency of each symbol in its tree's root to indicate the tree's ***weight***. (More generally, the weight of a tree will be equal to the sum of the frequencies in the tree's leaves.)

Step 2 Repeat the following operation until a single tree is obtained. Find two trees with the smallest weight (ties can be broken arbitrarily, but see Problem 2 in this section's exercises). Make them the left and right subtree of a new tree and record the sum of their weights in the root of the new tree as its weight.

A tree constructed by the above algorithm is called a ***Huffman tree***. It defines—in the manner described above—a ***Huffman code***.

EXAMPLE Consider the five-symbol alphabet {A, B, C, D, _} with the following occurrence frequencies in a text made up of these symbols:

symbol	A	B	C	D	_
frequency	0.35	0.1	0.2	0.2	0.15

The Huffman tree construction for this input is shown in Figure 9.12.

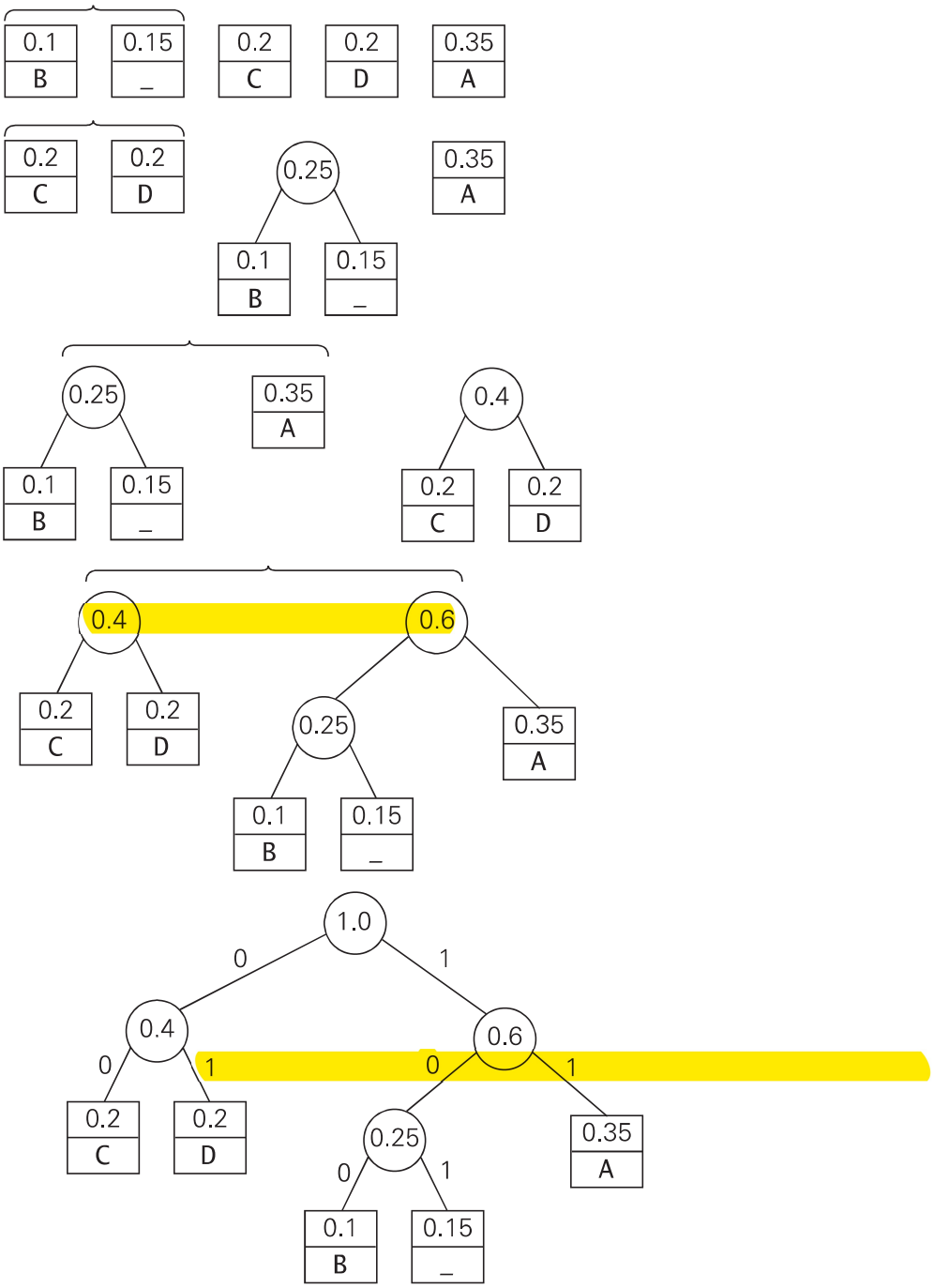


FIGURE 9.12 Example of constructing a Huffman coding tree.

The resulting codewords are as follows:

symbol	A	B	C	D	_
frequency	0.35	0.1	0.2	0.2	0.15
codeword	11	100	00	01	101

Hence, DAD is encoded as 011101, and 10011011011101 is decoded as BAD_AD.

With the occurrence frequencies given and the codeword lengths obtained, the average number of bits per symbol in this code is

$$2 \cdot 0.35 + 3 \cdot 0.1 + 2 \cdot 0.2 + 2 \cdot 0.2 + 3 \cdot 0.15 = 2.25.$$

Had we used a fixed-length encoding for the same alphabet, we would have to use at least 3 bits per each symbol. Thus, for this toy example, Huffman's code achieves the ***compression ratio***—a standard measure of a compression algorithm's effectiveness—of $(3 - 2.25)/3 \cdot 100\% = 25\%$. In other words, Huffman's encoding of the text will use 25% less memory than its fixed-length encoding. (Extensive experiments with Huffman codes have shown that the compression ratio for this scheme typically falls between 20% and 80%, depending on the characteristics of the text being compressed.) ■

Huffman's encoding is one of the most important file-compression methods. In addition to its simplicity and versatility, it yields an optimal, i.e., minimal-length, encoding (provided the frequencies of symbol occurrences are independent and known in advance). The simplest version of Huffman compression calls, in fact, for a preliminary scanning of a given text to count the frequencies of symbol occurrences in it. Then these frequencies are used to construct a Huffman coding tree and encode the text as described above. This scheme makes it necessary, however, to include the coding table into the encoded text to make its decoding possible. This drawback can be overcome by using ***dynamic Huffman encoding***, in which the coding tree is updated each time a new symbol is read from the source text. Further, modern alternatives such as ***Lempel-Ziv*** algorithms (e.g., [Say05]) assign codewords not to individual symbols but to strings of symbols, allowing them to achieve better and more robust compressions in many applications.

It is important to note that applications of Huffman's algorithm are not limited to data compression. Suppose we have n positive numbers $w_1, w_2, \ldots, w_n$ that have to be assigned to n leaves of a binary tree, one per node. If we define the ***weighted path length*** as the sum $\sum_{i=1}^{n} l_i w_i$, where l_i is the length of the simple path from the root to the ith leaf, how can we construct a binary tree with minimum weighted path length? It is this more general problem that Huffman's algorithm actually solves. (For the coding application, l_i and w_i are the length of the codeword and the frequency of the ith symbol, respectively.)

This problem arises in many situations involving decision making. Consider, for example, the game of guessing a chosen object from n possibilities (say, an integer between 1 and n) by asking questions answerable by yes or no. Different strategies for playing this game can be modeled by ***decision trees***[5] such as those depicted in Figure 9.13 for $n = 4$. The length of the simple path from the root to a leaf in such a tree is equal to the number of questions needed to get to the chosen number represented by the leaf. If number i is chosen with probability p_i, the sum

5. Decision trees are discussed in more detail in Section 11.2.

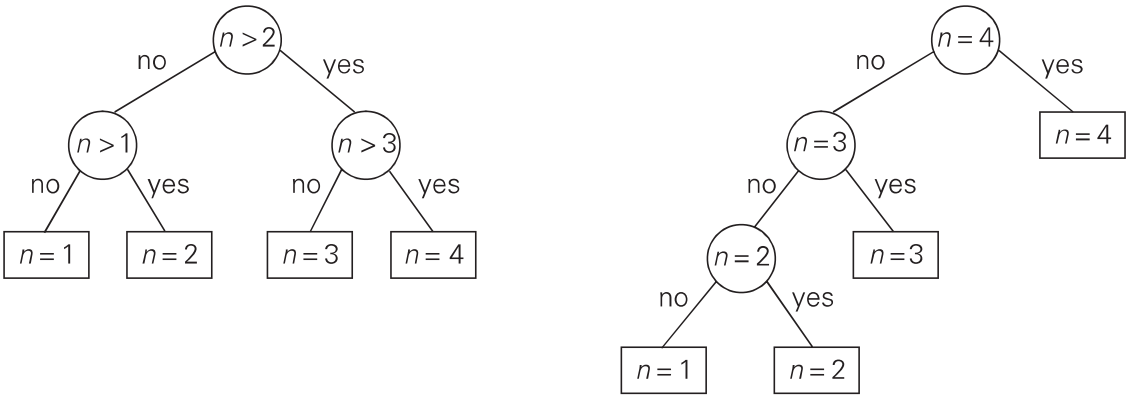


FIGURE 9.13 Two decision trees for guessing an integer between 1 and 4.

$\sum_{i=1}^{n} l_i p_i$, where l_i is the length of the path from the root to the ith leaf, indicates the average number of questions needed to "guess" the chosen number with a game strategy represented by its decision tree. If each of the numbers is chosen with the same probability of $1/n$, the best strategy is to successively eliminate half (or almost half) the candidates as binary search does. This may not be the case for arbitrary p_i's, however. For example, if $n = 4$ and $p_1 = 0.1$, $p_2 = 0.2$, $p_3 = 0.3$, and $p_4 = 0.4$, the minimum weighted path tree is the rightmost one in Figure 9.13. Thus, we need Huffman's algorithm to solve this problem in its general case.

Note that this is the second time we are encountering the problem of constructing an optimal binary tree. In Section 8.3, we discussed the problem of constructing an optimal binary search tree with positive numbers (the search probabilities) assigned to every node of the tree. In this section, given numbers are assigned just to leaves. The latter problem turns out to be easier: it can be solved by the greedy algorithm, whereas the former is solved by the more complicated dynamic programming algorithm.

Exercises 9.4

1. a. Construct a Huffman code for the following data:

symbol	A	B	C	D	_
frequency	0.4	0.1	0.2	0.15	0.15

b. Encode ABACABAD using the code of question (a).

c. Decode 100010111001010 using the code of question (a).

2. For data transmission purposes, it is often desirable to have a code with a minimum variance of the codeword lengths (among codes of the same average length). Compute the average and variance of the codeword length in two

Huffman codes that result from a different tie breaking during a Huffman code construction for the following data:

symbol	A	B	C	D	E
probability	0.1	0.1	0.2	0.2	0.4

3. Indicate whether each of the following properties is true for every Huffman code.
   a. The codewords of the two least frequent symbols have the same length.
   b. The codeword's length of a more frequent symbol is always smaller than or equal to the codeword's length of a less frequent one.
4. What is the maximal length of a codeword possible in a Huffman encoding of an alphabet of n symbols?
5. a. Write pseudocode of the Huffman-tree construction algorithm.
   b. What is the time efficiency class of the algorithm for constructing a Huffman tree as a function of the alphabet size?
6. Show that a Huffman tree can be constructed in linear time if the alphabet symbols are given in a sorted order of their frequencies.
7. Given a Huffman coding tree, which algorithm would you use to get the codewords for all the symbols? What is its time-efficiency class as a function of the alphabet size?
8. Explain how one can generate a Huffman code without an explicit generation of a Huffman coding tree.
9. a. Write a program that constructs a Huffman code for a given English text and encode it.
   b. Write a program for decoding of an English text which has been encoded with a Huffman code.
   c. Experiment with your encoding program to find a range of typical compression ratios for Huffman's encoding of English texts of, say, 1000 words.
   d. Experiment with your encoding program to find out how sensitive the compression ratios are to using standard estimates of frequencies instead of actual frequencies of symbol occurrences in English texts.
10. *Card guessing* Design a strategy that minimizes the expected number of questions asked in the following game [Gar94]. You have a deck of cards that consists of one ace of spades, two deuces of spades, three threes, and on up to nine nines, making 45 cards in all. Someone draws a card from the shuffled deck, which you have to identify by asking questions answerable with yes or no.

Chapter 10

(Limitations of Algorithm Power)

a. Prove that any algorithm for this problem must make at least $\lceil \log_3(2n+1) \rceil$ weighings in the worst case.

b. Draw a decision tree for an algorithm that solves the problem for $n = 3$ coins in two weighings.

c. Prove that there exists no algorithm that solves the problem for $n = 4$ coins in two weighings.

d. Draw a decision tree for an algorithm that solves the problem for $n = 4$ coins in two weighings by using an extra coin known to be genuine.

e. Draw a decision tree for an algorithm that solves the classic version of the problem—that for $n = 12$ coins in three weighings (with no extra coins being used).

11. *Jigsaw puzzle* A jigsaw puzzle contains n pieces. A "section" of the puzzle is a set of one or more pieces that have been connected to each other. A "move" consists of connecting two sections. What algorithm will minimize the number of moves required to complete the puzzle?

11.3 *P*, *NP*, and *NP*-Complete Problems

In the study of the computational complexity of problems, the first concern of both computer scientists and computing professionals is whether a given problem can be solved in polynomial time by some algorithm.

DEFINITION 1 We say that an algorithm solves a problem in polynomial time if its worst-case time efficiency belongs to $O(p(n))$ where $p(n)$ is a polynomial of the problem's input size n. (Note that since we are using big-oh notation here, problems solvable in, say, logarithmic time are solvable in polynomial time as well.) Problems that can be solved in polynomial time are called ***tractable***, and problems that cannot be solved in polynomial time are called ***intractable***.

There are several reasons for drawing the intractability line in this way. First, the entries of Table 2.1 and their discussion in Section 2.1 imply that we cannot solve arbitrary instances of intractable problems in a reasonable amount of time unless such instances are very small. Second, although there might be a huge difference between the running times in $O(p(n))$ for polynomials of drastically different degrees, there are very few useful polynomial-time algorithms with the degree of a polynomial higher than three. In addition, polynomials that bound running times of algorithms do not usually have extremely large coefficients. Third, polynomial functions possess many convenient properties; in particular, both the sum and composition of two polynomials are always polynomials too. Fourth, the choice of this class has led to a development of an extensive theory called ***computational complexity***, which seeks to classify problems according to their inherent difficulty. And according to this theory, a problem's intractability

remains the same for all principal models of computations and all reasonable input-encoding schemes for the problem under consideration.

We just touch on some basic notions and ideas of complexity theory in this section. If you are interested in a more formal treatment of this theory, you will have no trouble finding a wealth of textbooks devoted to the subject (e.g., [Sip05], [Aro09]).

P and *NP* Problems

Most problems discussed in this book can be solved in polynomial time by some algorithm. They include computing the product and the greatest common divisor of two integers, sorting a list, searching for a key in a list or for a pattern in a text string, checking connectivity and acyclicity of a graph, and finding a minimum spanning tree and shortest paths in a weighted graph. (You are invited to add more examples to this list.) Informally, we can think about problems that can be solved in polynomial time as the set that computer science theoreticians call P. A more formal definition includes in P only ***decision problems***, which are problems with yes/no answers.

DEFINITION 2 Class P is a class of decision problems that can be solved in polynomial time by (deterministic) algorithms. This class of problems is called ***polynomial***.

The restriction of P to decision problems can be justified by the following reasons. First, it is sensible to exclude problems not solvable in polynomial time because of their exponentially large output. Such problems do arise naturally—e.g., generating subsets of a given set or all the permutations of n distinct items—but it is apparent from the outset that they cannot be solved in polynomial time. Second, many important problems that are not decision problems in their most natural formulation can be reduced to a series of decision problems that are easier to study. For example, instead of asking about the minimum number of colors needed to color the vertices of a graph so that no two adjacent vertices are colored the same color, we can ask whether there exists such a coloring of the graph's vertices with no more than m colors for $m = 1, 2, \ldots$. (The latter is called the ***m-coloring problem***.) The first value of m in this series for which the decision problem of m-coloring has a solution solves the optimization version of the graph-coloring problem as well.

It is natural to wonder whether *every* decision problem can be solved in polynomial time. The answer to this question turns out to be no. In fact, some decision problems cannot be solved at all by any algorithm. Such problems are called ***undecidable***, as opposed to ***decidable*** problems that can be solved by an algorithm. A famous example of an undecidable problem was given by Alan

Turing in 1936.[1] The problem in question is called the ***halting problem***: given a computer program and an input to it, determine whether the program will halt on that input or continue working indefinitely on it.

Here is a surprisingly short proof of this remarkable fact. By way of contradiction, assume that A is an algorithm that solves the halting problem. That is, for any program P and input I,

$$A(P, I) = \begin{cases} 1, & \text{if program } P \text{ halts on input } I; \\ 0, & \text{if program } P \text{ does not halt on input } I. \end{cases}$$

We can consider program P as an input to itself and use the output of algorithm A for pair (P, P) to construct a program Q as follows:

$$Q(P) = \begin{cases} \text{halts}, & \text{if } A(P, P) = 0, \text{ i.e., if program } P \text{ does not halt on input } P; \\ \text{does not halt}, & \text{if } A(P, P) = 1, \text{ i.e., if program } P \text{ halts on input } P. \end{cases}$$

Then on substituting Q for P, we obtain

$$Q(Q) = \begin{cases} \text{halts}, & \text{if } A(Q, Q) = 0, \text{ i.e., if program } Q \text{ does not halt on input } Q; \\ \text{does not halt}, & \text{if } A(Q, Q) = 1, \text{ i.e., if program } Q \text{ halts on input } Q. \end{cases}$$

This is a contradiction because neither of the two outcomes for program Q is possible, which completes the proof.

Are there decidable but intractable problems? Yes, there are, but the number of *known* examples is surprisingly small, especially of those that arise naturally rather than being constructed for the sake of a theoretical argument.

There are many important problems, however, for which no polynomial-time algorithm has been found, nor has the impossibility of such an algorithm been proved. The classic monograph by M. Garey and D. Johnson [Gar79] contains a list of several hundred such problems from different areas of computer science, mathematics, and operations research. Here is just a small sample of some of the best-known problems that fall into this category:

Hamiltonian circuit problem Determine whether a given graph has a Hamiltonian circuit—a path that starts and ends at the same vertex and passes through all the other vertices exactly once.

Traveling salesman problem Find the shortest tour through n cities with known positive integer distances between them (find the shortest Hamiltonian circuit in a complete graph with positive integer weights).

1. This was just one of many breakthrough contributions to theoretical computer science made by the English mathematician and computer science pioneer Alan Turing (1912–1954). In recognition of this, the ACM—the principal society of computing professionals and researchers—has named after him an award given for outstanding contributions to theoretical computer science. A lecture given on such an occasion by Richard Karp [Kar86] provides an interesting historical account of the development of complexity theory.

Knapsack problem Find the most valuable subset of n items of given positive integer weights and values that fit into a knapsack of a given positive integer capacity.

Partition problem Given n positive integers, determine whether it is possible to partition them into two disjoint subsets with the same sum.

Bin-packing problem Given n items whose sizes are positive rational numbers not larger than 1, put them into the smallest number of bins of size 1.

Graph-coloring problem For a given graph, find its chromatic number, which is the smallest number of colors that need to be assigned to the graph's vertices so that no two adjacent vertices are assigned the same color.

Integer linear programming problem Find the maximum (or minimum) value of a linear function of several integer-valued variables subject to a finite set of constraints in the form of linear equalities and inequalities.

Some of these problems are decision problems. Those that are not have decision-version counterparts (e.g., the m-coloring problem for the graph-coloring problem). What all these problems have in common is an exponential (or worse) growth of choices, as a function of input size, from which a solution needs to be found. Note, however, that some problems that also fall under this umbrella can be solved in polynomial time. For example, the Eulerian circuit problem—the problem of the existence of a cycle that traverses all the edges of a given graph exactly once—can be solved in $O(n^2)$ time by checking, in addition to the graph's connectivity, whether all the graph's vertices have even degrees. This example is particularly striking: it is quite counterintuitive to expect that the problem about cycles traversing all the edges exactly once (Eulerian circuits) can be so much easier than the seemingly similar problem about cycles visiting all the vertices exactly once (Hamiltonian circuits).

Another common feature of a vast majority of decision problems is the fact that although solving such problems can be computationally difficult, checking whether a proposed solution actually solves the problem is computationally easy, i.e., it can be done in polynomial time. (We can think of such a proposed solution as being randomly generated by somebody leaving us with the task of verifying its validity.) For example, it is easy to check whether a proposed list of vertices is a Hamiltonian circuit for a given graph with n vertices. All we need to check is that the list contains $n + 1$ vertices of the graph in question, that the first n vertices are distinct whereas the last one is the same as the first, and that every consecutive pair of the list's vertices is connected by an edge. This general observation about decision problems has led computer scientists to the notion of a nondeterministic algorithm.

DEFINITION 3 A ***nondeterministic algorithm*** is a two-stage procedure that takes as its input an instance I of a decision problem and does the following.

Nondeterministic ("guessing") stage: An arbitrary string S is generated that can be thought of as a candidate solution to the given instance I (but may be complete gibberish as well).

Deterministic ("verification") stage: A deterministic algorithm takes both I and S as its input and outputs yes if S represents a solution to instance I. (If S is not a solution to instance I, the algorithm either returns no or is allowed not to halt at all.)

We say that a nondeterministic algorithm solves a decision problem if and only if for every yes instance of the problem it returns yes on some execution. (In other words, we require a nondeterministic algorithm to be capable of "guessing" a solution at least once and to be able to verify its validity. And, of course, we do not want it to ever output a yes answer on an instance for which the answer should be no.) Finally, a nondeterministic algorithm is said to be ***nondeterministic polynomial*** if the time efficiency of its verification stage is polynomial.

Now we can define the class of *NP* problems.

DEFINITION 4 Class *NP* is the class of decision problems that can be solved by nondeterministic polynomial algorithms. This class of problems is called ***nondeterministic polynomial***.

Most decision problems are in *NP*. First of all, this class includes all the problems in *P*:

$$P \subseteq NP.$$

This is true because, if a problem is in P, we can use the deterministic polynomial-time algorithm that solves it in the verification-stage of a nondeterministic algorithm that simply ignores string S generated in its nondeterministic ("guessing") stage. But *NP* also contains the Hamiltonian circuit problem, the partition problem, decision versions of the traveling salesman, the knapsack, graph coloring, and many hundreds of other difficult combinatorial optimization problems cataloged in [Gar79]. The halting problem, on the other hand, is among the rare examples of decision problems that are known not to be in *NP*.

This leads to the most important open question of theoretical computer science: Is P a proper subset of *NP*, or are these two classes, in fact, the same? We can put this symbolically as

$$P \stackrel{?}{=} NP.$$

Note that $P = NP$ would imply that each of many hundreds of difficult combinatorial decision problems can be solved by a polynomial-time algorithm, although computer scientists have failed to find such algorithms despite their persistent efforts over many years. Moreover, many well-known decision problems are known to be "*NP*-complete" (see below), which seems to cast more doubts on the possibility that $P = NP$.

NP-Complete Problems

Informally, an *NP*-complete problem is a problem in *NP* that is as difficult as any other problem in this class because, by definition, any other problem in *NP* can be reduced to it in polynomial time (shown symbolically in Figure 11.6).

Here are more formal definitions of these concepts.

DEFINITION 5 A decision problem D_1 is said to be ***polynomially reducible*** to a decision problem D_2, if there exists a function t that transforms instances of D_1 to instances of D_2 such that:

1. t maps all yes instances of D_1 to yes instances of D_2 and all no instances of D_1 to no instances of D_2
2. t is computable by a polynomial time algorithm

This definition immediately implies that if a problem D_1 is polynomially reducible to some problem D_2 that can be solved in polynomial time, then problem D_1 can also be solved in polynomial time (why?).

DEFINITION 6 A decision problem D is said to be ***NP-complete*** if:

1. it belongs to class *NP*
2. every problem in *NP* is polynomially reducible to D

The fact that closely related decision problems are polynomially reducible to each other is not very surprising. For example, let us prove that the Hamiltonian circuit problem is polynomially reducible to the decision version of the traveling

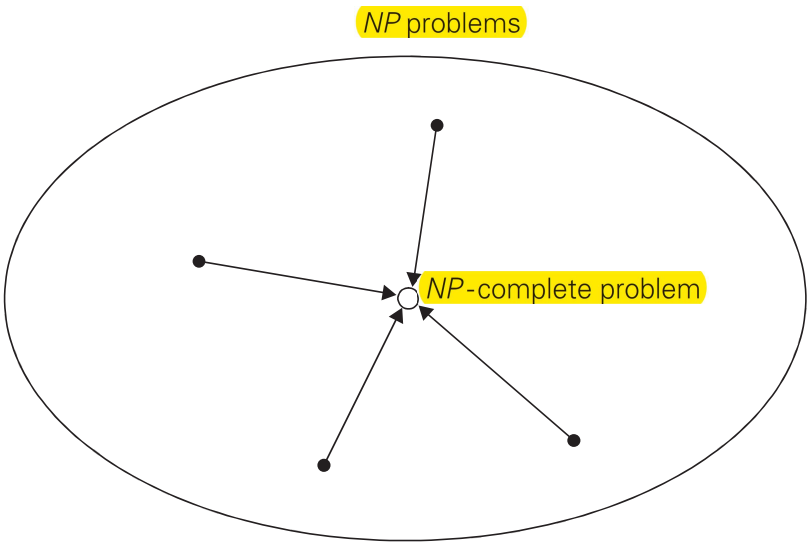


FIGURE 11.6 Notion of an *NP*-complete problem. Polynomial-time reductions of *NP* problems to an *NP*-complete problem are shown by arrows.

salesman problem. The latter can be stated as the existence problem of a Hamiltonian circuit not longer than a given positive integer m in a given complete graph with positive integer weights. We can map a graph G of a given instance of the Hamiltonian circuit problem to a complete weighted graph G' representing an instance of the traveling salesman problem by assigning 1 as the weight to each edge in G and adding an edge of weight 2 between any pair of nonadjacent vertices in G. As the upper bound m on the Hamiltonian circuit length, we take $m = n$, where n is the number of vertices in G (and G'). Obviously, this transformation can be done in polynomial time.

Let G be a yes instance of the Hamiltonian circuit problem. Then G has a Hamiltonian circuit, and its image in G' will have length n, making the image a yes instance of the decision traveling salesman problem. Conversely, if we have a Hamiltonian circuit of the length not larger than n in G', then its length must be exactly n (why?) and hence the circuit must be made up of edges present in G, making the inverse image of the yes instance of the decision traveling salesman problem be a yes instance of the Hamiltonian circuit problem. This completes the proof.

The notion of *NP*-completeness requires, however, polynomial reducibility of *all* problems in *NP,* both known and unknown, to the problem in question. Given the bewildering variety of decision problems, it is nothing short of amazing that specific examples of *NP*-complete problems have been actually found. Nevertheless, this mathematical feat was accomplished independently by Stephen Cook in the United States and Leonid Levin in the former Soviet Union.[2] In his 1971 paper, Cook [Coo71] showed that the so-called ***CNF-satisfiability problem*** is *NP*-complete. The CNF-satisfiability problem deals with boolean expressions. Each boolean expression can be represented in conjunctive normal form, such as the following expression involving three boolean variables x_1, x_2, and x_3 and their negations denoted $\bar{x}_1$, $\bar{x}_2$, and $\bar{x}_3$, respectively:

$$(x_1 \vee \bar{x}_2 \vee \bar{x}_3)\&(\bar{x}_1 \vee x_2)\&(\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3).$$

The CNF-satisfiability problem asks whether or not one can assign values *true* and *false* to variables of a given boolean expression in its CNF form to make the entire expression *true*. (It is easy to see that this can be done for the above formula: if $x_1 = \mathit{true}$, $x_2 = \mathit{true}$, and $x_3 = \mathit{false}$, the entire expression is *true*.)

Since the Cook-Levin discovery of the first known *NP*-complete problems, computer scientists have found many hundreds, if not thousands, of other examples. In particular, the well-known problems (or their decision versions) mentioned above—Hamiltonian circuit, traveling salesman, partition, bin packing, and graph coloring—are all *NP*-complete. It is known, however, that if $P \neq NP$ there must exist *NP* problems that neither are in P nor are *NP*-complete.

2. As it often happens in the history of science, breakthrough discoveries are made independently and almost simultaneously by several scientists. In fact, Levin introduced a more general notion than *NP*-completeness, which was not limited to decision problems, but his paper [Lev73] was published two years after Cook's.

For a while, the leading candidate to be such an example was the problem of determining whether a given integer is prime or composite. But in an important theoretical breakthrough, Professor Manindra Agrawal and his students Neeraj Kayal and Nitin Saxena of the Indian Institute of Technology in Kanpur announced in 2002 a discovery of a deterministic polynomial-time algorithm for primality testing [Agr04]. Their algorithm does not solve, however, the related problem of factoring large composite integers, which lies at the heart of the widely used encryption method called the ***RSA algorithm*** [Riv78].

Showing that a decision problem is *NP*-complete can be done in two steps. First, one needs to show that the problem in question is in *NP*; i.e., a randomly generated string can be checked in polynomial time to determine whether or not it represents a solution to the problem. Typically, this step is easy. The second step is to show that every problem in *NP* is reducible to the problem in question in polynomial time. Because of the transitivity of polynomial reduction, this step can be done by showing that a known *NP*-complete problem can be transformed to the problem in question in polynomial time (see Figure 11.7). Although such a transformation may need to be quite ingenious, it is incomparably simpler than proving the existence of a transformation for every problem in *NP*. For example, if we already know that the Hamiltonian circuit problem is *NP*-complete, its polynomial reducibility to the decision traveling salesman problem implies that the latter is also *NP*-complete (after an easy check that the decision traveling salesman problem is in class *NP*).

The definition of *NP*-completeness immediately implies that if there exists a deterministic polynomial-time algorithm for just one *NP*-complete problem, then every problem in *NP* can be solved in polynomial time by a deterministic algorithm, and hence $P = NP$. In other words, finding a polynomial-time algorithm

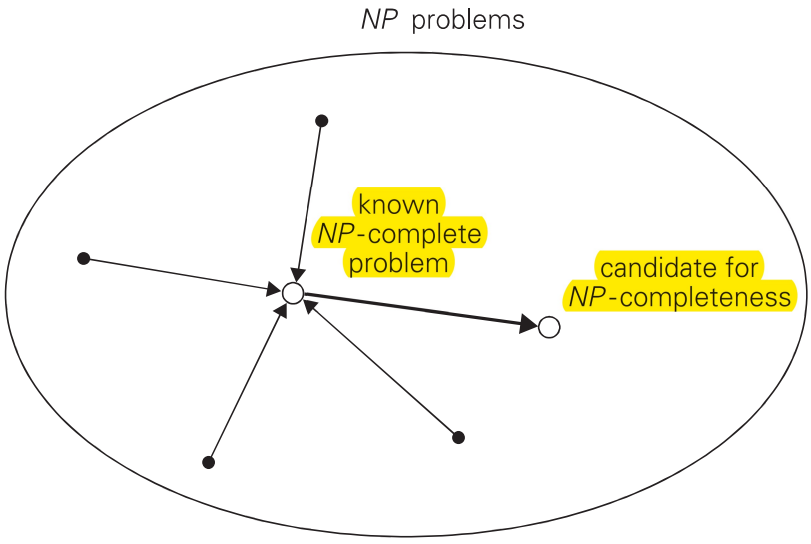


FIGURE 11.7 Proving *NP*-completeness by reduction.