Chapter 4

ALGORITHMS

An **algorithm** is a step-by-step method of solving some problem. Such an approach to problem-solving is not new; indeed, the word "algorithm" derives from the name of the ninth-century Persian mathematician al-Khowārizmī. Today, "algorithm" typically refers to a solution that can be executed by a computer. In this book, we will be concerned primarily with algorithms that can be executed by a "traditional" computer, that is, a computer, such as a personal computer, with a single processor that executes instructions step-by-step.

After introducing algorithms and providing several examples, we turn to the analysis of algorithms, which refers to the time and space required to execute algorithms. We conclude by discussing recursive algorithms—algorithms that refer to themselves.

4.1 Introduction

Algorithms typically have the following characteristics:

- ***Input*** The algorithm receives *input*.
- ***Output*** The algorithm produces *output*.
- ***Precision*** The steps are precisely stated.
- ***Determinism*** The intermediate results of each step of execution are unique and are determined only by the inputs and the results of the preceding steps.
- ***Finiteness*** The algorithm *terminates;* that is, it stops after finitely many instructions have been executed.
- ***Correctness*** The output produced by the algorithm is correct; that is, the algorithm correctly solves the problem.
- ***Generality*** The algorithm applies to a set of inputs.

As an example, consider the following algorithm that finds the maximum of three numbers a, b, and c:

1. $large = a$.
2. If $b > large$, then $large = b$.
3. If $c > large$, then $large = c$.

(As explained in Appendix C, $=$ is the assignment operator.)

The idea of the algorithm is to inspect the numbers one by one and copy the largest value seen into a variable *large*. At the conclusion of the algorithm, *large* will then be equal to the largest of the three numbers.

We show how the preceding algorithm executes for some specific values of a, b, and c. Such a simulation is called a **trace.** First suppose that $a = 1$, $b = 5$, and $c = 3$. At line 1, we set *large* to a (1). At line 2, $b > large$ $(5 > 1)$ is true, so we set *large* to b (5). At line 3, $c > large$ $(3 > 5)$ is false, so we do nothing. At this point *large* is 5, the largest of a, b, and c.

Suppose that $a = 6$, $b = 1$, and $c = 9$. At line 1, we set *large* to a (6). At line 2, $b > large$ $(1 > 6)$ is false, so we do nothing. At line 3, $c > large$ $(9 > 6)$ is true, so we set *large* to 9. At this point *large* is 9, the largest of a, b, and c.

We verify that our example algorithm has the properties set forth at the beginning of this section.

The algorithm receives the three values a, b, and c as input and produces the value *large* as output.

The steps of the algorithm are stated sufficiently precisely so that the algorithm could be written in a programming language and executed by a computer.

Given values for the input, each intermediate step of an algorithm produces a unique result. For example, given the values $a = 1$, $b = 5$, and $c = 3$, at line 2, *large* will be set to 5 regardless of who executes the algorithm.

The algorithm terminates after finitely many steps (three steps) correctly answering the given question (find the largest of the three values input).

The algorithm is general; it can find the largest value of *any* three numbers.

Our description of what an algorithm is will suffice for our needs in this book. However, it should be noted that it is possible to give a precise, mathematical definition of "algorithm" (see the Notes for Chapter 12).

Although ordinary language is sometimes adequate to specify an algorithm, most mathematicians and computer scientists prefer **pseudocode** because of its precision, structure, and universality. Pseudocode is so named because it resembles the actual code of computer languages such as C++ and Java. There are many versions of pseudocode. Unlike actual computer languages, which must be concerned about semicolons, uppercase and lowercase letters, special words, and so on, any version of pseudocode is acceptable as long as its instructions are unambiguous. Our pseudocode is described in detail in Appendix C.

As our first example of an algorithm written in pseudocode, we rewrite the first algorithm in this section, which finds the maximum of three numbers.

Algorithm 4.1.1 **Finding the Maximum of Three Numbers**

This algorithm finds the largest of the numbers a, b, and c.

Input: a, b, c

Output: *large* (the largest of a, b, and c)

```
1.  max3(a, b, c) {
2.      large = a
3.      if (b > large) // if b is larger than large, update large
4.          large = b
5.      if (c > large) // if c is larger than large, update large
6.          large = c
7.      return large
8.  }
```

Our algorithms consist of a title, a brief description of the algorithm, the input to and output from the algorithm, and the functions containing the instructions of the algorithm. Algorithm 4.1.1 consists of a single function. To make it convenient to refer to individual lines within a function, we sometimes number some of the lines. The function in Algorithm 4.1.1 has eight numbered lines.

When the function in Algorithm 4.1.1 executes, at line 2 we set *large* to *a*. At line 3, *b* and *large* are compared. If *b* is greater than *large,* we execute line 4

$$large = b$$

but if *b* is not greater than *large,* we skip to line 5. At line 5, *c* and *large* are compared. If *c* is greater than *large,* we execute line 6

$$large = c$$

but if *c* is not greater than *large,* we skip to line 7. Thus when we arrive at line 7, *large* will correctly hold the largest of *a*, *b*, and *c*.

At line 7 we return the value of *large,* which is equal to the largest of the numbers *a*, *b*, and *c*, to the invoker of the function and terminate the function. Algorithm 4.1.1 has correctly found the largest of three numbers.

The method of Algorithm 4.1.1 can be used to find the largest value in a sequence.

Algorithm 4.1.2 **Finding the Maximum Value in a Sequence**

This algorithm finds the largest of the numbers $s_1, \ldots, s_n$.

```
  Input:  s, n
 Output:  large (the largest value in the sequence s)

max(s, n) {
   large = s_1
   for i = 2 to n
      if (s_i > large)
         large = s_i
   return large
}
```

We verify that Algorithm 4.1.2 is correct by proving that

$$large \text{ is the largest value in the subsequence } s_1, \ldots, s_i \qquad \textbf{(4.1.1)}$$

is a loop invariant using induction on *i*.

For the Basis Step ($i = 1$), we note that just before the for loop begins executing, *large* is set to s_1; so *large* is surely the largest value in the subsequence s_1.

Assume that *large* is the largest value in the subsequence $s_1, \ldots, s_i$. If $i < n$ is true (so that the for loop body executes again), *i* becomes $i + 1$. Suppose first that $s_{i+1} > large$. It then follows that s_{i+1} is the largest value in the subsequence $s_1, \ldots, s_i, s_{i+1}$. In this case, the algorithm assigns *large* the value s_{i+1}. Now *large* is equal to the largest value in the subsequence $s_1, \ldots, s_i, s_{i+1}$. Suppose next that $s_{i+1} \leq large$. It then follows that *large* is the largest value in the subsequence $s_1, \ldots, s_i, s_{i+1}$. In this case, the algorithm does *not* change the value of *large;* thus, *large* is the largest value in the

subsequence $s_1, \ldots, s_i, s_{i+1}$. We have proved the Inductive Step. Therefore, (4.1.1) is a loop invariant.

The for loop terminates when $i = n$. Because (4.1.1) is a loop invariant, at this point *large* is the largest value in the sequence $s_1, \ldots, s_n$. Therefore, Algorithm 4.1.2 is correct.

4.1 Problem-Solving Tips

To construct an algorithm, it is often helpful to assume that you are in the middle of the algorithm and part of the problem has been solved. For example, in finding the largest element in a sequence $s_1, \ldots, s_n$ (Algorithm 4.1.2), it was helpful to *assume* that we had already found the largest element *large* in the subsequence $s_1, \ldots, s_i$. Then, all we had to do was look at the next element s_{i+1} and, if s_{i+1} was larger than *large,* we simply updated *large*. If s_{i+1} was not larger than *large,* we did not modify *large*. Iterating this procedure yields the algorithm. These observations also led to the loop invariant (4.1.1) which allowed us to *prove* that Algorithm 4.1.2 is correct.

4.1 Review Exercises

1. What is an algorithm?
2. Describe the following properties an algorithm typically has: input, output, precision, determinism, finiteness, correctness, and generality.
3. What is a trace of an algorithm?
4. What are the advantages of pseudocode over ordinary text in writing an algorithm?
5. How do algorithms relate to pseudocode functions?

4.1 Exercises

1. Consult the instructions for connecting a DVD or Blu-ray player to a TV. Which properties of an algorithm—input, output, precision, determinism, finiteness, correctness, generality—are present? Which properties are lacking?

2. Consult the instructions for adding a contact to a cell phone. Which properties of an algorithm—input, output, precision, determinism, finiteness, correctness, generality—are present? Which properties are lacking?

3. Goldbach's conjecture states that every even number greater than 2 is the sum of two prime numbers. Here is a proposed algorithm that checks whether Goldbach's conjecture is true:

1. Let $n = 4$.
2. If n is not the sum of two primes, output "no" and stop.
3. Else increase n by 2 and continue with step 2.
4. Output "yes" and stop.

Which properties of an algorithm—input, output, precision, determinism, finiteness, correctness, generality—does this proposed algorithm have? Do any of them depend on the truth of Goldbach's conjecture (which mathematicians have not yet settled)?

4. Write an algorithm that finds the smallest element among a, b, and c.

5. Write an algorithm that finds the second-smallest element among a, b, and c. Assume that the values of a, b, and c are distinct.

6. Write an algorithm that returns the smallest value in the sequence $s_1, \ldots, s_n$.

7. Write an algorithm that returns the largest and second-largest values in the sequence $s_1, \ldots, s_n$. Assume that $n > 1$ and the values in the sequence are distinct.

8. Write an algorithm that returns the smallest and second-smallest values in the sequence $s_1, \ldots, s_n$. Assume that $n > 1$ and the values in the sequence are distinct.

9. Write an algorithm that outputs the smallest and largest values in the sequence $s_1, \ldots, s_n$.

10. Write an algorithm that returns the index of the first occurrence of the largest element in the sequence $s_1, \ldots, s_n$. *Example:* If the sequence is 6.2, 8.9, 4.2, 8.9, the algorithm returns the value 2.

11. Write an algorithm that returns the index of the last occurrence of the largest element in the sequence $s_1, \ldots, s_n$.

Example: If the sequence is 6.2, 8.9, 4.2, 8.9, the algorithm returns the value 4.

12. Write an algorithm that returns the sum of the sequence of numbers $s_1, \ldots, s_n$.

13. Write an algorithm that returns the index of the first item that is less than its predecessor in the sequence $s_1, \ldots, s_n$. If s is in nondecreasing order, the algorithm returns the value 0. *Example:* If the sequence is

    AMY BRUNO ELIE DAN ZEKE,

    the algorithm returns the value 4.

14. Write an algorithm that returns the index of the first item that is greater than its predecessor in the sequence $s_1, \ldots, s_n$. If s is in nonincreasing order, the algorithm returns the value 0. *Example:* If the sequence is

    AMY BRUNO ELIE DAN ZEKE,

    the algorithm returns the value 2.

15. Write an algorithm that reverses the sequence $s_1, \ldots, s_n$. *Example:* If the sequence is

    AMY BRUNO ELIE,

    the reversed sequence is

    ELIE BRUNO AMY.

16. Write the standard method of adding two positive decimal integers, taught in elementary schools, as an algorithm.

17. Write an algorithm that receives as input the $n \times n$ matrix A and outputs the transpose A^T.

18. Write an algorithm that receives as input the matrix of a relation R and tests whether R is reflexive.

19. Write an algorithm that receives as input the matrix of a relation R and tests whether R is symmetric.

20. Write an algorithm that receives as input the matrix of a relation R and tests whether R is transitive.

21. Write an algorithm that receives as input the matrix of a relation R and tests whether R is antisymmetric.

22. Write an algorithm that receives as input the matrix of a relation R and tests whether R is a function.

23. Write an algorithm that receives as input the matrix of a relation R and produces as output the matrix of the inverse relation R^{-1}.

24. Write an algorithm that receives as input the matrices of relations R_1 and R_2 and produces as output the matrix of the composition $R_2 \circ R_1$.

25. Write an algorithm whose input is a sequence $s_1, \ldots, s_n$ and a value x. (Assume that all the values are real numbers.) The algorithm returns true if $s_i + s_j = x$, for some $i \neq j$, and false otherwise. *Example:* If the input sequence is 2, 12, 6, 14 and $x = 26$, the algorithm returns true because $12 + 14 = 26$. If the input sequence is 2, 12, 6, 14 and $x = 4$, the algorithm returns false because no *distinct* pair in the sequence sums to 4.

4.2 Examples of Algorithms

Algorithms have been devised to solve many problems. In this section, we give examples of several useful algorithms. Throughout the remainder of the book, we will investigate many additional algorithms.

Searching

A large amount of computer time is devoted to searching. When a teller looks for a record in a bank, a computer program searches for the record. Looking for a solution to a puzzle or for an optimal move in a game can be stated as a searching problem. Using a search engine on the web is another example of a searching problem. Looking for specified text in a document when running a word processor is yet another example of a searching problem. We discuss an algorithm to solve the text-searching problem.

Suppose that we are given text t (e.g., a word processor document) and we want to find the first occurrence of pattern p in t (e.g., we want to find the first occurrence of the string $p =$ "Nova Scotia" in t) or determine that p does not occur in t. We index the characters in t starting at 1. One approach to searching for p is to check whether p occurs at index 1 in t. If so, we stop, having found the first occurrence of p in t. If not, we check

whether p occurs at index 2 in t. If so, we stop, having found the first occurrence of p in t. If not, we next check whether p occurs at index 3 in t, and so on.

We state the text-searching algorithm as Algorithm 4.2.1.

Algorithm 4.2.1 **Text Search**

This algorithm searches for an occurrence of the pattern p in text t. It returns the smallest index i such that p occurs in t starting at index i. If p does not occur in t, it returns 0.

```
  Input:  p (indexed from 1 to m), m, t (indexed from 1 to n), n
 Output:  i

text_search(p, m, t, n) {
   for i = 1 to n − m + 1 {
      j = 1

      // i is the index in t of the first character of the substring
      // to compare with p, and j is the index in p

      // the while loop compares t_i ··· t_{i+m−1} and p_1 ··· p_m
      while (t_{i+j−1} == p_j) {
         j = j + 1
         if (j > m)
            return i
      }
   }
   return 0
}
```

The variable i marks the index in t of the first character of the substring to compare with p. The algorithm first tries $i = 1$, then $i = 2$, and so on. Index $n - m + 1$ is the last possible value for i since, at this point, the string $t_{n-m+1}t_{n-m+2} \cdots t_m$ has length exactly m.

After the value of i is set, the while loop compares $t_i \cdots t_{i+m-1}$ and $p_1 \cdots p_m$. If the characters match,

$$t_{i+j-1} == p_j$$

j is incremented

$$j = j + 1$$

and the next characters are compared. If j is $m + 1$, all m characters have matched and we have found p at index i in t. In this case, the algorithm returns i:

```
if (j > m)
   return i
```

If the for loop runs to completion, a match was never found; so the algorithm returns 0.

Example 4.2.2 Figure 4.2.1 shows a trace of Algorithm 4.2.1 where we are searching for the pattern "001" in the text "010001".

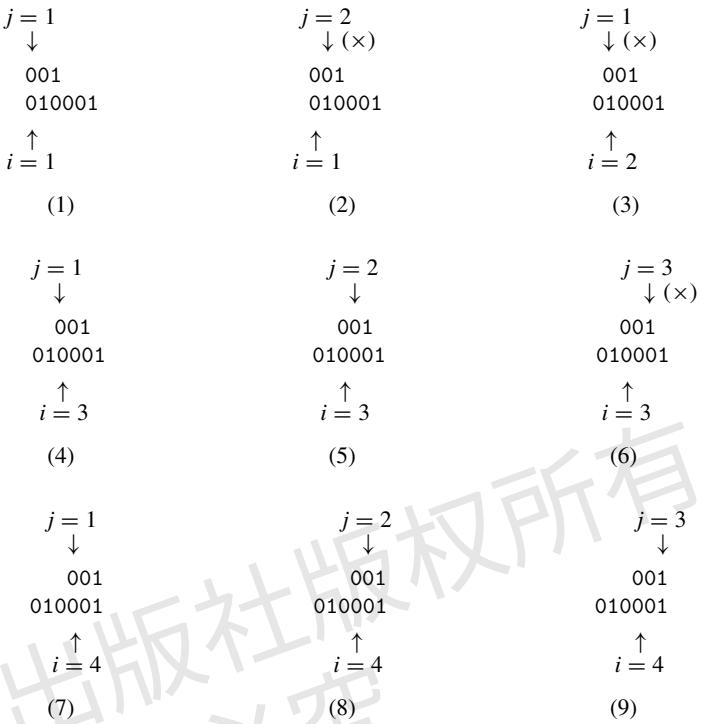


Figure 4.2.1 Searching for "001" in "010001" using Algorithm 4.2.1. The cross (×) in steps (2), (3), and (6) marks a mismatch. ◀

Sorting

To **sort** a sequence is to put it in some specified order. If we have a sequence of names, we might want the sequence sorted in nondecreasing order according to dictionary order. For example, if the sequence is

Jones, Johnson, Appel, Zamora, Chu,

after sorting the sequence in nondecreasing order, we would obtain

Appel, Chu, Johnson, Jones, Zamora.

A major advantage of using a sorted sequence rather than an unsorted sequence is that it is much easier to find a particular item. Imagine trying to find the phone number of a particular individual in the New York City telephone book if the names were not sorted!

Many sorting algorithms have been devised (see, e.g., [Knuth, 1998b]). Which algorithm is preferred in a particular situation depends on factors such as the size of the data and how the data are represented. We discuss **insertion sort,** which is one of the fastest algorithms for sorting small sequences (less than 50 or so items).

We assume that the input to insertion sort is $s_1, \ldots, s_n$ and that the goal is to sort the data in nondecreasing order. At the ith iteration of insertion sort, the first part of the sequence $s_1, \ldots, s_i$ will have been rearranged so that it is sorted. (We will explain

shortly how $s_1, \ldots, s_i$ gets sorted.) Insertion sort then *inserts* s_{i+1} in $s_1, \ldots, s_i$ so that $s_1, \ldots, s_i, s_{i+1}$ is sorted.

For example, suppose that $i = 4$ and $s_1, \ldots, s_4$ is

8	13	20	27

If s_5 is 16, after it is inserted, $s_1, \ldots, s_5$ becomes

8	13	16	20	27

Notice that 20 and 27, being *greater than* 16, move one index to the right to make room for 16. Thus the "insert" part of the algorithm is: Beginning at the right of the sorted subsequence, move an element one index to the right if it is greater than the element to insert. Repeat until reaching the first index or encountering an element that is less than or equal to the element to insert.

For example, to insert 16 in

8	13	20	27

we first compare 16 and 27. Since 27 is greater than 16, 27 moves one index to the right:

8	13	20		27

We next compare 16 with 20. Since 20 is greater than 16, 20 moves one index to the right:

8	13		20	27

We next compare 16 with 13. Since 13 is less than or equal to 16, we insert (i.e., copy) 16 to the third index:

8	13	16	20	27

This subsequence is now sorted.

Having explained the key idea of insertion sort, we now complete the explanation of the algorithm. Insertion sort begins by inserting s_2 into the subsequence s_1. Note that s_1 by itself is sorted! Now s_1, s_2 is sorted. Next, insertion sort inserts s_3 into the now-sorted subsequence s_1, s_2. Now s_1, s_2, s_3 is sorted. This procedure continues until insertion sort inserts s_n into the sorted subsequence $s_1, \ldots, s_{n-1}$. Now the entire sequence $s_1, \ldots, s_n$ is sorted. We obtain the following algorithm.

Algorithm 4.2.3 **Insertion Sort**

This algorithm sorts the sequence $s_1, \ldots, s_n$ in nondecreasing order.

Input: s, n
Output: s (sorted)

```
insertion_sort(s, n) {
    for i = 2 to n {
        val = s_i // save s_i so it can be inserted into the correct place
        j = i − 1
        // if val < s_j, move s_j right to make room for s_i
        while (j ≥ 1 ∧ val < s_j) {
            s_{j+1} = s_j
            j = j − 1
        }
        s_{j+1} = val // insert val
    }
}
```

We leave proving that Algorithm 4.2.3 is correct as an exercise (see Exercise 14).

Time and Space for Algorithms

It is important to know or be able to estimate the time (e.g., the number of steps) and space (e.g., the number of variables, length of the sequences involved) required by algorithms. Knowing the time and space required by algorithms allows us to compare algorithms that solve the same problem. For example, if one algorithm takes n steps to solve a problem and another algorithm takes n^2 steps to solve the same problem, we would surely prefer the first algorithm, assuming that the space requirements are acceptable. In Section 4.3, we will give the technical definitions that allow us to make rigorous statements about the time and space required by algorithms.

The for loop in Algorithm 4.2.3 always executes $n - 1$ times, but the number of times that the while loop executes for a particular value of i depends on the input. Thus, even for a fixed size n, the time required by Algorithm 4.2.3 depends on the input. For example, if the input sequence is already sorted in nondecreasing order,

$$val < s_j \tag{4.2.1}$$

will always be false and the body of the while loop will never be executed. We call this time the **best-case time.**

On the other hand, if the sequence is sorted in *decreasing* order, (4.2.1) will always be true and the while loop will execute the maximum number of times. (The while loop will execute $i - 1$ times during the ith iteration of the for loop.) We call this time the **worst-case time.**

Randomized Algorithms

It is occasionally necessary to relax the requirements of an algorithm stated in Section 4.1. Many algorithms currently in use are not general, deterministic, or even finite. An operating system (e.g., Windows), for example, is better thought of as a program that never terminates rather than as a finite program with input and output. Algorithms written for more than one processor, whether for a multiprocessor machine or for a distributed

environment (such as the internet), are rarely deterministic, for example, because of different execution speeds of the processors. Also, many practical problems are too difficult to be solved efficiently, and compromises either in generality or correctness are necessary. As an illustration, we present an example that shows the usefulness of allowing an algorithm to make random decisions, thereby violating the requirement of determinism.

A **randomized algorithm** does *not* require that the intermediate results of each step of execution be uniquely defined and depend only on the inputs and results of the preceding steps. By definition, when a randomized algorithm executes, at some points it makes *random* choices. In practice, a pseudorandom number generator is used (see Example 3.1.16).

We shall assume the existence of a function $rand(i, j)$, which returns a random integer between the integers i and j, inclusive. As an example, we describe a randomized algorithm that shuffles a sequence of numbers. More precisely, it inputs a sequence $a_1, \ldots, a_n$ and moves the numbers to random positions. Major bridge tournaments use computer programs to shuffle the cards.

The algorithm first swaps (i.e., interchanges the values of) a_1 and $a_{rand(1,n)}$. At this point, the value of a_1 might be equal to *any* one of the original values in the sequence. Next, the algorithm swaps a_2 and $a_{rand(2,n)}$. Now the value of a_2 might be equal to *any* of the remaining values in the sequence. The algorithm continues in this manner until it swaps a_{n-1} and $a_{rand(n-1,n)}$. Now the entire sequence is shuffled.

Algorithm 4.2.4 **Shuffle**

This algorithm shuffles the values in the sequence

$$a_1, \ldots, a_n.$$

Input: a, n

Output: a (shuffled)

```
shuffle(a, n) {
  for i = 1 to n − 1
    swap(a_i, a_rand(i,n))
}
```

Example 4.2.5 Suppose that the sequence a

17	9	5	23	21

is input to *shuffle*. We first swap a_i and a_j, where $i = 1$ and $j = rand(1, 5)$. If $j = 3$, after the swap we have

5	9	17	23	21
↑ i		↑ j		

Next, $i = 2$. If $j = rand(2, 5) = 5$, after the swap we have

5	21	17	23	9
	↑ i			↑ j

Next, $i = 3$. If $j = rand(3, 5) = 3$, the sequence does not change.
Finally, $i = 4$. If $j = rand(4, 5) = 5$, after the swap we have

5	21	17	9	23
			↑ i	↑ j

Notice that the output (i.e., the rearranged sequence) depends on the random choices made by the random number generator. ◀

Randomized algorithms can be used to search for nonrandom goals. For example, a person searching for the exit in a maze could randomly make a choice at each intersection. Of course, such an algorithm might not terminate (because of bad choices at the intersections). In Chapter 8, Graph Theory, we will present a randomized algorithm that searches for a particular structure in a graph (see Algorithm 8.3.10).

4.2 Problem-Solving Tips

Again, we emphasize that to construct an algorithm, it is often helpful to assume that you are in the middle of the algorithm and that part of the problem has been solved. In insertion sort (Algorithm 4.2.3), it was helpful to *assume* that the subsequence $s_1, \ldots, s_i$ was sorted. Then, all we had to do was insert the next element s_{i+1} in the proper place. Iterating this procedure yields the algorithm. These observations lead to a loop invariant that can be used to prove that Algorithm 4.2.3 is correct (see Exercise 14).

4.2 Review Exercises

1. Give examples of searching problems.
2. What is text searching?
3. Describe, in words, an algorithm that solves the text-searching problem.
4. What does it mean to sort a sequence?
5. Give an example that illustrates why we might want to sort a sequence.
6. Describe insertion sort in words.
7. What do we mean by the time and space required by an algorithm?
8. Why is it useful to know or be able to estimate the time and space required by an algorithm?
9. Why is it sometimes necessary to relax the requirements of an algorithm as stated in Section 4.1?
10. What is a randomized algorithm?
11. Which requirements of an algorithm as stated in Section 4.1 does a randomized algorithm violate?
12. Describe the shuffle algorithm in words.
13. Give an application of the shuffle algorithm.

4.2 Exercises

1. Trace Algorithm 4.2.1 for the input t = "balalaika" and p = "bala".
2. Trace Algorithm 4.2.1 for the input t = "balalaika" and p = "lai".
3. Trace Algorithm 4.2.1 for the input t = "000000000" and p = "001".
4. Trace Algorithm 4.2.3 for the input 34, 20, 144, 55.
5. Trace Algorithm 4.2.3 for the input 34, 20, 19, 5.
6. Trace Algorithm 4.2.3 for the input 34, 55, 144, 259.
7. Trace Algorithm 4.2.3 for the input 34, 34, 34, 34.
8. Trace Algorithm 4.2.4 for the input 34, 57, 72, 101, 135. Assume that the values of *rand* are

$$rand(1, 5) = 5, \quad rand(2, 5) = 4,$$
$$rand(3, 5) = 3, \quad rand(4, 5) = 5.$$

9. Trace Algorithm 4.2.4 for the input 34, 57, 72, 101, 135.

Assume that the values of *rand* are

$$rand(1, 5) = 2, \quad rand(2, 5) = 5,$$
$$rand(3, 5) = 3, \quad rand(4, 5) = 4.$$

10. Trace Algorithm 4.2.4 for the input 34, 57, 72, 101, 135. Assume that the values of *rand* are

$$rand(1, 5) = 5, \quad rand(2, 5) = 5,$$
$$rand(3, 5) = 4, \quad rand(4, 5) = 4.$$

11. Is it possible that Algorithm 4.2.4 sorts the input 67, 32, 6, 89, 52 in increasing order? If so, show possible values for *rand* that perform the sort. It not, prove that Algorithm 4.2.4 cannot sort this input in increasing order.

12. Is it possible that Algorithm 4.2.4 sorts the input 67, 32, 6, 89, 52 in decreasing order? If so, show possible values for *rand* that perform the sort. It not, prove that Algorithm 4.2.4 cannot sort this input in decreasing order.

13. Prove that Algorithm 4.2.1 is correct.

14. Prove that Algorithm 4.2.3 is correct.

15. Write an algorithm that returns the index of the first occurrence of the value *key* in the sequence $s_1, \ldots, s_n$. If *key* is not in the sequence, the algorithm returns the value 0. *Example:* If the sequence is 12, 11, 12, 23 and *key* is 12, the algorithm returns the value 1.

16. Write an algorithm that returns the index of the last occurrence of the value *key* in the sequence $s_1, \ldots, s_n$. If *key* is not in the sequence, the algorithm returns the value 0. *Example:* If the sequence is 12, 11, 12, 23 and *key* is 12, the algorithm returns the value 3.

17. Write an algorithm whose input is a sequence $s_1, \ldots, s_n$ sorted in nondecreasing order and a value x. (Assume that all the values are real numbers.) The algorithm inserts x into the sequence so that the resulting sequence is sorted in nondecreasing order. *Example:* If the input sequence is 2, 6, 12, 14 and $x = 5$, the resulting sequence is 2, 5, 6, 12, 14.

18. Modify Algorithm 4.2.1 so that it finds *all* occurrences of p in t.

19. Describe best-case input for Algorithm 4.2.1.

20. Describe worst-case input for Algorithm 4.2.1.

21. Modify Algorithm 4.2.3 so that it sorts the sequence $s_1, \ldots, s_n$ in *nonincreasing* order.

22. The *selection sort* algorithm sorts the sequence $s_1, \ldots, s_n$ in nondecreasing order by first finding the smallest item, say s_i, and placing it first by swapping s_1 and s_i. It then finds the smallest item in $s_2, \ldots, s_n$, again say s_i, and places it second by swapping s_2 and s_i. It continues until the sequence is sorted. Write selection sort in pseudocode.

23. Trace selection sort (see Exercise 22) for the input of Exercises 4–7.

24. Show that the time for selection sort (see Exercise 22) is the same for all inputs of size n.

4.3 Analysis of Algorithms

A computer program, even though derived from a correct algorithm, might be useless for certain types of input because the time needed to run the program or the space needed to hold the data, program variables, and so on, is too great. **Analysis of an algorithm** refers to the process of deriving estimates for the time and space needed to execute the algorithm. In this section we deal with the problem of estimating the time required to execute algorithms.

Suppose that we are given a set X of n elements, some labeled "red" and some labeled "black," and we want to find the number of subsets of X that contain at least one red item. Suppose we construct an algorithm that examines all subsets of X and counts those that contain at least one red item and then implement this algorithm as a computer program. Since a set that has n elements has 2^n subsets (see Theorem 2.4.6), the program would require at least 2^n units of time to execute. It does not matter what the units of time are—2^n grows so fast as n increases (see Table 4.3.1) that, except for small values of n, it would be impractical to run the program.

Determining the performance parameters of a computer program is a difficult task and depends on a number of factors such as the computer that is being used, the way the data are represented, and how the program is translated into machine instructions. Although precise estimates of the execution time of a program must take such factors into account, useful information can be obtained by analyzing the time of the underlying algorithm.

The time needed to execute an algorithm is a function of the input. Usually, it is difficult to obtain an explicit formula for this function, and we settle for less. Instead of dealing directly with the input, we use parameters that characterize the *size* of the input.

TABLE 4.3.1 ■ Time to Execute an Algorithm if One Step Takes 1 Microsecond to Execute. $\lg n$ Denotes $\log_2 n$ (the logarithm of n to base 2)

Number of Steps to Termination for Input of Size n	*Time to Execute if n =*				
	3	6	9	12	
1	10^{-6} sec	10^{-6} sec	10^{-6} sec	10^{-6} sec	
$\lg \lg n$	10^{-6} sec	10^{-6} sec	2×10^{-6} sec	2×10^{-6} sec	
$\lg n$	2×10^{-6} sec	3×10^{-6} sec	3×10^{-6} sec	4×10^{-6} sec	
n	3×10^{-6} sec	6×10^{-6} sec	9×10^{-6} sec	10^{-5} sec	
$n \lg n$	5×10^{-6} sec	2×10^{-5} sec	3×10^{-5} sec	4×10^{-5} sec	
n^2	9×10^{-6} sec	4×10^{-5} sec	8×10^{-5} sec	10^{-4} sec	
n^3	3×10^{-5} sec	2×10^{-4} sec	7×10^{-4} sec	2×10^{-3} sec	
2^n	8×10^{-6} sec	6×10^{-5} sec	5×10^{-4} sec	4×10^{-3} sec	
	50	100	1000	10^5	10^6
1	10^{-6} sec	10^{-6} sec	10^{-6} sec	10^{-6} sec	10^{-6} sec
$\lg \lg n$	2×10^{-6} sec	3×10^{-6} sec	3×10^{-6} sec	4×10^{-6} sec	4×10^{-6} sec
$\lg n$	6×10^{-6} sec	7×10^{-6} sec	10^{-5} sec	2×10^{-5} sec	2×10^{-5} sec
n	5×10^{-5} sec	10^{-4} sec	10^{-3} sec	0.1 sec	1 sec
$n \lg n$	3×10^{-4} sec	7×10^{-4} sec	10^{-2} sec	2 sec	20 sec
n^2	3×10^{-3} sec	0.01 sec	1 sec	3 hr	12 days
n^3	0.13 sec	1 sec	16.7 min	32 yr	31,710 yr
2^n	36 yr	4×10^{16} yr	3×10^{287} yr	3×10^{30089} yr	3×10^{301016} yr

For example, if the input is a set containing n elements, we would say that the size of the input is n. We can ask for the minimum time needed to execute the algorithm among all inputs of size n. This time is called the **best-case time** for inputs of size n. We can also ask for the maximum time needed to execute the algorithm among all inputs of size n. This time is called the **worst-case time** for inputs of size n. Another important case is **average-case time**—the average time needed to execute the algorithm over some finite set of inputs all of size n.

Since we are primarily concerned with *estimating* the time of an algorithm rather than computing its exact time, as long as we count some fundamental, dominating steps of the algorithm, we will obtain a useful measure of the time. For example, if the principal activity of an algorithm is making comparisons, as might happen in a sorting routine, we might count the number of comparisons. As another example, if an algorithm consists of a single loop whose body executes in at most C steps, for some constant C, we might count the number of iterations of the loop.

Example 4.3.1 A reasonable definition of the size of input for Algorithm 4.1.2 that finds the largest value in a finite sequence is the number of elements in the input sequence. A reasonable definition of the execution time is the number of iterations of the while loop. With these definitions, the worst-case, best-case, and average-case times for Algorithm 4.1.2 for input of size n are each $n - 1$ since the loop is always executed $n - 1$ times. ◀

Usually we are less interested in the exact best-case or worst-case time required for an algorithm to execute than we are in how the best-case or worst-case time grows as the size of the input increases. For example, suppose that the worst-case time of an algorithm is

$$t(n) = 60n^2 + 5n + 1$$

for input of size n. For large n, the term $60n^2$ is approximately equal to $t(n)$ (see Table 4.3.2). In this sense, $t(n)$ grows like $60n^2$.

TABLE 4.3.2 ■ Comparing Growth of $t(n)$ with $60n^2$

n	$t(n) = 60n^2 + 5n + 1$	$60n^2$
10	6,051	6,000
100	600,501	600,000
1,000	60,005,001	60,000,000
10,000	6,000,050,001	6,000,000,000

If $t(n)$ measures the worst-case time for input of size n in seconds, then

$$T(n) = n^2 + \frac{5}{60}n + \frac{1}{60}$$

measures the worst-case time for input of size n in minutes. Now this change of units does not affect how the worst-case time grows as the size of the input increases but only the units in which we measure the worst-case time for input of size n. Thus when we describe how the best-case or worst-case time grows as the size of the input increases, we not only seek the dominant term [e.g., $60n^2$ in the formula for $t(n)$], but we also may ignore constant coefficients. Under these assumptions, $t(n)$ grows like n^2 as n increases. We say that $t(n)$ is of **order** n^2 and write $t(n) = \Theta(n^2)$, which is read "$t(n)$ is theta of n^2." The basic idea is to replace an expression, such as $t(n) = 60n^2 + 5n + 1$, with a simpler expression, such as n^2, that grows at the same rate as $t(n)$. The formal definitions follow.

Definition 4.3.2 ▶ Let f and g be functions with domain $\{1, 2, 3, \ldots\}$.

We write

$$f(n) = O(g(n))$$

and say that $f(n)$ *is of order at most* $g(n)$ or $f(n)$ *is big oh of* $g(n)$ if there exists a positive constant C_1 such that

$$|f(n)| \leq C_1|g(n)|$$

for all but finitely many positive integers n.

We write

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

and say that $f(n)$ *is of order at least* $g(n)$ or $f(n)$ *is omega of* $g(n)$ if there exists a positive constant C_2 such that

$$|f(n)| \geq C_2|g(n)|$$

for all but finitely many positive integers n.

We write

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

and say that $f(n)$ *is of order* $g(n)$ or $f(n)$ *is theta of* $g(n)$ if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$. ◀

Go Online
For more on these order notations, see goo.gl/ZwpPlu

Definition 4.3.2 can be loosely paraphrased as follows: $f(n) = O(g(n))$ if, except for a constant factor and a finite number of exceptions, f is bounded above by g. We also say that g is an **asymptotic upper bound** for f. Similarly, $f(n) = \Omega(g(n))$ if, except for

a constant factor and a finite number of exceptions, f is bounded below by g. We also say that g is an **asymptotic lower bound** for f. Also, $f(n) = \Theta(g(n))$ if, except for constant factors and a finite number of exceptions, f is bounded above and below by g. We also say that g is an **asymptotic tight bound** for f.

According to Definition 4.3.2, if $f(n) = O(g(n))$, all that we can conclude is that, except for a constant factor and a finite number of exceptions, f is bounded *above* by g, so g grows at least as fast as f. For example, if $f(n) = n$ and $g(n) = 2^n$, then $f(n) = O(g(n))$, but g grows considerably faster than f. The statement $f(n) = O(g(n))$ says nothing about a *lower* bound for f. On the other hand, if $f(n) = \Theta(g(n))$, we can draw the conclusion that, except for constant factors and a finite number of exceptions, f is bounded *above* and *below* by g, so f and g grow at the same rate. Notice that $n = O(2^n)$, but $n \neq \Theta(2^n)$.

Example 4.3.3 Since

$$60n^2 + 5n + 1 \leq 60n^2 + 5n^2 + n^2 = 66n^2 \qquad \text{for all } n \geq 1,$$

we may take $C_1 = 66$ in Definition 4.3.2 to obtain

$$60n^2 + 5n + 1 = O(n^2).$$

Since

$$60n^2 + 5n + 1 \geq 60n^2 \qquad \text{for all } n \geq 1,$$

we may take $C_2 = 60$ in Definition 4.3.2 to obtain

$$60n^2 + 5n + 1 = \Omega(n^2).$$

Since $60n^2 + 5n + 1 = O(n^2)$ and $60n^2 + 5n + 1 = \Omega(n^2)$,

$$60n^2 + 5n + 1 = \Theta(n^2).$$

◀

The method of Example 4.3.3 can be used to show that a polynomial in n of degree k with nonnegative coefficients is $\Theta(n^k)$. [In fact, *any* polynomial in n of degree k is $\Theta(n^k)$, even if some of its coefficients are negative. To prove this more general result, the method of Example 4.3.3 has to be modified.]

Theorem 4.3.4 Let

$$p(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0$$

be a polynomial in n of degree k, where each a_i is nonnegative. Then

$$p(n) = \Theta(n^k).$$

Proof We first show that $p(n) = O(n^k)$. Let

$$C_1 = a_k + a_{k-1} + \cdots + a_1 + a_0.$$

Then, for all n,

$$\begin{aligned} p(n) &= a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0 \\ &\le a_k n^k + a_{k-1} n^k + \cdots + a_1 n^k + a_0 n^k \\ &= (a_k + a_{k-1} + \cdots + a_1 + a_0) n^k = C_1 n^k. \end{aligned}$$

Therefore, $p(n) = O(n^k)$.

Next, we show that $p(n) = \Omega(n^k)$. For all n,

$$p(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0 \ge a_k n^k = C_2 n^k,$$

where $C_2 = a_k$. Therefore, $p(n) = \Omega(n^k)$.

Since $p(n) = O(n^k)$ and $p(n) = \Omega(n^k)$, $p(n) = \Theta(n^k)$. ◀

Example 4.3.5 In this book, we let $\lg n$ denote $\log_2 n$ (the logarithm of n to the base 2). Since $\lg n < n$ for all $n \ge 1$ (see Figure 4.3.1†),

$$2n + 3\lg n < 2n + 3n = 5n \qquad \text{for all } n \ge 1.$$

Thus,

$$2n + 3\lg n = O(n).$$

Also, $2n + 3\lg n \ge 2n$, for all $n \ge 1$. Thus,

$$2n + 3\lg n = \Omega(n).$$

Therefore,

$$2n + 3\lg n = \Theta(n).$$ ◀

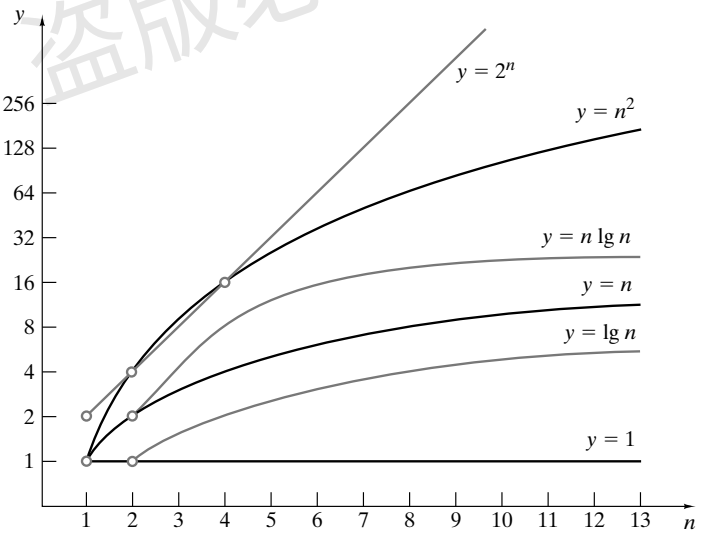


Figure 4.3.1 Growth of some common functions.

†In Figure 4.3.1, the spacing on the y-axis is proportional to the logarithm of the number rather than to the number itself so that we can plot large y-values against smaller x-values. This y-axis scale is called a *logarithmic scale*. On the standard xy-graph, $y = n$ would be a straight line; here it is curved.

Example 4.3.6 If $a > 1$ and $b > 1$ (to ensure that $\log_b a > 0$), by the change-of-base formula for logarithms [Theorem B.37(e)],

$$\log_b n = \log_b a \log_a n \qquad \text{for all } n \geq 1.$$

Therefore,

$$\log_b n \leq C \log_a n \qquad \text{for all } n \geq 1,$$

where $C = \log_b a$. Thus, $\log_b n = O(\log_a n)$.

Also,

$$\log_b n \geq C \log_a n \qquad \text{for all } n \geq 1;$$

so $\log_b n = \Omega(\log_a n)$. Since $\log_b n = O(\log_a n)$ and $\log_b n = \Omega(\log_a n)$, we conclude that $\log_b n = \Theta(\log_a n)$.

Because $\log_b n = \Theta(\log_a n)$, when using asymptotic notation we need not worry about which number is used as the base for the logarithm function (as long as the base is greater than 1). For this reason, we sometimes simply write log without specifying the base. ◀

Example 4.3.7 If we replace each integer $1, 2, \ldots, n$ by n in the sum $1 + 2 + \cdots + n$, the sum does not decrease and we have

$$1 + 2 + \cdots + n \leq n + n + \cdots + n = n \cdot n = n^2 \qquad \text{for all } n \geq 1. \tag{4.3.1}$$

It follows that

$$1 + 2 + \cdots + n = O(n^2).$$

To obtain a lower bound, we might imitate the preceding argument and replace each integer $1, 2, \ldots, n$ by 1 in the sum $1 + 2 + \cdots + n$ to obtain

$$1 + 2 + \cdots + n \geq 1 + 1 + \cdots + 1 = n \qquad \text{for all } n \geq 1.$$

In this case we conclude that

$$1 + 2 + \cdots + n = \Omega(n),$$

and while the preceding expression is true, we cannot deduce a Θ-estimate for $1 + 2 + \cdots + n$, since the upper bound n^2 and lower bound n are not equal. We must be craftier in deriving a lower bound.

One way to get a sharper lower bound is to argue as in the previous paragraph, but first throw away approximately the first half of the terms, to obtain

$$\begin{aligned} 1 + 2 + \cdots + n &\geq \lceil (n+1)/2 \rceil + \cdots + (n-1) + n \\ &\geq \lceil (n+1)/2 \rceil + \cdots + \lceil (n+1)/2 \rceil + \lceil (n+1)/2 \rceil \\ &= \lceil n/2 \rceil \lceil (n+1)/2 \rceil \geq (n/2)(n/2) = \frac{n^2}{4} \end{aligned} \tag{4.3.2}$$

for all $n \geq 1$. We can now conclude that

$$1 + 2 + \cdots + n = \Omega(n^2).$$

Therefore,

$$1 + 2 + \cdots + n = \Theta(n^2).$$ ◀

Example 4.3.8 If k is a positive integer and, as in Example 4.3.7, we replace each integer $1, 2, \ldots, n$ by n, we have

$$1^k + 2^k + \cdots + n^k \le n^k + n^k + \cdots + n^k = n \cdot n^k = n^{k+1}$$

for all $n \ge 1$; hence

$$1^k + 2^k + \cdots + n^k = O(n^{k+1}).$$

We can also obtain a lower bound as in Example 4.3.7:

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

for all $n \ge 1$. We conclude that

$$1^k + 2^k + \cdots + n^k = \Omega(n^{k+1}),$$

and hence

$$1^k + 2^k + \cdots + n^k = \Theta(n^{k+1}).$$ ◀

Notice the difference between the polynomial

$$a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0$$

in Theorem 4.3.4 and the expression

$$1^k + 2^k + \cdots + n^k$$

in Example 4.3.8. A polynomial has a fixed number of terms, whereas the number of terms in the expression in Example 4.3.8 is dependent on the value of n. Furthermore, the polynomial in Theorem 4.3.4 is $\Theta(n^k)$, but the expression in Example 4.3.8 is $\Theta(n^{k+1})$.

Our next example gives a theta notation for $\lg n!$.

Example 4.3.9 Use an argument similar to that in Example 4.3.7, to show that $\lg n! = \Theta(n \lg n)$.

SOLUTION By properties of logarithms, we have

$$\lg n! = \lg n + \lg(n-1) + \cdots + \lg 2 + \lg 1$$

for all $n \ge 1$. Since lg is an increasing function,

$$\lg n + \lg(n-1) + \cdots + \lg 2 + \lg 1 \le \lg n + \lg n + \cdots + \lg n + \lg n = n \lg n$$

for all $n \ge 1$. We conclude that $\lg n! = O(n \lg n)$.

For all $n \ge 4$, we have

$$\begin{aligned} \lg n + \lg(n-1) + \cdots + \lg 2 + \lg 1 &\ge \lg n + \lg(n-1) + \cdots + \lg\lceil (n+1)/2 \rceil \\ &\ge \lg\lceil (n+1)/2 \rceil + \cdots + \lg\lceil (n+1)/2 \rceil \\ &= \lceil n/2 \rceil \lg\lceil (n+1)/2 \rceil \end{aligned}$$

$$\begin{aligned}
&\geq (n/2)\lg(n/2)\\
&= (n/2)[\lg n - \lg 2]\\
&= (n/2)[(\lg n)/2 + ((\lg n)/2 - 1)]\\
&\geq (n/2)(\lg n)/2\\
&= n\lg n/4
\end{aligned}$$

[since $(\lg n)/2 \geq 1$ for all $n \geq 4$]. Therefore, $\lg n! = \Omega(n \lg n)$. It follows that $\lg n! = \Theta(n \lg n)$. ◀

Example 4.3.10 Show that if $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$, then $f(n) = \Theta(h(n))$.

SOLUTION Because $f(n) = \Theta(g(n))$, there are constants C_1 and C_2 such that

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

for all but finitely many positive integers n. Because $g(n) = \Theta(h(n))$, there are constants C_3 and C_4 such that

$$C_3|h(n)| \leq |g(n)| \leq C_4|h(n)|$$

for all but finitely many positive integers n. Therefore,

$$C_1C_3|h(n)| \leq C_1|g(n)| \leq |f(n)| \leq C_2|g(n)| \leq C_2C_4|h(n)|$$

for all but finitely many positive integers n. It follows that $f(n) = \Theta(h(n))$. ◀

We next define what it means for the best-case, worst-case, or average-case time of an algorithm to be of order at most $g(n)$.

Definition 4.3.11 ▶ If an algorithm requires $t(n)$ units of time to terminate in the best case for an input of size n and

$$t(n) = O(g(n)),$$

we say that the *best-case time required by the algorithm is of order at most* $g(n)$ or that the *best-case time required by the algorithm* is $O(g(n))$.

If an algorithm requires $t(n)$ units of time to terminate in the worst case for an input of size n and

$$t(n) = O(g(n)),$$

we say that the *worst-case time required by the algorithm is of order at most* $g(n)$ or that the *worst-case time required by the algorithm* is $O(g(n))$.

If an algorithm requires $t(n)$ units of time to terminate in the average case for an input of size n and

$$t(n) = O(g(n)),$$

we say that the *average-case time required by the algorithm is of order at most* $g(n)$ or that the *average-case time required by the algorithm* is $O(g(n))$. ◀

By replacing O by Ω and "at most" by "at least" in Definition 4.3.11, we obtain the definition of what it means for the best-case, worst-case, or average-case time of an algorithm to be of order at least $g(n)$. If the best-case time required by an algorithm is $O(g(n))$ and $\Omega(g(n))$, we say that the best-case time required by the algorithm is

$\Theta(g(n))$. An analogous definition applies to the worst-case and average-case times of an algorithm.

Example 4.3.12 Suppose that an algorithm is known to take $60n^2+5n+1$ units of time to terminate in the worst case for inputs of size n. We showed in Example 4.3.3 that $60n^2+5n+1 = \Theta(n^2)$. Thus the worst-case time required by this algorithm is $\Theta(n^2)$. ◀

Example 4.3.13 Find a theta notation in terms of n for the number of times the statement $x = x + 1$ is executed.

```
1.  for i = 1 to n
2.      for j = 1 to i
3.          x = x + 1
```

SOLUTION First, i is set to 1 and, as j runs from 1 to 1, line 3 is executed one time. Next, i is set to 2 and, as j runs from 1 to 2, line 3 is executed two times, and so on. Thus the total number of times line 3 is executed is (see Example 4.3.7) $1+2+\cdots+n = \Theta(n^2)$. Thus a theta notation for the number of times the statement $x = x+1$ is executed is $\Theta(n^2)$. ◀

Example 4.3.14 Find a theta notation in terms of n for the number of times the statement $x = x + 1$ is executed:

```
1.  i = n
2.  while (i ≥ 1) {
3.      x = x + 1
4.      i = ⌊i/2⌋
5.  }
```

SOLUTION First, we examine some specific cases. Because of the floor function, the computations are simplified if n is a power of 2. Consider, for example, the case $n = 8$. At line 1, i is set to 8. At line 2, the condition $i \geq 1$ is true. At line 3, we execute the statement $x = x + 1$ the first time. At line 4, i is reset to 4 and we return to line 2.

At line 2, the condition $i \geq 1$ is again true. At line 3, we execute the statement $x = x + 1$ the second time. At line 4, i is reset to 2 and we return to line 2.

At line 2, the condition $i \geq 1$ is again true. At line 3, we execute the statement $x = x + 1$ the third time. At line 4, i is reset to 1 and we return to line 2.

At line 2, the condition $i \geq 1$ is again true. At line 3, we execute the statement $x = x + 1$ the fourth time. At line 4, i is reset to 0 and we return to line 2.

This time at line 2, the condition $i \geq 1$ is false. The statement $x = x + 1$ was executed four times.

Now suppose that n is 16. At line 1, i is set to 16. At line 2, the condition $i \geq 1$ is true. At line 3, we execute the statement $x = x + 1$ the first time. At line 4, i is reset to 8 and we return to line 2. Now execution proceeds as before; the statement $x = x + 1$ is executed four more times, for a total of five times.

Similarly, if n is 32, the statement $x = x + 1$ is executed a total of six times.

A pattern is emerging. Each time the initial value of n is doubled, the statement $x = x + 1$ is executed one more time. More precisely, if $n = 2^k$, the statement $x = x + 1$ is executed $k + 1$ times. Since k is the exponent for 2, $k = \lg n$. Thus if $n = 2^k$, the statement $x = x + 1$ is executed $1 + \lg n$ times.

If n is an arbitrary positive integer (not necessarily a power of 2), it lies between two powers of 2; that is, for some $k \geq 1$,

$$2^{k-1} \leq n < 2^k.$$

We use induction on k to show that in this case the statement $x = x + 1$ is executed k times.

If $k = 1$, we have

$$1 = 2^{1-1} \leq n < 2^1 = 2.$$

Therefore, n is 1. In this case, the statement $x = x + 1$ is executed once. Thus the Basis Step is proved.

Now suppose that if n satisfies

$$2^{k-1} \leq n < 2^k, \tag{4.3.3}$$

the statement $x = x + 1$ is executed k times. We must show that if n satisfies

$$2^k \leq n < 2^{k+1}, \tag{4.3.4}$$

the statement $x = x + 1$ is executed $k + 1$ times.

Suppose that n satisfies (4.3.4). At line 1, i is set to n. At line 2, the condition $i \geq 1$ is true. At line 3, we execute the statement $x = x + 1$ the first time. At line 4, i is reset to $\lfloor n/2 \rfloor$ and we return to line 2. Notice that

$$2^{k-1} \leq n/2 < 2^k.$$

Because 2^{k-1} is an integer, we must also have

$$2^{k-1} \leq \lfloor n/2 \rfloor < 2^k.$$

By the inductive assumption (4.3.3), the statement $x = x + 1$ is executed k more times, for a total of $k + 1$ times. The Inductive Step is complete. Therefore, if n satisfies (4.3.3), the statement $x = x + 1$ is executed k times.

Suppose that n satisfies (4.3.3). Taking logarithms to the base 2, we have

$$k - 1 \leq \lg n < k.$$

Therefore, k, the number of times the statement $x = x + 1$ is executed, satisfies

$$\lg n < k \leq 1 + \lg n.$$

Because k is an integer, we must have $k \leq 1 + \lfloor \lg n \rfloor$. Furthermore, $\lfloor \lg n \rfloor < k$. It follows from the last two inequalities that $k = 1 + \lfloor \lg n \rfloor$. Since $1 + \lfloor \lg n \rfloor = \Theta(\lg n)$, a theta notation for the number of times the statement $x = x + 1$ is executed is $\Theta(\lg n)$. ◀

Many algorithms are based on the idea of repeated halving. Example 4.3.14 shows that for size n, repeated halving takes time $\Theta(\lg n)$. Of course, the algorithm may do work in addition to the halving that will increase the overall time.

Example 4.3.15 Find a theta notation in terms of n for the number of times the statement $x = x + 1$ is executed.

```
1.  j = n
2.  while (j ≥ 1) {
3.      for i = 1 to j
4.          x = x + 1
5.      j = ⌊j/2⌋
6.  }
```

SOLUTION Let $t(n)$ denote the number of times we execute the statement $x = x + 1$. The first time we arrive at the body of the while loop, the statement $x = x+1$ is executed n times. Therefore $t(n) \geq n$ for all $n \geq 1$ and $t(n) = \Omega(n)$.

Next we derive a big oh notation for $t(n)$. After j is set to n, we arrive at the while loop for the first time. The statement $x = x+1$ is executed n times. At line 5, j is replaced by $\lfloor n/2 \rfloor$; hence $j \leq n/2$. If $j \geq 1$, we will execute $x = x + 1$ at most $n/2$ additional times in the next iteration of the while loop, and so on. If we let k denote the number of times we execute the body of the while loop, the number of times we execute $x = x + 1$ is at most

$$n + \frac{n}{2} + \frac{n}{4} + \cdots + \frac{n}{2^{k-1}}.$$

This geometric sum (see Example 2.4.4) is equal to

$$\frac{n\left(1 - \frac{1}{2^k}\right)}{1 - \frac{1}{2}}.$$

Now

$$t(n) \leq \frac{n\left(1 - \frac{1}{2^k}\right)}{1 - \frac{1}{2}} = 2n\left(1 - \frac{1}{2^k}\right) \leq 2n \qquad \text{for all } n \geq 1,$$

so $t(n) = O(n)$. Thus a theta notation for the number of times we execute $x = x + 1$ is $\Theta(n)$. ◀

Example 4.3.16 Determine, in theta notation, the best-case, worst-case, and average-case times required to execute Algorithm 4.3.17, which follows. Assume that the input size is n and that the run time of the algorithm is the number of comparisons made at line 3. Also, assume that the $n + 1$ possibilities of *key* being at any particular position in the sequence or not being in the sequence are equally likely.

SOLUTION The best-case time can be analyzed as follows. If $s_1 = key$, line 3 is executed once. Thus the best-case time of Algorithm 4.3.17 is $\Theta(1)$.

The worst-case time of Algorithm 4.3.17 is analyzed as follows. If *key* is not in the sequence, line 3 will be executed n times, so the worst-case time of Algorithm 4.3.17 is $\Theta(n)$.

Finally, consider the average-case time of Algorithm 4.3.17. If *key* is found at the ith position, line 3 is executed i times; if *key* is not in the sequence, line 3 is executed n times. Thus the average number of times line 3 is executed is

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

Now

$$\begin{aligned} \frac{(1 + 2 + \cdots + n) + n}{n + 1} &\leq \frac{n^2 + n}{n + 1} \qquad \text{by (4.3.1)} \\ &= \frac{n(n + 1)}{n + 1} = n. \end{aligned}$$

Therefore, the average-case time of Algorithm 4.3.17 is $O(n)$. Also,

$$\frac{(1+2+\cdots+n)+n}{n+1} \geq \frac{n^2/4+n}{n+1} \qquad \text{by (4.3.2)}$$
$$\geq \frac{n^2/4+n/4}{n+1} = \frac{n}{4}.$$

Therefore the average-case time of Algorithm 4.3.17 is $\Omega(n)$. Thus the average-case time of Algorithm 4.3.17 is $\Theta(n)$. For this algorithm, the worst-case and average-case times are both $\Theta(n)$. ◀

Algorithm 4.3.17 **Searching an Unordered Sequence**

Given the sequence $s_1, \ldots, s_n$ and a value *key*, this algorithm returns the index of *key*. If *key* is not found, the algorithm returns 0.

Input: $s_1, s_2, \ldots, s_n, n$, and *key* (the value to search for)
Output: The index of *key*, or if *key* is not found, 0

```
1.  linear_search(s, n, key) {
2.      for i = 1 to n
3.          if (key == s_i)
4.              return i // successful search
5.      return 0 // unsuccessful search
6.  }
```

Example 4.3.18 **Matrix Multiplication and Transitive Relations** If A is a matrix, we let A_{ij} denote the entry in row i, column j. The product of $n \times n$ matrices A and B (i.e., A and B have n rows and n columns) is defined as the $n \times n$ matrix C, where

$$C_{ij} = \sum_{k=1}^{n} A_{ik}B_{kj} \qquad 1 \leq i \leq n,\ 1 \leq j \leq n.$$

Algorithm 4.3.19, which computes the matrix product, is a direct translation of the preceding definition. Because of the nested loops, it runs in time $\Theta(n^3)$.

Recall (see the discussion following Theorem 3.5.6) that we can test whether a relation R on an n-element set is transitive by squaring its adjacency matrix, say A, and then comparing A^2 with A. The relation R is transitive if and only if, whenever the entry in row i, column j in A^2 is nonzero, the corresponding entry in A is also nonzero. Since there are n^2 entries in A and A^2, the worst-case time to compare the entries is $\Theta(n^2)$. Using Algorithm 4.3.19 to compute A^2 requires time $\Theta(n^3)$. Therefore, the overall time to test whether a relation on an n-element set is transitive, using Algorithm 4.3.19 to compute A^2, is $\Theta(n^3)$.

For many years it was believed that the minimum time to multiply two $n \times n$ matrices was $\Theta(n^3)$; thus it was quite a surprise when a more efficient algorithm was discovered. Strassen's algorithm (see [Johnsonbaugh: Section 5.4]) to multiply two $n \times n$ matrices runs in time $\Theta(n^{\lg 7})$. Since $\lg 7$ is approximately 2.807, Strassen's algorithm runs in time approximately $\Theta(n^{2.807})$, which is asymptotically faster than Algorithm 4.3.19. An algorithm by Coppersmith and Winograd (see [Coppersmith]) runs in time $\Theta(n^{2.376})$ and, so, is even asymptotically faster than Strassen's algorithm. Since the product of two $n \times n$ matrices contains n^2 terms, *any* algorithm that multiplies two $n \times n$ matrices requires time at least $\Omega(n^2)$. At the present time, no sharper lower bound is known. ◀

Algorithm 4.3.19 **Matrix Multiplication**

This algorithm computes the product C of the $n \times n$ matrices A and B directly from the definition of matrix multiplication.

Input: A, B, n

Output: C, the product of A and B

```
matrix_product(A, B, n) {
    for i = 1 to n
        for j = 1 to n {
            C_ij = 0
            for k = 1 to n
                C_ij = C_ij + A_ik * B_kj
            }
    return C
}
```

The constants that are suppressed in the theta notation may be important. Even if for any input of size n, algorithm A requires exactly $C_1 n$ time units and algorithm B requires exactly $C_2 n^2$ time units, for certain sizes of inputs algorithm B may be superior. For example, suppose that for any input of size n, algorithm A requires $300n$ units of time and algorithm B requires $5n^2$ units of time. For an input size of $n = 5$, algorithm A requires 1500 units of time and algorithm B requires 125 units of time, and thus algorithm B is faster. Of course, for sufficiently large inputs, algorithm A is considerably faster than algorithm B.

TABLE 4.3.3 ■ Common Growth Functions

Theta Form	*Name*
$\Theta(1)$	Constant
$\Theta(\lg \lg n)$	Log log
$\Theta(\lg n)$	Log
$\Theta(n)$	Linear
$\Theta(n \lg n)$	$n \log n$
$\Theta(n^2)$	Quadratic
$\Theta(n^3)$	Cubic
$\Theta(n^k), k \geq 1$	Polynomial
$\Theta(c^n), c > 1$	Exponential
$\Theta(n!)$	Factorial

A real-world example of the importance of constants in the theta notation is provided by matrix multiplication. Algorithm 4.3.19, which runs in time $\Theta(n^3)$, is typically used to multiply matrices even though the Strassen and Coppersmith-Winograd algorithms (see Example 4.3.18), which run in times $\Theta(n^{2.807})$ and $\Theta(n^{2.376})$, are asymptotically faster. The constants in the Strassen and Coppersmith-Winograd algorithms are so large that they are faster than Algorithm 4.3.19 only for very large matrices.

Certain growth functions occur so often that they are given special names, as shown in Table 4.3.3. The functions in Table 4.3.3, with the exception of $\Theta(n^k)$, are arranged so that if $\Theta(f(n))$ is above $\Theta(g(n))$, then $f(n) \leq g(n)$ for all but finitely many positive integers n. Thus, if algorithms A and B have run times that are $\Theta(f(n))$ and $\Theta(g(n))$, respectively, and $\Theta(f(n))$ is above $\Theta(g(n))$ in Table 4.3.3, then algorithm A is more time-efficient than algorithm B for sufficiently large inputs.

It is important to develop some feeling for the relative sizes of the functions in Table 4.3.3. In Figure 4.3.1 we have graphed some of these functions. Another way to develop some appreciation for the relative sizes of the functions $f(n)$ in Table 4.3.3 is to determine how long it would take an algorithm to terminate whose run time is exactly $f(n)$. For this purpose, let us assume that we have a computer that can execute one step in 1 microsecond (10^{-6} sec). Table 4.3.1 shows the execution times, under this assumption, for various input sizes. Notice that it is practical to implement an algorithm that requires 2^n steps for an input of size n only for very small input sizes. Algorithms requiring n^2 or n^3 steps also become impractical to implement, but for relatively larger input sizes. Also, notice the dramatic improvement that results when we move from n^2 steps to $n \lg n$ steps.

A problem that has a worst-case polynomial-time algorithm is considered to have a "good" algorithm; the interpretation is that such a problem has an efficient solution. Such problems are called **feasible** or **tractable.** Of course, if the worst-case time to solve the problem is proportional to a high-degree polynomial, the problem can still take a long time to solve. Fortunately, in many important cases, the polynomial bound has small degree.

A problem that does not have a worst-case polynomial-time algorithm is said to be **intractable.** Any algorithm, if there is one, that solves an intractable problem is guaranteed to take a long time to execute in the worst case, even for modest sizes of the input.

Certain problems are so hard that they have no algorithms at all. A problem for which there is no algorithm is said to be **unsolvable.** A large number of problems are known to be unsolvable, some of considerable practical importance. One of the earliest problems to be proved unsolvable is the **halting problem:** Given an arbitrary program and a set of inputs, will the program eventually halt?

A large number of solvable problems have an as yet undetermined status; they are thought to be intractable, but none of them has been proved to be intractable. (Most of these problems belong to the class of NP-complete problems; see [Johnsonbaugh] for details.) An example of an NP-complete problem is:

> Given a collection $\mathcal{C}$ of finite sets and a positive integer $k \leq |\mathcal{C}|$, does $\mathcal{C}$ contain at least k mutually disjoint sets?

Other NP-complete problems include the traveling-salesperson problem and the Hamiltonian-cycle problem (see Section 8.3).

NP-complete problems have efficient (i.e., polynomial-time) algorithms to check whether a proposed solution is, in fact, a solution. For example, given a collection C of finite sets and k sets in C, it is easy and fast to check whether the k sets are mutually disjoint. (Just check each pair of sets!) On the other hand, no NP-complete problem is known to have an efficient algorithm. For example, given a collection C of finite sets, *finding* k mutually disjoint sets in C is, in general, difficult and time consuming. NP-complete problems also have the property that if any one of them has a polynomial-time algorithm, then *all* NP-complete problems have polynomial-time algorithms.

4.3 Problem-Solving Tips

- To derive a big oh notation for an expression $f(n)$ directly, you must find a constant C_1 and a simple expression $g(n)$ (e.g., n, $n \lg n$, n^2) such that $|f(n)| \leq C_1|g(n)|$ for all but finitely many n. Remember you're trying to derive an *inequality, not* an equality, so you can replace terms in $f(n)$ with other terms if the result is *larger* (see, e.g., Example 4.3.3).
- To derive an omega notation for an expression $f(n)$ directly, you must find a constant C_2 and a simple expression $g(n)$ such that $|f(n)| \geq C_2|g(n)|$ for all but finitely many n. Again, you're trying to derive an *inequality* so you can replace terms in $f(n)$ with other terms if the result is *smaller* (again, see Example 4.3.3).
- To derive a theta notation, you must derive both big oh and omega notations.
- Another way to derive big oh, omega, and theta estimates is to use known results:

Expression	*Name*	*Estimate*	*Reference*
$a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0$	Polynomial	$\Theta(n^k)$	Theorem 4.3.4
$1 + 2 + \cdots + n$	Arithmetic Sum (Case $k = 1$ for Next Entry)	$\Theta(n^2)$	Example 4.3.7
$1^k + 2^k + \cdots + n^k$	Sum of Powers	$\Theta(n^{k+1})$	Example 4.3.8
$\lg n!$	log n Factorial	$\Theta(n \lg n)$	Example 4.3.9

- To derive an asymptotic estimate for the time of an algorithm, count the number of steps $t(n)$ required by the algorithm, and then derive an estimate for $t(n)$ as described previously. Algorithms typically contain loops, in which case, deriving $t(n)$ requires counting the number of iterations of the loops.

4.3 Review Exercises

1. To what does "analysis of algorithms" refer?
2. What is the worst-case time of an algorithm?
3. What is the best-case time of an algorithm?
4. What is the average-case time of an algorithm?
5. Define $f(n) = O(g(n))$. What is this notation called?
6. Give an intuitive interpretation of how f and g are related if $f(n) = O(g(n))$.
7. Define $f(n) = \Omega(g(n))$. What is this notation called?
8. Give an intuitive interpretation of how f and g are related if $f(n) = \Omega(g(n))$.
9. Define $f(n) = \Theta(g(n))$. What is this notation called?
10. Give an intuitive interpretation of how f and g are related if $f(n) = \Theta(g(n))$.

4.3 Exercises

Select a theta notation from Table 4.3.3 for each expression in Exercises 1–13.

1. $6n + 1$
2. $2n^2 + 1$
3. $6n^3 + 12n^2 + 1$
4. $3n^2 + 2n \lg n$
5. $2 \lg n + 4n + 3n \lg n$
6. $6n^6 + n + 4$
7. $2 + 4 + 6 + \cdots + 2n$
8. $(6n + 1)^2$
9. $(6n + 4)(1 + \lg n)$
10. $\dfrac{(n + 1)(n + 3)}{n + 2}$
11. $\dfrac{(n^2 + \lg n)(n + 1)}{n + n^2}$
12. $2 + 4 + 8 + 16 + \cdots + 2^n$
13. $\lg[(2n)!]$

In Exercises 14–16, select a theta notation for $f(n) + g(n)$.

14. $f(n) = \Theta(1), \quad g(n) = \Theta(n^2)$
15. $f(n) = 6n^3 + 2n^2 + 4, \quad g(n) = \Theta(n \lg n)$
16. $f(n) = \Theta(n^{3/2}), \quad g(n) = \Theta(n^{5/2})$

In Exercises 17–26, select a theta notation from among

$$\Theta(1), \quad \Theta(\lg n), \quad \Theta(n), \quad \Theta(n \lg n),$$
$$\Theta(n^2), \quad \Theta(n^3), \quad \Theta(2^n), \quad \textit{or} \quad \Theta(n!)$$

for the number of times the statement $x = x + 1$ is executed.

17.
```
for i = 1 to 2n
    x = x + 1
```

18.
```
i = 1
while (i ≤ 2n) {
    x = x + 1
    i = i + 2
}
```

19.
```
for i = 1 to n
    for j = 1 to n
        x = x + 1
```

20.
```
for i = 1 to 2n
    for j = 1 to n
        x = x + 1
```

21.
```
for i = 1 to n
    for j = 1 to ⌊i/2⌋
        x = x + 1
```

22.
```
for i = 1 to n
    for j = 1 to n
        for k = 1 to n
            x = x + 1
```

23.
```
for i = 1 to n
    for j = 1 to n
        for k = 1 to i
            x = x + 1
```

24.
```
for i = 1 to n
    for j = 1 to i
        for k = 1 to j
            x = x + 1
```

25.
```
j = n
while (j ≥ 1) {
    for i = 1 to j
        x = x + 1
    j = ⌊j/3⌋
}
```

26.
```
i = n
while (i ≥ 1) {
    for j = 1 to n
        x = x + 1
    i = ⌊i/2⌋
}
```

27. Find a theta notation for the number of times the statement $x = x + 1$ is executed.

```
i = 2
while (i < n) {
    i = i^2
    x = x + 1
}
```

28. Let $t(n)$ be the total number of times that i is incremented and j is decremented in the following pseudocode, where $a_1, a_2, \ldots$ is a sequence of real numbers.

```
i = 1
j = n
while (i < j) {
    while (i < j ∧ a_i < 0)
        i = i + 1
    while (i < j ∧ a_j ≥ 0)
        j = j − 1
    if (i < j)
        swap(a_i, a_j)
}
```

Find a theta notation for $t(n)$.

29. Find a theta notation for the worst-case time required by the following algorithm:

```
iskey(s, n, key) {
    for i = 1 to n − 1
        for j = i + 1 to n
            if (s_i + s_j == key)
                return 1
            else
                return 0
}
```

30. In addition to finding a theta notation in Exercises 1–29, prove that it is correct.

31. Find the exact number of comparisons (lines 10, 15, 17, 24, and 26) required by the following algorithm when n is even and when n is odd. Find a theta notation for this algorithm.

Input: $s_1, s_2, \ldots, s_n, n$

Output: *large* (the largest item in $s_1, s_2, \ldots, s_n$), *small* (the smallest item in $s_1, s_2, \ldots, s_n$)

```
1.  large_small(s, n, large, small) {
2.      if (n == 1) {
3.          large = s_1
4.          small = s_1
5.          return
6.      }
7.      m = 2⌊n/2⌋
8.      i = 1
9.      while (i ≤ m − 1) {
10.         if (s_i > s_{i+1})
11.             swap(s_i, s_{i+1})
12.         i = i + 2
13.     }
14.     if (n > m) {
15.         if (s_{m−1} > s_n)
16.             swap(s_{m−1}, s_n)
17.         if (s_n > s_m)
18.             swap(s_m, s_n)
19.     }
20.     small = s_1
21.     large = s_2
22.     i = 3
23.     while (i ≤ m − 1) {
24.         if (s_i < small)
25.             small = s_i
26.         if (s_{i+1} > large)
27.             large = s_{i+1}
28.         i = i + 2
29.     }
30. }
```

32. This exercise shows another way to guess a formula for $1 + 2 + \cdots + n$.

Example 4.3.7 suggests that

$$1 + 2 + \cdots + n = An^2 + Bn + C \qquad \text{for all } n,$$

for some constants A, B, and C. Assuming that this is true, plug in $n = 1, 2, 3$ to obtain three equations in the three unknowns A, B, and C. Now solve for A, B, and C. The resulting formula can now be proved using mathematical induction (see Section 2.4).

33. Suppose that $a > 1$ and that $f(n) = \Theta(\log_a n)$. Show that $f(n) = \Theta(\lg n)$.

34. Show that $n! = O(n^n)$.

35. Show that $2^n = O(n!)$.

36. By using an argument like the one shown in Examples 4.3.7–4.3.9 or otherwise, prove that $\sum_{i=1}^{n} i \lg i = \Theta(n^2 \lg n)$.

⋆**37.** Show that $n^{n+1} = O(2^{n^2})$.

38. Show that $\lg(n^k + c) = \Theta(\lg n)$ for every fixed $k > 0$ and $c > 0$.

39. Show that if n is a power of 2, say $n = 2^k$, then

$$\sum_{i=0}^{k} \lg(n/2^i) = \Theta(\lg^2 n).$$

40. Suppose that $f(n) = O(g(n))$, and $f(n) \geq 0$ and $g(n) > 0$ for all $n \geq 1$. Show that for some constant C, $f(n) \leq Cg(n)$ for all $n \geq 1$.

41. State and prove a result for Ω similar to that for Exercise 40.

42. State and prove a result for Θ similar to that for Exercises 40 and 41.

Determine whether each statement in Exercises 43–68 is true or false. If the statement is true, prove it. If the statement is false, give a counterexample. Assume that the functions f, g, and h take on only positive values.

43. $n^n = O(2^n)$

44. $2 + \sin n = O(2 + \cos n)$

45. $(2n)^2 = O(n^2)$

46. $(2n)^2 = \Omega(n^2)$

47. $(2n)^2 = \Theta(n^2)$

48. $2^{2n} = O(2^n)$

49. $2^{2n} = \Omega(2^n)$

50. $2^{2n} = \Theta(2^n)$

51. $n! = O((n+1)!)$

52. $n! = \Omega((n+1)!)$

53. $n! = \Theta((n+1)!)$

54. $\lg(2n)^2 = O(\lg n^2)$

55. $\lg(2n)^2 = \Omega(\lg n^2)$

56. $\lg(2n)^2 = \Theta(\lg n^2)$

57. $\lg 2^{2n} = O(\lg 2^n)$

58. $\lg 2^{2n} = \Omega(\lg 2^n)$

59. $\lg 2^{2n} = \Theta(\lg 2^n)$

60. If $f(n) = \Theta(h(n))$ and $g(n) = \Theta(h(n))$, then $f(n) + g(n) = \Theta(h(n))$.

61. If $f(n) = \Theta(g(n))$, then $cf(n) = \Theta(g(n))$ for any $c \neq 0$.

62. If $f(n) = \Theta(g(n))$, then $2^{f(n)} = \Theta(2^{g(n)})$.

63. If $f(n) = \Theta(g(n))$, then $\lg f(n) = \Theta(\lg g(n))$. Assume that $f(n) \geq 1$ and $g(n) \geq 1$ for all $n = 1, 2, \ldots$.

64. If $f(n) = O(g(n))$, then $g(n) = O(f(n))$.

65. If $f(n) = O(g(n))$, then $g(n) = \Omega(f(n))$.

66. If $f(n) = \Theta(g(n))$, then $g(n) = \Theta(f(n))$.

67. $f(n) + g(n) = \Theta(h(n))$, where $h(n) = \max\{f(n), g(n)\}$

68. $f(n) + g(n) = \Theta(h(n))$, where $h(n) = \min\{f(n), g(n)\}$

69. Write out exactly what $f(n) \neq O(g(n))$ means.

70. What is wrong with the following argument that purports to show that we cannot simultaneously have $f(n) \neq O(g(n))$ and $g(n) \neq O(f(n))$?

If $f(n) \neq O(g(n))$, then for every $C > 0$, $|f(n)| > C|g(n)|$. In particular, $|f(n)| > 2|g(n)|$. If $g(n) \neq O(f(n))$, then for every $C > 0$, $|g(n)| > C|f(n)|$. In particular, $|g(n)| > 2|f(n)|$. But now

$$|f(n)| > 2|g(n)| > 4|f(n)|.$$

Cancelling $|f(n)|$ gives $1 > 4$, which is a contradiction. Therefore, we cannot simultaneously have $f(n) \neq O(g(n))$ and $g(n) \neq O(f(n))$.

★71. Find functions f and g satisfying

$$f(n) \neq O(g(n)) \qquad \text{and} \qquad g(n) \neq O(f(n)).$$

★72. Give an example of increasing positive functions f and g defined on the positive integers for which

$$f(n) \neq O(g(n)) \qquad \text{and} \qquad g(n) \neq O(f(n)).$$

★73. Prove that $n^k = O(c^n)$ for all $k = 1, 2, \ldots$ and $c > 1$.

74. Find functions f, g, h, and t satisfying

$$f(n) = \Theta(g(n)), \qquad h(n) = \Theta(t(n)),$$
$$f(n) - h(n) \neq \Theta(g(n) - t(n)).$$

75. Suppose that the worst-case time of an algorithm is $\Theta(n)$. What is the error in the following reasoning? Since $2n = \Theta(n)$, the worst-case time to run the algorithm with input of size $2n$ will be approximately the same as the worst-case time to run the algorithm with input of size n.

76. Does $f(n) = O(g(n))$ define an equivalence relation on the set of real-valued functions on $\{1, 2, \ldots\}$?

77. Does $f(n) = \Theta(g(n))$ define an equivalence relation on the set of real-valued functions on $\{1, 2, \ldots\}$?

78. [Requires the integral]

(a) Show, by consulting the figure, that

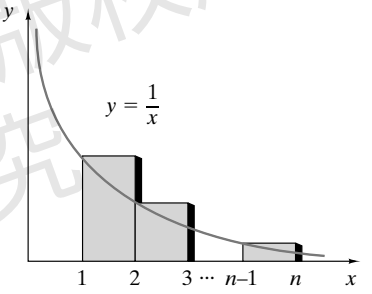
$$\frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} < \log_e n.$$

(b) Show, by consulting the figure, that

$$\log_e n < 1 + \frac{1}{2} + \cdots + \frac{1}{n-1}.$$

(c) Use parts (a) and (b) to show that

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



79. [Requires the integral] Use an argument like the one shown in Exercise 78 to show that

$$\frac{n^{m+1}}{m+1} < 1^m + 2^m + \cdots + n^m < \frac{(n+1)^{m+1}}{m+1},$$

where m is a positive integer.

80. By using the formula

$$\frac{b^{n+1} - a^{n+1}}{b-a} = \sum_{i=0}^{n} a^i b^{n-i} \qquad 0 \leq a < b$$

or otherwise, prove that

$$\frac{b^{n+1} - a^{n+1}}{b-a} < (n+1)b^n \qquad 0 \leq a < b.$$

81. Take $a = 1 + 1/(n+1)$ and $b = 1 + 1/n$ in the inequality of Exercise 80 to prove that the sequence $\{(1 + 1/n)^n\}$ is increasing.

82. Take $a = 1$ and $b = 1 + 1/(2n)$ in the inequality of Exercise 80 to prove that

$$\left(1 + \frac{1}{2n}\right)^n < 2$$

for all $n \geq 1$. Use the preceding exercise to conclude that

$$\left(1 + \frac{1}{n}\right)^n < 4$$

for all $n \geq 1$.

The method used to prove the results of this exercise and its predecessor is apparently due to Fort in 1862 (see [Chrystal, vol. II, page 77]).

83. By using the preceding two exercises or otherwise, prove that

$$\frac{1}{n} \leq \lg(n+1) - \lg n < \frac{2}{n}$$

for all $n \geq 1$.

84. Use the preceding exercise to prove that

$$\sum_{i=1}^{n} \frac{1}{i} = \Theta(\lg n).$$

(Compare with Exercise 78.)

85. Prove that the sequence $\{n^{1/n}\}_{n=3}^{\infty}$ is decreasing.

86. Prove that if $0 \leq a < b$, then

$$\frac{b^{n+1} - a^{n+1}}{b - a} > (n+1)a^n.$$

87. Find appropriate values for a and b in the inequality in the preceding exercise to prove that the sequence $\{(1 - 1/n)^n\}_{n=1}^{\infty}$ is increasing and bounded above by $4/9$.

88. By using the result of the preceding exercise, or otherwise, prove that the sequence $\{(1 + 1/n)^{n+1}\}_{n=1}^{\infty}$ is decreasing.

89. By using the result of the preceding exercise, or otherwise, prove that

$$\lg(n+1) - \lg n \leq \frac{2}{n+1}$$

for all $n \geq 1$.

90. What is wrong with the following "proof" that any algorithm has a run time that is $O(n)$?

We must show that the time required for an input of size n is at most a constant times n.

Basis Step

Suppose that $n = 1$. If the algorithm takes C units of time for an input of size 1, the algorithm takes at most $C \cdot 1$ units of time. Thus the assertion is true for $n = 1$.

Inductive Step

Assume that the time required for an input of size n is at most $C'n$ and that the time for processing an additional item is C''. Let C be the maximum of C' and C''. Then the total time required for an input of size $n + 1$ is at most

$$C'n + C'' \leq Cn + C = C(n+1).$$

The Inductive Step has been verified.

By induction, for input of size n, the time required is at most a constant time n. Therefore, the run time is $O(n)$.

In Exercises 91–96, determine whether the statement is true or false. If the statement is true, prove it. If the statement is false, give a counterexample. Assume that f and g are real-valued functions defined on the set of positive integers and that $g(n) \neq 0$ for $n \geq 1$. These exercises require calculus.

91. If

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

then $f(n) = O(g(n))$.

92. If

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

then $f(n) = \Theta(g(n))$.

93. If

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c \neq 0,$$

then $f(n) = O(g(n))$.

94. If

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c \neq 0,$$

then $f(n) = \Theta(g(n))$.

95. If $f(n) = O(g(n))$, then

$$\lim_{n\to\infty} \frac{f(n)}{g(n)}$$

exists and is equal to some real number.

96. If $f(n) = \Theta(g(n))$, then

$$\lim_{n\to\infty} \frac{f(n)}{g(n)}$$

exists and is equal to some real number.

★97. Use induction to prove that

$$\lg n! \geq \frac{n}{2} \lg \frac{n}{2}.$$

98. [Requires calculus] Let $\ln x$ denote the natural logarithm ($\log_e x$) of x. Use the integral to obtain the estimate

$$n \ln n - n \leq \sum_{k=1}^{n} \ln k = \ln n!, \qquad n \geq 1.$$

99. Use the result of Exercise 98 and the change-of-base formula for logarithms to obtain the formula

$$n \lg n - n \lg e \leq \lg n!, \qquad n \geq 1.$$

100. Deduce

$$\lg n! \geq \frac{n}{2} \lg \frac{n}{2}$$

from the inequality of Exercise 99.

Problem-Solving Corner Design and Analysis of an Algorithm

Problem

Develop and analyze an algorithm that returns the maximum sum of consecutive values in the numerical sequence $s_1, \ldots, s_n$. In mathematical notation, the problem is to find the maximum sum of the form $s_j + s_{j+1} + \cdots + s_i$. *Example:* If the sequence is

27 6 −50 21 −3 14 16 −8 42 33 −21 9,

the algorithm returns 115—the sum of

21 −3 14 16 −8 42 33.

If all the numbers in a sequence are negative, the maximum sum of consecutive values is defined to be 0. (The idea is that the maximum of 0 is achieved by taking an "empty" sum.)

Attacking the Problem

In developing an algorithm, a good way to start is to ask the question, "How would I solve this problem by hand?" At least initially, take a straightforward approach. Here we might just list the sums of *all* consecutive values and pick the largest. For the example sequence, the sums are as follows:

						j					
i	1	2	3	4	5	6	7	8	9	10	11	12
1	27											
2	33	6										
3	−17	−44	−50									
4	4	−23	−29	21								
5	1	−26	−32	18	−3							
6	15	−12	−18	32	11	14						
7	31	4	−2	48	27	30	16					
8	23	−4	−10	40	19	22	8	−8				
9	65	38	32	82	61	64	50	34	42			
10	98	71	65	115	94	97	83	67	75	33		
11	77	50	44	94	73	76	62	46	54	12	−21	
12	86	59	53	103	82	85	71	55	63	21	−12	9

The entry in column j, row i, is the sum $s_j + \cdots + s_i$. For example, the entry in column 4, row 7, is 48—the sum

$$s_4 + s_5 + s_6 + s_7 = 21 + -3 + 14 + 16 = 48.$$

By inspection, we find that 115 is the largest sum.

Finding a Solution

We begin by writing pseudocode for the straightforward algorithm that computes all consecutive sums and finds the largest:

```
Input:  s_1, ..., s_n
Output: max

max_sum1(s, n) {
   // sum_ji is the sum s_j + ... + s_i.
   for i = 1 to n {
      for j = 1 to i - 1
         sum_ji = sum_j,i-1 + s_i
      sum_ii = s_i
   }

   // step through sum_ji and find the maximum
   max = 0
   for i = 1 to n
      for j = 1 to i
         if (sum_ji > max)
            max = sum_ji
   return max
}
```

The first nested for loops compute the sums

$$sum_{ji} = s_j + \cdots + s_i.$$

The computation relies on the fact that

$$sum_{ji} = s_j + \cdots + s_i = s_j + \cdots + s_{i-1} + s_i$$
$$= sum_{j,i-1} + s_i.$$

The second nested for loops step through sum_{ji} and find the largest value.

Since each of the nested for loops takes time $\Theta(n^2)$, *max_sum1*'s time is $\Theta(n^2)$.

We can improve the actual time, but not the asymptotic time, of the algorithm by computing the maximum within the same nested for loops in which we compute sum_{ji}:

```
   Input:  s_1, ..., s_n
  Output:  max

max_sum2(s, n) {
   // sum_ji is the sum s_j + ... + s_i.
   max = 0
   for i = 1 to n {
      for j = 1 to i - 1 {
         sum_ji = sum_{j,i-1} + s_i
         if (sum_ji > max)
            max = sum_ji
      }
      sum_ii = s_i
      if (sum_ii > max)
         max = sum_ii
   }
   return max
}
```

Since the nested for loops take time $\Theta(n^2)$, *max_sum2*'s time is $\Theta(n^2)$. To reduce the asymptotic time, we need to take a hard look at the pseudocode to see where it can be improved.

Two key observations lead to improved time. First, since we are looking only for the *maximum* sum, there is no need to record all of the sums; we will store only the maximum sum that ends at index *i*. Second, the line

$$sum_{ji} = sum_{j,i-1} + s_i$$

shows how a consecutive sum that ends at index $i - 1$ is related to a consecutive sum that ends at index *i*. The maximum can be computed by using a similar formula. If *sum* is the maximum consecutive sum that ends at index $i-1$, the maximum consecutive sum that ends at index *i* is obtained by adding s_i to *sum* provided that $sum + s_i$ is positive. (If some sum of consecutive terms that ends at index *i* exceeds $sum + s_i$, we could remove s_i and obtain a sum of consecutive terms ending at index $i - 1$ that exceeds *sum*, which is impossible.) If $sum + s_i \leq 0$, the maximum consecutive sum that ends at index *i* is obtained by taking no terms and has value 0. Thus we may compute the maximum consecutive sum that ends at index *i* by executing

```
   if (sum + s_i > 0)
      sum = sum + s_i
   else
      sum = 0
```

Formal Solution

```
   Input:  s_1, ..., s_n
  Output:  max

max_sum3(s, n) {
   // max is the maximum sum seen so far.
   // After the ith iteration of the for
   // loop, sum is the largest consecutive
   // sum that ends at index i.
   max = 0
   sum = 0
   for i = 1 to n {
      if (sum + s_i > 0)
         sum = sum + s_i
      else
         sum = 0
      if (sum > max)
         max = sum
   }
   return max
}
```

Since this algorithm has a single for loop that runs from 1 to *n*, *max_sum3*'s time is $\Theta(n)$. The asymptotic time of this algorithm cannot be further improved. To find the maximum sum of consecutive values, we must at least look at each element in the sequence, which takes time $\Theta(n)$.

Summary of Problem-Solving Techniques

- In developing an algorithm, a good way to start is to ask the question, "How would I solve this problem by hand?"
- In developing an algorithm, initially take a straightforward approach.
- After developing an algorithm, take a close look at the pseudocode to see where it can be improved. Look at the parts that perform key computations to gain insight into how to enhance the algorithm's efficiency.
- As in mathematical induction, extend a solution of a smaller problem to a larger problem. (In this problem, we extended a sum that ends at index $i - 1$ to a sum that ends at index *i*.)

- Don't repeat computations. (In this problem, we extended a sum that ends at index $i-1$ to a sum that ends at index i by adding an additional term rather than by computing the sum that ends at index i from scratch. This latter method would have meant recomputing the sum that ends at index $i-1$.)

Comments

According to [Bentley], the problem discussed in this section is the one-dimensional version of the original two-dimensional problem that dealt with pattern matching in digital images. The original problem was to find the maximum sum in a rectangular submatrix of an $n \times n$ matrix of real numbers.

Exercises

1. Modify *max_sum3* so that it computes not only the maximum sum of consecutive values but also the indexes of the first and last terms of a maximum-sum subsequence. If there is no maximum-sum subsequence (which would happen, for example, if all of the values of the sequence were negative), the algorithm should set the first and last indexes to zero.

4.4 Recursive Algorithms

Go Online
For more on recursion, see goo.gl/ZwpPlu

A **recursive function** (pseudocode) is a function that invokes itself. A **recursive algorithm** is an algorithm that contains a recursive function. Recursion is a powerful, elegant, and natural way to solve a large class of problems. A problem in this class can be solved using a *divide-and-conquer* technique in which the problem is decomposed into problems of the same type as the original problem. Each subproblem, in turn, is decomposed further until the process yields subproblems that can be solved in a straightforward way. Finally, solutions to the subproblems are combined to obtain a solution to the original problem.

Example 4.4.1

Recall that if $n \geq 1$, $n! = n(n-1)\cdots 2 \cdot 1$, and $0! = 1$. Notice that if $n \geq 2$, n factorial can be written "in terms of itself" since, if we "peel off" n, the remaining product is simply $(n-1)!$; that is,

$$n! = n(n-1)(n-2)\cdots 2 \cdot 1 = n \cdot (n-1)!.$$

For example,

$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 5 \cdot 4!.$$

The equation

$$n! = n \cdot (n-1)!, \tag{4.4.1}$$

which happens to be true even when $n = 1$, shows how to decompose the original problem (compute $n!$) into increasingly simpler subproblems [compute $(n-1)!$, compute $(n-2)!, \ldots$] until the process reaches the straightforward problem of computing $0!$. The solutions to these subproblems can then be combined, by multiplying, to solve the original problem.

For example, the problem of computing $5!$ is reduced to computing $4!$; the problem of computing $4!$ is reduced to computing $3!$; and so on. Table 4.4.1 summarizes this process.

Once the problem of computing $5!$ has been reduced to solving subproblems, the solution to the simplest subproblem can be used to solve the next simplest subproblem, and so on, until the original problem has been solved. Table 4.4.2 shows how the subproblems are combined to compute $5!$. ◀

TABLE 4.4.1 ■ Decomposing the Factorial Problem

Problem	*Simplified Problem*
$5!$	$5 \cdot 4!$
$4!$	$4 \cdot 3!$
$3!$	$3 \cdot 2!$
$2!$	$2 \cdot 1!$
$1!$	$1 \cdot 0!$
$0!$	None

TABLE 4.4.2 ■ Combining Subproblems of the Factorial Problem

Problem	*Solution*
$0!$	1
$1!$	$1 \cdot 0! = 1$
$2!$	$2 \cdot 1! = 2$
$3!$	$3 \cdot 2! = 3 \cdot 2 = 6$
$4!$	$4 \cdot 3! = 4 \cdot 6 = 24$
$5!$	$5 \cdot 4! = 5 \cdot 24 = 120$

Next, we write a recursive algorithm that computes factorials. The algorithm is a direct translation of equation (4.4.1).

Algorithm 4.4.2 **Computing n Factorial**

This recursive algorithm computes $n!$.

Input: n, an integer greater than or equal to 0
Output: $n!$

```
1.  factorial(n) {
2.      if (n == 0)
3.          return 1
4.      return n * factorial(n − 1)
5.  }
```

We show how Algorithm 4.4.2 computes $n!$ for several values of n. If $n = 0$, at line 3 the function correctly returns the value 1.

If $n = 1$, we proceed to line 4 since $n \neq 0$. We use this function to compute 0!. We have just observed that the function computes 1 as the value of 0!. At line 4, the function correctly computes the value of 1!:

$$n \cdot (n-1)! = 1 \cdot 0! = 1 \cdot 1 = 1.$$

If $n = 2$, we proceed to line 4 since $n \neq 0$. We use this function to compute 1!. We have just observed that the function computes 1 as the value of 1!. At line 4, the function correctly computes the value of 2!:

$$n \cdot (n-1)! = 2 \cdot 1! = 2 \cdot 1 = 2.$$

If $n = 3$ we proceed to line 4 since $n \neq 0$. We use this function to compute 2!. We have just observed that the function computes 2 as the value of 2!. At line 4, the function correctly computes the value of 3!:

$$n \cdot (n-1)! = 3 \cdot 2! = 3 \cdot 2 = 6.$$

The preceding arguments may be generalized using mathematical induction to *prove* that Algorithm 4.4.2 correctly returns the value of $n!$ for any nonnegative integer n.

Theorem 4.4.3 Algorithm 4.4.2 returns the value of $n!$, $n \geq 0$.

Proof

Basis Step (n = 0)

We have already observed that if $n = 0$, Algorithm 4.4.2 correctly returns the value of 0! (1).

Inductive Step

Assume that Algorithm 4.4.2 correctly returns the value of $(n-1)!$, $n > 0$. Now suppose that n is input to Algorithm 4.4.2. Since $n \neq 0$, when we execute the function in Algorithm 4.4.2 we proceed to line 4. By the inductive assumption, the function correctly computes the value of $(n-1)!$. At line 4, the function correctly computes the value $(n-1)! \cdot n = n!$.

Therefore, Algorithm 4.4.2 correctly returns the value of $n!$ for every integer $n \geq 0$. ◀

If executed by a computer, Algorithm 4.4.2 would typically not be as efficient as a nonrecursive version because of the overhead of the recursive calls.

There must be some situations in which a recursive function does *not* invoke itself; otherwise, it would invoke itself forever. In Algorithm 4.4.2, if $n = 0$, the function does not invoke itself. We call the values for which a recursive function does not invoke itself the *base cases*. To summarize, every recursive function must have base cases.

We have shown how mathematical induction may be used to prove that a recursive algorithm computes the value it claims to compute. The link between mathematical induction and recursive algorithms runs deep. Often a proof by mathematical induction can be considered to be an algorithm to compute a value or to carry out a particular construction. The Basis Step of a proof by mathematical induction corresponds to the base cases of a recursive function, and the Inductive Step of a proof by mathematical induction corresponds to the part of a recursive function where the function calls itself.

In Example 2.4.7, we gave a proof using mathematical induction that, given an $n \times n$ deficient board (a board with one square removed), where n is a power of 2, we can tile the board with right trominoes (three squares that form an "L"; see Figure 2.4.4). We now translate the inductive proof into a recursive algorithm to construct a tiling by right trominoes of an $n \times n$ deficient board where n is a power of 2.

Algorithm 4.4.4

Go Online
For a C program implementing this algorithm, see `goo.gl/rZYmnK`

Tiling a Deficient Board with Trominoes

This algorithm constructs a tiling by right trominoes of an $n \times n$ deficient board where n is a power of 2.

Input: n, a power of 2 (the board size); and the location L of the missing square

Output: A tiling of an $n \times n$ deficient board

```
1.  tile(n, L) {
2.      if (n == 2) {
            // the board is a right tromino T
3.          tile with T
4.          return
5.      }
6.      divide the board into four (n/2) × (n/2) boards
7.      rotate the board so that the missing square is in the upper-left quadrant
8.      place one right tromino in the center // as in Figure 2.4.5
        // consider each of the squares covered by the center tromino as
        // missing, and denote the missing squares as m1, m2, m3, m4
9.      tile(n/2, m1)
10.     tile(n/2, m2)
11.     tile(n/2, m3)
12.     tile(n/2, m4)
13. }
```

Using the method of the proof of Theorem 4.4.3, we can prove that Algorithm 4.4.4 is correct (see Exercise 4).

We present one final example of a recursive algorithm.

Example 4.4.5 A robot can take steps of 1 meter or 2 meters. We write an algorithm to calculate the number of ways the robot can walk n meters. As examples:

Distance	*Sequence of Steps*	*Number of Ways to Walk*
1	1	1
2	1, 1 or 2	2
3	1, 1, 1 or 1, 2 or 2, 1	3
4	1, 1, 1, 1 or 1, 1, 2 or 1, 2, 1 or 2, 1, 1 or 2, 2	5

Let $walk(n)$ denote the number of ways the robot can walk n meters. We have observed that $walk(1) = 1$ and $walk(2) = 2$. Now suppose that $n > 2$. The robot can begin by taking a step of 1 meter or a step of 2 meters. If the robot begins by taking a 1-meter step, a distance of $n - 1$ meters remains; but, by definition, the remainder of the walk can be completed in $walk(n - 1)$ ways. Similarly, if the robot begins by taking a 2-meter step, a distance of $n - 2$ meters remains and, in this case, the remainder of the walk can be completed in $walk(n - 2)$ ways. Since the walk must begin with either a 1-meter or a 2-meter step, all of the ways to walk n meters are accounted for. We obtain the formula

$$walk(n) = walk(n - 1) + walk(n - 2). \tag{4.4.2}$$

For example,

$$walk(4) = walk(3) + walk(2) = 3 + 2 = 5.$$

We can write a recursive algorithm to compute $walk(n)$ by translating equation (4.4.2) directly into an algorithm. The base cases are $n = 1$ and $n = 2$. ◀

Algorithm 4.4.6 **Robot Walking**

This algorithm computes the function defined by

$$walk(n) = \begin{cases} 1, & n = 1 \\ 2, & n = 2 \\ walk(n-1) + walk(n-2) & n > 2. \end{cases}$$

Input: n
Output: $walk(n)$

```
walk(n) {
   if (n == 1 ∨ n == 2)
      return n
   return walk(n − 1) + walk(n − 2)
}
```

Go Online
For a C program implementing this algorithm, see goo.gl/m754WF

Using the method of the proof of Theorem 4.4.3, we can prove that Algorithm 4.4.6 is correct (see Exercise 7).

The sequence $walk(1), walk(2), walk(3), \ldots,$ whose values begin $1, 2, 3, 5, 8, 13, \ldots,$ is related to the Fibonacci sequence. The **Fibonacci sequence** $\{f_n\}$ is defined by the equations

$$f_1 = 1, \qquad f_2 = 1, \qquad f_n = f_{n-1} + f_{n-2} \qquad n \geq 3.$$

Go Online
For more on the Fibonacci sequence, see goo.gl/ZwpPlu

Figure 4.4.1 A pine cone. There are 13 clockwise spirals (marked with white thread) and 8 counterclockwise spirals (marked with dark thread). [*Photo by the author; pine cone courtesy of André Berthiaume and Sigrid (Anne) Settle.*]

The Fibonacci sequence begins

$$1, 1, 2, 3, 5, 8, 13, \ldots.$$

Since $walk(1) = f_2$, $walk(2) = f_3$, and

$$walk(n) = walk(n-1) + walk(n-2), \qquad f_n = f_{n-1} + f_{n-2} \qquad \text{for all } n \geq 3,$$

it follows that

$$walk(n) = f_{n+1} \qquad \text{for all } n \geq 1.$$

(The argument can be formalized using mathematical induction; see Exercise 8.)

The Fibonacci sequence is named in honor of Leonardo Fibonacci (ca. 1170–1250), an Italian merchant and mathematician. The sequence originally arose in a puzzle about rabbits (see Exercises 18 and 19). After returning from the Orient in 1202, Fibonacci wrote his most famous work, *Liber Abaci* (available in an English translation by [Sigler]), which, in addition to containing what we now call the Fibonacci sequence, advocated the use of Hindu-Arabic numerals. This book was one of the main influences in bringing the decimal number system to Western Europe. Fibonacci signed much of his work "Leonardo Bigollo." *Bigollo* translates as "traveler" or "blockhead." There is some evidence that Fibonacci enjoyed having his contemporaries consider him a blockhead for advocating the new number system.

The Fibonacci sequence pops up in unexpected places. Figure 4.4.1 shows a pine cone with 13 clockwise spirals and 8 counterclockwise spirals. Many plants distribute their seeds as evenly as possible, thus maximizing the space available for each seed. The pattern in which the number of spirals is a Fibonacci number provides the most even distribution (see [Naylor, Mitchison]). In Section 5.3, the Fibonacci sequence appears in the analysis of the Euclidean algorithm.

Example 4.4.7 Use mathematical induction to show that

$$\sum_{k=1}^{n} f_k = f_{n+2} - 1 \qquad \text{for all } n \geq 1.$$

SOLUTION For the basis step ($n = 1$), we must show that

$$\sum_{k=1}^{1} f_k = f_3 - 1.$$

Since $\sum_{k=1}^{1} f_k = f_1 = 1$ and $f_3 - 1 = 2 - 1 = 1$, the equation is verified.

For the inductive step, we assume case n

$$\sum_{k=1}^{n} f_k = f_{n+2} - 1$$

and prove case $n + 1$

$$\sum_{k=1}^{n+1} f_k = f_{n+3} - 1.$$

Now

$$\begin{aligned}\sum_{k=1}^{n+1} f_k &= \sum_{k=1}^{n} f_k + f_{n+1} \\ &= (f_{n+2} - 1) + f_{n+1} \qquad \text{by the inductive assumption} \\ &= f_{n+1} + f_{n+2} - 1 \\ &= f_{n+3} - 1.\end{aligned}$$

The last equality is true because of the definition of the Fibonacci numbers:

$$f_n = f_{n-1} + f_{n-2} \qquad \text{for all } n \geq 3.$$

Since the basis step and the inductive step have been verified, the given equation is true for all $n \geq 1$. ◀

4.4 Problem-Solving Tips

A recursive function is a function that invokes itself. The key to writing a recursive function is to find a smaller instance of the problem within the larger problem. For example, we can compute $n!$ recursively because $n! = n \cdot (n - 1)!$ for all $n \geq 1$. The situation is analogous to the inductive step in mathematical induction when we must find a smaller case (e.g., case n) within the larger case (e.g., case $n + 1$).

As another example, tiling an $n \times n$ deficient board with trominoes when n is a power of 2 can be done recursively because we can find four $(n/2) \times (n/2)$ subboards within the original $n \times n$ board. Note the similarity of the tiling algorithm to the inductive step of the proof that every $n \times n$ deficient board can be tiled with trominoes when n is a power of 2.

To prove a statement about the Fibonacci numbers, use the equation

$$f_n = f_{n-1} + f_{n-2} \qquad \text{for all } n \geq 3.$$

The proof will often use mathematical induction *and* the previous equation (see Example 4.4.7).

4.4 Review Exercises

1. What is a recursive algorithm?
2. What is a recursive function?
3. Give an example of a recursive function.
4. Explain how the divide-and-conquer technique works.
5. What is a base case in a recursive function?
6. Why must every recursive function have a base case?
7. How is the Fibonacci sequence defined?
8. Give the first four values of the Fibonacci sequence.

4.4 Exercises

1. Trace Algorithm 4.4.2 for $n = 4$.
2. Trace Algorithm 4.4.4 when $n = 4$ and the missing square is the upper-left corner square.
3. Trace Algorithm 4.4.4 when $n = 8$ and the missing square is four from the left and six from the top.
4. Prove that Algorithm 4.4.4 is correct.
5. Trace Algorithm 4.4.6 for $n = 4$.
6. Trace Algorithm 4.4.6 for $n = 5$.
7. Prove that Algorithm 4.4.6 is correct.
8. Prove that
$$walk(n) = f_{n+1} \qquad \text{for all } n \geq 1.$$
9. (a) Use the formulas
$$s_1 = 1, \qquad s_n = s_{n-1} + n \qquad \text{for all } n \geq 2,$$
to write a recursive algorithm that computes
$$s_n = 1 + 2 + 3 + \cdots + n.$$
(b) Give a proof using mathematical induction that your algorithm for part (a) is correct.
10. (a) Use the formulas
$$s_1 = 2, \qquad s_n = s_{n-1} + 2n \qquad \text{for all } n \geq 2,$$
to write a recursive algorithm that computes
$$s_n = 2 + 4 + 6 + \cdots + 2n.$$
(b) Give a proof using mathematical induction that your algorithm for part (a) is correct.
11. (a) A robot can take steps of 1 meter, 2 meters, or 3 meters. Write a recursive algorithm to calculate the number of ways the robot can walk n meters.
(b) Give a proof using mathematical induction that your algorithm for part (a) is correct.
12. Write a recursive algorithm to find the minimum of a finite sequence of numbers. Give a proof using mathematical induction that your algorithm is correct.
13. Write a recursive algorithm to find the maximum of a finite sequence of numbers. Give a proof using mathematical induction that your algorithm is correct.
14. Write a recursive algorithm that reverses a finite sequence. Give a proof using mathematical induction that your algorithm is correct.
15. Write a nonrecursive algorithm to compute $n!$.
⋆16. A robot can take steps of 1 meter or 2 meters. Write an algorithm to list all of the ways the robot can walk n meters.
⋆17. A robot can take steps of 1 meter, 2 meters, or 3 meters. Write an algorithm to list all of the ways the robot can walk n meters.

Exercises 18–36 concern the Fibonacci sequence $\{f_n\}$.

18. Suppose that at the beginning of the year, there is one pair of rabbits and that every month each pair produces a new pair that becomes productive after one month. Suppose further that no deaths occur. Let a_n denote the number of pairs of rabbits at the end of the nth month. Show that $a_1 = 1, a_2 = 2$, and $a_n - a_{n-1} = a_{n-2}$. Prove that $a_n = f_{n+1}$ for all $n \geq 1$.
19. Fibonacci's original question was: Under the conditions of Exercise 18, how many pairs of rabbits are there after one year? Answer Fibonacci's question.
20. Show that the number of ways to tile a $2 \times n$ board with 1×2 rectangular pieces is f_{n+1}, the $(n+1)$st Fibonacci number.
21. Use mathematical induction to show that
$$f_n^2 = f_{n-1}f_{n+1} + (-1)^{n+1} \qquad \text{for all } n \geq 2.$$
22. Show that
$$f_n^2 = f_{n-2}f_{n+2} + (-1)^n \qquad \text{for all } n \geq 3.$$
23. Show that
$$f_{n+2}^2 - f_{n+1}^2 = f_n f_{n+3} \qquad \text{for all } n \geq 1.$$
24. Use mathematical induction to show that
$$\sum_{k=1}^{n} f_k^2 = f_n f_{n+1} \qquad \text{for all } n \geq 1.$$
25. Use mathematical induction to show that for all $n \geq 1, f_n$ is even if and only if n is divisible by 3.
⋆26. Use mathematical induction to show that
$$f_{2n} = f_{n+1}^2 - f_{n-1}^2 \quad \text{and} \quad f_{2n+1} = f_n^2 + f_{n+1}^2 \quad \text{for all } n \geq 2.$$
27. Use mathematical induction to show that for all $n \geq 6$,
$$f_n > \left(\frac{3}{2}\right)^{n-1}.$$
28. Use mathematical induction to show that for all $n \geq 1$,
$$f_n \leq 2^{n-1}.$$

29. Use mathematical induction to show that for all $n \geq 1$,

$$\sum_{k=1}^{n} f_{2k-1} = f_{2n}, \qquad \sum_{k=1}^{n} f_{2k} = f_{2n+1} - 1.$$

⋆30. Use mathematical induction to show that every integer $n \geq 1$ can be expressed as the sum of distinct Fibonacci numbers, no two of which are consecutive.

⋆31. Show that the representation in Exercise 30 is unique if we do not allow f_1 as a summand.

32. Show that for all $n \geq 2$,

$$f_n = \frac{f_{n-1} + \sqrt{5f_{n-1}^2 + 4(-1)^{n+1}}}{2}.$$

Notice that this formula gives f_n in terms of one predecessor rather than two predecessors as in the original definition.

33. Prove that

$$1 + \sum_{k=1}^{n} \frac{(-1)^{k+1}}{f_k f_{k+1}} = \frac{f_{n+2}}{f_{n+1}} \quad \text{for all } n \geq 1.$$

34. Define a sequence $\{g_n\}$ as $g_1 = c_1$ and $g_2 = c_2$ for constants c_1 and c_2, and

$$g_n = g_{n-1} + g_{n-2}$$

for $n \geq 3$. Prove that

$$g_n = g_1 f_{n-2} + g_2 f_{n-1}$$

for all $n \geq 3$.

35. Prove that

$$\sum_{k=1}^{n} (-1)^k f_k = (-1)^n f_{n-1} - 1 \quad \text{for all } n \geq 2.$$

36. Prove that

$$\sum_{k=1}^{n} (-1)^k k f_k = (-1)^n (n f_{n-1} + f_{n-3}) - 2 \quad \text{for all } n \geq 4.$$

37. [Requires calculus] Assume the formula for differentiating products:

$$\frac{d(fg)}{dx} = f\frac{dg}{dx} + g\frac{df}{dx}.$$

Use mathematical induction to prove that

$$\frac{dx^n}{dx} = nx^{n-1} \qquad \text{for } n = 1, 2, \ldots.$$

38. [Requires calculus] Explain how the formula gives a recursive algorithm for integrating $\log^n |x|$:

$$\int \log^n |x|\, dx = x \log^n |x| - n \int \log^{n-1} |x|\, dx.$$

Give other examples of recursive integration formulas.

Chapter 4 Notes

The first half of [Knuth, 1977] introduces the concept of an algorithm and various mathematical topics, including mathematical induction. The second half is devoted to data structures.

Most general references on computer science contain some discussion of algorithms. Books specifically on algorithms are [Aho; Baase; Brassard; Cormen; Johnsonbaugh; Knuth, 1997, 1998a, 1998b; Manber; Miller; Nievergelt; and Reingold]. [McNaughton] contains a very thorough discussion on an introductory level of what an algorithm is. Knuth's expository article about algorithms ([Knuth, 1977]) and his article about the role of algorithms in the mathematical sciences ([Knuth, 1985]) are also recommended. [Gardner, 1992] contains a chapter about the Fibonacci sequence.

Chapter 4 Review

Section 4.1

1. Algorithm
2. Properties of an algorithm: Input, output, precision, determinism, finiteness, correctness, generality
3. Trace
4. Pseudocode

Section 4.2

5. Searching
6. Text search
7. Text-search algorithm
8. Sorting
9. Insertion sort
10. Time and space for algorithms
11. Best-case time
12. Worst-case time
13. Randomized algorithm
14. Shuffle algorithm

Section 4.3

15. Analysis of algorithms
16. Worst-case time of an algorithm
17. Best-case time of an algorithm
18. Average-case time of an algorithm

19. Big oh notation: $f(n) = O(g(n))$
20. Omega notation: $f(n) = \Omega(g(n))$
21. Theta notation: $f(n) = \Theta(g(n))$

Section 4.4

22. Recursive algorithm
23. Recursive function
24. Divide-and-conquer technique
25. Base cases: Situations where a recursive function does not invoke itself
26. Fibonacci sequence $\{f_n\} : f_1 = 1, f_2 = 1, f_n = f_{n-1} + f_{n-2}, n \geq 3$

Chapter 4 Self-Test

1. Trace Algorithm 4.1.1 for the values $a = 12, b = 3, c = 0$.
2. Which of the algorithm properties—input, output, precision, determinism, finiteness, correctness, generality—if any, are lacking in the following? Explain.

   Input: S (a set of integers), m (an integer)
   Output: All finite subsets of S that sum to m

   1. List all finite subsets of S and their sums.
   2. Step through the subsets listed in 1 and output each whose sum is m.

3. Trace Algorithm 4.2.1 for the input $t =$ "111011" and $p =$ "110".
4. Trace Algorithm 4.2.3 for the input 44, 64, 77, 15, 3.
5. Trace Algorithm 4.2.4 for the input 5, 51, 2, 44, 96. Assume that the values of *rand* are

   $rand(1, 5) = 1$, $rand(2, 5) = 3$, $rand(3, 5) = 5$, $rand(4, 5) = 5$.

6. Write an algorithm that receives as input the distinct numbers a, b, and c and assigns the values a, b, and c to the variables x, y, and z so that $x < y < z$.
7. Write an algorithm that receives as input the sequence $s_1, \ldots, s_n$ sorted in nondecreasing order and prints all values that appear more than once. *Example:* If the sequence is 1, 1, 1, 5, 8, 8, 9, 12, the output is 1 8.
8. Write an algorithm that returns true if the values of a, b, and c are distinct, and false otherwise.
9. Write an algorithm that tests whether two $n \times n$ matrices are equal and find a theta notation for its worst-case time.

Select a theta notation from among $\Theta(1)$, $\Theta(n)$, $\Theta(n^2)$, $\Theta(n^3)$, $\Theta(n^4)$, $\Theta(2^n)$, *or* $\Theta(n!)$ *for each of the expressions in Exercises 10 and 11.*

10. $4n^3 + 2n - 5$
11. $1^3 + 2^3 + \cdots + n^3$
12. Select a theta notation from among $\Theta(1)$, $\Theta(n)$, $\Theta(n^2)$, $\Theta(n^3)$, $\Theta(2^n)$, or $\Theta(n!)$ for the number of times the line $x = x + 1$ is executed.

```
for i = 1 to n
    for j = 1 to n
        x = x + 1
```

13. Trace Algorithm 4.4.4 (the tromino tiling algorithm) when $n = 8$ and the missing square is four from the left and two from the top.

Exercises 14–16 refer to the tribonacci sequence $\{t_n\}$ *defined by the equations*

$$t_1 = t_2 = t_3 = 1, \quad t_n = t_{n-1} + t_{n-2} + t_{n-3} \quad \textit{for all } n \geq 4.$$

14. Find t_4 and t_5.
15. Write a recursive algorithm to compute t_n, $n \geq 1$.
16. Give a proof using mathematical induction that your algorithm for Exercise 15 is correct.

Chapter 4 Computer Exercises

1. Implement Algorithm 4.1.2, finding the largest element in a sequence, as a program.
2. Implement Algorithm 4.2.1, text search, as a program.
3. Implement Algorithm 4.2.3, insertion sort, as a program.
4. Implement Algorithm 4.2.4, shuffle, as a program.
5. Run shuffle (Algorithm 4.2.4) many times for the same input sequence. How might the output be analyzed to determine if it is truly "random"?
6. Implement selection sort (see Exercise 22, Section 4.2) as a program.
7. Compare the running times of insertion sort (Algorithm 4.2.3) and selection sort (see Exercise 22, Section 4.2) for several inputs of different sizes. Include data sorted in nondecreasing order, data sorted in nonincreasing order, data containing many duplicates, and data in random order.
8. Write recursive and nonrecursive programs to compute $n!$. Compare the times required by the programs.

9. Write a program whose input is a $2^n \times 2^n$ board with one missing square and whose output is a tiling of the board by trominoes.

10. Write a program that uses a graphics display to show a tiling with trominoes of a $2^n \times 2^n$ board with one square missing.

11. Write a program that tiles with trominoes an $n \times n$ board with one square missing, provided that $n \neq 5$ and 3 does not divide n.

12. Write recursive and nonrecursive programs to compute the Fibonacci sequence. Compare the times required by the programs.

13. A robot can take steps of 1 meter or 2 meters. Write a program to list all of the ways the robot can walk n meters.

14. A robot can take steps of 1, 2, or 3 meters. Write a program to list all of the ways the robot can walk n meters.