3

Algorithms

Many problems can be solved by considering them as special cases of general problems. For instance, consider the problem of locating the largest integer in the sequence 101, 12, 144, 212, 98. This is a specific case of the problem of locating the largest integer in a sequence of integers. To solve this general problem we must give an algorithm, which specifies a sequence of steps used to solve this general problem. We will study algorithms for solving many different types of problems in this book. For example, in this chapter we will introduce algorithms for two of the most important problems in computer science, searching for an element in a list and sorting a list so its elements are in some prescribed order, such as increasing, decreasing, or alphabetic. Later in the book we will develop algorithms that find the greatest common divisor of two integers, that generate all the orderings of a finite set, that find the shortest path between nodes in a network, and for solving many other problems.

We will also introduce the notion of an algorithmic paradigm, which provides a general method for designing algorithms. In particular we will discuss brute-force algorithms, which find solutions using a straightforward approach without introducing any cleverness. We will also discuss greedy algorithms, a class of algorithms used to solve optimization problems. Proofs are important in the study of algorithms. In this chapter we illustrate this by proving that a particular greedy algorithm always finds an optimal solution.

One important consideration concerning an algorithm is its computational complexity, which measures the processing time and computer memory required by the algorithm to solve problems of a particular size. To measure the complexity of algorithms we use big-O and big-Theta notation, which we develop in this chapter. We will illustrate the analysis of the complexity of algorithms in this chapter, focusing on the time an algorithm takes to solve a problem. Furthermore, we will discuss what the time complexity of an algorithm means in practical and theoretical terms.

3.1 Algorithms

3.1.1 Introduction

There are many general classes of problems that arise in discrete mathematics. For instance: given a sequence of integers, find the largest one; given a set, list all its subsets; given a set of integers, put them in increasing order; given a network, find the shortest path between two vertices. When presented with such a problem, the first thing to do is to construct a model that translates the problem into a mathematical context. Discrete structures used in such models include sets, sequences, and functions—structures discussed in Chapter 2—as well as such other structures as permutations, relations, graphs, trees, networks, and finite state machines—concepts that will be discussed in later chapters.

Setting up the appropriate mathematical model is only part of the solution. To complete the solution, a method is needed that will solve the general problem using the model. Ideally, what is required is a procedure that follows a sequence of steps that leads to the desired answer. Such a sequence of steps is called an **algorithm**.

Definition 1   An *algorithm* is a finite sequence of precise instructions for performing a computation or for solving a problem.

The term *algorithm* is a corruption of the name *al-Khowarizmi*, a mathematician of the ninth century, whose book on Hindu numerals is the basis of modern decimal notation. Originally, the word *algorism* was used for the rules for performing arithmetic using decimal notation. *Algorism* evolved into the word *algorithm* by the eighteenth century. With the growing interest in computing machines, the concept of an algorithm was given a more general meaning, to include all definite procedures for solving problems, not just the procedures for performing arithmetic. (We will discuss algorithms for performing arithmetic with integers in Chapter 4.)

In this book, we will discuss algorithms that solve a wide variety of problems. In this section we will use the problem of finding the largest integer in a finite sequence of integers to illustrate the concept of an algorithm and the properties algorithms have. Also, we will describe algorithms for locating a particular element in a finite set. In subsequent sections, procedures for finding the greatest common divisor of two integers, for finding the shortest path between two points in a network, for multiplying matrices, and so on, will be discussed.

EXAMPLE 1   Describe an algorithm for finding the maximum (largest) value in a finite sequence of integers.

Extra Examples ❯

Even though the problem of finding the maximum element in a sequence is relatively trivial, it provides a good illustration of the concept of an algorithm. Also, there are many instances where the largest integer in a finite sequence of integers is required. For instance, a university may need to find the highest score on a competitive exam taken by thousands of students. Or a sports organization may want to identify the member with the highest rating each month. We want to develop an algorithm that can be used whenever the problem of finding the largest element in a finite sequence of integers arises.

We can specify a procedure for solving this problem in several ways. One method is simply to use the English language to describe the sequence of steps used. We now provide such a solution.

Solution of Example 1: We perform the following steps.

1. Set the temporary maximum equal to the first integer in the sequence. (The temporary maximum will be the largest integer examined at any stage of the procedure.)
2. Compare the next integer in the sequence to the temporary maximum, and if it is larger than the temporary maximum, set the temporary maximum equal to this integer.
3. Repeat the previous step if there are more integers in the sequence.
4. Stop when there are no integers left in the sequence. The temporary maximum at this point is the largest integer in the sequence. ◀

An algorithm can also be described using a computer language. However, when that is done, only those instructions permitted in the language can be used. This often leads to a description of the algorithm that is complicated and difficult to understand. Furthermore, because many programming languages are in common use, it would be undesirable to choose one particular language. So, instead of using a particular computer language to specify algorithms, a form of **pseudocode**, described in Appendix 3, will be used in this book. (We will also describe algorithms using the English language.) Pseudocode provides an intermediate

Links ❯

ABU JA'FAR MOHAMMED IBN MUSA AL-KHOWARIZMI (C. 780 – C. 850)   al-Khowarizmi, an astronomer and mathematician, was a member of the House of Wisdom, an academy of scientists in Baghdad. The name al-Khowarizmi means "from the town of Kowarizm," which was then part of Persia, but is now called *Khiva* and is part of Uzbekistan. al-Khowarizmi wrote books on mathematics, astronomy, and geography. Western Europeans first learned about algebra from his works. The word *algebra* comes from al-jabr, part of the title of his book *Kitab al-jabr w'al muquabala*. This book was translated into Latin and was a widely used textbook. His book on the use of Hindu numerals describes procedures for arithmetic operations using these numerals. European authors used a Latin corruption of his name, which later evolved to the word *algorithm,* to describe the subject of arithmetic with Hindu numerals.

©dbimages/Alamy Stock Photo

step between an English language description of an algorithm and an implementation of this algorithm in a programming language. The steps of the algorithm are specified using instructions resembling those used in programming languages. However, in pseudocode, the instructions used can include any well-defined operations or statements. A computer program can be produced in any computer language using the pseudocode description as a starting point.

The pseudocode used in this book is designed to be easily understood. It can serve as an intermediate step in the construction of programs implementing algorithms in one of a variety of different programming languages. Although this pseudocode does not follow the syntax of Java, C, C++, or any other programming language, students familiar with a modern programming language will find it easy to follow. A key difference between this pseudocode and code in a programming language is that we can use any well-defined instruction even if it would take many lines of code to implement this instruction. The details of the pseudocode used in the text are given in Appendix 3. The reader should refer to this appendix whenever the need arises.

A pseudocode description of the algorithm for finding the maximum element in a finite sequence follows.

ALGORITHM 1 Finding the Maximum Element in a Finite Sequence.

procedure $max(a_1, a_2, \ldots, a_n$: integers)
$max := a_1$
for $i := 2$ **to** n
    if $max < a_i$ **then** $max := a_i$
return $max\{max$ is the largest element$\}$

This algorithm first assigns the initial term of the sequence, a_1, to the variable max. The "for" loop is used to successively examine terms of the sequence. If a term is greater than the current value of max, it is assigned to be the new value of max. The algorithm terminates after all terms have been examined. The value of max on termination is the maximum element in the sequence.

To gain insight into how an algorithm works it is useful to construct a **trace** that shows its steps when given specific input. For instance, a trace of Algorithm 1 with input 8, 4, 11, 3, 10 begins with the algorithm setting max to 8, the value of the initial term. It then compares 4, the second term, with 8, the current value of max. Because $4 \leq 8$, max is unchanged. Next, the algorithm compares the third term, 11, with 8, the current value of max. Because $8 < 11$, max is set equal to 11. The algorithm then compares 3, the fourth term, and 11, the current value of max. Because $3 \leq 11$, max is unchanged. Finally, the algorithm compares 10, the first term, and 11, the current value of max. As $10 \leq 11$, max remains unchanged. Because there are five terms, we have $n = 5$. So after examining 10, the last term, the algorithm terminates, with $max = 11$. When it terminates, the algorithms reports that 11 is the largest term in the sequence.

PROPERTIES OF ALGORITHMS   There are several properties that algorithms generally share. They are useful to keep in mind when algorithms are described. These properties are:

▶ *Input.* An algorithm has input values from a specified set.
▶ *Output.* From each set of input values an algorithm produces output values from a specified set. The output values are the solution to the problem.
▶ *Definiteness.* The steps of an algorithm must be defined precisely.
▶ *Correctness.* An algorithm should produce the correct output values for each set of input values.
▶ *Finiteness.* An algorithm should produce the desired output after a finite (but perhaps large) number of steps for any input in the set.
▶ *Effectiveness.* It must be possible to perform each step of an algorithm exactly and in a finite amount of time.

▶ *Generality.* The procedure should be applicable for all problems of the desired form, not just for a particular set of input values.

EXAMPLE 2 Show that Algorithm 1 for finding the maximum element in a finite sequence of integers has all the properties listed.

Solution: The input to Algorithm 1 is a sequence of integers. The output is the largest integer in the sequence. Each step of the algorithm is precisely defined, because only assignments, a finite loop, and conditional statements occur. To show that the algorithm is correct, we must show that when the algorithm terminates, the value of the variable *max* equals the maximum of the terms of the sequence. To see this, note that the initial value of *max* is the first term of the sequence; as successive terms of the sequence are examined, *max* is updated to the value of a term if the term exceeds the maximum of the terms previously examined. This (informal) argument shows that when all the terms have been examined, *max* equals the value of the largest term. (A rigorous proof of this requires the use of mathematical induction, a proof technique developed in Section 5.1.) The algorithm uses a finite number of steps, because it terminates after all the integers in the sequence have been examined. The algorithm can be carried out in a finite amount of time because each step is either a comparison or an assignment, there are a finite number of these steps, and each of these two operations takes a finite amount of time. Finally, Algorithm 1 is general, because it can be used to find the maximum of any finite sequence of integers. ◀

3.1.2 Searching Algorithms

The problem of locating an element in an ordered list occurs in many contexts. For instance, a program that checks the spelling of words searches for them in a dictionary, which is just an ordered list of words. Problems of this kind are called **searching problems**. We will discuss several algorithms for searching in this section. We will study the number of steps used by each of these algorithms in Section 3.3.

The general searching problem can be described as follows: Locate an element x in a list of distinct elements $a_1, a_2, \ldots, a_n$, or determine that it is not in the list. The solution to this search problem is the location of the term in the list that equals x (that is, i is the solution if $x = a_i$) and is 0 if x is not in the list.

THE LINEAR SEARCH   The first algorithm that we will present is called the **linear search**, or **sequential search**, algorithm. The linear search algorithm begins by comparing x and a_1. When $x = a_1$, the solution is the location of a_1, namely, 1. When $x \neq a_1$, compare x with a_2. If $x = a_2$, the solution is the location of a_2, namely, 2. When $x \neq a_2$, compare x with a_3. Continue this process, comparing x successively with each term of the list until a match is found, where the solution is the location of that term, unless no match occurs. If the entire list has been searched without locating x, the solution is 0. The pseudocode for the linear search algorithm is displayed as Algorithm 2.

Links ❯

ALGORITHM 2 The Linear Search Algorithm.

procedure *linear search*(x: integer, $a_1, a_2, \ldots, a_n$: distinct integers)
$i := 1$
while ($i \leq n$ and $x \neq a_i$)
        $i := i + 1$
if $i \leq n$ **then** *location* $:= i$
else *location* $:= 0$
return *location*{*location* is the subscript of the term that equals x, or is 0 if x is not found}

THE BINARY SEARCH   We will now consider another searching algorithm. This algorithm can be used when the list has terms occurring in order of increasing size (for instance: if the terms are numbers, they are listed from smallest to largest; if they are words, they are listed in lexicographic, or alphabetic, order). This second searching algorithm is called the **binary search algorithm**. It proceeds by comparing the element to be located to the middle term of the list. The list is then split into two smaller sublists of the same size, or where one of these smaller lists has one fewer term than the other. The search continues by restricting the search to the appropriate sublist based on the comparison of the element to be located and the middle term. In Section 3.3, it will be shown that the binary search algorithm is much more efficient than the linear search algorithm. Example 3 demonstrates how a binary search works.

Links

EXAMPLE 3   To search for 19 in the list

1 2 3 5 6 7 8 10 12 13 15 16 18 19 20 22*,*

first split this list, which has 16 terms, into two smaller lists with eight terms each, namely,

1 2 3 5 6 7 8 10   12 13 15 16 18 19 20 22.

Then, compare 19 and the largest term in the first list. Because $10 < 19$, the search for 19 can be restricted to the list containing the 9th through the 16th terms of the original list. Next, split this list, which has eight terms, into the two smaller lists of four terms each, namely,

12 13 15 16   18 19 20 22.

Because $16 < 19$ (comparing 19 with the largest term of the first list) the search is restricted to the second of these lists, which contains the 13th through the 16th terms of the original list. The list 18 19 20 22 is split into two lists, namely,

18 19   20 22.

Because 19 is not greater than the largest term of the first of these two lists, which is also 19, the search is restricted to the first list: 18 19, which contains the 13th and 14th terms of the original list. Next, this list of two terms is split into two lists of one term each: 18 and 19. Because $18 < 19$, the search is restricted to the second list: the list containing the 14th term of the list, which is 19. Now that the search has been narrowed down to one term, a comparison is made, and 19 is located as the 14th term in the original list.   ◄

We now specify the steps of the binary search algorithm. To search for the integer x in the list $a_1, a_2, \ldots, a_n$, where $a_1 < a_2 < \cdots < a_n$, begin by comparing x with the middle term a_m of the list, where $m = \lfloor (n + 1)/2 \rfloor$. (Recall that $\lfloor x \rfloor$ is the greatest integer not exceeding x.) If $x > a_m$, the search for x is restricted to the second half of the list, which is $a_{m+1}, a_{m+2}, \ldots, a_n$. If x is not greater than a_m, the search for x is restricted to the first half of the list, which is $a_1, a_2, \ldots, a_m$.

The search has now been restricted to a list with no more than $\lceil n/2 \rceil$ elements. (Recall that $\lceil x \rceil$ is the smallest integer greater than or equal to x.) Using the same procedure, compare x to the middle term of the restricted list. Then restrict the search to the first or second half of the list. Repeat this process until a list with one term is obtained. Then determine whether this term is x. Pseudocode for the binary search algorithm is displayed as Algorithm 3.

ALGORITHM 3 **The Binary Search Algorithm.**

procedure *binary search* (x: integer, $a_1, a_2, \ldots, a_n$: increasing integers)
$i := 1 \{i$ is left endpoint of search interval$\}$
$j := n \{j$ is right endpoint of search interval$\}$
while $i < j$
  $m := \lfloor (i + j)/2 \rfloor$
  if $x > a_m$ **then** $i := m + 1$
  else $j := m$
if $x = a_i$ **then** *location* := i
else *location* := 0
return *location*$\{location$ is the subscript i of the term a_i equal to x, or 0 if x is not found$\}$

Algorithm 3 proceeds by successively narrowing down the part of the sequence being searched. At any given stage only the terms from a_i to a_j are under consideration. In other words, i and j are the smallest and largest subscripts of the remaining terms, respectively. Algorithm 3 continues narrowing the part of the sequence being searched until only one term of the sequence remains. When this is done, a comparison is made to see whether this term equals x.

3.1.3 Sorting

Demo ❯

Ordering the elements of a list is a problem that occurs in many contexts. For example, to produce a telephone directory it is necessary to alphabetize the names of subscribers. Similarly, producing a directory of songs available for downloading requires that their titles be put in alphabetic order. Putting addresses in order in an e-mail mailing list can determine whether there are duplicated addresses. Creating a useful dictionary requires that words be put in alphabetical order. Similarly, generating a parts list requires that we order them according to increasing part number.

Suppose that we have a list of elements of a set. Furthermore, suppose that we have a way to order elements of the set. (The notion of ordering elements of sets will be discussed in detail in Section 9.6.) **Sorting** is putting these elements into a list in which the elements are in increasing order. For instance, sorting the list 7, 2, 1, 4, 5, 9 produces the list 1, 2, 4, 5, 7, 9. Sorting the list d, h, c, a, f (using alphabetical order) produces the list a, c, d, f, h.

An amazingly large percentage of computing resources is devoted to sorting one thing or another. Hence, much effort has been devoted to the development of sorting algorithms. A surprisingly large number of sorting algorithms have been devised using distinct strategies, with new ones introduced regularly. In the third volume of his fundamental work *The Art of Computer Programming* [Kn98], Donald Knuth devotes close to 400 pages to sorting, covering around 15 different sorting algorithms in depth! More than 100 sorting algorithms have been devised, and it is surprising how often new sorting algorithms are developed. Among the newest sorting algorithms that have caught on is a widely useful algorithm called Timsort, which was invented in 2002, and the library sort, also known as the gapped insertion sort, invented in 2006.

Sorting is thought to hold the record as the problem solved by the most fundamentally different algorithms!

There are many reasons why sorting algorithms interest computer scientists and mathematicians. Among these reasons are that some algorithms are easier to implement, some algorithms are more efficient (either in general, or when given input with certain characteristics, such as lists slightly out of order), some algorithms take advantage of particular computer architectures, and some algorithms are particularly clever. In this section we will introduce two sorting algorithms, the bubble sort and the insertion sort. Two other sorting algorithms, the selection

sort and the binary insertion sort, are introduced in the exercises, and the shaker sort is introduced in the Supplementary Exercises. In Section 5.4 we will discuss the merge sort and introduce the quick sort in the exercises in that section; the tournament sort is introduced in the exercise set in Section 11.2. We cover sorting algorithms both because sorting is an important problem and because these algorithms can serve as examples for many important concepts.

THE BUBBLE SORT   The **bubble sort** is one of the simplest sorting algorithms, but not one of the most efficient. It puts a list into increasing order by successively comparing adjacent elements, interchanging them if they are in the wrong order. To carry out the bubble sort, we perform the basic operation, that is, interchanging a larger element with a smaller one following it, starting at the beginning of the list, for a full pass. We iterate this procedure until the sort is complete. Pseudocode for the bubble sort is given as Algorithm 4. We can imagine the elements in the list placed in a column. In the bubble sort, the smaller elements "bubble" to the top as they are interchanged with larger elements. The larger elements "sink" to the bottom. This is illustrated in Example 4.

Links

EXAMPLE 4   Use the bubble sort to put 3, 2, 4, 1, 5 into increasing order.

Solution: The steps of this algorithm are illustrated in Figure 1. Begin by comparing the first two elements, 3 and 2. Because 3 > 2, interchange 3 and 2, producing the list 2, 3, 4, 1, 5. Because 3 < 4, continue by comparing 4 and 1. Because 4 > 1, interchange 1 and 4, producing the list 2, 3, 1, 4, 5. Because 4 < 5, the first pass is complete. The first pass guarantees that the largest element, 5, is in the correct position.

The second pass begins by comparing 2 and 3. Because these are in the correct order, 3 and 1 are compared. Because 3 > 1, these numbers are interchanged, producing 2, 1, 3, 4, 5. Because 3 < 4, these numbers are in the correct order. It is not necessary to do any more comparisons for this pass because 5 is already in the correct position. The second pass guarantees that the two largest elements, 4 and 5, are in their correct positions.

The third pass begins by comparing 2 and 1. These are interchanged because 2 > 1, producing 1, 2, 3, 4, 5. Because 2 < 3, these two elements are in the correct order. It is not necessary to do any more comparisons for this pass because 4 and 5 are already in the correct positions. The third pass guarantees that the three largest elements, 3, 4, and 5, are in their correct positions.

The fourth pass consists of one comparison, namely, the comparison of 1 and 2. Because 1 < 2, these elements are in the correct order. This completes the bubble sort. ◀
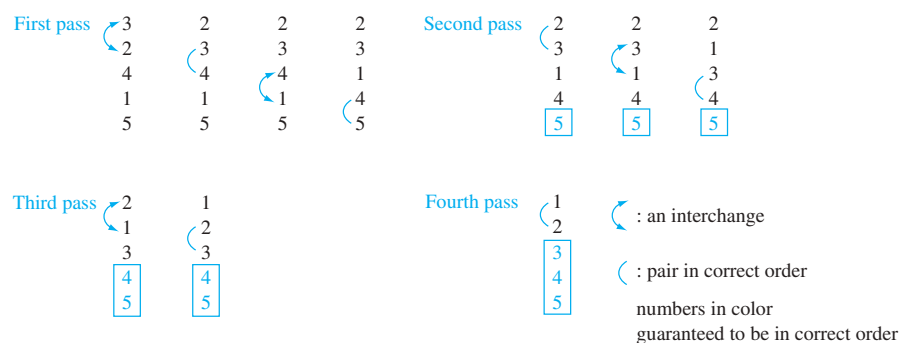

FIGURE 1   **The steps of a bubble sort.**

ALGORITHM 4 The Bubble Sort.

procedure *bubblesort*($a_1, \ldots, a_n$: real numbers with $n \geq 2$)
for $i := 1$ **to** $n - 1$
    for $j := 1$ **to** $n - i$
        if $a_j > a_{j+1}$ **then** interchange a_j and a_{j+1}
$\{a_1, \ldots, a_n$ is in increasing order$\}$

THE INSERTION SORT   The **insertion sort** is a simple sorting algorithm, but it is usually not the most efficient. To sort a list with n elements, the insertion sort begins with the second element. The insertion sort compares this second element with the first element and inserts it before the first element if it does not exceed the first element and after the first element if it exceeds the first element. At this point, the first two elements are in the correct order. The third element is then compared with the first element, and if it is larger than the first element, it is compared with the second element; it is inserted into the correct position among the first three elements.

Links ❯

In general, in the jth step of the insertion sort, the jth element of the list is inserted into the correct position in the list of the previously sorted $j - 1$ elements. To insert the jth element in the list, a linear search technique is used (see Exercise 45); the jth element is successively compared with the already sorted $j - 1$ elements at the start of the list until the first element that is not less than this element is found or until it has been compared with all $j - 1$ elements; the jth element is inserted in the correct position so that the first j elements are sorted. The algorithm continues until the last element is placed in the correct position relative to the already sorted list of the first $n - 1$ elements. The insertion sort is described in pseudocode in Algorithm 5.

EXAMPLE 5   Use the insertion sort to put the elements of the list 3, 2, 4, 1, 5 in increasing order.

Solution: The insertion sort first compares 2 and 3. Because $3 > 2$, it places 2 in the first position, producing the list 2, 3, 4, 1, 5 (the sorted part of the list is shown in color). At this point, 2 and 3 are in the correct order. Next, it inserts the third element, 4, into the already sorted part of the list by making the comparisons $4 > 2$ and $4 > 3$. Because $4 > 3$, 4 remains in the third position. At this point, the list is 2, 3, 4, 1, 5 and we know that the ordering of the first three elements is correct. Next, we find the correct place for the fourth element, 1, among the already sorted elements, 2, 3, 4. Because $1 < 2$, we obtain the list 1, 2, 3, 4, 5. Finally, we insert 5 into the correct position by successively comparing it to 1, 2, 3, and 4. Because $5 > 4$, it stays at the end of the list, producing the correct order for the entire list.   ◀

ALGORITHM 5 The Insertion Sort.

procedure *insertion sort*($a_1, a_2, \ldots, a_n$: real numbers with $n \geq 2$)
for $j := 2$ **to** n
    $i := 1$
    while $a_j > a_i$
        $i := i + 1$
    $m := a_j$
    for $k := 0$ **to** $j - i - 1$
        $a_{j-k} := a_{j-k-1}$
    $a_i := m$
$\{a_1, \ldots, a_n$ is in increasing order$\}$

3.1.4 String Matching

Although searching and sorting are the most commonly encountered problems in computer science, many other problems arise frequently. One of these problems asks where a particular string of characters P, called the **pattern**, occurs, if it does, within another string T, called the **text**. For instance, we can ask whether the pattern 101 can be found within the string 11001011. By inspection we can see that the pattern 101 occurs within the text 11001011 at a shift of four characters, because 101 is the string formed by the fifth, sixth, and seventh characters of the text. On the other hand, the pattern 111 does not occur within the text 110110001101.

Finding where a pattern occurs in a text string is called **string matching**. String matching plays an essential role in a wide variety of applications, including text editing, spam filters, systems that look for attacks in a computer network, search engines, plagiarism detection, bioinformatics, and many other important applications. For example, in text editing, the string matching problem arises whenever we need to find all occurrences of a string so that we can replace this string with a different string. Search engines look for matching of search keywords with words on web pages. Many problems in bioinformatics arise in the study of DNA molecules, which are made up of four bases: thymine (T), adenine (A), cytosine (C), and guanine (G). The process of DNA sequencing is the determination of the order of the four bases in DNA. This leads to string matching problems involving strings made up from the four letters T, A, C, and G. For instance, we can ask whether the pattern CAG occurs in the text CATCACAGAGA. The answer is yes, because it occurs with a shift of five characters. Solving questions about the genome requires the use of efficient algorithms for string matching, especially because a string representing a human genome is about 3×10^9 characters long.

We will now describe a brute force algorithm, Algorithm 6, for string matching, called the **naive string matcher**. The input to this algorithm is the pattern we wish to match, $P = p_1 p_2 \ldots p_m$, and the text, $T = t_1 t_2 \ldots t_n$. When this pattern begins at position $s + 1$ in the text T, we say that P occurs with **shift** s in T, that is, when $t_{s+1} = p_1, t_{s+2} = p_2, \ldots, t_{s+m} = p_m$. To find all valid shifts, the naive string matcher runs through all possible shifts s from $s = 0$ to $s = n - m$, checking whether s is a valid shift. In Figure 2, we display the operation of Algorithm 6 when it is used to search for the pattern $P = eye$ in the text $T = eceyeye$.

ALGORITHM 6 Naive String Matcher.

procedure *string match* (n, m: positive integers, $m \leq n, t_1, t_2, \ldots, t_n, p_1, p_2, \ldots, p_m$: characters)
    for $s := 0$ **to** $n - m$
        $j := 1$
        while ($j \leq m$ and $t_{s+j} = p_j$)
            $j := j + 1$
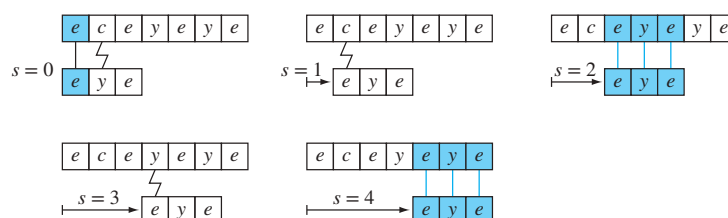   **if** $j > m$ **then print** "s is a valid shift"

FIGURE 2 **The steps of the naive string matcher with $P = eye$ in $T = eceyeye$. Matches are identified with a solid line and mismatches with a jagged line. The algorithm finds two valid shifts, $s = 2$ and $s = 4$.**

Many other string matching algorithms have been developed besides the naive string matcher. These algorithms use a surprisingly wide variety of approaches to make them more efficient than the naive string matcher. To learn more about these algorithms, consult [CoLeRiSt09], as well as books on algorithms in bioinformatics.

3.1.5 Greedy Algorithms

Many algorithms we will study in this book are designed to solve **optimization problems**. The goal of such problems is to find a solution to the given problem that either minimizes or maximizes the value of some parameter. Optimization problems studied later in this text include finding a route between two cities with least total mileage, determining a way to encode messages using the fewest bits possible, and finding a set of fiber links between network nodes using the least amount of fiber.

Surprisingly, one of the simplest approaches often leads to a solution of an optimization problem. This approach selects the best choice at each step, instead of considering all sequences of steps that may lead to an optimal solution. Algorithms that make what seems to be the "best" choice at each step are called **greedy algorithms**. Once we know that a greedy algorithm finds a feasible solution, we need to determine whether it has found an optimal solution. (Note that we call the algorithm "greedy" whether or not it finds an optimal solution.) To do this, we either prove that the solution is optimal or we show that there is a counterexample where the algorithm yields a nonoptimal solution. To make these concepts more concrete, we will consider the **cashier's algorithm** that makes change using coins. (This algorithm is called the cashier's algorithm because cashiers often used this algorithm for making change in the days before cash registers became electronic.)

EXAMPLE 6 Consider the problem of making n cents change with quarters, dimes, nickels, and pennies, and using the least total number of coins. We can devise a greedy algorithm for making change for n cents by making a locally optimal choice at each step; that is, at each step we choose the coin of the largest denomination possible to add to the pile of change without exceeding n cents. For example, to make change for 67 cents, we first select a quarter (leaving 42 cents). We next select a second quarter (leaving 17 cents), followed by a dime (leaving 7 cents), followed by a nickel (leaving 2 cents), followed by a penny (leaving 1 cent), followed by a penny. ◀

Demo

We display the cashier's algorithm for n cents, using any set of denominations of coins, as Algorithm 7.

ALGORITHM 7 Cashier's Algorithm.

procedure *change*($c_1, c_2, \ldots, c_r$: values of denominations of coins, where
$\quad c_1 > c_2 > \cdots > c_r$; n: a positive integer)
for $i := 1$ **to** r
$\quad d_i := 0$ $\{d_i$ counts the coins of denomination c_i used$\}$
$\quad$ **while** $n \geq c_i$
$\quad\quad d_i := d_i + 1$ $\{$add a coin of denomination $c_i\}$
$\quad\quad n := n - c_i$
$\{d_i$ is the number of coins of denomination c_i in the change for $i = 1, 2, \ldots, r\}$

We have described the cashier's algorithm, a greedy algorithm for making change, using any finite set of coins with denominations $c_1, c_2, \ldots, c_r$. In the particular case where the four denominations are quarters, dimes, nickels, and pennies, we have $c_1 = 25$, $c_2 = 10$, $c_3 = 5$, and $c_4 = 1$. For this case, we will show that this algorithm leads to an optimal solution in the sense

that it uses the fewest coins possible. Before we embark on our proof, we show that there are sets of coins for which the cashier's algorithm (Algorithm 7) does not necessarily produce change using the fewest coins possible. For example, if we have only quarters, dimes, and pennies (and no nickels) to use, the cashier's algorithm would make change for 30 cents using six coins—a quarter and five pennies—whereas we could have used three coins, namely, three dimes.

LEMMA 1   If n is a positive integer, then n cents in change using quarters, dimes, nickels, and pennies using the fewest coins possible has at most two dimes, at most one nickel, at most four pennies, and cannot have two dimes and a nickel. The amount of change in dimes, nickels, and pennies cannot exceed 24 cents.

Proof: We use a proof by contradiction. We will show that if we had more than the specified number of coins of each type, we could replace them using fewer coins that have the same value. We note that if we had three dimes we could replace them with a quarter and a nickel, if we had two nickels we could replace them with a dime, if we had five pennies we could replace them with a nickel, and if we had two dimes and a nickel we could replace them with a quarter. Because we can have at most two dimes, one nickel, and four pennies, but we cannot have two dimes and a nickel, it follows that 24 cents is the most money we can have in dimes, nickels, and pennies when we make change using the fewest number of coins for n cents.   ◁

THEOREM 1   The cashier's algorithm (Algorithm 7) always makes changes using the fewest coins possible when change is made from quarters, dimes, nickels, and pennies.

Proof: We will use a proof by contradiction. Suppose that there is a positive integer n such that there is a way to make change for n cents using quarters, dimes, nickels, and pennies that uses fewer coins than the greedy algorithm finds. We first note that q', the number of quarters used in this optimal way to make change for n cents, must be the same as q, the number of quarters used by the greedy algorithm. To show this, first note that the greedy algorithm uses the most quarters possible, so $q' \leq q$. However, it is also the case that q' cannot be less than q. If it were, we would need to make up at least 25 cents from dimes, nickels, and pennies in this optimal way to make change. But this is impossible by Lemma 1.

Because there must be the same number of quarters in the two ways to make change, the value of the dimes, nickels, and pennies in these two ways must be the same, and these coins are worth no more than 24 cents. There must be the same number of dimes, because the greedy algorithm used the most dimes possible and by Lemma 1, when change is made using the fewest coins possible, at most one nickel and at most four pennies are used, so that the most dimes possible are also used in the optimal way to make change. Similarly, we have the same number of nickels and, finally, the same number of pennies.   ◁

A greedy algorithm makes the best choice at each step according to a specified criterion. The next example shows that it can be difficult to determine which of many possible criteria to choose.

EXAMPLE 7   Suppose we have a group of proposed talks with preset start and end times. Devise a greedy algorithm to schedule as many of these talks as possible in a lecture hall, under the assumptions that once a talk starts, it continues until it ends, no two talks can proceed at the same time, and a talk can begin at the same time another one ends. Assume that talk j begins at time s_j (where s stands for *start*) and ends at time e_j (where e stands for *end*).

Solution: To use a greedy algorithm to schedule the most talks, that is, an optimal schedule, we need to decide how to choose which talk to add at each step. There are many criteria we could

use to select a talk at each step, where we chose from the talks that do not overlap talks already selected. For example, we could add talks in order of earliest start time, we could add talks in order of shortest time, we could add talks in order of earliest finish time, or we could use some other criterion.

We now consider these possible criteria. Suppose we add the talk that starts earliest among the talks compatible with those already selected. We can construct a counterexample to see that the resulting algorithm does not always produce an optimal schedule. For instance, suppose that we have three talks: Talk 1 starts at 8 A.M. and ends at 12 noon, Talk 2 starts at 9 A.M. and ends at 10 A.M., and Talk 3 starts at 11 A.M. and ends at 12 noon. We first select the Talk 1 because it starts earliest. But once we have selected Talk 1 we cannot select either Talk 2 or Talk 3 because both overlap Talk 1. Hence, this greedy algorithm selects only one talk. This is not optimal because we could schedule Talk 2 and Talk 3, which do not overlap.

Now suppose we add the talk that is shortest among the talks that do not overlap any of those already selected. Again we can construct a counterexample to show that this greedy algorithm does not always produce an optimal schedule. So, suppose that we have three talks: Talk 1 starts at 8 A.M. and ends at 9:15 A.M., Talk 2 starts at 9 A.M. and ends at 10 A.M., and Talk 3 starts at 9:45 A.M. and ends at 11 A.M. We select Talk 2 because it is shortest, requiring one hour. Once we select Talk 2, we cannot select either Talk 1 or Talk 3 because neither is compatible with Talk 2. Hence, this greedy algorithm selects only one talk. However, it is possible to select two talks, Talk 1 and Talk 3, which are compatible.

However, it can be shown that we schedule the most talks possible if in each step we select the talk with the earliest ending time among the talks compatible with those already selected. We will prove this in Chapter 5 using the method of mathematical induction. The first step we will make is to sort the talks according to increasing finish time. After this sorting, we relabel the talks so that $e_1 \leq e_2 \leq \cdots \leq e_n$. The resulting greedy algorithm is given as Algorithm 8. ◀

ALGORITHM 8 Greedy Algorithm for Scheduling Talks.

procedure *schedule*($s_1 \leq s_2 \leq \cdots \leq s_n$: start times of talks,
    $e_1 \leq e_2 \leq \cdots \leq e_n$: ending times of talks)
sort talks by finish time and reorder so that $e_1 \leq e_2 \leq \cdots \leq e_n$
$S := \emptyset$
for $j := 1$ **to** n
    if talk j is compatible with S **then**
        $S := S \cup \{\text{talk } j\}$
return $S\{S$ is the set of talks scheduled$\}$

3.1.6 The Halting Problem

We will now describe a proof of one of the most famous theorems in computer science. We will show that there is a problem that cannot be solved using any procedure. That is, we will show there are unsolvable problems. The problem we will study is the **halting problem**. It asks whether there is a procedure that does this: It takes as input a computer program and input to the program and determines whether the program will eventually stop when run with this input. It would be convenient to have such a procedure, if it existed. Certainly being able to test whether a program entered into an infinite loop would be helpful when writing and debugging programs. However, in 1936 Alan Turing showed that no such procedure exists (see his biography in Section 13.4).

Before we present a proof that the halting problem is unsolvable, first note that we cannot simply run a program and observe what it does to determine whether it terminates when run
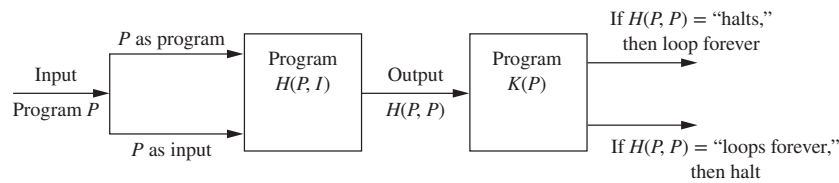
FIGURE 3 **Showing that the halting problem is unsolvable.**

with the given input. If the program halts, we have our answer, but if it is still running after any fixed length of time has elapsed, we do not know whether it will never halt or we just did not wait long enough for it to terminate. After all, it is not hard to design a program that will stop only after more than a billion years has elapsed.

We will describe Turing's proof that the halting problem is unsolvable; it is a proof by contradiction. (The reader should note that our proof is not completely rigorous, because we have not explicitly defined what a procedure is. To remedy this, the concept of a Turing machine is needed. This concept is introduced in Section 13.5.)

Proof: Assume there is a solution to the halting problem, a procedure called $H(P, I)$. The procedure $H(P, I)$ takes two inputs, one a program P and the other I, an input to the program P. $H(P, I)$ generates the string "halt" as output if H determines that P stops when given I as input. Otherwise, $H(P, I)$ generates the string "loops forever" as output. We will now derive a contradiction.

When a procedure is coded, it is expressed as a string of characters; this string can be interpreted as a sequence of bits. This means that a program itself can be used as data. Therefore, a program can be thought of as input to another program, or even itself. Hence, H can take a program P as both of its inputs, which are a program and input to this program. H should be able to determine whether P will halt when it is given a copy of itself as input.

To show that no procedure H exists that solves the halting problem, we construct a simple procedure $K(P)$, which works as follows, making use of the output $H(P, P)$. If the output of $H(P, P)$ is "loops forever," which means that P loops forever when given a copy of itself as input, then $K(P)$ halts. If the output of $H(P, P)$ is "halt," which means that P halts when given a copy of itself as input, then $K(P)$ loops forever. That is, $K(P)$ does the opposite of what the output of $H(P, P)$ specifies. (See Figure 3.)

Now suppose we provide K as intput to K. We note that if the output of $H(K, K)$ is "loops forever," then by the definition of K, we see that $K(K)$ halts. This means that by the definition of H, the output of $H(K, K)$ is "halt," which is a contradiction. Otherwise, if the output of $H(K, K)$ is "halts," then by the definition of K, we see that $K(K)$ loops forever, which means that by the definition of H, the output of $H(K, K)$ is "loops forever." This is also a contradiction. Thus, H cannot always give the correct answers. Consequently, there is no procedure that solves the halting problem. ◁

Exercises

1. List all the steps used by Algorithm 1 to find the maximum of the list 1, 8, 12, 9, 11, 2, 14, 5, 10, 4.

2. Determine which characteristics of an algorithm described in the text (after Algorithm 1) the following procedures have and which they lack.

a) **procedure** *double*(*n*: positive integer)
    while $n > 0$
        $n := 2n$

b) **procedure** *divide*(*n*: positive integer)
    while $n \geq 0$
        $m := 1/n$
        $n := n - 1$

c) **procedure** *sum*(*n*: positive integer)
    $sum := 0$
    while $i < 10$
        $sum := sum + i$

d) **procedure** *choose*(*a*, *b*: integers)
    $x := $ either a or b

3. Devise an algorithm that finds the sum of all the integers in a list.

4. Describe an algorithm that takes as input a list of n integers and produces as output the largest difference obtained by subtracting an integer in the list from the one following it.

5. Describe an algorithm that takes as input a list of n integers in nondecreasing order and produces the list of all values that occur more than once. (Recall that a list of integers is **nondecreasing** if each integer in the list is at least as large as the previous integer in the list.)

6. Describe an algorithm that takes as input a list of n integers and finds the number of negative integers in the list.

7. Describe an algorithm that takes as input a list of n integers and finds the location of the last even integer in the list or returns 0 if there are no even integers in the list.

8. Describe an algorithm that takes as input a list of n distinct integers and finds the location of the largest even integer in the list or returns 0 if there are no even integers in the list.

9. A **palindrome** is a string that reads the same forward and backward. Describe an algorithm for determining whether a string of n characters is a palindrome.

10. Devise an algorithm to compute x^n, where x is a real number and n is an integer. [*Hint:* First give a procedure for computing x^n when n is nonnegative by successive multiplication by x, starting with 1. Then extend this procedure, and use the fact that $x^{-n} = 1/x^n$ to compute x^n when n is negative.]

11. Describe an algorithm that interchanges the values of the variables x and y, using only assignments. What is the minimum number of assignment statements needed to do this?

12. Describe an algorithm that uses only assignment statements that replaces the triple (x, y, z) with (y, z, x). What is the minimum number of assignment statements needed?

13. List all the steps used to search for 9 in the sequence 1, 3, 4, 5, 6, 8, 9, 11 using

    a) a linear search.   **b)** a binary search.

14. List all the steps used to search for 7 in the sequence given in Exercise 13 for both a linear search and a binary search.

15. Describe an algorithm that inserts an integer x in the appropriate position into the list $a_1, a_2, \ldots, a_n$ of integers that are in increasing order.

16. Describe an algorithm for finding the smallest integer in a finite sequence of natural numbers.

17. Describe an algorithm that locates the first occurrence of the largest element in a finite list of integers, where the integers in the list are not necessarily distinct.

18. Describe an algorithm that locates the last occurrence of the smallest element in a finite list of integers, where the integers in the list are not necessarily distinct.

19. Describe an algorithm that produces the maximum, median, mean, and minimum of a set of three integers. (The **median** of a set of integers is the middle element in the list when these integers are listed in order of increasing size. The **mean** of a set of integers is the sum of the integers divided by the number of integers in the set.)

20. Describe an algorithm for finding both the largest and the smallest integers in a finite sequence of integers.

21. Describe an algorithm that puts the first three terms of a sequence of integers of arbitrary length in increasing order.

22. Describe an algorithm to find the longest word in an English sentence (where a sentence is a sequence of symbols, either a letter or a blank, which can then be broken into alternating words and blanks).

23. Describe an algorithm that determines whether a function from a finite set of integers to another finite set of integers is onto.

24. Describe an algorithm that determines whether a function from a finite set to another finite set is one-to-one.

25. Describe an algorithm that will count the number of 1s in a bit string by examining each bit of the string to determine whether it is a 1 bit.

26. Change Algorithm 3 so that the binary search procedure compares x to a_m at each stage of the algorithm, with the algorithm terminating if $x = a_m$. What advantage does this version of the algorithm have?

27. The **ternary search algorithm** locates an element in a list of increasing integers by successively splitting the list into three sublists of equal (or as close to equal as possible) size, and restricting the search to the appropriate piece. Specify the steps of this algorithm.

28. Specify the steps of an algorithm that locates an element in a list of increasing integers by successively splitting the list into four sublists of equal (or as close to equal as possible) size, and restricting the search to the appropriate piece.

In a list of elements the same element may appear several times. A **mode** of such a list is an element that occurs at least as often as each of the other elements; a list has more than one mode when more than one element appears the maximum number of times.

29. Devise an algorithm that finds a mode in a list of nondecreasing integers. (Recall that a list of integers is nondecreasing if each term is at least as large as the preceding term.)

30. Devise an algorithm that finds all modes. (Recall that a list of integers is nondecreasing if each term of the list is at least as large as the preceding term.)

31. Two strings are **anagrams** if each can be formed from the other string by rearranging its characters. Devise an algorithm to determine whether two strings are anagrams

    a) by first finding the frequency of each character that appears in the strings.

    b) by first sorting the characters in both strings.

32. Given n real numbers $x_1, x_2, \ldots, x_n$, find the two that are closest together by
  a) a brute force algorithm that finds the distance between every pair of these numbers.
  b) sorting the numbers and computing the least number of distances needed to solve the problem.

33. Devise an algorithm that finds the first term of a sequence of integers that equals some previous term in the sequence.

34. Devise an algorithm that finds all terms of a finite sequence of integers that are greater than the sum of all previous terms of the sequence.

35. Devise an algorithm that finds the first term of a sequence of positive integers that is less than the immediately preceding term of the sequence.

36. Use the bubble sort to sort 6, 2, 3, 1, 5, 4, showing the lists obtained at each step.

37. Use the bubble sort to sort 3, 1, 5, 7, 4, showing the lists obtained at each step.

38. Use the bubble sort to sort d, f, k, m, a, b, showing the lists obtained at each step.

∗39. Adapt the bubble sort algorithm so that it stops when no interchanges are required. Express this more efficient version of the algorithm in pseudocode.

40. Use the insertion sort to sort the list in Exercise 36, showing the lists obtained at each step.

41. Use the insertion sort to sort the list in Exercise 37, showing the lists obtained at each step.

42. Use the insertion sort to sort the list in Exercise 38, showing the lists obtained at each step.

The **selection sort** begins by finding the least element in the list. This element is moved to the front. Then the least element among the remaining elements is found and put into the second position. This procedure is repeated until the entire list has been sorted.

Links

43. Sort these lists using the selection sort.
  a) 3, 5, 4, 1, 2   **b)** 5, 4, 3, 2, 1
  c) 1, 2, 3, 4, 5

44. Write the selection sort algorithm in pseudocode.

45. Describe an algorithm based on the linear search for determining the correct position in which to insert a new element in an already sorted list.

46. Describe an algorithm based on the binary search for determining the correct position in which to insert a new element in an already sorted list.

47. How many comparisons does the insertion sort use to sort the list 1, 2, …, n?

48. How many comparisons does the insertion sort use to sort the list $n, n − 1, \ldots, 2, 1$?

The **binary insertion sort** is a variation of the insertion sort that uses a binary search technique (see Exercise 46) rather than a linear search technique to insert the ith element in the correct place among the previously sorted elements.

49. Show all the steps used by the binary insertion sort to sort the list 3, 2, 4, 5, 1, 6.

50. Compare the number of comparisons used by the insertion sort and the binary insertion sort to sort the list 7, 4, 3, 8, 1, 5, 4, 2.

∗51. Express the binary insertion sort in pseudocode.

52. a) Devise a variation of the insertion sort that uses a linear search technique that inserts the jth element in the correct place by first comparing it with the $(j − 1)$st element, then the $(j − 2)$th element if necessary, and so on.
  b) Use your algorithm to sort 3, 2, 4, 5, 1, 6.
  c) Answer Exercise 47 using this algorithm.
  d) Answer Exercise 48 using this algorithm.

53. When a list of elements is in close to the correct order, would it be better to use an insertion sort or its variation described in Exercise 52?

54. List all the steps the naive string matcher uses to find all occurrences of the pattern FE in the text COVFEFE.

55. List all the steps the naive string matcher uses to find all occurrences of the pattern ACG in the text TACAGACG.

56. Use the cashier's algorithm to make change using quarters, dimes, nickels, and pennies for
  a) 87 cents.   **b)** 49 cents.
  c) 99 cents.   **d)** 33 cents.

57. Use the cashier's algorithm to make change using quarters, dimes, nickels, and pennies for
  a) 51 cents.   **b)** 69 cents.
  c) 76 cents.   **d)** 60 cents.

58. Use the cashier's algorithm to make change using quarters, dimes, and pennies (but no nickels) for each of the amounts given in Exercise 56. For which of these amounts does the greedy algorithm use the fewest coins of these denominations possible?

59. Use the cashier's algorithm to make change using quarters, dimes, and pennies (but no nickels) for each of the amounts given in Exercise 57. For which of these amounts does the greedy algorithm use the fewest coins of these denominations possible?

60. Show that if there were a coin worth 12 cents, the cashier's algorithm using quarters, 12-cent coins, dimes, nickels, and pennies would not always produce change using the fewest coins possible.

61. Use Algorithm 7 to schedule the largest number of talks in a lecture hall from a proposed set of talks, if the starting and ending times of the talks are 9:00 A.M. and 9:45 A.M.; 9:30 A.M. and 10:00 A.M.; 9:50 A.M. and 10:15 A.M.; 10:00 A.M. and 10:30 A.M.; 10:10 A.M. and 10:25 A.M.; 10:30 A.M. and 10:55 A.M.; 10:15 A.M. and 10:45 A.M.; 10:30 A.M. and 11:00 A.M.; 10:45 A.M. and 11:30 A.M.; 10:55 A.M. and 11:25 A.M.; 11:00 A.M. and 11:15 A.M.

62. Show that a greedy algorithm that schedules talks in a lecture hall, as described in Example 7, by selecting at each step the talk that overlaps the fewest other talks, does not always produce an optimal schedule.

*63. a) Devise a greedy algorithm that determines the fewest lecture halls needed to accommodate n talks given the starting and ending time for each talk.

   b) Prove that your algorithm is optimal.

Suppose we have s men $m_1, m_2, \ldots, m_s$ and s women $w_1, w_2, \ldots, w_s$. We wish to match each person with a member of the opposite gender. Furthermore, suppose that each person ranks, in order of preference, with no ties, the people of the opposite gender. We say that a matching of people of opposite genders to form couples is **stable** if we cannot find a man m and a woman w who are not assigned to each other such that m prefers w over his assigned partner and w prefers m to her assigned partner.

64. Suppose we have three men m_1, m_2, and m_3 and three women w_1, w_2, and w_3. Furthermore, suppose that the preference rankings of the men for the three women, from highest to lowest, are $m_1: w_3, w_1, w_2$; $m_2: w_1, w_2, w_3$; $m_3: w_2, w_3, w_1$; and the preference rankings of the women for the three men, from highest to lowest, are $w_1: m_1, m_2, m_3$; $w_2: m_2, m_1, m_3$; $w_3: m_3, m_2, m_1$. For each of the six possible matchings of men and women to form three couples, determine whether this matching is stable.

The **deferred acceptance algorithm**, also known as the **Gale-Shapley algorithm**, can be used to construct a stable matching of men and women. In this algorithm, members of one gender are the **suitors** and members of the other gender the **suitees**. The algorithm uses a sequence of rounds; in each round every suitor whose proposal was rejected in the previous round proposes to his or her highest ranking suitee who has not already rejected a proposal from this suitor. A suitee rejects all proposals except that from the suitor that this suitee ranks highest among all the suitors who have proposed to this suitee in this round or previous rounds. The proposal of this highest ranking suitor remains pending and is rejected in a later round if a more appealing suitor proposes in that round. The series of rounds ends when every suitor has exactly one pending proposal. All pending proposals are then accepted.

65. Write the deferred acceptance algorithm in pseudocode.

66. Show that the deferred acceptance algorithm terminates.

*67. Show that the deferred acceptance always terminates with a stable assignment.

An element of a sequence is called a **majority element** if it occurs repeatedly for more than half the terms of the sequence. The **Boyer-Moore majority vote algorithm** (named after Robert Boyer and J. Strother Moore) finds the majority element of a sequence, if it exists. The algorithm maintains a counter that is initially set to 0 and a temporary candidate for a majority element initially with no assigned value. The algorithm processes the elements in order. When it processes the first element, this element becomes the majority candidate and the counter is set equal to 1. Then, as it processes the remaining elements in order, if the counter is 0, this element becomes the majority candidate and the counter is set equal to 1, while if the counter is nonzero, the counter is incremented (that is, 1 is added to it) or decremented (that is, 1 is subtracted from it), depending on whether this element equals the current candidate. After all the terms are processed, the majority candidate is the majority element, if it exists.

68. a) Explain why a sequence has at most one majority element.

   b) Show all the steps of the Boyer-Moore majority vote algorithm when given the sequence 2, 1, 3, 3, 2, 3.

   c) Express the Boyer-Moore majority vote algorithm in pseudocode.

   d) Explain how you can determine whether the majority candidate element produced by the Boyer-Moore algorithm is actually a majority element.

*69. a) Prove that the Boyer-Moore majority vote algorithm outputs the majority element of a sequence, if it exists.

   b) Prove or disprove that the majority candidate of the Boyer-Moore majority vote algorithm will be a mode of the sequence (that is, its most common element) even when no majority element exists.

70. Show that the problem of determining whether a program with a given input ever prints the digit 1 is unsolvable.

71. Show that the following problem is solvable. Given two programs with their inputs and the knowledge that exactly one of them halts, determine which halts.

72. Show that the problem of deciding whether a specific program with a specific input halts is solvable.

3.2 The Growth of Functions

3.2.1 Introduction

In Section 3.1 we discussed the concept of an algorithm. We introduced algorithms that solve a variety of problems, including searching for an element in a list and sorting a list. In Section 3.3 we will study the number of operations used by these algorithms. In particular, we will estimate the number of comparisons used by the linear and binary search algorithms to find an element in a sequence of n elements. We will also estimate the number of comparisons used by the bubble sort and by the insertion sort to sort a list of n elements. The time required to solve a problem depends on more than only the number of operations it uses. The time also depends on the hardware and software used to run the program that implements the algorithm. However, when we change the hardware and software used to implement an algorithm, we can closely

approximate the time required to solve a problem of size n by multiplying the previous time required by a constant. For example, on a supercomputer we might be able to solve a problem of size n a million times faster than we can on a PC. However, this factor of one million will not depend on n (except perhaps in some minor ways). One of the advantages of using **big-O notation**, which we introduce in this section, is that we can estimate the growth of a function without worrying about constant multipliers or smaller order terms. This means that, using big-O notation, we do not have to worry about the hardware and software used to implement an algorithm. Furthermore, using big-O notation, we can assume that the different operations used in an algorithm take the same time, which simplifies the analysis considerably.

Big-O notation is used extensively to estimate the number of operations an algorithm uses as its input grows. With the help of this notation, we can determine whether it is practical to use a particular algorithm to solve a problem as the size of the input increases. Furthermore, using big-O notation, we can compare two algorithms to determine which is more efficient as the size of the input grows. For instance, if we have two algorithms for solving a problem, one using $100n^2 + 17n + 4$ operations and the other using n^3 operations, big-O notation can help us see that the first algorithm uses far fewer operations when n is large, even though it uses more operations for small values of n, such as $n = 10$.

This section introduces big-O notation and the related big-Omega and big-Theta notations. We will explain how big-O, big-Omega, and big-Theta estimates are constructed and establish estimates for some important functions that are used in the analysis of algorithms.

3.2.2   Big-O Notation

The growth of functions is often described using a special notation. Definition 1 describes this notation.

Definition 1

Let f and g be functions from the set of integers or the set of real numbers to the set of real numbers. We say that $f(x)$ is $O(g(x))$ if there are constants C and k such that

$$|f(x)| \leq C|g(x)|$$

whenever $x > k$. [This is read as "$f(x)$ is big-oh of $g(x)$."]

Remark: Intuitively, the definition that $f(x)$ is $O(g(x))$ says that $f(x)$ grows slower than some fixed multiple of $g(x)$ as x grows without bound.

The constants C and k in the definition of big-O notation are called **witnesses** to the relationship $f(x)$ is $O(g(x))$. To establish that $f(x)$ is $O(g(x))$ we need only one pair of witnesses to this relationship. That is, to show that $f(x)$ is $O(g(x))$, we need find only *one* pair of constants C and k, the witnesses, such that $|f(x)| \leq C|g(x)|$ whenever $x > k$.

Note that when there is one pair of witnesses to the relationship $f(x)$ is $O(g(x))$, there are *infinitely many* pairs of witnesses. To see this, note that if C and k are one pair of witnesses, then any pair C' and k', where $C < C'$ and $k < k'$, is also a pair of witnesses, because $|f(x)| \leq C|g(x)| \leq C'|g(x)|$ whenever $x > k' > k$.

THE HISTORY OF BIG-O NOTATION   Big-O notation has been used in mathematics for more than a century. In computer science it is widely used in the analysis of algorithms, as will be seen in Section 3.3. The German mathematician Paul Bachmann first introduced big-O notation in 1892 in an important book on number theory. The big-O symbol is sometimes called a **Landau symbol** after the German mathematician Edmund Landau, who used this notation throughout his work. The use of big-O notation in computer science was popularized by Donald Knuth, who also introduced the big-Ω and big-Θ notations defined later in this section.

WORKING WITH THE DEFINITION OF BIG-O NOTATION   A useful approach for finding a pair of witnesses is to first select a value of k for which the size of $|f(x)|$ can be readily estimated when $x > k$ and to see whether we can use this estimate to find a value of C for which $|f(x)| \leq C|g(x)|$ for $x > k$. This approach is illustrated in Example 1.

EXAMPLE 1   Show that $f(x) = x^2 + 2x + 1$ is $O(x^2)$.

> *Extra*
> *Examples*

Solution: We observe that we can readily estimate the size of $f(x)$ when $x > 1$ because $x < x^2$ and $1 < x^2$ when $x > 1$. It follows that

$$0 \leq x^2 + 2x + 1 \leq x^2 + 2x^2 + x^2 = 4x^2$$

whenever $x > 1$, as shown in Figure 1. Consequently, we can take $C = 4$ and $k = 1$ as witnesses to show that $f(x)$ is $O(x^2)$. That is, $f(x) = x^2 + 2x + 1 < 4x^2$ whenever $x > 1$. (Note that it is not necessary to use absolute values here because all functions in these equalities are positive when x is positive.)

Alternatively, we can estimate the size of $f(x)$ when $x > 2$. When $x > 2$, we have $2x \leq x^2$ and $1 \leq x^2$. Consequently, if $x > 2$, we have

$$0 \leq x^2 + 2x + 1 \leq x^2 + x^2 + x^2 = 3x^2.$$

It follows that $C = 3$ and $k = 2$ are also witnesses to the relation $f(x)$ is $O(x^2)$.

Observe that in the relationship "$f(x)$ is $O(x^2)$," x^2 can be replaced by any function that has larger values than x^2 for all $x \geq k$ for some positive real number k. For example, $f(x)$ is $O(x^3)$, $f(x)$ is $O(x^2 + x + 7)$, and so on.

It is also true that x^2 is $O(x^2 + 2x + 1)$, because $x^2 < x^2 + 2x + 1$ whenever $x > 1$. This means that $C = 1$ and $k = 1$ are witnesses to the relationship x^2 is $O(x^2 + 2x + 1)$.   ◄

Note that in Example 1 we have two functions, $f(x) = x^2 + 2x + 1$ and $g(x) = x^2$, such that $f(x)$ is $O(g(x))$ and $g(x)$ is $O(f(x))$—the latter fact following from the inequality $x^2 \leq x^2 + 2x + 1$, which holds for all nonnegative real numbers x. We say that two functions
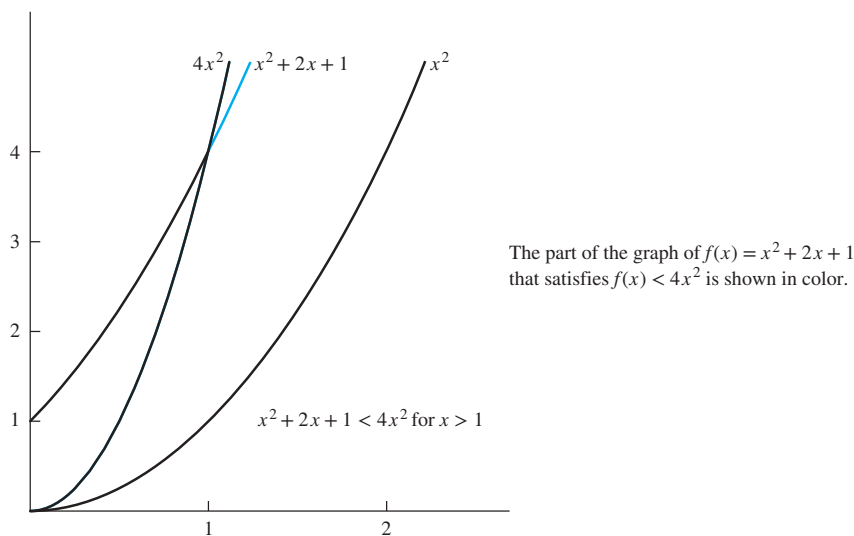

FIGURE 1   **The function $x^2 + 2x + 1$ is $O(x^2)$.**

$f(x)$ and $g(x)$ that satisfy both of these big-O relationships are of the **same order**. We will return to this notion later in this section.

Remark: The fact that $f(x)$ is $O(g(x))$ is sometimes written $f(x) = O(g(x))$. However, the equals sign in this notation does *not* represent a genuine equality. Rather, this notation tells us that an inequality holds relating the values of the functions f and g for sufficiently large numbers in the domains of these functions. However, it is acceptable to write $f(x) \in O(g(x))$ because $O(g(x))$ represents the set of functions that are $O(g(x))$.

When $f(x)$ is $O(g(x))$, and $h(x)$ is a function that has larger absolute values than $g(x)$ does for sufficiently large values of x, it follows that $f(x)$ is $O(h(x))$. In other words, the function $g(x)$ in the relationship $f(x)$ is $O(g(x))$ can be replaced by a function with larger absolute values. To see this, note that if

$$|f(x)| \leq C|g(x)| \qquad \text{if } x > k,$$

and if $|h(x)| > |g(x)|$ for all $x > k$, then

$$|f(x)| \leq C|h(x)| \qquad \text{if } x > k.$$

Hence, $f(x)$ is $O(h(x))$.

When big-O notation is used, the function g in the relationship $f(x)$ is $O(g(x))$ is often chosen to have the smallest growth rate of the functions belonging to a set of reference functions, such as functions of the form x^n, where n is a positive real number. (Important reference functions are discussed later in this section.)

In subsequent discussions, we will almost always deal with functions that take on only positive values. All references to absolute values can be dropped when working with big-O estimates for such functions. Figure 2 illustrates the relationship $f(x)$ is $O(g(x))$.
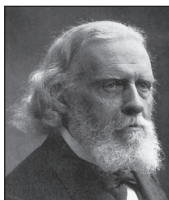
PAUL GUSTAV HEINRICH BACHMANN (1837–1920) Paul Bachmann, the son of a Lutheran pastor, shared his father's pious lifestyle and love of music. His mathematical talent was discovered by one of his teachers, even though he had difficulties with some of his early mathematical studies. After recuperating from tuberculosis in Switzerland, Bachmann studied mathematics, first at the University of Berlin and later at Göttingen, where he attended lectures presented by the famous number theorist Dirichlet. He received his doctorate under the German number theorist Kummer in 1862; his thesis was on group theory. Bachmann was a professor at Breslau and later at Münster. After he retired from his professorship, he continued his mathematical writing, played the piano, and served as a music critic for newspapers. Bachmann's mathematical writings include a five-volume survey of results and methods in number theory, a two-volume work on elementary number theory, a book on irrational numbers, and a book on the famous conjecture known as Fermat's Last Theorem. He introduced big-O notation in his 1892 book *Analytische Zahlentheorie*.

EDMUND LANDAU (1877–1938) Edmund Landau, the son of a Berlin gynecologist, attended high school and university in Berlin. He received his doctorate in 1899, under the direction of Frobenius. Landau first taught at the University of Berlin and then moved to Göttingen, where he was a full professor until the Nazis forced him to stop teaching. Landau's main contributions to mathematics were in the field of analytic number theory. In particular, he established several important results concerning the distribution of primes. He authored a three-volume exposition on number theory as well as other books on number theory and mathematical analysis.
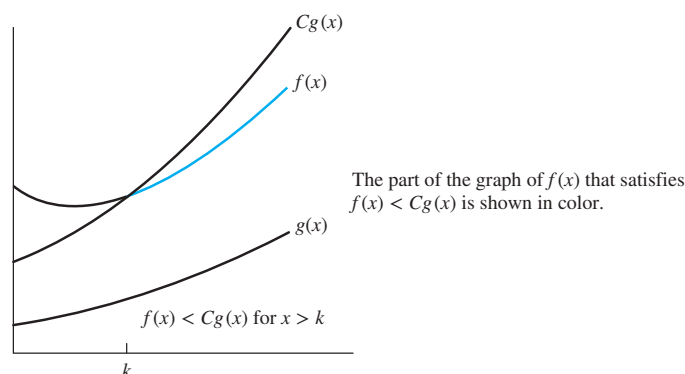
The part of the graph of $f(x)$ that satisfies $f(x) < Cg(x)$ is shown in color.

$f(x) < Cg(x)$ for $x > k$

FIGURE 2 **The function $f(x)$ is $O(g(x))$.**

Example 2 illustrates how big-O notation is used to estimate the growth of functions.

EXAMPLE 2   Show that $7x^2$ is $O(x^3)$.

Solution: Note that when $x > 7$, we have $7x^2 < x^3$. (We can obtain this inequality by multiplying both sides of $x > 7$ by x^2.) Consequently, we can take $C = 1$ and $k = 7$ as witnesses to establish the relationship $7x^2$ is $O(x^3)$. Alternatively, when $x > 1$, we have $7x^2 < 7x^3$, so that $C = 7$ and $k = 1$ are also witnesses to the relationship $7x^2$ is $O(x^3)$.   ◄

Links ▶

DONALD E. KNUTH (BORN 1938)   Knuth grew up in Milwaukee, where his father taught bookkeeping at a Lutheran high school and owned a small printing business. He was an excellent student, earning academic achievement awards. He applied his intelligence in unconventional ways, winning a contest when he was in the eighth grade by finding over 4500 words that could be formed from the letters in "Ziegler's Giant Bar." This won a television set for his school and a candy bar for everyone in his class.

Courtesy of Stanford University News Service

Knuth had a difficult time choosing physics over music as his major at the Case Institute of Technology. He then switched from physics to mathematics, and in 1960 he received his bachelor of science degree, simultaneously receiving a master of science degree by a special award of the faculty who considered his work outstanding. At Case, he managed the basketball team and applied his talents by constructing a formula for the value of each player. This novel approach was covered by *Newsweek* and by Walter Cronkite on the CBS television network. Knuth began graduate work at the California Institute of Technology in 1960 and received his Ph.D. there in 1963. During this time he worked as a consultant, writing compilers for different computers.

Knuth joined the staff of the California Institute of Technology in 1963, where he remained until 1968, when he took a job as a full professor at Stanford University. He retired as Professor Emeritus in 1992 to concentrate on writing. He is especially interested in updating and completing new volumes of his series *The Art of Computer Programming,* a work that has had a profound influence on the development of computer science, which he began writing as a graduate student in 1962, focusing on compilers. In common jargon, "Knuth," referring to *The Art of Computer Programming*, has come to mean the reference that answers all questions about such topics as data structures and algorithms.

Knuth is the founder of the modern study of computational complexity. He has made fundamental contributions to the subject of compilers. His dissatisfaction with mathematics typography sparked him to invent the now widely used TeX and Metafont systems. TeX has become a standard language for computer typography. Two of the many awards Knuth has received are the 1974 Turing Award and the 1979 National Medal of Technology, awarded to him by President Carter.

Knuth has written for a wide range of professional journals in computer science and in mathematics. However, his first publication, in 1957, when he was a college freshman, was a parody of the metric system called "The Potrzebie Systems of Weights and Measures," which appeared in *MAD Magazine* and has been in reprint several times. He is a church organist, as his father was. He is also a composer of music for the organ. Knuth believes that writing computer programs can be an aesthetic experience, much like writing poetry or composing music.

Knuth pays $2.56 for the first person to find each error in his books and $0.32 for significant suggestions. If you send him a letter with an error (you will need to use regular mail, because he has given up reading e-mail), he will eventually inform you whether you were the first person to tell him about this error. Be prepared for a long wait, because he receives an overwhelming amount of mail. (The author received a letter years after sending an error report to Knuth, noting that this report arrived several months after the first report of this error.)

Remark: In Example 2 we did not choose the smallest possible power of x the reference function in the big-O estimate. Note that $7x^2$ is also big-O of x^2 and x^2 grows much slower than x^3. In fact, x^2 would be the smallest possible power of x suitable as the reference function in the big-O estimate.

Example 3 illustrates how to show that a big-O relationship does not hold.

EXAMPLE 3   Show that n^2 is not $O(n)$.

Solution: To show that n^2 is not $O(n)$, we must show that no pair of witnesses C and k exist such that $n^2 \le Cn$ whenever $n > k$. We will use a proof by contradiction to show this.

Suppose that there are constants C and k for which $n^2 \le Cn$ whenever $n > k$. Observe that when $n > 0$ we can divide both sides of the inequality $n^2 \le Cn$ by n to obtain the equivalent inequality $n \le C$. However, no matter what C and k are, the inequality $n \le C$ cannot hold for all n with $n > k$. In particular, once we set a value of k, we see that when n is larger than the maximum of k and C, it is not true that $n \le C$ even though $n > k$. This contradiction shows that n^2 is not $O(n)$. ◄

EXAMPLE 4   Example 2 shows that $7x^2$ is $O(x^3)$. Is it also true that x^3 is $O(7x^2)$?

Solution: To determine whether x^3 is $O(7x^2)$, we need to determine whether witnesses C and k exist, so that $x^3 \le C(7x^2)$ whenever $x > k$. We will show that no such witnesses exist using a proof by contradiction.

If C and k are witnesses, the inequality $x^3 \le C(7x^2)$ holds for all $x > k$. Observe that the inequality $x^3 \le C(7x^2)$ is equivalent to the inequality $x \le 7C$, which follows by dividing both sides by the positive quantity x^2. However, no matter what C is, it is not the case that $x \le 7C$ for all $x > k$ no matter what k is, because x can be made arbitrarily large. It follows that no witnesses C and k exist for this proposed big-O relationship. Hence, x^3 is *not* $O(7x^2)$. ◄

3.2.3   Big-O Estimates for Some Important Functions

Polynomials can often be used to estimate the growth of functions. Instead of analyzing the growth of polynomials each time they occur, we would like a result that can always be used to estimate the growth of a polynomial. Theorem 1 does this. It shows that the leading term of a polynomial dominates its growth by asserting that a polynomial of degree n or less is $O(x^n)$.

THEOREM 1   Let $f(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$, where $a_0, a_1, \ldots, a_{n-1}, a_n$ are real numbers. Then $f(x)$ is $O(x^n)$.

Proof: Using the triangle inequality (see Exercise 9 in Section 1.8), if $x > 1$ we have

$$|f(x)| = |a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0|$$

$$\le |a_n|x^n + |a_{n-1}|x^{n-1} + \cdots + |a_1|x + |a_0|$$

$$= x^n \left(|a_n| + |a_{n-1}|/x + \cdots + |a_1|/x^{n-1} + |a_0|/x^n\right)$$

$$\le x^n \left(|a_n| + |a_{n-1}| + \cdots + |a_1| + |a_0|\right).$$

This shows that

$$|f(x)| \leq Cx^n,$$

where $C = |a_n| + |a_{n-1}| + \cdots + |a_0|$ whenever $x > 1$. Hence, the witnesses $C = |a_n| + |a_{n-1}| + \cdots + |a_0|$ and $k = 1$ show that $f(x)$ is $O(x^n)$. ◄

We now give some examples involving functions that have the set of positive integers as their domains.

EXAMPLE 5 How can big-O notation be used to estimate the sum of the first n positive integers?

Solution: Because each of the integers in the sum of the first n positive integers does not exceed n, it follows that

$$1 + 2 + \cdots + n \leq n + n + \cdots + n = n^2.$$

From this inequality it follows that $1 + 2 + 3 + \cdots + n$ is $O(n^2)$, taking $C = 1$ and $k = 1$ as witnesses. (In this example the domains of the functions in the big-O relationship are the set of positive integers.) ◄

In Example 6 big-O estimates will be developed for the factorial function and its logarithm. These estimates will be important in the analysis of the number of steps used in sorting procedures.

EXAMPLE 6 Give big-O estimates for the factorial function and the logarithm of the factorial function, where the factorial function $f(n) = n!$ is defined by

$$n! = 1 \cdot 2 \cdot 3 \cdot \cdots \cdot n$$

whenever n is a positive integer, and $0! = 1$. For example,

$$1! = 1, \quad 2! = 1 \cdot 2 = 2, \quad 3! = 1 \cdot 2 \cdot 3 = 6, \quad 4! = 1 \cdot 2 \cdot 3 \cdot 4 = 24.$$

Note that the function $n!$ grows rapidly. For instance,

$$20! = 2{,}432{,}902{,}008{,}176{,}640{,}000.$$

Solution: A big-O estimate for $n!$ can be obtained by noting that each term in the product does not exceed n. Hence,

$$n! = 1 \cdot 2 \cdot 3 \cdot \cdots \cdot n$$
$$\leq n \cdot n \cdot n \cdot \cdots \cdot n$$
$$= n^n.$$

This inequality shows that $n!$ is $O(n^n)$, taking $C = 1$ and $k = 1$ as witnesses. Taking logarithms of both sides of the inequality established for $n!$, we obtain

$$\log n! \leq \log n^n = n \log n.$$

This implies that $\log n!$ is $O(n \log n)$, again taking $C = 1$ and $k = 1$ as witnesses. ◄

EXAMPLE 7   In Section 5.1 , we will show that $n < 2^n$ whenever n is a positive integer. Show that this in-equality implies that n is $O(2^n)$, and use this inequality to show that $\log n$ is $O(n)$.

Solution: Using the inequality $n < 2^n$, we quickly can conclude that n is $O(2^n)$ by taking $k = C = 1$ as witnesses. Note that because the logarithm function is increasing, taking logarithms (base 2) of both sides of this inequality shows that

$$\log n < n.$$

It follows that

$$\log n \text{ is } O(n).$$

(Again we take $C = k = 1$ as witnesses.)

If we have logarithms to a base b, where b is different from 2, we still have $\log_b n$ is $O(n)$ because

$$\log_b n = \frac{\log n}{\log b} < \frac{n}{\log b}$$

whenever n is a positive integer. We take $C = 1/\log b$ and $k = 1$ as witnesses. (We have used Theorem 3 in Appendix 2 to see that $\log_b n = \log n / \log b$.)   ◄

As mentioned before, big-O notation is used to estimate the number of operations needed to solve a problem using a specified procedure or algorithm. The functions used in these estimates often include the following:

$$1, \ \log n, \ n, \ n \log n, \ n^2, \ 2^n, \ n!$$

Using calculus it can be shown that each function in the list is smaller than the succeeding function, in the sense that the ratio of a function and the succeeding function tends to zero as n grows without bound. Figure 3 displays the graphs of these functions, using a scale for the values of the functions that doubles for each successive marking on the graph. That is, the vertical scale in this graph is logarithmic.
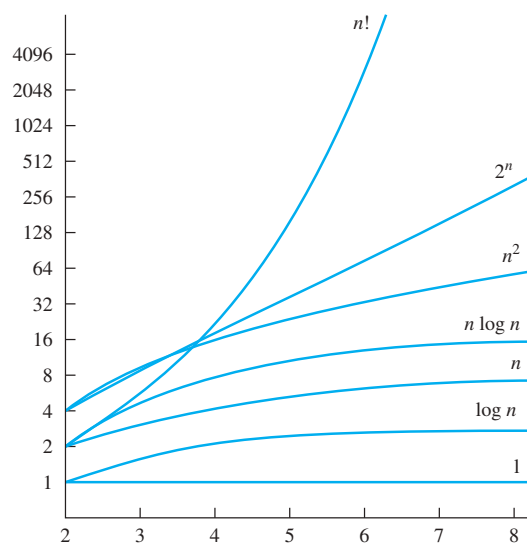

FIGURE 3   **A display of the growth of functions commonly used in big-O estimates.**

USEFUL BIG-O ESTIMATES INVOLVING LOGARITHMS, POWERS, AND EXPONEN-TIAL FUNCTIONS We now give some useful facts that help us determine whether big-O relationships hold between pairs of functions when each of the functions is a power of a logarithm, a power, or an exponential function of the form b^n where $b > 1$. Their proofs are left as Exercises 57–62 for readers skilled with calculus.

Theorem 1 shows that if $f(n)$ is a polynomial of degree d or less, then $f(n)$ is $O(n^d)$. Applying this theorem, we see that if $d > c > 1$, then n^c is $O(n^d)$. We leave it to the reader to show that the reverse of this relationship does not hold. Putting these facts together, we see that if $d > c > 1$, then

$$n^c \text{ is } O(n^d), \text{ but } n^d \text{ is not } O(n^c).$$

In Example 7 we showed that $\log_b n$ is $O(n)$ whenever $b > 1$. More generally, whenever $b > 1$ and c and d are positive, we have

$$(\log_b n)^c \text{ is } O(n^d), \text{ but } n^d \text{ is not } (O(\log_b n)^c).$$

This tells us that every positive power of the logarithm of n to the base b, where $b > 1$, is big-O of every positive power of n, but the reverse relationship never holds.

In Example 7, we also showed that n is $O(2^n)$. More generally, whenever d is positive and $b > 1$, we have

$$n^d \text{ is } O(b^n), \text{ but } b^n \text{ is not } O(n^d).$$

This tells us that every power of n is big-O of every exponential function of n with a base that is greater than one, but the reverse relationship never holds. Furthermore, when $c > b > 1$ we have

$$b^n \text{ is } O(c^n), \text{ but } c^n \text{ is not } O(b^n).$$

This tells us that if we have two exponential functions with different bases greater than one, one of these functions is big-O of the other if and only if its base is smaller or equal.

Finally, we note that if $c > 1$, we have

$$c^n \text{ is } O(n!), \text{ but } n! \text{ is not } O(c^n).$$

We can use the big-O estimates discussed here to help us order the growth of different functions, as Example 8 illustrates.

EXAMPLE 8 Arrange the functions $f_1(n) = 8\sqrt{n}, f_2(n) = (\log n)^2, f_3(n) = 2n \log n, f_4(n) = n!, f_5(n) = (1.1)^n$, and $f_6(n) = n^2$ in a list so that each function is big-O of the next function.

Solution: From the big-O estimates described in this subsection, we see that $f_2(n) = (\log n)^2$ is the slowest growing of these functions. (This follows because $\log n$ grows slower than any positive power of n.) The next three functions, in order, are $f_1(n) = 8\sqrt{n} = f_3(n) = 2n \log n$, and $f_6(n) = n^2$. (We know this because $f_1(n) = 8n^{1/2}, f_3(n) = 2n \log n$ is a function that grows faster than n but slower than n^c for every $c > 1$, and $f_6(n) = n^2$ is of the form n^c where $c = 2$.) The next function in the list is $f_5(n) = (1.1)^n$, because it is an exponential function with base 1.1. Finally, $f_4(n) = n!$ is the fastest growing function on the list, because $f(n) = n!$ grows faster than any exponential function of n. ◀

3.2.4 The Growth of Combinations of Functions

Many algorithms are made up of two or more separate subprocedures. The number of steps used by a computer to solve a problem with input of a specified size using such an algorithm is the sum of the number of steps used by these subprocedures. To give a big-O estimate for the number of steps needed, it is necessary to find big-O estimates for the number of steps used by each subprocedure and then combine these estimates.

Big-O estimates of combinations of functions can be provided if care is taken when different big-O estimates are combined. In particular, it is often necessary to estimate the growth of the sum and the product of two functions. What can be said if big-O estimates for each of two functions are known? To see what sort of estimates hold for the sum and the product of two functions, suppose that $f_1(x)$ is $O(g_1(x))$ and $f_2(x)$ is $O(g_2(x))$.

From the definition of big-O notation, there are constants C_1, C_2, k_1, and k_2 such that

$$|f_1(x)| \le C_1|g_1(x)|$$

when $x > k_1$, and

$$|f_2(x)| \le C_2|g_2(x)|$$

when $x > k_2$. To estimate the sum of $f_1(x)$ and $f_2(x)$, note that

$$|(f_1 + f_2)(x)| = |f_1(x) + f_2(x)|$$
$$\le |f_1(x)| + |f_2(x)| \quad \text{using the triangle inequality } |a + b| \le |a| + |b|.$$

When x is greater than both k_1 and k_2, it follows from the inequalities for $|f_1(x)|$ and $|f_2(x)|$ that

$$|f_1(x)| + |f_2(x)| \le C_1|g_1(x)| + C_2|g_2(x)|$$
$$\le C_1|g(x)| + C_2|g(x)|$$
$$= (C_1 + C_2)|g(x)|$$
$$= C|g(x)|,$$

where $C = C_1 + C_2$ and $g(x) = \max(|g_1(x)|, |g_2(x)|)$. [Here $\max(a, b)$ denotes the maximum, or larger, of a and b.]

This inequality shows that $|(f_1 + f_2)(x)| \le C|g(x)|$ whenever $x > k$, where $k = \max(k_1, k_2)$. We state this useful result as Theorem 2.

THEOREM 2   Suppose that $f_1(x)$ is $O(g_1(x))$ and that $f_2(x)$ is $O(g_2(x))$. Then $(f_1 + f_2)(x)$ is $O(g(x))$, where $g(x) = (\max(|g_1(x)|, |g_2(x)|)$ for all x.

We often have big-O estimates for f_1 and f_2 in terms of the same function g. In this situation, Theorem 2 can be used to show that $(f_1 + f_2)(x)$ is also $O(g(x))$, because $\max(g(x), g(x)) = g(x)$. This result is stated in Corollary 1.

COROLLARY 1   Suppose that $f_1(x)$ and $f_2(x)$ are both $O(g(x))$. Then $(f_1 + f_2)(x)$ is $O(g(x))$.

In a similar way big-O estimates can be derived for the product of the functions f_1 and f_2. When x is greater than $\max(k_1, k_2)$ it follows that

$$|(f_1 f_2)(x)| = |f_1(x)||f_2(x)|$$
$$\leq C_1|g_1(x)|C_2|g_2(x)|$$
$$\leq C_1 C_2 |(g_1 g_2)(x)|$$
$$\leq C|(g_1 g_2)(x)|,$$

where $C = C_1 C_2$. From this inequality, it follows that $f_1(x)f_2(x)$ is $O(g_1 g_2(x))$, because there are constants C and k, namely, $C = C_1 C_2$ and $k = \max(k_1, k_2)$, such that $|(f_1 f_2)(x)| \leq C|g_1(x)g_2(x)|$ whenever $x > k$. This result is stated in Theorem 3.

THEOREM 3   Suppose that $f_1(x)$ is $O(g_1(x))$ and $f_2(x)$ is $O(g_2(x))$. Then $(f_1 f_2)(x)$ is $O(g_1(x)g_2(x))$.

The goal in using big-O notation to estimate functions is to choose a function $g(x)$ as simple as possible, that grows relatively slowly so that $f(x)$ is $O(g(x))$. Examples 9 and 10 illustrate how to use Theorems 2 and 3 to do this. The type of analysis given in these examples is often used in the analysis of the time used to solve problems using computer programs.

EXAMPLE 9   Give a big-O estimate for $f(n) = 3n \log(n!) + (n^2 + 3) \log n$, where n is a positive integer.

Solution: First, the product $3n \log(n!)$ will be estimated. From Example 6 we know that $\log(n!)$ is $O(n \log n)$. Using this estimate and the fact that $3n$ is $O(n)$, Theorem 3 gives the estimate that $3n \log(n!)$ is $O(n^2 \log n)$.

Next, the product $(n^2 + 3) \log n$ will be estimated. Because $(n^2 + 3) < 2n^2$ when $n > 2$, it follows that $n^2 + 3$ is $O(n^2)$. Thus, from Theorem 3 it follows that $(n^2 + 3) \log n$ is $O(n^2 \log n)$. Using Theorem 2 to combine the two big-O estimates for the products shows that $f(n) = 3n \log(n!) + (n^2 + 3) \log n$ is $O(n^2 \log n)$. ◀

EXAMPLE 10   Give a big-O estimate for $f(x) = (x + 1) \log(x^2 + 1) + 3x^2$.

Solution: First, a big-O estimate for $(x + 1) \log(x^2 + 1)$ will be found. Note that $(x + 1)$ is $O(x)$. Furthermore, $x^2 + 1 \leq 2x^2$ when $x > 1$. Hence,

$$\log(x^2 + 1) \leq \log(2x^2) = \log 2 + \log x^2 = \log 2 + 2 \log x \leq 3 \log x,$$

if $x > 2$. This shows that $\log(x^2 + 1)$ is $O(\log x)$.

From Theorem 3 it follows that $(x + 1) \log(x^2 + 1)$ is $O(x \log x)$. Because $3x^2$ is $O(x^2)$, Theorem 2 tells us that $f(x)$ is $O(\max(x \log x, x^2))$. Because $x \log x \leq x^2$, for $x > 1$, it follows that $f(x)$ is $O(x^2)$. ◀

3.2.5   Big-Omega and Big-Theta Notation

Big-O notation is used extensively to describe the growth of functions, but it has limitations. In particular, when $f(x)$ is $O(g(x))$, we have an upper bound, in terms of $g(x)$, for the size of $f(x)$ for large values of x. However, big-O notation does not provide a lower bound for the size of $f(x)$ for large x. For this, we use **big-Omega (big-Ω) notation**. When we want to give both an upper and a lower bound on the size of a function $f(x)$, relative to a reference function $g(x)$, we use **big-Theta (big-Θ) notation**. Both big-Omega and big-Theta notation were introduced

Ω and Θ are the Greek uppercase letters omega and theta, respectively.

by Donald Knuth in the 1970s. His motivation for introducing these notations was the common misuse of big-O notation when both an upper and a lower bound on the size of a function are needed. We now define big-Omega notation and illustrate its use. After doing so, we will do the same for big-Theta notation.

Definition 2

Let f and g be functions from the set of integers or the set of real numbers to the set of real numbers. We say that $f(x)$ is $\Omega(g(x))$ if there are constants C and k with C positive such that

$$|f(x)| \geq C|g(x)|$$

whenever $x > k$. [This is read as "$f(x)$ is big-Omega of $g(x)$."]

There is a strong connection between big-O and big-Omega notation. In particular, $f(x)$ is $\Omega(g(x))$ if and only if $g(x)$ is $O(f(x))$. We leave the verification of this fact as a straightforward exercise for the reader.

EXAMPLE 11 The function $f(x) = 8x^3 + 5x^2 + 7$ is $\Omega(g(x))$, where $g(x)$ is the function $g(x) = x^3$. This is easy to see because $f(x) = 8x^3 + 5x^2 + 7 \geq 8x^3$ for all positive real numbers x. This is equivalent to saying that $g(x) = x^3$ is $O(8x^3 + 5x^2 + 7)$, which can be established directly by turning the inequality around. ◄

Often, it is important to know the order of growth of a function in terms of some relatively simple reference function such as x^n when n is a positive integer or c^x, where $c > 1$. Knowing the order of growth requires that we have both an upper bound and a lower bound for the size of the function. That is, given a function $f(x)$, we want a reference function $g(x)$ such that $f(x)$ is $O(g(x))$ and $f(x)$ is $\Omega(g(x))$. Big-Theta notation, defined as follows, is used to express both of these relationships, providing both an upper and a lower bound on the size of a function.

Definition 3

Let f and g be functions from the set of integers or the set of real numbers to the set of real numbers. We say that $f(x)$ is $\Theta(g(x))$ if $f(x)$ is $O(g(x))$ and $f(x)$ is $\Omega(g(x))$. When $f(x)$ is $\Theta(g(x))$, we say that f is big-Theta of $g(x)$, that $f(x)$ is of *order* $g(x)$, and that $f(x)$ and $g(x)$ are of the *same order*.

When $f(x)$ is $\Theta(g(x))$, it is also the case that $g(x)$ is $\Theta(f(x))$. Also note that $f(x)$ is $\Theta(g(x))$ if and only if $f(x)$ is $O(g(x))$ and $g(x)$ is $O(f(x))$ (see Exercise 31). Furthermore, note that $f(x)$ is $\Theta(g(x))$ if and only if there are positive real numbers C_1 and C_2 and a positive real number k such that

$$C_1|g(x)| \leq |f(x)| \leq C_2|g(x)|$$

whenever $x > k$. The existence of the constants C_1, C_2, and k tells us that $f(x)$ is $\Omega(g(x))$ and that $f(x)$ is $O(g(x))$, respectively.

Usually, when big-Theta notation is used, the function $g(x)$ in $\Theta(g(x))$ is a relatively simple reference function, such as x^n, c^x, $\log x$, and so on, while $f(x)$ can be relatively complicated.

EXAMPLE 12 We showed (in Example 5) that the sum of the first n positive integers is $O(n^2)$. Determine whether this sum is of order n^2 without using the summation formula for this sum.

Extra Examples ❯

Solution: Let $f(n) = 1 + 2 + 3 + \cdots + n$. Because we already know that $f(n)$ is $O(n^2)$, to show that $f(n)$ is of order n^2 we need to find a positive constant C such that $f(n) > Cn^2$ for sufficiently

large integers n. To obtain a lower bound for this sum, we can ignore the first half of the terms. Summing only the terms greater than $\lceil n/2 \rceil$, we find that

$$
\begin{aligned}
1 + 2 + \cdots + n &\geq \lceil n/2 \rceil + (\lceil n/2 \rceil + 1) + \cdots + n \\
&\geq \lceil n/2 \rceil + \lceil n/2 \rceil + \cdots + \lceil n/2 \rceil \\
&= (n - \lceil n/2 \rceil + 1) \lceil n/2 \rceil \\
&\geq (n/2)(n/2) \\
&= n^2/4.
\end{aligned}
$$

This shows that $f(n)$ is $\Omega(n^2)$. We conclude that $f(n)$ is of order n^2, or in symbols, $f(n)$ is $\Theta(n^2)$. ◄

Remark: Note that we can also show that $f(n) = \sum_{i=1}^{n} i$ is $\Theta(n^2)$ using the closed formula $\sum_{i=1}^{n} = n(n+1)/2$ from Table 2 in Section 2.4 and derived in Exercise 37(b) of that section.

EXAMPLE 13 Show that $3x^2 + 8x \log x$ is $\Theta(x^2)$.

Extra Examples **▶** *Solution:* Because $0 \leq 8x \log x \leq 8x^2$, it follows that $3x^2 + 8x \log x \leq 11x^2$ for $x > 1$. Consequently, $3x^2 + 8x \log x$ is $O(x^2)$. Clearly, x^2 is $O(3x^2 + 8x \log x)$. Consequently, $3x^2 + 8x \log x$ is $\Theta(x^2)$. ◄

One useful fact is that the leading term of a polynomial determines its order. For example, if $f(x) = 3x^5 + x^4 + 17x^3 + 2$, then $f(x)$ is of order x^5. This is stated in Theorem 4, whose proof is left as Exercise 50.

THEOREM 4   Let $f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$, where $a_0, a_1, \ldots, a_n$ are real numbers with $a_n \neq 0$. Then $f(x)$ is of order x^n.

EXAMPLE 14 The polynomials $3x^8 + 10x^7 + 221x^2 + 1444$, $x^{19} - 18x^4 - 10,112$, and $-x^{99} + 40,001x^{98} + 100,003x$ are of orders x^8, x^{19}, and x^{99}, respectively. ◄

Unfortunately, as Knuth observed, big-O notation is often used by careless writers and speakers as if it had the same meaning as big-Theta notation. Keep this in mind when you see big-O notation used. The recent trend has been to use big-Theta notation whenever both upper and lower bounds on the size of a function are needed.

Exercises

In Exercises 1–14, to establish a big-O relationship, find witnesses C and k such that $|f(x)| \leq C|g(x)|$ whenever $x > k$.

1. Determine whether each of these functions is $O(x)$.
 a) $f(x) = 10$ **b)** $f(x) = 3x + 7$
 c) $f(x) = x^2 + x + 1$ **d)** $f(x) = 5 \log x$
 e) $f(x) = \lfloor x \rfloor$ **f)** $f(x) = \lceil x/2 \rceil$

2. Determine whether each of these functions is $O(x^2)$.
 a) $f(x) = 17x + 11$ **b)** $f(x) = x^2 + 1000$
 c) $f(x) = x \log x$ **d)** $f(x) = x^4/2$
 e) $f(x) = 2^x$ **f)** $f(x) = \lfloor x \rfloor \cdot \lceil x \rceil$

3. Use the definition of "$f(x)$ is $O(g(x))$" to show that $x^4 + 9x^3 + 4x + 7$ is $O(x^4)$.

4. Use the definition of "$f(x)$ is $O(g(x))$" to show that $2^x + 17$ is $O(3^x)$.

5. Show that $(x^2 + 1)/(x + 1)$ is $O(x)$.

6. Show that $(x^3 + 2x)/(2x + 1)$ is $O(x^2)$.

7. Find the least integer n such that $f(x)$ is $O(x^n)$ for each of these functions.
   a) $f(x) = 2x^3 + x^2 \log x$
   b) $f(x) = 3x^3 + (\log x)^4$
   c) $f(x) = (x^4 + x^2 + 1)/(x^3 + 1)$
   d) $f(x) = (x^4 + 5 \log x)/(x^4 + 1)$

8. Find the least integer n such that $f(x)$ is $O(x^n)$ for each of these functions.
   a) $f(x) = 2x^2 + x^3 \log x$
   b) $f(x) = 3x^5 + (\log x)^4$
   c) $f(x) = (x^4 + x^2 + 1)/(x^4 + 1)$
   d) $f(x) = (x^3 + 5 \log x)/(x^4 + 1)$

9. Show that $x^2 + 4x + 17$ is $O(x^3)$ but that x^3 is not $O(x^2 + 4x + 17)$.

10. Show that x^3 is $O(x^4)$ but that x^4 is not $O(x^3)$.

11. Show that $3x^4 + 1$ is $O(x^4/2)$ and $x^4/2$ is $O(3x^4 + 1)$.

12. Show that $x \log x$ is $O(x^2)$ but that x^2 is not $O(x \log x)$.

13. Show that 2^n is $O(3^n)$ but that 3^n is not $O(2^n)$. (Note that this is a special case of Exercise 60.)

14. Determine whether x^3 is $O(g(x))$ for each of these functions $g(x)$.
   a) $g(x) = x^2$ **b)** $g(x) = x^3$
   c) $g(x) = x^2 + x^3$ **d)** $g(x) = x^2 + x^4$
   e) $g(x) = 3^x$ **f)** $g(x) = x^3/2$

15. Explain what it means for a function to be $O(1)$.

16. Show that if $f(x)$ is $O(x)$, then $f(x)$ is $O(x^2)$.

17. Suppose that $f(x)$, $g(x)$, and $h(x)$ are functions such that $f(x)$ is $O(g(x))$ and $g(x)$ is $O(h(x))$. Show that $f(x)$ is $O(h(x))$.

18. Let k be a positive integer. Show that $1^k + 2^k + \cdots + n^k$ is $O(n^{k+1})$.

19. Determine whether each of the functions 2^{n+1} and 2^{2n} is $O(2^n)$.

20. Determine whether each of the functions $\log(n + 1)$ and $\log(n^2 + 1)$ is $O(\log n)$.

21. Arrange the functions $\sqrt{n}$, $1000 \log n$, $n \log n$, $2n!$, 2^n, 3^n, and $n^2/1{,}000{,}000$ in a list so that each function is big-O of the next function.

22. Arrange the functions $(1.5)^n$, n^{100}, $(\log n)^3$, $\sqrt{n} \log n$, 10^n, $(n!)^2$, and $n^{99} + n^{98}$ in a list so that each function is big-O of the next function.

23. Suppose that you have two different algorithms for solving a problem. To solve a problem of size n, the first algorithm uses exactly $n(\log n)$ operations and the second algorithm uses exactly $n^{3/2}$ operations. As n grows, which algorithm uses fewer operations?

24. Suppose that you have two different algorithms for solving a problem. To solve a problem of size n, the first algorithm uses exactly $n^2 2^n$ operations and the second algorithm uses exactly $n!$ operations. As n grows, which algorithm uses fewer operations?

25. Give as good a big-O estimate as possible for each of these functions.
   a) $(n^2 + 8)(n + 1)$ **b)** $(n \log n + n^2)(n^3 + 2)$
   c) $(n! + 2^n)(n^3 + \log(n^2 + 1))$

26. Give a big-O estimate for each of these functions. For the function g in your estimate $f(x)$ is $O(g(x))$, use a simple function g of smallest order.
   a) $(n^3 + n^2 \log n)(\log n + 1) + (17 \log n + 19)(n^3 + 2)$
   b) $(2^n + n^2)(n^3 + 3^n)$
   c) $(n^n + n2^n + 5^n)(n! + 5^n)$

27. Give a big-O estimate for each of these functions. For the function g in your estimate that $f(x)$ is $O(g(x))$, use a simple function g of the smallest order.
   a) $n \log(n^2 + 1) + n^2 \log n$
   b) $(n \log n + 1)^2 + (\log n + 1)(n^2 + 1)$
   c) $n^{2^n} + n^{n^2}$

28. For each function in Exercise 1, determine whether that function is $\Omega(x)$ and whether it is $\Theta(x)$.

29. For each function in Exercise 2, determine whether that function is $\Omega(x^2)$ and whether it is $\Theta(x^2)$.

30. Show that each of these pairs of functions are of the same order.
   a) $3x + 7$, x
   b) $2x^2 + x - 7$, x^2
   c) $\lfloor x + 1/2 \rfloor$, x
   d) $\log(x^2 + 1)$, $\log_2 x$
   e) $\log_{10} x$, $\log_2 x$

31. Show that $f(x)$ is $\Theta(g(x))$ if and only if $f(x)$ is $O(g(x))$ and $g(x)$ is $O(f(x))$.

32. Show that if $f(x)$ and $g(x)$ are functions from the set of real numbers to the set of real numbers, then $f(x)$ is $O(g(x))$ if and only if $g(x)$ is $\Omega(f(x))$.

33. Show that if $f(x)$ and $g(x)$ are functions from the set of real numbers to the set of real numbers, then $f(x)$ is $\Theta(g(x))$ if and only if there are positive constants k, C_1, and C_2 such that $C_1|g(x)| \leq |f(x)| \leq C_2|g(x)|$ whenever $x > k$.

34. a) Show that $3x^2 + x + 1$ is $\Theta(3x^2)$ by directly finding the constants k, C_1, and C_2 in Exercise 33.
   b) Express the relationship in part (a) using a picture showing the functions $3x^2 + x + 1$, $C_1 \cdot 3x^2$, and $C_2 \cdot 3x^2$, and the constant k on the x-axis, where C_1, C_2, and k are the constants you found in part (a) to show that $3x^2 + x + 1$ is $\Theta(3x^2)$.

35. Express the relationship $f(x)$ is $\Theta(g(x))$ using a picture. Show the graphs of the functions $f(x)$, $C_1|g(x)|$, and $C_2|g(x)|$, as well as the constant k on the x-axis.

36. Explain what it means for a function to be $\Omega(1)$.

37. Explain what it means for a function to be $\Theta(1)$.

38. Give a big-O estimate of the product of the first n odd positive integers.

39. Show that if f and g are real-valued functions such that $f(x)$ is $O(g(x))$, then for every positive integer n, $f^n(x)$ is $O(g^n(x))$. [Note that $f^n(x) = f(x)^n$.]

40. Show that for all real numbers a and b with $a > 1$ and $b > 1$, if $f(x)$ is $O(\log_b x)$, then $f(x)$ is $O(\log_a x)$.

41. Suppose that $f(x)$ is $O(g(x))$, where f and g are increasing and unbounded functions. Show that $\log |f(x)|$ is $O(\log |g(x)|)$.

42. Suppose that $f(x)$ is $O(g(x))$. Does it follow that $2^{f(x)}$ is $O(2^{g(x)})$?

43. Let $f_1(x)$ and $f_2(x)$ be functions from the set of real numbers to the set of positive real numbers. Show that if $f_1(x)$ and $f_2(x)$ are both $\Theta(g(x))$, where $g(x)$ is a function from the set of real numbers to the set of positive real numbers, then $f_1(x) + f_2(x)$ is $\Theta(g(x))$. Is this still true if $f_1(x)$ and $f_2(x)$ can take negative values?

44. Suppose that $f(x)$, $g(x)$, and $h(x)$ are functions such that $f(x)$ is $\Theta(g(x))$ and $g(x)$ is $\Theta(h(x))$. Show that $f(x)$ is $\Theta(h(x))$.

45. If $f_1(x)$ and $f_2(x)$ are functions from the set of positive integers to the set of positive real numbers and $f_1(x)$ and $f_2(x)$ are both $\Theta(g(x))$, is $(f_1 - f_2)(x)$ also $\Theta(g(x))$? Either prove that it is or give a counterexample.

46. Show that if $f_1(x)$ and $f_2(x)$ are functions from the set of positive integers to the set of real numbers and $f_1(x)$ is $\Theta(g_1(x))$ and $f_2(x)$ is $\Theta(g_2(x))$, then $(f_1 f_2)(x)$ is $\Theta((g_1 g_2)(x))$.

47. Find functions f and g from the set of positive integers to the set of real numbers such that $f(n)$ is not $O(g(n))$ and $g(n)$ is not $O(f(n))$.

48. Express the relationship $f(x)$ is $\Omega(g(x))$ using a picture. Show the graphs of the functions $f(x)$ and $Cg(x)$, as well as the constant k on the real axis.

49. Show that if $f_1(x)$ is $\Theta(g_1(x))$, $f_2(x)$ is $\Theta(g_2(x))$, and $f_2(x) \neq 0$ and $g_2(x) \neq 0$ for all real numbers $x > 0$, then $(f_1/f_2)(x)$ is $\Theta((g_1/g_2)(x))$.

50. Show that if $f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$, where $a_0, a_1, \ldots, a_{n-1}$, and a_n are real numbers and $a_n \neq 0$, then $f(x)$ is $\Theta(x^n)$.

Big-O, big-Theta, and big-Omega notation can be extended to functions in more than one variable. For example, the statement $f(x, y)$ is $O(g(x, y))$ means that there exist constants C, k_1, and k_2 such that $|f(x, y)| \leq C|g(x, y)|$ whenever $x > k_1$ and $y > k_2$.

51. Define the statement $f(x, y)$ is $\Theta(g(x, y))$.

52. Define the statement $f(x, y)$ is $\Omega(g(x, y))$.

53. Show that $(x^2 + xy + x \log y)^3$ is $O(x^6 y^3)$.

54. Show that $x^5 y^3 + x^4 y^4 + x^3 y^5$ is $\Omega(x^3 y^3)$.

55. Show that $\lfloor xy \rfloor$ is $O(xy)$.

56. Show that $\lceil xy \rceil$ is $\Omega(xy)$.

57. (*Requires calculus*) Show that if $c > d > 0$, then n^d is $O(n^c)$, but n^c is not $O(n^d)$.

58. (*Requires calculus*) Show that if $b > 1$ and c and d are positive, then $(\log_b n)^c$ is $O(n^d)$, but n^d is not $O((\log_b n)^c)$.

59. (*Requires calculus*) Show that if d is positive and $b > 1$, then n^d is $O(b^n)$, but b^n is not $O(n^d)$.

60. (*Requires calculus*) Show that if $c > b > 1$, then b^n is $O(c^n)$, but c^n is not $O(b^n)$.

61. (*Requires calculus*) Show that if $c > 1$, then c^n is $O(n!)$, but $n!$ is not $O(c^n)$.

62. (*Requires calculus*) Prove or disprove that $(2n)!$ is $O(n!)$.

The following problems deal with another type of asymptotic notation, called **little-o** notation. Because little-o notation is based on the concept of limits, a knowledge of calculus is needed for these problems. We say that $f(x)$ is $o(g(x))$ [read $f(x)$ is "little-oh" of $g(x)$], when

$$\lim_{x \to \infty} \frac{f(x)}{g(x)} = 0.$$

63. (*Requires calculus*) Show that
a) x^2 is $o(x^3)$. **b)** $x \log x$ is $o(x^2)$.
c) x^2 is $o(2^x)$. **d)** $x^2 + x + 1$ is not $o(x^2)$.

64. (*Requires calculus*)
a) Show that if $f(x)$ and $g(x)$ are functions such that $f(x)$ is $o(g(x))$ and c is a constant, then $cf(x)$ is $o(g(x))$, where $(cf)(x) = cf(x)$.
b) Show that if $f_1(x)$, $f_2(x)$, and $g(x)$ are functions such that $f_1(x)$ is $o(g(x))$ and $f_2(x)$ is $o(g(x))$, then $(f_1 + f_2)(x)$ is $o(g(x))$, where $(f_1 + f_2)(x) = f_1(x) + f_2(x)$.

65. (*Requires calculus*) Represent pictorially that $x \log x$ is $o(x^2)$ by graphing $x \log x$, x^2, and $x \log x / x^2$. Explain how this picture shows that $x \log x$ is $o(x^2)$.

66. (*Requires calculus*) Express the relationship $f(x)$ is $o(g(x))$ using a picture. Show the graphs of $f(x)$, $g(x)$, and $f(x)/g(x)$.

*67. (*Requires calculus*) Suppose that $f(x)$ is $o(g(x))$. Does it follow that $2^{f(x)}$ is $o(2^{g(x)})$?

*68. (*Requires calculus*) Suppose that $f(x)$ is $o(g(x))$. Does it follow that $\log |f(x)|$ is $o(\log |g(x)|)$?

69. (*Requires calculus*) The two parts of this exercise describe the relationship between little-o and big-O notation.
a) Show that if $f(x)$ and $g(x)$ are functions such that $f(x)$ is $o(g(x))$, then $f(x)$ is $O(g(x))$.
b) Show that if $f(x)$ and $g(x)$ are functions such that $f(x)$ is $O(g(x))$, then it does not necessarily follow that $f(x)$ is $o(g(x))$.

70. (*Requires calculus*) Show that if $f(x)$ is a polynomial of degree n and $g(x)$ is a polynomial of degree m where $m > n$, then $f(x)$ is $o(g(x))$.

71. (*Requires calculus*) Show that if $f_1(x)$ is $O(g(x))$ and $f_2(x)$ is $o(g(x))$, then $f_1(x) + f_2(x)$ is $O(g(x))$.

72. (*Requires calculus*) Let H_n be the nth **harmonic number**

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}.$$

Show that H_n is $O(\log n)$. [*Hint:* First establish the inequality

$$\sum_{j=2}^{n} \frac{1}{j} < \int_1^n \frac{1}{x} dx$$

by showing that the sum of the areas of the rectangles of height $1/j$ with base from $j-1$ to j, for $j = 2, 3, \ldots, n$, is less than the area under the curve $y = 1/x$ from 2 to n.]

*73. Show that $n \log n$ is $O(\log n!)$.

☞ 74. Determine whether $\log n!$ is $\Theta(n \log n)$. Justify your answer.

*75. Show that $\log n!$ is greater than $(n \log n)/4$ for $n > 4$. [*Hint:* Begin with the inequality $n! > n(n-1)(n-2) \cdots \lceil n/2 \rceil$.]

Let $f(x)$ and $g(x)$ be functions from the set of real numbers to the set of real numbers. We say that the functions f and g are **asymptotic** and write $f(x) \sim g(x)$ if $\lim_{x \to \infty} f(x)/g(x) = 1$.

76. (*Requires calculus*) For each of these pairs of functions, determine whether f and g are asymptotic.
  a) $f(x) = x^2 + 3x + 7$, $g(x) = x^2 + 10$
  b) $f(x) = x^2 \log x$, $g(x) = x^3$
  c) $f(x) = x^4 + \log(3x^8 + 7)$,
      $g(x) = (x^2 + 17x + 3)^2$
  d) $f(x) = (x^3 + x^2 + x + 1)^4$,
      $g(x) = (x^4 + x^3 + x^2 + x + 1)^3$.

77. (*Requires calculus*) For each of these pairs of functions, determine whether f and g are asymptotic.
  a) $f(x) = \log(x^2 + 1)$, $g(x) = \log x$
  b) $f(x) = 2^{x+3}$, $g(x) = 2^{x+7}$
  c) $f(x) = 2^{2^x}$, $g(x) = 2^{x^2}$
  d) $f(x) = 2^{x^2+x+1}$, $g(x) = 2^{x^2+2x}$

3.3   Complexity of Algorithms

3.3.1   Introduction

When does an algorithm provide a satisfactory solution to a problem? First, it must always produce the correct answer. How this can be demonstrated will be discussed in Chapter 5. Second, it should be efficient. The efficiency of algorithms will be discussed in this section.

How can the efficiency of an algorithm be analyzed? One measure of efficiency is the time used by a computer to solve a problem using the algorithm, when input values are of a specified size. A second measure is the amount of computer memory required to implement the algorithm when input values are of a specified size.

Questions such as these involve the **computational complexity** of the algorithm. An analysis of the time required to solve a problem of a particular size involves the **time complexity** of the algorithm. An analysis of the computer memory required involves the **space complexity** of the algorithm. Considerations of the time and space complexity of an algorithm are essential when algorithms are implemented. It is important to know whether an algorithm will produce an answer in a microsecond, a minute, or a billion years. Likewise, the required memory must be available to solve a problem, so that space complexity must be taken into account.

Considerations of space complexity are tied in with the particular data structures used to implement the algorithm. Because data structures are not dealt with in detail in this book, space complexity will not be considered. We will restrict our attention to time complexity.

3.3.2   Time Complexity

The time complexity of an algorithm can be expressed in terms of the number of operations used by the algorithm when the input has a particular size. The operations used to measure time complexity can be the comparison of integers, the addition of integers, the multiplication of integers, the division of integers, or any other basic operation.

Time complexity is described in terms of the number of operations required instead of actual computer time because of the difference in time needed for different computers to perform basic operations. Moreover, it is quite complicated to break all operations down to the basic bit operations that a computer uses. Furthermore, the fastest computers in existence can perform basic bit operations (for instance, adding, multiplying, comparing, or exchanging two bits) in 10^{-11} second (10 picoseconds), but personal computers may require 10^{-8} second (10 nanoseconds), which is 1000 times as long, to do the same operations.

We illustrate how to analyze the time complexity of an algorithm by considering Algorithm 1 of Section 3.1, which finds the maximum of a finite set of integers.

EXAMPLE 1 Describe the time complexity of Algorithm 1 of Section 3.1 for finding the maximum element in a finite set of integers.

Extra Examples ▷

Solution: The number of comparisons will be used as the measure of the time complexity of the algorithm, because comparisons are the basic operations used.

To find the maximum element of a set with n elements, listed in an arbitrary order, the temporary maximum is first set equal to the initial term in the list. Then, after a comparison $i \leq n$ has been done to determine that the end of the list has not yet been reached, the temporary maximum and second term are compared, updating the temporary maximum to the value of the second term if it is larger. This procedure is continued, using two additional comparisons for each term of the list—one $i \leq n$, to determine that the end of the list has not been reached and another max $< a_i$, to determine whether to update the temporary maximum. Because two comparisons are used for each of the second through the nth elements and one more comparison is used to exit the loop when $i = n + 1$, exactly $2(n - 1) + 1 = 2n - 1$ comparisons are used whenever this algorithm is applied. Hence, the algorithm for finding the maximum of a set of n elements has time complexity $\Theta(n)$, measured in terms of the number of comparisons used. Note that for this algorithm the number of comparisons is independent of particular input of n numbers. ◀

Next, we will analyze the time complexity of searching algorithms.

EXAMPLE 2 Describe the time complexity of the linear search algorithm (specified as Algortihm 2 in Section 3.1).

Solution: The number of comparisons used by Algorithm 2 in Section 3.1 will be taken as the measure of the time complexity. At each step of the loop in the algorithm, two comparisons are performed—one $i \leq n$, to see whether the end of the list has been reached and one $x \leq a_i$, to compare the element x with a term of the list. Finally, one more comparison $i \leq n$ is made outside the loop. Consequently, if $x = a_i$, $2i + 1$ comparisons are used. The most comparisons, $2n + 2$, are required when the element is not in the list. In this case, $2n$ comparisons are used to determine that x is not a_i, for $i = 1, 2, \ldots, n$, an additional comparison is used to exit the loop, and one comparison is made outside the loop. So when x is not in the list, a total of $2n + 2$ comparisons are used. Hence, a linear search requires $\Theta(n)$ comparisons in the worst case, because $2n + 2$ is $\Theta(n)$. ◀

WORST-CASE COMPLEXITY The type of complexity analysis done in Example 2 is a **worst-case** analysis. By the worst-case performance of an algorithm, we mean the largest number of operations needed to solve the given problem using this algorithm on input of specified size. Worst-case analysis tells us how many operations an algorithm requires to guarantee that it will produce a solution.

EXAMPLE 3 Describe the time complexity of the binary search algorithm (specified as Algorithm 3 in Section 3.1) in terms of the number of comparisons used (and ignoring the time required to compute $m = \lfloor (i + j)/2 \rfloor$ in each iteration of the loop in the algorithm).

Solution: For simplicity, assume there are $n = 2^k$ elements in the list $a_1, a_2, \ldots, a_n$, where k is a nonnegative integer. Note that $k = \log n$. (If n, the number of elements in the list, is not a power of 2, the list can be considered part of a larger list with 2^{k+1} elements, where $2^k < n < 2^{k+1}$. Here 2^{k+1} is the smallest power of 2 larger than n.)

At each stage of the algorithm, i and j, the locations of the first term and the last term of the restricted list at that stage, are compared to see whether the restricted list has more than one term. If $i < j$, a comparison is done to determine whether x is greater than the middle term of the restricted list.

At the first stage the search is restricted to a list with 2^{k-1} terms. So far, two comparisons have been used. This procedure is continued, using two comparisons at each stage to restrict the search to a list with half as many terms. In other words, two comparisons are used at the first stage of the algorithm when the list has 2^k elements, two more when the search has been reduced to a list with 2^{k-1} elements, two more when the search has been reduced to a list with 2^{k-2} elements, and so on, until two comparisons are used when the search has been reduced to a list with $2^1 = 2$ elements. Finally, when one term is left in the list, one comparison tells us that there are no additional terms left, and one more comparison is used to determine if this term is x.

Hence, at most $2k + 2 = 2 \log n + 2$ comparisons are required to perform a binary search when the list being searched has 2^k elements. (If n is not a power of 2, the original list is expanded to a list with 2^{k+1} terms, where $k = \lfloor \log n \rfloor$, and the search requires at most $2 \lceil \log n \rceil + 2$ comparisons.) It follows that in the worst case, binary search requires $O(\log n)$ comparisons.

Note that in the worst case, $2 \log n + 2$ comparisons are used by the binary search. Hence, the binary search uses $\Theta(\log n)$ comparisons in the worst case, because $2 \log n + 2 = \Theta(\log n)$. From this analysis it follows that in the worst case, the binary search algorithm is more efficient than the linear search algorithm, because we know by Example 2 that the linear search algorithm has $\Theta(n)$ worst-case time complexity. ◀

AVERAGE-CASE COMPLEXITY   Another important type of complexity analysis, besides worst-case analysis, is called **average-case** analysis. The average number of operations used to solve the problem over all possible inputs of a given size is found in this type of analysis. Average-case time complexity analysis is usually much more complicated than worst-case analysis. However, the average-case analysis for the linear search algorithm can be done without difficulty, as shown in Example 4.

EXAMPLE 4   Describe the average-case performance of the linear search algorithm in terms of the average number of comparisons used, assuming that the integer x is in the list and it is equally likely that x is in any position.

Solution: By hypothesis, the integer x is one of the integers $a_1, a_2, \ldots, a_n$ in the list. If x is the first term a_1 of the list, three comparisons are needed, one $i \le n$ to determine whether the end of the list has been reached, one $x \neq a_i$ to compare x and the first term, and one $i \le n$ outside the loop. If x is the second term a_2 of the list, two more comparisons are needed, so that a total of five comparisons are used. In general, if x is the ith term of the list a_i, two comparisons will be used at each of the i steps of the loop, and one outside the loop, so that a total of $2i + 1$ comparisons are needed. Hence, the average number of comparisons used equals

$$\frac{3 + 5 + 7 + \cdots + (2n + 1)}{n} = \frac{2(1 + 2 + 3 + \cdots + n) + n}{n}.$$

Using the formula from line 2 of Table 2 in Section 2.4 (and see Exercise 37(b) of Section 2.4),

$$1 + 2 + 3 + \cdots + n = \frac{n(n + 1)}{2}.$$

Hence, the average number of comparisons used by the linear search algorithm (when x is known to be in the list) is

$$\frac{2[n(n + 1)/2]}{n} + 1 = n + 2,$$

which is $\Theta(n)$. ◀

Remark: In the analysis in Example 4 we assumed that x is in the list being searched. It is also possible to do an average-case analysis of this algorithm when x may not be in the list (see Exercise 23).

Remark: Although we have counted the comparisons needed to determine whether we have reached the end of a loop, these comparisons are often not counted. From this point on we will ignore such comparisons.

WORST-CASE COMPLEXITY OF TWO SORTING ALGORITHMS We analyze the worst-case complexity of the bubble sort and the insertion sort in Examples 5 and 6.

EXAMPLE 5 What is the worst-case complexity of the bubble sort in terms of the number of comparisons made?

Solution: The bubble sort described before Example 4 in Section 3.1 sorts a list by performing a sequence of passes through the list. During each pass the bubble sort successively compares adjacent elements, interchanging them if necessary. When the ith pass begins, the $i - 1$ largest elements are guaranteed to be in the correct positions. During this pass, $n - i$ comparisons are used. Consequently, the total number of comparisons used by the bubble sort to order a list of n elements is

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

using a summation formula from line 2 in Table 2 in Section 2.4 (and Exercise 37(b) in Section 2.4). Note that the bubble sort always uses this many comparisons, because it continues even if the list becomes completely sorted at some intermediate step. Consequently, the bubble sort uses $(n - 1)n/2$ comparisons, so it has $\Theta(n^2)$ worst-case complexity in terms of the number of comparisons used. ◄

EXAMPLE 6 What is the worst-case complexity of the insertion sort in terms of the number of comparisons made?

Solution: The insertion sort (described in Section 3.1) inserts the jth element into the correct position among the first $j - 1$ elements that have already been put into the correct order. It does this by using a linear search technique, successively comparing the jth element with successive terms until a term that is greater than or equal to it is found or it compares a_j with itself and stops because a_j is not less than itself. Consequently, in the worst case, j comparisons are required to insert the jth element into the correct position. Therefore, the total number of comparisons used by the insertion sort to sort a list of n elements is

$$2 + 3 + \cdots + n = \frac{n(n + 1)}{2} - 1,$$

using the summation formula for the sum of consecutive integers in line 2 of Table 2 of Section 2.4 (and see Exercise 37(b) of Section 2.4), and noting that the first term, 1, is missing in this sum. Note that the insertion sort may use considerably fewer comparisons if the smaller elements started out at the end of the list. We conclude that the insertion sort has worst-case complexity $\Theta(n^2)$. ◄

In Examples 5 and 6 we showed that both the bubble sort and the insertion sort have worst-case time complexity $\Theta(n^2)$. However, the most efficient sorting algorithms can sort n items in $O(n \log n)$ time, as we will show in Sections 8.3 and 11.1 using techniques we develop in those sections. From this point on, we will assume that sorting n items can be done in $O(n \log n)$ time.

Links

You can run animations found on many different websites that simultaneously run different sorting algorithms on the same lists. Doing so will help you gain insights into the efficiency of different sorting algorithms. Among the sorting algorithms that you can find are the bubble sort, the insertion sort, the shell sort, the merge sort, and the quick sort. Some of these animations allow you to test the relative performance of these sorting algorithms on lists of randomly selected items, lists that are nearly sorted, and lists that are in reversed order.

3.3.3 Complexity of Matrix Multiplication

The definition of the product of two matrices can be expressed as an algorithm for computing the product of two matrices. Suppose that $\mathbf{C} = [c_{ij}]$ is the $m \times n$ matrix that is the product of the $m \times k$ matrix $\mathbf{A} = [a_{ij}]$ and the $k \times n$ matrix $\mathbf{B} = [b_{ij}]$. The algorithm based on the definition of the matrix product is expressed in pseudocode in Algorithm 1.

ALGORITHM 1 Matrix Multiplication.

procedure *matrix multiplication*($\mathbf{A}$, $\mathbf{B}$: matrices)
for $i := 1$ **to** m
    for $j := 1$ **to** n
        $c_{ij} := 0$
        for $q := 1$ **to** k
            $c_{ij} := c_{ij} + a_{iq}b_{qj}$
return $\mathbf{C}$ {$\mathbf{C} = [c_{ij}]$ is the product of $\mathbf{A}$ and $\mathbf{B}$}

We can determine the complexity of this algorithm in terms of the number of additions and multiplications used.

EXAMPLE 7 How many additions of integers and multiplications of integers are used by Algorithm 1 to multiply two $n \times n$ matrices with integer entries?

Solution: There are n^2 entries in the product of $\mathbf{A}$ and $\mathbf{B}$. To find each entry requires a total of n multiplications and $n - 1$ additions. Hence, a total of n^3 multiplications and $n^2(n - 1)$ additions are used. ◀

Surprisingly, there are more efficient algorithms for matrix multiplication than that given in Algorithm 1. As Example 7 shows, multiplying two $n \times n$ matrices directly from the definition requires $O(n^3)$ multiplications and additions. Using other algorithms, two $n \times n$ matrices can be multiplied using $O(n^{\sqrt{7}})$ multiplications and additions. (Details of such algorithms can be found in [CoLeRiSt09].)

We can also analyze the complexity of the algorithm we described in Chapter 2 for computing the Boolean product of two matrices, which we display as Algorithm 2.

ALGORITHM 2 **The Boolean Product of Zero–One Matrices.**

procedure *Boolean product of Zero–One Matrices* (**A**, **B**: zero–one matrices)
for $i := 1$ **to** m
    for $j := 1$ **to** n
        $c_{ij} := 0$
        for $q := 1$ **to** k
            $c_{ij} := c_{ij} \vee (a_{iq} \wedge b_{qj})$
return C {**C** $= [c_{ij}]$ is the Boolean product of **A** and **B**}

The number of bit operations used to find the Boolean product of two $n \times n$ matrices can be easily determined.

EXAMPLE 8   How many bit operations are used to find **A** $\odot$ **B**, where **A** and **B** are $n \times n$ zero–one matrices?

Solution: There are n^2 entries in **A** $\odot$ **B**. Using Algorithm 2, a total of n *OR*s and n *AND*s are used to find an entry of **A** $\odot$ **B**. Hence, $2n$ bit operations are used to find each entry. Therefore, $2n^3$ bit operations are required to compute **A** $\odot$ **B** using Algorithm 2. ◀

MATRIX-CHAIN MULTIPLICATION   There is another important problem involving the complexity of the multiplication of matrices. How should the **matrix-chain $A_1 A_2 \cdots A_n$** be computed using the fewest multiplications of integers, where $A_1, A_2, \ldots, A_n$ are $m_1 \times m_2, m_2 \times m_3, \ldots, m_n \times m_{n+1}$ matrices, respectively, and each has integers as entries? (Because matrix multiplication is associative, as shown in Exercise 13 in Section 2.6, the order of the multiplication used does not change the product.) Note that $m_1 m_2 m_3$ multiplications of integers are performed to multiply an $m_1 \times m_2$ matrix and an $m_2 \times m_3$ matrix using Algorithm 1. Example 9 illustrates this problem.

Links ❯

EXAMPLE 9   In which order should the matrices A_1, A_2, and A_3—where A_1 is 30×20, A_2 is 20×40, and A_3 is 40×10, all with integer entries—be multiplied to use the least number of multiplications of integers?

Solution: There are two possible ways to compute $A_1 A_2 A_3$. These are $A_1(A_2 A_3)$ and $(A_1 A_2)A_3$.

If A_2 and A_3 are first multiplied, a total of $20 \cdot 40 \cdot 10 = 8000$ multiplications of integers are used to obtain the 20×10 matrix $A_2 A_3$. Then, to multiply A_1 and $A_2 A_3$ requires $30 \cdot 20 \cdot 10 = 6000$ multiplications. Hence, a total of

$$8000 + 6000 = 14{,}000$$

multiplications are used. On the other hand, if A_1 and A_2 are first multiplied, then $30 \cdot 20 \cdot 40 = 24{,}000$ multiplications are used to obtain the 30×40 matrix $A_1 A_2$. Then, to multiply $A_1 A_2$ and A_3 requires $30 \cdot 40 \cdot 10 = 12{,}000$ multiplications. Hence, a total of

$$24{,}000 + 12{,}000 = 36{,}000$$

multiplications are used.

Clearly, the first method is more efficient. ◀

We will return to this problem in Exercise 57 in Section 8.1. Algorithms for determining the most efficient way to carry out matrix-chain multiplication are discussed in [CoLeRiSt09].

3.3.4   Algorithmic Paradigms

In Section 3.1 we introduced the basic notion of an algorithm. We provided examples of many different algorithms, including searching and sorting algorithms. We also introduced the concept of a greedy algorithm, giving examples of several problems that can be solved by greedy algorithms. Greedy algorithms provide an example of an **algorithmic paradigm**, that is, a general approach based on a particular concept that can be used to construct algorithms for solving a variety of problems.

In this book we will construct algorithms for solving many different problems based on a variety of algorithmic paradigms, including the most widely used algorithmic paradigms. These paradigms can serve as the basis for constructing efficient algorithms for solving a wide range of problems.

Some of the algorithms we have already studied are based on an algorithmic paradigm known as brute force, which we will describe in this section. Algorithmic paradigms, studied later in this book, include divide-and-conquer algorithms studied in Chapter 8, dynamic programming, also studied in Chapter 8, backtracking, studied in Chapter 10, and probabilistic algorithms, studied in Chapter 7. There are many important algorithmic paradigms besides those described in this book. Consult books on algorithm design such as [KlTa06] to learn more about them.

BRUTE-FORCE ALGORITHMS   Brute force is an important, and basic, algorithmic paradigm. In a **brute-force algorithm**, a problem is solved in the most straightforward manner based on the statement of the problem and the definitions of terms. Brute-force algorithms are designed to solve problems without regard to the computing resources required. For example, in some brute-force algorithms the solution to a problem is found by examining every possible solution, looking for the best possible. In general, brute-force algorithms are naive approaches for solving problems that do not take advantage of any special structure of the problem or clever ideas.

Note that Algorithm 1 in Section 3.1 for finding the maximum number in a sequence is a brute-force algorithm because it examines each of the n numbers in a sequence to find the maximum term. The algorithm for finding the sum of n numbers by adding one additional number at a time is also a brute-force algorithm, as is the algorithm for matrix multiplication based on its definition (Algorithm 1). The bubble, insertion, and selection sorts (described in Section 3.1 in Algorithms 4 and 5 and in the preamble of Exercise 43, respectively) are also considered to be brute-force algorithms; all three of these sorting algorithms are straightforward approaches much less efficient than other sorting algorithms such as the merge sort and the quick sort discussed in Chapters 5 and 8.

Although brute-force algorithms are often inefficient, they are often quite useful. A brute-force algorithm may be able to solve practical instances of problems, particularly when the input is not too large, even if it is impractical to use this algorithm for larger inputs. Furthermore, when designing new algorithms to solve a problem, the goal is often to find a new algorithm that is more efficient than a brute-force algorithm. One such problem of this type is described in Example 10.

EXAMPLE 10   Construct a brute-force algorithm for finding the closest pair of points in a set of n points in the plane and provide a worst-case big-O estimate for the number of bit operations used by the algorithm.

Solution: Suppose that we are given as input the points $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$. Recall that the distance between (x_i, y_i) and (x_j, y_j) is $\sqrt{(x_j - x_i)^2 + (y_j - y_i)^2}$. A brute-force algorithm can find the closest pair of these points by computing the distances between all pairs of the n points and determining the smallest distance. (We can make one small simplification to make the computation easier; we can compute the square of the distance between pairs of points to find the

closest pair, rather than the distance between these points. We can do this because the square of the distance between a pair of points is smallest when the distance between these points is smallest.)

ALGORITHM 3 Brute-Force Algorithm for Closest Pair of Points.

procedure *closest-pair*($(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$): pairs of real numbers)
min$= \infty$
for $i := 2$ **to** n
    for $j := 1$ **to** $i - 1$
        if $(x_j - x_i)^2 + (y_j - y_i)^2 < min$ **then**
            $min := (x_j - x_i)^2 + (y_j - y_i)^2$
            closest pair $:= ((x_i, y_i), (x_j, y_j))$
return *closest pair*

To estimate the number of operations used by the algorithm, first note that there are $n(n - 1)/2$ pairs of points $((x_i, y_i), (x_j, y_j))$ that we loop through (as the reader should verify). For each such pair we compute $(x_j - x_i)^2 + (y_j - y_i)^2$, compare it with the current value of *min*, and if it is smaller than *min*, replace the current value of *min* by this new value. It follows that this algorithm uses $\Theta(n^2)$ operations, in terms of arithmetic operations and comparisons.

In Chapter 8 we will devise an algorithm that determines the closest pair of points when given n points in the plane as input that has $O(n \log n)$ worst-case complexity. The original discovery of such an algorithm, much more efficient than the brute-force approach, was considered quite surprising. ◀

3.3.5 Understanding the Complexity of Algorithms

Table 1 displays some common terminology used to describe the time complexity of algorithms. For example, an algorithm that finds the largest of the first 100 terms of a list of n elements by applying Algorithm 1 to the sequence of the first 100 terms, where n is an integer with $n \geq 100$, has **constant complexity** because it uses 99 comparisons no matter what n is (as the reader can verify). The linear search algorithm has **linear** (worst-case or average-case) **complexity** and the binary search algorithm has **logarithmic** (worst-case) **complexity**. Many important algorithms have $n \log n$, or **linearithmic** (worst-case) **complexity**, such as the merge sort, which we will introduce in Chapter 4. (The word *linearithmic* is a combination of the words *linear* and log*arithmic*.)

TABLE 1 Commonly Used Terminology for the Complexity of Algorithms.

Complexity	Terminology
$\Theta(1)$	Constant complexity
$\Theta(\log n)$	Logarithmic complexity
$\Theta(n)$	Linear complexity
$\Theta(n \log n)$	Linearithmic complexity
$\Theta(n^b)$	Polynomial complexity
$\Theta(b^n)$, where $b > 1$	Exponential complexity
$\Theta(n!)$	Factorial complexity

An algorithm has **polynomial complexity** if it has complexity $\Theta(n^b)$, where b is an integer with $b \geq 1$. For example, the bubble sort algorithm is a polynomial-time algorithm because it uses $\Theta(n^2)$ comparisons in the worst case. An algorithm has **exponential complexity** if it has time complexity $\Theta(b^n)$, where $b > 1$. The algorithm that determines whether a compound proposition in n variables is satisfiable by checking all possible assignments of truth variables is an algorithm with exponential complexity, because it uses $\Theta(2^n)$ operations. Finally, an algorithm has **factorial complexity** if it has $\Theta(n!)$ time complexity. The algorithm that finds all orders that a traveling salesperson could use to visit n cities has factorial complexity; we will discuss this algorithm in Chapter 9.

TRACTABILITY   A problem that is solvable using an algorithm with polynomial (or better) worst-case complexity is called **tractable**, because the expectation is that the algorithm will produce the solution to the problem for reasonably sized input in a relatively short time. However, if the polynomial in the big-Θ estimate has high degree (such as degree 100) or if the coefficients are extremely large, the algorithm may take an extremely long time to solve the problem. Consequently, that a problem can be solved using an algorithm with polynomial worst-case time complexity is no guarantee that the problem can be solved in a reasonable amount of time for even relatively small input values. Fortunately, in practice, the degree and coefficients of polynomials in such estimates are often small.

The situation is much worse for problems that cannot be solved using an algorithm with worst-case polynomial time complexity. Such problems are called **intractable**. Usually, but not always, an extremely large amount of time is required to solve the problem for the worst cases of even small input values. In practice, however, there are situations where an algorithm with a certain worst-case time complexity may be able to solve a problem much more quickly for most cases than for its worst case. When we are willing to allow that some, perhaps small, number of cases may not be solved in a reasonable amount of time, the average-case time complexity is a better measure of how long an algorithm takes to solve a problem. Many problems important in industry are thought to be intractable but can be practically solved for essentially all sets of input that arise in daily life. Another way that intractable problems are handled when they arise in practical applications is that instead of looking for exact solutions of a problem, approximate solutions are sought. It may be the case that fast algorithms exist for finding such approximate solutions, perhaps even with a guarantee that they do not differ by very much from an exact solution.

Some problems even exist for which it can be shown that no algorithm exists for solving them. Such problems are called **unsolvable** (as opposed to **solvable** problems that can be solved using an algorithm). The first proof that there are unsolvable problems was provided by the great English mathematician and computer scientist Alan Turing when he showed that the halting problem is unsolvable. Recall that we proved that the halting problem is unsolvable in Section 3.1. (A biography of Alan Turing and a description of some of his other work can be found in Chapter 13.)

P VERSUS NP   The study of the complexity of algorithms goes far beyond what we can describe here. Note, however, that many solvable problems are believed to have the property that no algorithm with polynomial worst-case time complexity solves them, but that a solution, if known, can be checked in polynomial time. Problems for which a solution can be checked in polynomial time are said to belong to the **class NP** (tractable problems are said to belong to **class P**). The abbreviation NP stands for *nondeterministic polynomial* time. The satisfiability problem, discussed in Section 1.3, is an example of an NP problem—we can quickly verify that an assignment of truth values to the variables of a compound proposition makes it true, but no polynomial time algorithm has been discovered for finding such an assignment of truth values. (For example, an exhaustive search of all possible truth values requires $\Omega(2^n)$ bit operations where n is the number of variables in the compound proposition.)

There is also an important class of problems, called **NP-complete problems**, with the property that if any of these problems can be solved by a polynomial worst-case time algorithm, then

Links

all problems in the class NP can be solved by polynomial worst-case time algorithms. The satisfiability problem is also an example of an NP-complete problem. It is an NP problem and if a polynomial time algorithm for solving it were known, there would be polynomial time algorithms for all problems known to be in this class of problems (and there are many important problems in this class). This last statement follows from the fact that every problem in NP can be reduced in polynomial time to the satisfiability problem. Although more than 3000 NP-complete problems are now known, the satisfiability problem was the first problem shown to be NP-complete. The theorem that asserts this is known as the **Cook-Levin theorem** after Stephen Cook and Leonid Levin, who independently proved it in the early 1970s.

The **P versus NP problem** asks whether NP, the class of problems for which it is possible to check solutions in polynomial time, equals P, the class of tractable problems. If P $\neq$ NP, there would be some problems that cannot be solved in polynomial time, but whose solutions could be verified in polynomial time. The concept of NP-completeness is helpful in research aimed at solving the P versus NP problem, because NP-complete problems are the problems in NP considered most likely not to be in P, as every problem in NP can be reduced to an NP-complete problem in polynomial time. A large majority of theoretical computer scientists believe that P $\neq$ NP, which would mean that no NP-complete problem can be solved in polynomial time. One reason for this belief is that despite extensive research, no one has succeeded in showing that P $=$ NP. In particular, no one has been able to find an algorithm with worst-case polynomial time complexity that solves any NP-complete problem. The P versus NP problem is one of the most famous unsolved problems in the mathematical sciences (which include theoretical computer science). It is one of the seven famous Millennium Prize Problems, of which six remain unsolved. A prize of $1,000,000 is offered by the Clay Mathematics Institute for its solution.

For more information about the complexity of algorithms, consult the references, including [CoLeRiSt09], for this section listed at the end of this book. (Also, for a more formal discussion of computational complexity in terms of Turing machines, see Section 13.5.)

PRACTICAL CONSIDERATIONS   Note that a big-Θ estimate of the time complexity of an algorithm expresses how the time required to solve the problem increases as the input grows in size. In practice, the best estimate (that is, with the smallest reference function) that can be shown is used. However, big-Θ estimates of time complexity cannot be directly translated into the actual amount of computer time used. One reason is that a big-Θ estimate $f(n)$ is $\Theta(g(n))$, where $f(n)$ is the time complexity of an algorithm and $g(n)$ is a reference function, means that $C_1 g(n) \leq f(n) \leq C_2 g(n)$ when $n > k$, where C_1, C_2, and k are constants. So without knowing the constants C_1, C_2, and k in the inequality, this estimate cannot be used to determine a lower bound and an upper bound on the number of operations used in the worst case. As remarked before, the time required for an operation depends on the type of operation and the

TABLE 2 The Computer Time Used by Algorithms.						
Problem Size	*Bit Operations Used*					
n	$\log n$	*n*	$n \log n$	n^2	2^n	$n!$
10	3×10^{-11} s	10^{-10} s	3×10^{-10} s	10^{-9} s	10^{-8} s	3×10^{-7} s
10^2	7×10^{-11} s	10^{-9} s	7×10^{-9} s	10^{-7} s	4×10^{11} yr	*
10^3	1.0×10^{-10} s	10^{-8} s	1×10^{-7} s	10^{-5} s	*	*
10^4	1.3×10^{-10} s	10^{-7} s	1×10^{-6} s	10^{-3} s	*	*
10^5	1.7×10^{-10} s	10^{-6} s	2×10^{-5} s	0.1 s	*	*
10^6	2×10^{-10} s	10^{-5} s	2×10^{-4} s	0.17 min	*	*

computer being used. Often, instead of a big-Θ estimate on the worst-case time complexity of an algorithm, we have only a big-O estimate. Note that a big-O estimate on the time complexity of an algorithm provides an upper, but not a lower, bound on the worst-case time required for the algorithm as a function of the input size. Nevertheless, for simplicity, we will often use big-O estimates when describing the time complexity of algorithms, with the understanding that big-Θ estimates would provide more information.

Table 2 displays the time needed to solve problems of various sizes with an algorithm using the indicated number n of bit operations, assuming that each bit operation takes 10^{-11} seconds, a reasonable estimate of the time required for a bit operation using the fastest computers available in 2018. Times of more than 10^{100} years are indicated with an asterisk. In the future, these times will decrease as faster computers are developed. We can use the times shown in Table 2 to see whether it is reasonable to expect a solution to a problem of a specified size using an algorithm with known worst-case time complexity when we run this algorithm on a modern computer. Note that we cannot determine the exact time a computer uses to solve a problem with input of a particular size because of a myriad of issues involving computer hardware and the particular software implementation of the algorithm.

It is important to have a reasonable estimate for how long it will take a computer to solve a problem. For instance, if an algorithm requires approximately 10 hours, it may be worthwhile to spend the computer time (and money) required to solve this problem. But, if an algorithm requires approximately 10 billion years to solve a problem, it would be unreasonable to use resources to implement this algorithm. One of the most interesting phenomena of modern technology is the tremendous increase in the speed and memory space of computers. Another important factor that decreases the time needed to solve problems on computers is **parallel processing**, which is the technique of performing sequences of operations simultaneously.

Efficient algorithms, including most algorithms with polynomial time complexity, benefit most from significant technology improvements. However, these technology improvements offer little help in overcoming the complexity of algorithms of exponential or factorial time complexity. Because of the increased speed of computation, increases in computer memory, and the use of algorithms that take advantage of parallel processing, many problems that were considered impossible to solve five years ago are now routinely solved, and certainly five years from now this statement will still be true. This is even true when the algorithms used are intractable.

Exercises

1. Give a big-O estimate for the number of operations (where an operation is an addition or a multiplication) used in this segment of an algorithm.

$t := 0$
for $i := 1$ **to** 3
  for $j := 1$ **to** 4
    $t := t + ij$

2. Give a big-O estimate for the number additions used in this segment of an algorithm.

$t := 0$
for $i := 1$ **to** n
  for $j := 1$ **to** n
    $t := t + i + j$

3. Give a big-O estimate for the number of operations, where an operation is a comparison or a multiplication, used in this segment of an algorithm (ignoring comparisons used to test the conditions in the **for** loops, where $a_1, a_2, ..., a_n$ are positive real numbers).

$m := 0$
for $i := 1$ **to** n
  for $j := i + 1$ **to** n
    $m := \max(a_i a_j, m)$

4. Give a big-O estimate for the number of operations, where an operation is an addition or a multiplication, used in this segment of an algorithm (ignoring comparisons used to test the conditions in the **while** loop).

$i := 1$
$t := 0$
while $i \leq n$
  $t := t + i$
  $i := 2i$

5. How many comparisons are used by the algorithm given in Exercise 16 of Section 3.1 to find the smallest natural number in a sequence of n natural numbers?

6. a) Use pseudocode to describe the algorithm that puts the first four terms of a list of real numbers of arbitrary length in increasing order using the insertion sort.

  b) Show that this algorithm has time complexity $O(1)$ in terms of the number of comparisons used.

7. Suppose that an element is known to be among the first four elements in a list of 32 elements. Would a linear search or a binary search locate this element more rapidly?

8. Given a real number x and a positive integer k, determine the number of multiplications used to find x^{2^k} starting with x and successively squaring (to find x^2, x^4, and so on). Is this a more efficient way to find x^{2^k} than by multiplying x by itself the appropriate number of times?

9. Give a big-O estimate for the number of comparisons used by the algorithm that determines the number of 1s in a bit string by examining each bit of the string to determine whether it is a 1 bit (see Exercise 25 of Section 3.1).

*10. a) Show that this algorithm determines the number of 1 bits in the bit string S:

    procedure *bit count*(S: bit string)
    count := 0
    while $S \neq 0$
      count := *count* + 1
      $S := S \wedge (S - 1)$
    return *count* {*count* is the number of 1s in S}

  Here $S - 1$ is the bit string obtained by changing the rightmost 1 bit of S to a 0 and all the 0 bits to the right of this to 1s. [Recall that $S \wedge (S - 1)$ is the bitwise *AND* of S and $S - 1$.]

  b) How many bitwise *AND* operations are needed to find the number of 1 bits in a string S using the algorithm in part (a)?

11. a) Suppose we have n subsets $S_1, S_2, ..., S_n$ of the set $\{1, 2, ..., n\}$. Express a brute-force algorithm that determines whether there is a disjoint pair of these subsets. [*Hint:* The algorithm should loop through the subsets; for each subset S_i, it should then loop through all other subsets; and for each of these other subsets S_j, it should loop through all elements k in S_i to determine whether k also belongs to S_j.]

  b) Give a big-O estimate for the number of times the algorithm needs to determine whether an integer is in one of the subsets.

12. Consider the following algorithm, which takes as input a sequence of n integers $a_1, a_2, ..., a_n$ and produces as output a matrix $\mathbf{M} = \{m_{ij}\}$ where m_{ij} is the minimum term in the sequence of integers $a_i, a_{i+1}, ..., a_j$ for $j \geq i$ and $m_{ij} = 0$ otherwise.

    initialize $\mathbf{M}$ so that $m_{ij} = a_i$ if $j \geq i$ and $m_{ij} = 0$
      otherwise
    for $i := 1$ **to** n
      for $j := i + 1$ **to** n
        for $k := i + 1$ **to** j
          $m_{ij} := \min(m_{ij}, a_k)$
    return $\mathbf{M} = \{m_{ij}\}$ {m_{ij} is the minimum term of
      $a_i, a_{i+1}, ..., a_j$}

  a) Show that this algorithm uses $O(n^3)$ comparisons to compute the matrix $\mathbf{M}$.

  b) Show that this algorithm uses $\Omega(n^3)$ comparisons to compute the matrix $\mathbf{M}$. Using this fact and part (a), conclude that the algorithms uses $\Theta(n^3)$ comparisons. [*Hint:* Only consider the cases where $i \leq n/4$ and $j \geq 3n/4$ in the two outer loops in the algorithm.]

13. The conventional algorithm for evaluating a polynomial $a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$ at $x = c$ can be expressed in pseudocode by

  procedure *polynomial*($c, a_0, a_1, ..., a_n$: real numbers)
    power := 1
    $y := a_0$
    for $i := 1$ **to** n
      power := *power* $* c$
      $y := y + a_i * power$
    return y {$y = a_n c^n + a_{n-1} c^{n-1} + \cdots + a_1 c + a_0$}

  where the final value of y is the value of the polynomial at $x = c$.

  a) Evaluate $3x^2 + x + 1$ at $x = 2$ by working through each step of the algorithm showing the values assigned at each assignment step.

  b) Exactly how many multiplications and additions are used to evaluate a polynomial of degree n at $x = c$? (Do not count additions used to increment the loop variable.)

14. There is a more efficient algorithm (in terms of the number of multiplications and additions used) for evaluating polynomials than the conventional algorithm described in the previous exercise. It is called **Horner's method**. This pseudocode shows how to use this method

to find the value of $a_nx^n + a_{n-1}x^{n-1} + \cdots + a_1x + a_0$ at $x = c$.

procedure $Horner(c, a_0, a_1, a_2, \ldots, a_n$: real numbers)
$\quad y := a_n$
$\quad$ **for** $i := 1$ **to** n
$\qquad y := y * c + a_{n-i}$
$\quad$ **return** y $\{y = a_nc^n + a_{n-1}c^{n-1} + \cdots + a_1c + a_0\}$

a) Evaluate $3x^2 + x + 1$ at $x = 2$ by working through each step of the algorithm showing the values assigned at each assignment step.

b) Exactly how many multiplications and additions are used by this algorithm to evaluate a polynomial of degree n at $x = c$? (Do not count additions used to increment the loop variable.)

15. What is the largest n for which one can solve within one second a problem using an algorithm that requires $f(n)$ bit operations, where each bit operation is carried out in 10^{-9} seconds, with these functions $f(n)$?

a) $\log n$　　　**b)** n　　　**c)** $n \log n$
d) n^2　　　**e)** 2^n　　　**f)** $n!$

16. What is the largest n for which one can solve within a day using an algorithm that requires $f(n)$ bit operations, where each bit operation is carried out in 10^{-11} seconds, with these functions $f(n)$?

a) $\log n$　　　**b)** $1000n$　　　**c)** n^2
d) $1000n^2$　　　**e)** n^3　　　**f)** 2^n
g) 2^{2n}　　　**h)** 2^{2^n}

17. What is the largest n for which one can solve within a minute using an algorithm that requires $f(n)$ bit operations, where each bit operation is carried out in 10^{-12} seconds, with these functions $f(n)$?

a) $\log \log n$　　　**b)** $\log n$　　　**c)** $(\log n)^2$
d) $1{,}000{,}000n$　　　**e)** n^2　　　**f)** 2^n
g) 2^{n^2}

18. How much time does an algorithm take to solve a problem of size n if this algorithm uses $2n^2 + 2^n$ operations, each requiring 10^{-9} seconds, with these values of n?

a) 10　　　**b)** 20　　　**c)** 50　　　**d)** 100

19. How much time does an algorithm using 2^{50} operations need if each operation takes these amounts of time?

a) 10^{-6} s　　　**b)** 10^{-9} s　　　**c)** 10^{-12} s

20. What is the effect in the time required to solve a problem when you double the size of the input from n to $2n$, assuming that the number of milliseconds the algorithm uses to solve the problem with input size n is each of these functions? [Express your answer in the simplest form possible, either as a ratio or a difference. Your answer may be a function of n or a constant.]

a) $\log \log n$　　　**b)** $\log n$　　　**c)** $100n$
d) $n \log n$　　　**e)** n^2　　　**f)** n^3
g) 2^n

21. What is the effect in the time required to solve a problem when you increase the size of the input from n to $n + 1$, assuming that the number of milliseconds the algorithm uses to solve the problem with input size n is

each of these functions? [Express your answer in the simplest form possible, either as a ratio or a difference. Your answer may be a function of n or a constant.]

a) $\log n$　　　**b)** $100n$　　　**c)** n^2
d) n^3　　　**e)** 2^n　　　**f)** 2^{n^2}
g) $n!$

22. Determine the least number of comparisons, or best-case performance,

a) required to find the maximum of a sequence of n integers, using Algorithm 1 of Section 3.1.

b) used to locate an element in a list of n terms with a linear search.

c) used to locate an element in a list of n terms using a binary search.

23. Analyze the average-case performance of the linear search algorithm, if exactly half the time the element x is not in the list, and if x is in the list, it is equally likely to be in any position.

24. An algorithm is called **optimal** for the solution of a problem with respect to a specified operation if there is no algorithm for solving this problem using fewer operations.

a) Show that Algorithm 1 in Section 3.1 is an optimal algorithm with respect to the number of comparisons of integers. [*Note:* Comparisons used for bookkeeping in the loop are not of concern here.]

b) Is the linear search algorithm optimal with respect to the number of comparisons of integers (not including comparisons used for bookkeeping in the loop)?

25. Describe the worst-case time complexity, measured in terms of comparisons, of the ternary search algorithm described in Exercise 27 of Section 3.1.

26. Describe the worst-case time complexity, measured in terms of comparisons, of the search algorithm described in Exercise 28 of Section 3.1.

27. Analyze the worst-case time complexity of the algorithm you devised in Exercise 29 of Section 3.1 for locating a mode in a list of nondecreasing integers.

28. Analyze the worst-case time complexity of the algorithm you devised in Exercise 30 of Section 3.1 for locating all modes in a list of nondecreasing integers.

29. Analyze the worst-case time complexity of the algorithm you devised in Exercise 33 of Section 3.1 for finding the first term of a sequence of integers equal to some previous term.

30. Analyze the worst-case time complexity of the algorithm you devised in Exercise 34 of Section 3.1 for finding all terms of a sequence that are greater than the sum of all previous terms.

31. Analyze the worst-case time complexity of the algorithm you devised in Exercise 35 of Section 3.1 for finding the first term of a sequence less than the immediately preceding term.

32. Determine the worst-case complexity in terms of comparisons of the algorithm from Exercise 5 in Section 3.1 for determining all values that occur more than once in a sorted list of integers.

33. Determine the worst-case complexity in terms of comparisons of the algorithm from Exercise 9 in Section 3.1 for determining whether a string of n characters is a palindrome.

34. How many comparisons does the selection sort (see preamble to Exercise 43 in Section 3.1) use to sort n items? Use your answer to give a big-O estimate of the complexity of the selection sort in terms of number of comparisons for the selection sort.

35. Determine a big-O estimate for the worst-case complexity in terms of number of comparisons used and the number of terms swapped by the binary insertion sort described in the preamble to Exercise 49 in Section 3.1.

36. Determine the number of character comparisons used by the naive string matcher to look for a pattern of m characters in a text with n characters if the first character of the pattern does not occur in the text.

37. Determine a big-O estimate of the number of character comparisons used by the naive string matcher to find all occurrences of a pattern of m characters in a text with n characters, in terms of the parameters m and n.

38. Determine big-O estimates for the algorithms for deciding whether two strings are anagrams from parts (a) and (b) of Exercise 31 of Section 3.1.

39. Determine big-O estimates for the algorithms for finding the closest of n real numbers from parts (a) and (b) of Exercise 32 of Section 3.1.

40. Show that the greedy algorithm for making change for n cents using quarters, dimes, nickels, and pennies has $O(n)$ complexity measured in terms of comparisons needed.

Exercises 41 and 42 deal with the problem of scheduling the most talks possible given the start and end times of n talks.

41. Find the complexity of a brute-force algorithm for scheduling the talks by examining all possible subsets of the talks. [*Hint:* Use the fact that a set with n elements has 2^n subsets.]

42. Find the complexity of the greedy algorithm for scheduling the most talks by adding at each step the talk with the earliest end time compatible with those already scheduled (Algorithm 7 in Section 3.1). Assume that the talks are not already sorted by earliest end time and assume that the worst-case time complexity of sorting is $O(n \log n)$.

43. Describe how the number of comparisons used in the worst case changes when these algorithms are used to search for an element of a list when the size of the list doubles from n to $2n$, where n is a positive integer.
    a) linear search   b) binary search

44. Describe how the number of comparisons used in the worst case changes when the size of the list to be sorted doubles from n to $2n$, where n is a positive integer when these sorting algorithms are used.
    a) bubble sort   b) insertion sort
    c) selection sort (described in the preamble to Exercise 43 in Section 3.1)
    d) binary insertion sort (described in the preamble to Exercise 49 in Section 3.1)

An $n \times n$ matrix is called **upper triangular** if $a_{ij} = 0$ whenever $i > j$.

45. From the definition of the matrix product, describe an algorithm in English for computing the product of two upper triangular matrices that ignores those products in the computation that are automatically equal to zero.

46. Give a pseudocode description of the algorithm in Exercise 45 for multiplying two upper triangular matrices.

47. How many multiplications of entries are used by the algorithm found in Exercise 45 for multiplying two $n \times n$ upper triangular matrices?

In Exercises 48–49 assume that the number of multiplications of entries used to multiply a $p \times q$ matrix and a $q \times r$ matrix is pqr.

48. What is the best order to form the product **ABC** if **A**, **B**, and **C** are matrices with dimensions 3×9, 9×4, and 4×2, respectively?

49. What is the best order to form the product **ABCD** if **A**, **B**, **C**, and **D** are matrices with dimensions 30×10, 10×40, 40×50, and 50×30, respectively?

Key Terms and Results

TERMS

algorithm: a finite sequence of precise instructions for performing a computation or solving a problem

searching algorithm: the problem of locating an element in a list

linear search algorithm: a procedure for searching a list element by element

binary search algorithm: a procedure for searching an ordered list by successively splitting the list in half

sorting: the reordering of the elements of a list into prescribed order

string searching: given a string, determining all the occurrences where this string occurs within a longer string

$f(x)$ is $O(g(x))$: the fact that $|f(x)| \leq C|g(x)|$ for all $x > k$ for some constants C and k

witness to the relationship $f(x)$ is $O(g(x))$: a pair C and k such that $|f(x)| \leq C|g(x)|$ whenever $x > k$

$f(x)$ is $\Omega(g(x))$: the fact that $|f(x)| \geq C|g(x)|$ for all $x > k$ for some positive constants C and k

$f(x)$ is $\Theta(g(x))$: the fact that $f(x)$ is both $O(g(x))$ and $\Omega(g(x))$

time complexity: the amount of time required for an algorithm to solve a problem

space complexity: the amount of space in computer memory required for an algorithm to solve a problem

worst-case time complexity: the greatest amount of time required for an algorithm to solve a problem of a given size

average-case time complexity: the average amount of time required for an algorithm to solve a problem of a given size

algorithmic paradigm: a general approach for constructing algorithms based on a particular concept

brute force: the algorithmic paradigm based on constructing algorithms for solving problems in a naive manner from the statement of the problem and definitions

greedy algorithm: an algorithm that makes the best choice at each step according to some specified condition

tractable problem: a problem for which there is a worst-case polynomial-time algorithm that solves it

intractable problem: a problem for which no worst-case polynomial-time algorithm exists for solving it

solvable problem: a problem that can be solved by an algorithm

unsolvable problem: a problem that cannot be solved by an algorithm

RESULTS

linear and binary search algorithms: (given in Section 3.1)

bubble sort: a sorting that uses passes where successive items are interchanged if they are in the wrong order

insertion sort: a sorting that at the jth step inserts the jth element into the correct position in the list, when the first $j - 1$ elements of the list are already sorted

The linear search has $O(n)$ worst case time complexity.
The binary search has $O(\log n)$ worst case time complexity.
The bubble and insertion sorts have $O(n^2)$ worst case time complexity.
$\log n!$ is $O(n \log n)$.

If $f_1(x)$ is $O(g_1(x))$ and $f_2(x)$ is $O(g_2(x))$, then $(f_1 + f_2)(x)$ is $O(\max(g_1(x), g_2(x)))$ and $(f_1 f_2)(x)$ is $O((g_1 g_2(x)))$.

If $a_0, a_1, \ldots, a_n$ are real numbers with $a_n \neq 0$, then $a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$ is $\Theta(x^n)$, and hence $O(n)$ and $\Omega(n)$.

Review Questions

1. a) Define the term *algorithm*.

   b) What are the different ways to describe algorithms?

   c) What is the difference between an algorithm for solving a problem and a computer program that solves this problem?

2. a) Describe, using English, an algorithm for finding the largest integer in a list of n integers.

   b) Express this algorithm in pseudocode.

   c) How many comparisons does the algorithm use?

3. a) State the definition of the fact that $f(n)$ is $O(g(n))$, where $f(n)$ and $g(n)$ are functions from the set of positive integers to the set of real numbers.

   b) Use the definition of the fact that $f(n)$ is $O(g(n))$ directly to prove or disprove that $n^2 + 18n + 107$ is $O(n^3)$.

   c) Use the definition of the fact that $f(n)$ is $O(g(n))$ directly to prove or disprove that n^3 is $O(n^2 + 18n + 107)$.

4. List these functions so that each function is big-O of the next function in the list: $(\log n)^3$, $n^3/1{,}000{,}000$, $\sqrt{n}$, $100n + 101$, 3^n, $n!$, $2^n n^2$.

5. a) How can you produce a big-O estimate for a function that is the sum of different terms where each term is the product of several functions?

   b) Give a big-O estimate for the function $f(n) = (n! + 1)(2^n + 1) + (n^{n-2} + 8n^{n-3})(n^3 + 2^n)$. For the function g in your estimate $f(x)$ is $O(g(x))$, use a simple function of smallest possible order.

6. a) Define what the worst-case time complexity, average-case time complexity, and best-case time complexity (in terms of comparisons) mean for an algorithm that finds the smallest integer in a list of n integers.

   b) What are the worst-case, average-case, and best-case time complexities, in terms of comparisons, of the algorithm that finds the smallest integer in a list of n integers by comparing each of the integers with the smallest integer found so far?

7. a) Describe the linear search and binary search algorithm for finding an integer in a list of integers in increasing order.

   b) Compare the worst-case time complexities of these two algorithms.

   c) Is one of these algorithms always faster than the other (measured in terms of comparisons)?

8. a) Describe the bubble sort algorithm.

   b) Use the bubble sort algorithm to sort the list 5, 2, 4, 1, 3.

   c) Give a big-O estimate for the number of comparisons used by the bubble sort.

9. a) Describe the insertion sort algorithm.

   b) Use the insertion sort algorithm to sort the list 2, 5, 1, 4, 3.

   c) Give a big-O estimate for the number of comparisons used by the insertion sort.

10. a) Explain the concept of a greedy algorithm.

   b) Provide an example of a greedy algorithm that produces an optimal solution and explain why it produces an optimal solution.

c) Provide an example of a greedy algorithm that does not always produce an optimal solution and explain why it fails to do so.

Supplementary Exercises

1. a) Describe an algorithm for locating the last occurrence of the largest number in a list of integers.
   b) Estimate the number of comparisons used.

2. a) Describe an algorithm for finding the first and second largest elements in a list of integers.
   b) Estimate the number of comparisons used.

3. a) Give an algorithm to determine whether a bit string contains a pair of consecutive zeros.
   b) How many comparisons does the algorithm use?

4. a) Suppose that a list contains integers that are in order of largest to smallest and an integer can appear repeatedly in this list. Devise an algorithm that locates all occurrences of an integer x in the list.
   b) Estimate the number of comparisons used.

5. a) Adapt Algorithm 1 in Section 3.1 to find the maximum and the minimum of a sequence of n elements by employing a temporary maximum and a temporary minimum that is updated as each successive element is examined.
   b) Describe the algorithm from part (a) in pseudocode.
   c) How many comparisons of elements in the sequence are carried out by this algorithm? (Do not count comparisons used to determine whether the end of the sequence has been reached.)

6. a) Describe in detail (and in English) the steps of an algorithm that finds the maximum and minimum of a sequence of n elements by examining pairs of successive elements, keeping track of a temporary maximum and a temporary minimum. If n is odd, both the temporary maximum and temporary minimum should initially equal the first term, and if n is even, the temporary minimum and temporary maximum should be found by comparing the initial two elements. The temporary maximum and temporary minimum should be updated by comparing them with the maximum and minimum of the pair of elements being examined.
   b) Express the algorithm described in part (a) in pseudocode.
   c) How many comparisons of elements of the sequence are carried out by this algorithm? (Do not count comparisons used to determine whether the end of the sequence has been reached.) How does this compare to the number of comparisons used by the algorithm in Exercise 5?

*7. Show that the worst-case complexity in terms of comparisons of an algorithm that finds the maximum and minimum of n elements is at least $\lceil 3n/2 \rceil - 2$.

8. Devise an efficient algorithm for finding the second largest element in a sequence of n elements and determine the worst-case complexity of your algorithm.

9. Devise an algorithm that finds all equal pairs of sums of two terms of a sequence of n numbers, and determine the worst-case complexity of your algorithm.

10. Devise an algorithm that finds the closest pair of integers in a sequence of n integers, and determine the worst-case complexity of your algorithm. [*Hint:* Sort the sequence. Use the fact that sorting can be done with worst-case time complexity $O(n \log n)$.]

The **shaker sort** (or **bidirectional bubble sort**) successively compares pairs of adjacent elements, exchanging them if they are out of order, and alternately passing through the list from the beginning to the end and then from the end to the beginning until no exchanges are needed.

11. Show the steps used by the shaker sort to sort the list 3, 5, 1, 4, 6, 2.

12. Express the shaker sort in pseudocode.

13. Show that the shaker sort has $O(n^2)$ complexity measured in terms of the number of comparisons it uses.

14. Explain why the shaker sort is efficient for sorting lists that are already in close to the correct order.

15. Show that $(n \log n + n^2)^3$ is $O(n^6)$.

16. Show that $8x^3 + 12x + 100 \log x$ is $O(x^3)$.

17. Give a big-O estimate for $(x^2 + x(\log x)^3) \cdot (2^x + x^3)$.

18. Find a big-O estimate for $\sum_{j=1}^{n} j(j+1)$.

*19. Show that $n!$ is not $O(2^n)$.

*20. Show that n^n is not $O(n!)$.

21. Find all pairs of functions of the same order in this list of functions: $n^2 + (\log n)^2$, $n^2 + n$, $n^2 + \log 2^n + 1$, $(n+1)^3 - (n-1)^3$, and $(n + \log n)^2$.

22. Find all pairs of functions of the same order in this list of functions $n^2 + 2^n$, $n^2 + 2^{100}$, $n^2 + 2^{2n}$, $n^2 + n!$, $n^2 + 3^n$, and $(n^2 + 1)^2$.

23. Find an integer n with $n > 2$ for which $n^{2^{100}} < 2^n$.

24. Find an integer n with $n > 2$ for which $(\log n)^{2^{100}} < \sqrt{n}$.

*25. Arrange the functions n^n, $(\log n)^2$, $n^{1.0001}$, $(1.0001)^n$, $2^{\sqrt{\log_2 n}}$, and $n(\log n)^{1001}$ in a list so that each function is big-O of the next function. [*Hint:* To determine the relative size of some of these functions, take logarithms.]

*26. Arrange the function 2^{100n}, 2^{n^2}, $2^{n!}$, 2^{2^n}, $n^{\log n}$, $n \log n \log \log n$, $n^{3/2}$, $n(\log n)^{3/2}$, and $n^{4/3}(\log n)^2$ in a list so that each function is big-O of the next function. [*Hint:* To determine the relative size of some of these functions, take logarithms.]

*27. Give an example of two increasing functions $f(n)$ and $g(n)$ from the set of positive integers to the set of positive integers such that neither $f(n)$ is $O(g(n))$ nor $g(n)$ is $O(f(n))$.

11. Define what it means for a problem to be tractable and what it means for a problem to be solvable.

28. Show that if the denominations of coins are $c^0, c^1, \ldots, c^k$, where k is a positive integer and c is a positive integer, $c > 1$, the greedy algorithm always produces change using the fewest coins possible.

29. a) Use pseudocode to specify a brute-force algorithm that determines when given as input a sequence of n positive integers whether there are two distinct terms of the sequence that have as sum a third term. The algorithm should loop through all triples of terms of the sequence, checking whether the sum of the first two terms equals the third.

   b) Give a big-O estimate for the complexity of the brute-force algorithm from part (a).

30. a) Devise a more efficient algorithm for solving the problem described in Exercise 29 that first sorts the input sequence and then checks for each pair of terms whether their difference is in the sequence.

   b) Give a big-O estimate for the complexity of this algorithm. Is it more efficient than the brute-force algorithm from Exercise 29?

Suppose we have s men and s women each with their preference lists for the members of the opposite gender, as described in the preamble to Exercise 64 in Section 3.1. We say that a woman w is a **valid partner** for a man m if there is some stable matching in which they are paired. Similarly, a man m is a **valid partner** for a woman w if there is some stable matching in which they are paired. A matching in which each man is assigned his valid partner ranking highest on his preference list is called **male optimal**, and a matching in which each woman is assigned her valid partner ranking lowest on her preference list is called **female pessimal**.

31. Find all valid partners for each man and each woman if there are three men m_1, m_2, and m_3 and three women w_1, w_2, w_3 with these preference rankings of the men for the women, from highest to lowest: m_1: w_3, w_1, w_2; m_2: w_3, w_2, w_1; m_3: w_2, w_3, w_1; and with these preference rankings of the women for the men, from highest to lowest: w_1: m_3, m_2, m_1; w_2: m_1, m_3, m_2; w_3: m_3, m_2, m_1.

∗32. Show that the deferred acceptance algorithm given in the preamble to Exercise 65 of Section 3.1, always produces a male optimal and female pessimal matching.

33. Define what it means for a matching to be female optimal and for a matching to be male pessimal.

∗34. Show that when woman do the proposing in the deferred acceptance algorithm, the matching produced is female optimal and male pessimal.

In Exercises 35 and 36 we consider variations on the problem of finding stable matchings of men and women described in the preamble to Exercise 65 in Section 3.1.

∗35. In this exercise we consider matching problems where there may be different numbers of men and women, so that it is impossible to match everyone with a member of the opposite gender.

   a) Extend the definition of a stable matching from that given in the preamble to Exercise 64 in Section 3.1 to cover the case where there are unequal numbers of men and women. Avoid all cases where a man and a woman would prefer each other to their current situation, including those involving unmatched people. (Assume that an unmatched person prefers a match with a member of the opposite gender to remaining unmatched.)

   b) Adapt the deferred acceptance algorithm to find stable matchings, using the definition of stable matchings from part (a), when there are different numbers of men and women.

   c) Prove that all matchings produced by the algorithm from part (b) are stable, according to the definition from part (a).

∗36. In this exercise we consider matching problems where some man-woman pairs are not allowed.

   a) Extend the definition of a stable matching to cover the situation where there are the same number of men and women, but certain pairs of men and women are forbidden. Avoid all cases where a man and a woman would prefer each other to their current situation, including those involving unmatched people.

   b) Adapt the deferred acceptance algorithm to find stable matchings when there are the same number of men and women, but certain man-woman pairs are forbidden. Be sure to consider people who are unmatched at the end of the algorithm. (Assume that an unmatched person prefers a match with a member of the opposite gender who is not a forbidden partner to remaining unmatched.)

   c) Prove that all matchings produced by the algorithm from (b) are stable, according to the definition in part (a).

Exercises 37–40 deal with the problem of scheduling n jobs on a single processor. To complete job j, the processor must run job j for time t_j without interruption. Each job has a deadline d_j. If we start job j at time s_j, it will be completed at time $e_j = s_j + t_j$. The **lateness** of the job measures how long it finishes after its deadline, that is, the lateness of job j is $\max(0, e_j - d_j)$. We wish to devise a greedy algorithm that minimizes the maximum lateness of a job among the n jobs.

37. Suppose we have five jobs with specified required times and deadlines: $t_1 = 25$, $d_1 = 50$; $t_2 = 15$, $d_2 = 60$; $t_3 = 20$, $d_3 = 60$; $t_4 = 5$, $d_4 = 55$; $t_5 = 10$, $d_5 = 75$. Find the maximum lateness of any job when the jobs are scheduled in this order (and they start at time 0): Job 3, Job 1, Job 4, Job 2, Job 5. Answer the same question for the schedule Job 5, Job 4, Job 3, Job 1, Job 2.

38. The **slackness** of a job requiring time t and with deadline d is $d - t$, the difference between its deadline and the time it requires. Find an example that shows that scheduling jobs by increasing slackness does not always yield a schedule with the smallest possible maximum lateness.

39. Find an example that shows that scheduling jobs in order of increasing time required does not always yield a schedule with the smallest possible maximum lateness.

*40. Prove that scheduling jobs in order of increasing deadlines always produces a schedule that minimizes the maximum lateness of a job. [*Hint:* First show that for a schedule to be optimal, jobs must be scheduled with no idle time between them and so that no job is scheduled before another with an earlier deadline.]

41. Suppose that we have a knapsack with total capacity of W kg. We also have n items where item j has mass w_j. The **knapsack problem** asks for a subset of these n items with the largest possible total mass not exceeding W.

a) Devise a brute-force algorithm for solving the knapsack problem.

b) Solve the knapsack problem when the capacity of the knapsack is 18 kg and there are five items: a 5-kg sleeping bag, an 8-kg tent, a 7-kg food pack, a 4-kg container of water, and an 11-kg portable stove.

In Exercises 42–46 we will study the problem of load balancing. The input to the problem is a collection of p processors and n jobs, t_j is the time required to run job j, jobs run without interruption on a single machine until finished, and a processor can run only one job at a time. The **load** L_k of processor k is the sum over all jobs assigned to processor k of the times required to run these jobs. The **makespan** is the maximum load over all the p processors. The load balancing problem asks for an assignment of jobs to processors to minimize the makespan.

42. Suppose we have three processors and five jobs requiring times $t_1 = 3$, $t_2 = 5$, $t_3 = 4$, $t_4 = 7$, and $t_5 = 8$. Solve the load balancing problem for this input by finding the assignment of the five jobs to the three processors that minimizes the makespan.

43. Suppose that L^* is the minimum makespan when p processors are given n jobs, where t_j is the time required to run job j.

a) Show that $L^* \geq \max_{j=1,2,\ldots,n} t_j$.

b) Show that $L^* \geq \frac{1}{p} \sum_{j=1}^{n} t_j$.

44. Write out in pseudocode the greedy algorithm that goes through the jobs in order and assigns each job to the processor with the smallest load at that point in the algorithm.

45. Run the algorithm from Exercise 44 on the input given in Exercise 42.

An **approximation algorithm** for an optimization problem produces a solution guaranteed to be close to an optimal solution. More precisely, suppose that the optimization problem asks for an input S that minimizes $F(X)$ where F is some function of the input X. If an algorithm always finds an input T with $F(T) \leq cF(S)$, where c is a fixed positive real number, the algorithm is called a c-**approximation algorithm** for the problem.

*46. Prove that the algorithm from Exercise 44 is a 2-approximation algorithm for the load balancing problem. [*Hint:* Use both parts of Exercise 43.]

Computer Projects

Write programs with these inputs and outputs.

1. Given a list of n integers, find the largest integer in the list.

2. Given a list of n integers, find the first and last occurrences of the largest integer in the list.

3. Given a list of n distinct integers, determine the position of an integer in the list using a linear search.

4. Given an ordered list of n distinct integers, determine the position of an integer in the list using a binary search.

5. Given a list of n integers, sort them using a bubble sort.

6. Given a list of n integers, sort them using an insertion sort.

7. Given two strings of characters use the naive string matching algorithm to determine whether the shorter string occurs in the longer string.

8. Given an integer n, use the cashier's algorithm to find the change for n cents using quarters, dimes, nickels, and pennies.

9. Given the starting and ending times of n talks, use the appropriate greedy algorithm to schedule the most talks possible in a single lecture hall.

10. Given an ordered list of n integers and an integer x in the list, find the number of comparisons used to determine the position of x in the list using a linear search and using a binary search.

11. Given a list of integers, determine the number of comparisons used by the bubble sort and by the insertion sort to sort this list.

Computations and Explorations

Use a computational program or programs you have written to do these exercises.

1. We know that n^b is $O(d^n)$ when b and d are positive numbers with $d \geq 2$. Give values of the constants C and k such that $n^b \leq Cd^n$ whenever $x > k$ for each of these sets of values: $b = 10$, $d = 2$; $b = 20$, $d = 3$; $b = 1000$, $d = 7$.

2. Compute the change for different values of *n* with coins of different denominations using the cashier's algorithm and determine whether the smallest number of coins was used. Can you find conditions so that the cashier's algorithm is guaranteed to use the fewest coins possible?

3. Using a generator of random orderings of the integers $1, 2, \ldots, n$, find the number of comparisons used by the bubble sort, insertion sort, binary insertion sort, and selection sort to sort these integers.

4. Collect experimental evidence comparing the number of comparisons used by the sorting algorithms in Question 3 when used to sort sequences that only have a small fraction of terms out of order.

*5. Write a program that animates the progress of all the sorting algorithms in Question 3 when given the numbers from 1 to 100 in random order.

Writing Projects

Respond to these with essays using outside sources.

1. Examine the history of the word *algorithm* and describe the use of this word in early writings.

2. Look up Bachmann's original introduction of big-*O* notation. Explain how he and others have used this notation.

3. Explain how sorting algorithms can be classified into a taxonomy based on the underlying principle on which they are based.

4. Describe the radix sort algorithm.

5. Describe some of the different algorithms for string matching.

6. Describe some of the different applications of string matching in bioinformatics.

7. Describe the historic trends in how quickly processors can perform operations and use these trends to estimate how quickly processors will be able to perform operations in the next 20 years.

8. Develop a detailed list of algorithmic paradigms and provide examples using each of these paradigms.

9. Explain what the Turing Award is and describe the criteria used to select winners. List six past winners of the award and why they received the award.

10. Describe what is meant by a parallel algorithm. Explain how the pseudocode used in this book can be extended to handle parallel algorithms.

11. Explain how the complexity of parallel algorithms can be measured. Give some examples to illustrate this concept, showing how a parallel algorithm can work more quickly than one that does not operate in parallel.

12. Describe six different NP-complete problems.

13. Demonstrate how one of the many different NP-complete problems can be reduced to the satisfiability problem.