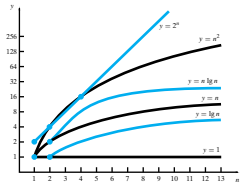
3



ALGORITHMS

It's so simple.

Step 1: We find the worst play in the world—a sure fire flop.

Step 2: I raise a million bucks—there are a lot of little old ladies in the world.

Step 3: You go back to work on the books. Phony lists of backers—one for the government, one for us. You can do it, Bloom, you're a wizard.

Step 4: We open on Broadway and before you can say

Step 5: We close on Broadway.

Step 6: We take our million bucks and we fly to Rio de Janeiro.

FROM *The Producers*

An algorithm is a step-by-step method of solving some problem. Adlai Stevenson's recipe for carp furnishes us an example of an algorithm:

1. Take a 1- to 2-pound carp and allow it to swim in clear water for 24 hours.
2. Scale and fillet the carp.
3. Rub fillets with butter and season with salt and pepper.
4. Place on board and bake in moderate oven for 20 minutes.
5. Throw away carp and eat board.

Examples of algorithms can be found throughout history, going back at least as far as ancient Babylonia. Indeed, the word "algorithm" derives from the name of the ninth-century Persian mathematician al-Khowārizmī. Algorithms based on sound mathematical principles play a central role in mathematics and computer science. In order for a computer to execute a solution to a problem, the solution must be described as a sequence of precise steps.

After introducing algorithms and our notation for them, we discuss the greatest common divisor algorithm, an ancient Greek algorithm that is still much used. We then turn to complexity of algorithms, which refers to the time and space required to execute algorithms, and the analysis of the resources required for particular algorithms. We conclude by discussing the RSA public-key cryptosystem—a method of encoding and decoding messages whose security relies primarily on the lack of an efficient algorithm for finding the prime divisors of an arbitrary integer.

† This section can be omitted without loss of continuity.

3.1 INTRODUCTION

An **algorithm** is a finite set of instructions having the following characteristics:

- ***Precision***. The steps are precisely stated.
- ***Uniqueness***. The intermediate results of each step of execution are uniquely defined and depend only on the inputs and the results of the preceding steps.
- ***Finiteness***. The algorithm stops after finitely many instructions have been executed.
- ***Input***. The algorithm receives input.
- ***Output***. The algorithm produces output.
- ***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. $x := a$.
2. If $b > x$, then $x := b$.
3. If $c > x$, then $x := c$.

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

The notation $y := z$ means "copy the value of z into y" or, equivalently, "replace the current value of y by the value of z." When $y := z$ is executed, the value of z is unchanged. We call := the **assignment operator**.

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

$$a = 1, \qquad b = 5, \qquad c = 3.$$

At line 1, we set x to a (1). At line 2, $b > x$ (5 > 1) is true, so we set x to b (5). At line 3, $c > x$ (3 > 5) is false, so we do nothing. At this point x is 5, the largest of a, b, and c.

Suppose that

$$a = 6, \qquad b = 1, \qquad c = 9.$$

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

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

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

Given values of the input, each intermediate step of an algorithm produces a unique result. For example, given the values

$$a = 1, \qquad b = 5, \qquad c = 3,$$

at line 2 of the example algorithm, x will be set to 5 regardless of what person or machine executes the algorithm.

An algorithm stops after finitely many steps answering the given question. For example, the example algorithm stops after three steps and produces the largest of the three given values.

An algorithm receives input and produces output. The example algorithm receives, as input, the values a, b, and c and produces, as output, the value x.

An algorithm must be general. The example algorithm 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 10).

In Section 3.2 we introduce a more formal way to specify algorithms and we give several additional examples of algorithms.

SECTION REVIEW EXERCISES

1. What is an algorithm?
2. Describe the following properties an algorithm must have: precision, uniqueness, finiteness, input, output, and generality.
3. What is a trace of an algorithm?

EXERCISES

1. Write an algorithm that finds the smallest element among a, b, and c.
2. 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.
3. Write the standard method of adding two positive decimal integers, taught in elementary schools, as an algorithm.
4. Consult the telephone book for the instructions for making a long-distance call. Which properties of an algorithm—precision, uniqueness, finiteness, input, output, generality—are present? Which properties are lacking?

3.2 NOTATION FOR ALGORITHMS

Although ordinary language is sometimes adequate to specify an algorithm, many mathematicians and computer scientists prefer **pseudocode** because of its precision, structure, and universality. Pseudocode is so named because it resembles the actual code (programs) of languages such as Pascal and C++. There are many versions of pseudocode. Unlike actual computer languages, which fuss over semicolons, uppercase and lowercase letters, special words, and so on, any version of pseudocode is acceptable as long as its instructions are unambiguous and it resembles in form, if not in exact syntax, the pseudocode described in this section.

As our first example of pseudocode, we rewrite the algorithm of Section 3.1, which finds the maximum of three numbers.

ALGORITHM 3.2.1 FINDING THE MAXIMUM OF THREE NUMBERS

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

Input: Three numbers a, b, and c
Output: x, the largest of a, b, and c

```
1.  procedure max(a, b, c)
2.    x := a
3.    if b > x then    // if b is larger than x, update x
4.      x := b
5.    if c > x then    // if c is larger than x, update x
6.      x := c
7.    return(x)
8.  end max
```

■

Our algorithms consist of a title, a brief description of the algorithm, the input to and output of the algorithm, and the procedures containing the instructions of the algorithm. Algorithm 3.2.1 consists of a single procedure. To make it

convenient to refer to individual lines within a procedure, we will sometimes number some of the lines. The procedure in Algorithm 3.2.1 has eight numbered lines. The first line of a procedure will consist of the word **procedure**, then the name of the procedure, and then, in parentheses, the parameters supplied to the procedure. The parameters describe the data, variables, arrays, and so on, that are available to the procedure. In Algorithm 3.2.1, the parameters supplied to the procedure are the three numbers a, b, and c. The last line of a procedure consists of the word **end** followed by the name of the procedure. Between the **procedure** and **end** lines are the executable lines of the procedure. Lines 2–7 are the executable lines of the procedure in Algorithm 3.2.1.

When the procedure in Algorithm 3.2.1 executes, at line 2 we set x to a. At line 3, b and x are compared. If b is greater than x, we execute line 4

$$x := b$$

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

$$x := c$$

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

At line 7 we return the value of x, which is equal to the largest of the numbers a, b, and c, to the invoker of the procedure and terminate the procedure. Algorithm 3.2.1 has correctly found the largest of three numbers.

In general, in the **if–then** structure

if p **then**
  action

if condition p is true, *action* is executed and control passes to the statement following *action*. If condition p is false, control immediately passes to the statement following *action*.

An alternative form is the **if–then–else** structure. In the if–then–else structure

if p **then**
  action 1
else
  action 2

if condition p is true, *action* 1 (but not *action* 2) is executed and control passes to the statement following *action* 2. If condition p is false, *action* 2 (but not *action* 1) is executed and control passes to the statement following *action* 2.

As shown, we use indentation to identify the statements that make up *action*. In addition, if *action* consists of multiple statements, we delimit those statements with the words **begin** and **end**. An example of a multiple-statement *action* in an if statement is

if $x \geq 0$ **then**
  begin
  $x := x - 1$
  $a := b + c$
  end

Two slash marks // signal the beginning of a **comment**, which then extends to the end of the line. An example of a comment in Algorithm 3.2.1 is

// if b is larger than x, update x

Comments help the reader understand the algorithm but are not executed.

The **return**(x) statement terminates a procedure and returns the value of x to the invoker of the procedure. The statement **return** [without the (x)] simply terminates a procedure. If there is no return statement, the procedure terminates just before the **end** line.

A procedure that contains a **return**(x) statement is a function. The domain consists of all valid values for the parameters, and the range is the set of all values that may be returned by the procedure.

When using pseudocode, we will use the usual arithmetic operators $+$, $-$, $*$ (for multiplication), and / as well as the relational operators $=$, $\neq$, $<$, $>$, $\leq$, and $\geq$ and the logical operators **and**, **or**, and **not**. We will use $=$ to denote the equality operator and $:=$ to denote the assignment operator. We will sometimes use less formal statements (*example*: Choose an element x in S.) when to do otherwise would obscure the meaning. In general, solutions to exercises that request algorithms should be written in the form illustrated by Algorithm 3.2.1.

The lines of a procedure, which are executed sequentially, are typically assignment statements, conditional statements (if statements), loops, return statements, and combinations of these statements. One useful loop structure is the **while loop**

while p **do**
  action

in which *action* is repeatedly executed as long as p is true. We call *action* the **body** of the loop. As in the if statement, if *action* consists of multiple statements, we delimit those statements with the words **begin** and **end**. We illustrate the while loop in Algorithm 3.2.2 that finds the largest value in a sequence. As in Algorithm 3.2.1, we step through the numbers one by one and update the variable that holds the largest. We use a while loop to step through the numbers.

ALGORITHM 3.2.2 FINDING THE LARGEST ELEMENT IN A FINITE SEQUENCE

This algorithm finds the largest number in the sequence $s_1, s_2, \ldots, s_n$. This version uses a while loop.

Input: The sequence $s_1, s_2, \ldots, s_n$ and the length n of the sequence

Output: *large*, the largest element in this sequence

```
1.  procedure find_large(s, n)
2.     large := s_1
3.     i := 2
4.     while i ≤ n do
5.        begin
6.        if s_i > large then    // a larger value was found
7.           large := s_i
8.        i := i + 1
9.        end
10.    return(large)
11. end find_large
```

■

We trace Algorithm 3.2.2 when $n = 4$ and s is the sequence

$$s_1 = -2, \qquad s_2 = 6, \qquad s_3 = 5, \qquad s_4 = 6.$$

At line 2 we set *large* to s_1; in this case we set *large* to -2. Next, at line 3, i is set to 2. At line 4 we test whether $i \leq n$; in this case we test whether $2 \leq 4$. Since this condition is true, we execute the body of the while loop (lines 5–9). At line 6 we test whether $s_i > large$; in this case we test whether $s_2 > large$ $(6 > -2)$. Since the condition is true, we execute line 7; *large* is set to 6. At line 8, i is set to 3. We then return to line 4.

We again test whether $i \leq n$; in this case we test whether $3 \leq 4$. Since this condition is true, we execute the body of the while loop. At line 6 we test whether $s_i > large$; in this case we test whether $s_3 > large$ $(5 > 6)$. Since the condition is false, we skip line 7. At line 8, i is set to 4. We then return to line 4.

We again test whether $i \leq n$; in this case we test whether $4 \leq 4$. Since this condition is true, we execute the body of the while loop. At line 6 we test whether $s_i > large$; in this case we test whether $s_4 > large$ $(6 > 6)$. Since the condition is false, we skip line 7. At line 8, i is set to 5. We then return to line 4.

We again test whether $i \leq n$; in this case we test whether $5 \leq 4$. Since the condition is false, we terminate the while loop and arrive at line 10, where we return *large* (6). We have found the largest element in the sequence.

In Algorithm 3.2.2 we stepped through a sequence by using the variable i that took on the integer values 1 through n. This kind of loop is so common that a special loop, called the **for loop**, is often used instead of the while loop. The form of the for loop is

for *var* := *init* **to** *limit* **do**
  action

As in the previous if statement and while loop, if *action* consists of multiple statements, we delimit the statements with the words **begin** and **end**. When the for loop is executed, *action* is executed for values of *var* from *init* to *limit*. More precisely, *init* and *limit* are expressions that have integer values. The variable *var* is first set to the value *init*. If $var \leq limit$, we execute *action* and then add 1 to *var*. The process is then repeated. Repetition continues until $var > limit$. Notice that if $init > limit$, *action* will not be executed at all.

Algorithm 3.2.2 may be rewritten in the following way using a for loop.

ALGORITHM 3.2.3 FINDING THE LARGEST ELEMENT IN A FINITE SEQUENCE

This algorithm finds the largest number in the sequence $s_1, s_2, \ldots, s_n$. This version uses a for loop.

Input: The sequence $s_1, s_2, \ldots, s_n$ and the length n of the sequence
Output: *large*, the largest element in this sequence

```
1.  procedure find_large(s, n)
2.    large := s_1
3.    for i := 2 to n do
4.      if s_i > large then    // a larger value was found
5.        large := s_i
6.    return(large)
7.  end find_large
```

■

When we develop an algorithm, it is often a good idea to break the original problem into two or more subproblems. A procedure can be developed to solve each subproblem, after which these procedures can be combined to provide a solution to the original problem. Our final algorithms illustrate these ideas.

Suppose that we want an algorithm to find the least prime number that exceeds a given positive integer. More precisely, the problem is: Given a positive integer n, find the least prime p satisfying $p > n$. We might break this problem up into two subproblems. We could first develop an algorithm to determine whether a positive integer is prime. We could then use this algorithm to find the least prime greater than a given positive integer.

Algorithm 3.2.4 tests whether a positive integer m is prime. We simply test whether any integer between 2 and $m-1$ divides m. If we find an integer between 2 and $m-1$ that divides m, m is not prime. If we fail to find an integer between 2 and $m-1$ that divides m, m is prime. (Exercise 34 shows that it suffices to check integers between 2 and $\sqrt{m}$ as possible divisors.) Algorithm 3.2.4 shows that we allow procedures to return **true** or **false**.

ALGORITHM 3.2.4 TESTING WHETHER A POSITIVE INTEGER IS PRIME

This algorithm tests whether the positive integer m is prime. The output is **true** if m is prime and **false** if m is not prime.

Input: m, a positive integer

Output: **true**, if m is prime; **false**, if m is not prime

```
procedure is_prime(m)
  for i := 2 to m − 1 do
    if m mod i = 0 then    // i divides m
      return(false)
  return(true)
end is_prime
```

■

Algorithm 3.2.5, which finds the least prime exceeding the positive integer n, uses Algorithm 3.2.4. To invoke a procedure that returns a value as in Algorithm 3.2.4, we name it. To invoke a procedure named, say, *proc*, that does not return a value, we write

$$\textbf{call } proc(p_1, p_2, \ldots, p_k),$$

where $p_1, p_2, \ldots, p_k$ are the arguments passed to *proc*.

ALGORITHM 3.2.5 FINDING A PRIME LARGER THAN A GIVEN INTEGER

This algorithm finds the smallest prime that exceeds the positive integer n.

Input: n, a positive integer

Output: m, the smallest prime greater than n

```
procedure large_prime(n)
  m := n + 1
  while not is_prime(m) do
    m := m + 1
  return(m)
end large_prime
```

■

Since the number of primes is infinite (see Exercise 35), the procedure in Algorithm 3.2.5 will eventually terminate.

SECTION REVIEW EXERCISES

1. What is pseudocode?
2. What is a procedure?
3. What is an if–then structure and how does it operate?
4. What is an if–then–else structure and how does it operate?
5. What is a comment?
6. What is a return statement and how does it operate?
7. What is a while loop and how does it operate?
8. What is a for loop and how does it operate?
9. What is a call statement and how does it operate?

EXERCISES

Write all algorithms in the style of Algorithms 3.2.1–3.2.5.

1. Show how Algorithm 3.2.1 finds the largest of the numbers $a = 4$, $b = -3$, and $c = 5$.
2. Show how Algorithm 3.2.1 finds the largest of the numbers $a = b = 4$ and $c = 2$.
3. Show how Algorithm 3.2.1 finds the largest of the numbers $a = b = c = 8$.
4. Show how Algorithm 3.2.2 finds the largest element in the sequence

$$s_1 = 2,\ s_2 = 3,\ s_3 = 8,\ s_4 = 6.$$

5. Show how Algorithm 3.2.2 finds the largest element in the sequence

$$s_1 = 8,\ s_2 = 8,\ s_3 = 4,\ s_4 = 1.$$

6. Show how Algorithm 3.2.2 finds the largest element in the sequence

$$s_1 = 1,\ s_2 = 1,\ s_3 = 1,\ s_4 = 1.$$

7. Show how Algorithm 3.2.4 tests whether 3 is prime.
8. Show how Algorithm 3.2.4 tests whether 4 is prime.
9. Show how Algorithm 3.2.4 tests whether 9 is prime.
10. Show how Algorithm 3.2.5 finds a prime that exceeds 1.
11. Show how Algorithm 3.2.5 finds a prime that exceeds 10.
12. Show how Algorithm 3.2.5 finds a prime that exceeds 1000.
13. Write an algorithm that outputs the smallest element in the sequence $s_1, \ldots, s_n$.
14. Write an algorithm that outputs the largest and second-largest elements in the sequence $s_1, \ldots, s_n$.
15. Write an algorithm that outputs the smallest and second-smallest elements in the sequence $s_1, \ldots, s_n$.
16. Write an algorithm that outputs the largest and smallest elements in the sequence $s_1, \ldots, s_n$.
17. Write an algorithm that outputs the index of the first occurrence of the largest element in the sequence $s_1, \ldots, s_n$. *Example*: If the sequence were

$$6.2 \quad 8.9 \quad 4.2 \quad 8.9,$$

the algorithm would output the value 2.

18. Write an algorithm that outputs the index of the last occurrence of the largest element in the sequence $s_1, \ldots, s_n$. *Example*: If the sequence were

$$6.2 \quad 8.9 \quad 4.2 \quad 8.9,$$

the algorithm would output the value 4.

19. Write an algorithm that outputs 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 outputs the value 0. *Example*: If the sequence were

$$\text{'MARY'} \quad \text{'JOE'} \quad \text{'MARK'} \quad \text{'RUDY'},$$

and *key* were 'MARK', the algorithm would output the value 3.

20. Write an algorithm that outputs 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 outputs the value 0.
21. Write an algorithm that outputs the index of the first item that is less than its predecessor in the sequence $s_1, \ldots, s_n$. If the items are in increasing order, the algorithm outputs the value 0. *Example*: If the sequence were

$$\text{'AMY'} \quad \text{'BRUNO'} \quad \text{'ELIE'} \quad \text{'DAN'} \quad \text{'ZEKE'},$$

the algorithm would output the value 4.

22. Write an algorithm that outputs the index of the first item that is greater than its predecessor in the sequence $s_1, \ldots, s_n$. If the items are in decreasing order, the algorithm outputs the value 0.
23. Write an algorithm that reverses the sequence $s_1, \ldots, s_n$. *Example*: If the sequence were

$$\text{'AMY'} \quad \text{'BRUNO'} \quad \text{'ELIE'},$$

the reversed sequence would be

$$\text{'ELIE'} \quad \text{'BRUNO'} \quad \text{'AMY'}.$$

24. Write the standard method of multiplying two positive decimal integers, taught in elementary schools, as an algorithm.
25. Write an algorithm that receives as input the matrix of a relation R and tests whether R is reflexive.
26. Write an algorithm that receives as input the matrix of a relation R and tests whether R is antisymmetric.
27. Write an algorithm that receives as input the matrix of a relation R and tests whether R is a function.
28. 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}.
29. 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 $R_2 \circ R_1$.
30. Write an algorithm that sums the sequence of numbers $s_1, s_2, \ldots, s_n$.
31. Write an algorithm whose input is a sequence of numbers $s_1, s_2, \ldots, s_n$, sorted in increasing order, and another number x. The algorithm inserts x into the sequence so that the resulting sequence is sorted. *Example*: If the input sequence is

$$2,\ 6,\ 12,\ 14$$

and $x = 5$, the resulting sequence is

$$2,\ 5,\ 6,\ 12,\ 14.$$

32. Write an algorithm whose input is a sequence $s_1, s_2, \ldots, s_n$ of numbers and another number x. 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.

33. Write an algorithm that receives a bit string $b_1 \ldots b_n$ (b_i is the ith bit). The algorithm rearranges the bit string so that all of the zeros precede all of the ones. *Example*: If the input is 01011, the rearranged bit string is 00111.

34. Show that the positive integer $m \geq 2$ is prime if and only if no integer between 2 and $\sqrt{m}$ divides m.

35. Show that the number of primes is infinite by completing the following argument.

It suffices to show that if p is prime, there is a prime larger than p. Let $p_1 < p_2 < \cdots < p_k = p$ denote the primes less than or equal to p, and let $n = p_1 p_2 \cdots p_k + 1$. Show that any prime that divides n is larger than p.

3.3 THE EUCLIDEAN ALGORITHM

An old and famous algorithm is the **Euclidean algorithm** for finding the greatest common divisor of two integers. The greatest common divisor of two integers m and n (not both zero) is the largest positive integer that divides both m and n. For example, the greatest common divisor of 4 and 6 is 2, and the greatest common divisor of 3 and 8 is 1. We use the notion of greatest common divisor when we check to see if a fraction m/n, where m and n are integers, is in lowest terms. If the greatest common divisor of m and n is 1, m/n is in lowest terms; otherwise, we can reduce m/n. For example, 4/6 is not in lowest terms because the greatest common divisor of 4 and 6 is 2, not 1. (We can divide both 4 and 6 by 2.) The fraction 3/8 is in lowest terms because the greatest common divisor of 3 and 8 is 1. After discussing the divisibility of integers, we examine the greatest common divisor in detail and present the Euclidean algorithm.

If a, b, and q are integers, $b \neq 0$, satisfying $a = bq$, we say that b **divides** a and we write $b \mid a$. In this case, we call q the **quotient** and call b a **divisor** of a. If b does not divide a, we write $b \nmid a$.

EXAMPLE 3.3.1 Since $21 = 3 \cdot 7$, 3 divides 21 and we write $3 \mid 21$. The quotient is 7. ■

Let m and n be integers that are not both zero. Among all the integers that divide both m and n, there is a largest divisor known as the **greatest common divisor** of m and n.

DEFINITION 3.3.2 Let m and n be integers with not both m and n zero. A *common divisor* of m and n is an integer that divides both m and n. The *greatest common divisor*, written

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

is the largest common divisor of m and n. ■

EXAMPLE 3.3.3 The positive divisors of 30 are

$$1,\quad 2,\quad 3,\quad 5,\quad 6,\quad 10,\quad 15,\quad 30$$

and the positive divisors of 105 are

$$1,\quad 3,\quad 5,\quad 7,\quad 15,\quad 21,\quad 35,\quad 105;$$

thus the positive common divisors of 30 and 105 are

$$1,\quad 3,\quad 5,\quad 15.$$

It follows that the greatest common divisor of 30 and 105, gcd(30, 105), is 15. ■

The properties of divisors given in the following theorem will be useful in our subsequent work in this section.

THEOREM 3.3.4

Let m, n, and c be integers.

(a) If c is a common divisor of m and n, then

$$c \mid (m + n).$$

(b) If c is a common divisor of m and n, then

$$c \mid (m - n).$$

(c) If $c \mid m$, then $c \mid mn$.

Proof. (a) Let c be a common divisor of m and n. Since $c \mid m$,

$$m = cq_1 \tag{3.3.1}$$

for some integer q_1. Similarly, since $c \mid n$,

$$n = cq_2 \tag{3.3.2}$$

for some integer q_2. If we add equations (3.3.1) and (3.3.2), we obtain

$$m + n = cq_1 + cq_2 = c(q_1 + q_2).$$

Therefore, c divides $m + n$ (with quotient $q_1 + q_2$). We have proved part (a).

The proofs of parts (b) and (c) are left to the reader (see Exercises 11 and 12). ■

Recall that if a is a nonnegative integer and b is a positive integer, $a \bmod b$ is the remainder when a is divided by b. For example, $105 \bmod 30 = 15$. The Euclidean algorithm is based on the fact that if $r = a \bmod b$, then

$$\gcd(a, b) = \gcd(b, r). \tag{3.3.3}$$

Before proving (3.3.3), we illustrate how the Euclidean algorithm uses it to find the greatest common divisor.

EXAMPLE 3.3.5 Since $105 \bmod 30 = 15$, by (3.3.3)

$$\gcd(105, 30) = \gcd(30, 15).$$

Since $30 \bmod 15 = 0$, by (3.3.3)

$$\gcd(30, 15) = \gcd(15, 0).$$

By inspection, $\gcd(15, 0) = 15$. Therefore,

$$\gcd(105, 30) = \gcd(30, 15) = \gcd(15, 0) = 15.$$ ■

In Example 3.3.3, we obtained the greatest common divisor of 105 and 30 by listing all of the divisors of 105 and 30. By our using (3.3.3), two simple modulus operations produce the greatest common divisor. We next prove (3.3.3).

THEOREM 3.3.6

If a is a nonnegative integer, b is a positive integer, and $r = a \bmod b$, then

$$\gcd(a, b) = \gcd(b, r).$$

Proof. If we divide a by b, we obtain a quotient q and modulus (remainder) r satisfying

$$a = bq + r, \quad 0 \leq r < b.$$

We show that the set of common divisors of a and b is equal to the set of common divisors of b and r, thus proving the theorem.

Let c be a common divisor of a and b. By Theorem 3.3.4(c), $c \mid bq$. Since $c \mid a$ and $c \mid bq$, by Theorem 3.3.4(b), $c \mid a - bq \ (= r)$. Thus c is a common divisor of b and r. Conversely, if c is a common divisor of b and r, then $c \mid bq$ and $c \mid bq + r \ (= a)$ and c is a common divisor of a and b. Thus the set of common divisors of a and b is equal to the set of common divisors of b and r. Therefore,

$$\gcd(a, b) = \gcd(b, r).$$ ■

We next formally state the Euclidean algorithm as Algorithm 3.3.7.

ALGORITHM 3.3.7 EUCLIDEAN ALGORITHM

This algorithm finds the greatest common divisor of the nonnegative integers a and b, where not both a and b are zero.

Input: a and b (nonnegative integers, not both zero)

Output: Greatest common divisor of a and b

```
1.  procedure gcd(a, b)
2.    // make a largest
3.    if a < b then
4.      swap(a, b)
        // that is, execute
        // temp := a
        // a := b
        // b := temp
5.    while b ≠ 0 do
6.      begin
7.      r := a mod b
8.      a := b
9.      b := r
10.     end
11.   return(a)
12. end gcd
```

■

EXAMPLE 3.3.8

We show how Algorithm 3.3.7 finds gcd(504,396).

Let $a = 504$ and $b = 396$. Since $a > b$, we move to line 5. Since $b \neq 0$, we proceed to line 7, where we set r to

$$a \bmod b = 504 \bmod 396 = 108.$$

We then move to lines 8 and 9, where we set a to 396 and b to 108. We then return to line 5.

Since $b \neq 0$, we proceed to line 7, where we set r to

$$a \bmod b = 396 \bmod 108 = 72.$$

We then move to lines 8 and 9, where we set a to 108 and b to 72. We then return to line 5.

Since $b \neq 0$, we proceed to line 7, where we set r to

$$a \bmod b = 108 \bmod 72 = 36.$$

We then move to lines 8 and 9, where we set a to 72 and b to 36. We then return to line 5.

Since $b \neq 0$, we proceed to line 7, where we set r to

$$a \bmod b = 72 \bmod 36 = 0.$$

We then move to lines 8 and 9, where we set a to 36 and b to 0. We then return to line 5.

This time $b = 0$, so we skip to line 11, where we return a (36), the greatest common divisor of 396 and 504. ■

We note that the while loop in the Euclidean algorithm (lines 5–10) always terminates since at the bottom of the loop (lines 8 and 9), the values of a and b are updated to *smaller* values. Since nonnegative integers cannot decrease indefinitely, eventually b becomes zero and the loop terminates. By Theorem 3.3.6, the value returned at line 11 is $\gcd(a, b)$.

SECTION REVIEW EXERCISES

1. Define *b divides a*.
2. Define *b is a divisor of a*.
3. Define *quotient*.
4. What is a common divisor?
5. What is the greatest common divisor?
6. State the Euclidean algorithm.
7. What key fact is the basis for the Euclidean algorithm?

EXERCISES

Use the Euclidean algorithm to find the greatest common divisor of each pair of integers in Exercises 1–10.

1. 60, 90
2. 110, 273
3. 220, 1400
4. 315, 825
5. 20, 40
6. 331, 993
7. 2091, 4807
8. 2475, 32670
9. 67942, 4209
10. 490256, 337
11. Let m, n, and c be integers. Show that if c is a common divisor of m and n, then $c \mid (m - n)$.
12. Let m, n, and c be integers. Show that if $c \mid m$, then $c \mid mn$.
13. Suppose that a, b, and c are positive integers. Show that if $a \mid b$ and $b \mid c$, then $a \mid c$.
14. Find two numbers a and b, each less than 100, that maximize the number of iterations of the while loop of Algorithm 3.3.7.
15. Show that Algorithm 3.3.7 correctly finds $\gcd(a, b)$ even if lines 3 and 4 are deleted.
16. If a and b are positive integers, show that $\gcd(a, b) = \gcd(a, a + b)$.

★ 17. Given $a > b \geq 0$, let $r_0 = a$, $r_1 = b$, and r_i equal the value of r after the $(i-1)$st time the while loop is executed in Algorithm 3.3.7 (e.g., $r_2 = a \bmod b$). Suppose that r_n is the first r-value that is zero so that $\gcd(a, b) = r_{n-1}$. Show that we may successively write

$$\gcd(a, b) = s_{n-3}r_{n-2} + t_{n-3}r_{n-3}$$

$$\gcd(a, b) = s_{n-4}r_{n-3} + t_{n-4}r_{n-4}$$

$$\vdots$$

$$\gcd(a, b) = s_0 r_1 + t_0 r_0$$

for some integers s_i and t_i.

★ 18. Use the method of Exercise 17 to write the greatest common divisor of each pair of integers a and b in Exercises 1–10 in the form $ta + sb$.

★ 19. Show that if p is a prime number, a and b are positive integers, and $p \mid ab$, then $p \mid a$ or $p \mid b$.

20. Give an example of positive integers p, a, and b where $p \mid ab$, $p \nmid a$, and $p \nmid b$.

21. Show that if $a > b \geq 0$, then

$$\gcd(a, b) = \gcd(a - b, b).$$

22. Using Exercise 21, write an algorithm to compute the greatest common divisor of two nonnegative integers a and b, not both zero, that uses subtraction but not the modulus operation.

★ 23. Show that for some n, postage of n cents or more can be achieved by using only p-cent and q-cent stamps provided that $\gcd(p, q) = 1$. *Hint*: Use Exercise 17 and the method of Example 1.6.7.

24. Show that if $\gcd(p, q) > 1$, the statement in Exercise 23 is false.

3.4 RECURSIVE ALGORITHMS

A **recursive procedure** is a procedure that invokes itself. A **recursive algorithm** is an algorithm that contains a recursive procedure. 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 3.4.1 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

$$n! = n \cdot (n-1)!,$$

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 $(n-2)!$, ...] 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 3.4.1 summarizes this process.

TABLE 3.4.1
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

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 3.4.2 shows how the subproblems are combined to compute 5!.

TABLE 3.4.2
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 the equation

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

Algorithm 3.4.2 Computing n Factorial

This recursive algorithm computes $n!$.

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

Output: $n!$

```
1.  procedure factorial(n)
2.    if n = 0 then
3.      return(1)
4.    return(n * factorial(n - 1))
5.  end factorial
```

■

We show how Algorithm 3.4.2 computes $n!$ for several values of n. If $n = 0$, at line 3 the procedure correctly returns the value 1.

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

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

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

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

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

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

The preceding arguments may be generalized using mathematical induction to *prove* that Algorithm 3.4.2 correctly outputs the value of $n!$ for any nonnegative integer n.

Theorem 3.4.3 *Algorithm 3.4.2 outputs the value of $n!$, $n \geq 0$.*

Proof.

Basis Step ($n = 0$). We have already observed that if $n = 0$, Algorithm 3.4.2 correctly outputs the value of 0! (1).

Inductive Step. Assume that Algorithm 3.4.2 correctly outputs the value of $(n-1)!$, $n > 0$. Now suppose that n is input to Algorithm 3.4.2. Since $n \neq 0$, when we execute the procedure in Algorithm 3.4.2 we proceed to line 4. By the inductive assumption, the procedure correctly computes the value of $(n-1)!$. At line 4, the procedure correctly computes the value $(n-1)! \cdot n = n!$.

Therefore, Algorithm 3.4.2 correctly outputs the value of $n!$ for every integer $n \geq 0$. ■

There must be some situations in which a recursive procedure does *not* invoke itself; otherwise, it would invoke itself forever. In Algorithm 3.4.2, if $n = 0$, the procedure does not invoke itself. We call the values for which a recursive procedure does not invoke itself the *base cases*. To summarize, every recursive procedure 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 procedure, and the Inductive Step of a proof by mathematical induction corresponds to the part of a recursive procedure where the procedure calls itself.

In Example 1.6.6, we gave a proof using mathematical induction that given a deficient $n \times n$ 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 "el"; see Figure 1.6.3). We now translate the inductive proof into a recursive algorithm to construct a tiling by right trominoes of a deficient $n \times n$ board where n is a power of 2.

ALGORITHM 3.4.4 TILING A DEFICIENT BOARD WITH TROMINOES

This algorithm constructs a tiling by right trominoes of a deficient $n \times n$ 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 a deficient $n \times n$ board

procedure *tile*(n, L)
1. **if** $n = 2$ **then**
2. **begin**
   // the board is a right tromino T
3. tile with T
4. **return**
5. **end**
6. divide the board into four $(n/2) \times (n/2)$ 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 1.6.5
   // consider each of the squares covered by the center tromino as
   // missing and denote the missing squares as m_1, m_2, m_3, m_4
9. **call** *tile*$(n/2, m_1)$
10. **call** *tile*$(n/2, m_2)$
11. **call** *tile*$(n/2, m_3)$
12. **call** *tile*$(n/2, m_4)$

end *tile* ■

We next give a recursive algorithm to compute the greatest common divisor of two nonnegative integers, not both zero.

Theorem 3.3.6 states that if a is a nonnegative integer, b is a positive integer, and $r = a \bmod b$, then

$$\gcd(a, b) = \gcd(b, r). \tag{3.4.1}$$

[$\gcd(x, y)$ denotes the greatest common divisor of x and y.] Equation (3.4.1) is inherently recursive; it reduces the problem of computing the greatest common divisor of a and b to a smaller problem—that of computing the greatest common divisor of b and r. Recursive Algorithm 3.4.5, which computes the greatest common divisor, is based on equation (3.4.1).

ALGORITHM 3.4.5 RECURSIVELY COMPUTING THE GREATEST COMMON DIVISOR

This algorithm recursively finds the greatest common divisor of the nonnegative integers a and b, where not both a and b are zero. (Algorithm 3.3.7 gives a nonrecursive algorithm for computing the greatest common divisor.)

Input: a and b (nonnegative integers, not both zero)
Output: Greatest common divisor of a and b

```
    procedure gcd_recurs(a, b)
       // make a largest
1.     if a < b then
2.        swap(a, b)
3.     if b = 0 then
4.        return(a)
5.     r := a mod b
6.     return(gcd_recurs(b, r))
    end gcd_recurs
```

■

We present one final example of a recursive algorithm.

Example 3.4.6 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

$$\text{walk}(1) = 1, \qquad \text{walk}(2) = 2.$$

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

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

For example,

$$\text{walk}(4) = \text{walk}(3) + \text{walk}(2) = 3 + 2 = 5.$$

We can write a recursive algorithm to compute walk(n) by translating the equation

$$\text{walk}(n) = \text{walk}(n-1) + \text{walk}(n-2)$$

directly into an algorithm. The base cases are $n = 1$ and $n = 2$. ■

Algorithm 3.4.7 Robot Walking

This algorithm computes the function defined by

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

Input: n

Output: walk(n)

procedure *robot_walk*(n)
 if $n = 1$ **or** $n = 2$ **then**
  return(n)
 return(*robot_walk*($n - 1$) + *robot_walk*($n - 2$))
end *robot_walk*

The sequence

$$\text{walk}(1),\ \text{walk}(2),\ \text{walk}(3),\quad \ldots,$$

whose values begin

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

is called the **Fibonacci sequence** in honor of Leonardo Fibonacci (ca. 1170–1250), an Italian merchant and mathematician. Subsequently, we denote the Fibonacci sequence as

$$f_1, f_2, \ldots.$$

This sequence is defined by the equations

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

FIGURE 3.4.1
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 originally arose in a puzzle about rabbits (see Exercises 20 and 21). After returning from the Orient in 1202, Fibonacci wrote his most famous work, *Liber Abaci*, 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. For example, the number of clockwise spirals and the number of counterclockwise spirals in pine cones are found in the Fibonacci sequence. Figure 3.4.1 shows a pine cone with 13 clockwise spirals and 8 counterclockwise spirals. In Section 3.6, the Fibonacci sequence appears in the analysis of the Euclidean algorithm.

SECTION REVIEW EXERCISES

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

Exercises

1. Trace Algorithm 3.4.2 for $n = 4$.
2. Trace Algorithm 3.4.4 when $n = 4$ and the missing square is the upper-left corner square.
3. Trace Algorithm 3.4.4 when $n = 8$ and the missing square is four from the left and six from the top.
4. Trace Algorithm 3.4.5 for $a = 5$ and $b = 0$.
5. Trace Algorithm 3.4.5 for $a = 55$ and $b = 20$.
6. Trace Algorithm 3.4.7 for $n = 4$.
7. Trace Algorithm 3.4.7 for $n = 5$.
8. (a) Use the formulas

$$s_1 = 1, \qquad s_n = s_{n-1} + n, \qquad n \geq 2,$$

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.

9. (a) Use the formulas

$$s_1 = 2, \qquad s_n = s_{n-1} + 2n, \qquad n \geq 2,$$

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.

10. (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.

11. Write a recursive algorithm that computes the greatest common divisor of two nonnegative integers, not both zero, that uses subtractions but no modulus operations (see Exercise 22, Section 3.3).
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 recursive algorithm to sort a finite sequence of numbers. Give a proof using mathematical induction that your algorithm is correct.
16. Write a recursive algorithm whose input is a bit string and whose output is a rearranged bit string in which all of the zeros precede all of the ones. Give a proof using mathematical induction that your algorithm is correct.
17. Write a nonrecursive algorithm to compute $n!$.
18. ★ 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.
19. ★ 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 20–32 concern the Fibonacci sequence $\{f_n\}$.

20. 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 $a_n - a_{n-1} = a_{n-2}$. Explain why $a_n = f_n, n \geq 1$.
21. Fibonacci's original question was: Under the conditions of Exercise 20, how many pairs of rabbits are there after one year? Answer Fibonacci's question.
22. Use mathematical induction to show that

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

23. Use mathematical induction to show that

$$f_n^2 = f_{n-1} f_{n+1} + (-1)^n, \qquad n \geq 2.$$

24. Show that

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

25. Use mathematical induction to show that

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

26. Use mathematical induction to show that f_n is even if and only if $n + 1$ is divisible by 3, $n \geq 1$.
27. Use mathematical induction to show that for $n \geq 5$,

$$f_n > \left(\frac{3}{2}\right)^n.$$

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

$$f_n < 2^n.$$

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

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

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.

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

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

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

33. [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

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

34. [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.

3.5 COMPLEXITY 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 storage 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. **Complexity of an algorithm** refers to the amount of time and space required 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 (Theorem 2.1.4), 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 3.5.1) that, except for small values of n, it would be infeasible 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 complexity 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. 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.

We could measure the time required by an algorithm by counting the number of instructions executed. Alternatively, we could use a cruder time estimate such as the number of times each loop is executed. If the principal activity of an algorithm is making comparisons, as might happen in a sorting routine, we might count the number of comparisons. Usually, we are interested in general estimates since, as we have already observed, the actual performance of a program implementation of an algorithm depends on many factors.

TABLE 3.5.1
Time to execute an algorithm if one step takes 1 microsecond to execute†

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

† $\lg n$ denotes $\log_2 n$ (the logarithm of n to base 2).

EXAMPLE 3.5.1 A reasonable definition of the size of input for Algorithm 3.2.2 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 3.2.2 for input of size n are each $n - 1$ since the loop is always executed $n - 1$ times. ■

Often 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

$$t(n) = 60n^2 + 5n + 1 \tag{3.5.1}$$

for input of size n. For large n, the term $60n^2$ is approximately equal to $t(n)$ (see Table 3.5.2). In this sense, $t(n)$ grows like $60n^2$.

TABLE 3.5.2
Comparing growth of $t(n)$ with $60n^2$

n	***$t(n) = 60n^2 + 5n + 1$***	***$60n^2$***
10	6051	6000
100	600,501	600,000
1000	60,005,001	60,000,000
10,000	6,000,050,001	6,000,000,000

If (3.5.1) 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 (3.5.1)], 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 3.5.2 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* $g(n)$ 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* $g(n)$ 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* $g(n)$ if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$. ■

Definition 3.5.2 can be loosely paraphrased as follows. $f(n) = O(g(n))$ if, except for constants and a finite number of exceptions, f is bounded above by g. $f(n) = \Omega(g(n))$ if, except for constants and a finite number of exceptions, f is bounded below by g. $f(n) = \Theta(g(n))$ if, except for constants and a finite number of exceptions, f is bounded above and below by g.

An expression of the form $f(n) = O(g(n))$ is sometimes referred to as a **big oh notation** for f. Similarly, $f(n) = \Omega(g(n))$ is sometimes referred to as an **omega notation** for f, and $f(n) = \Theta(g(n))$ is sometimes referred to as a **theta notation** for f.

According to the definition, if $f(n) = O(g(n))$, all that one can conclude is that, except for constants 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 $f(n) = \Theta(g(n))$, one can draw the conclusion that, except for constants 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 $n \neq \Theta(2^n)$. Unfortunately, it is not uncommon in the literature to find big oh notation used as if it were theta notation.

EXAMPLE 3.5.3 Since

$$60n^2 + 5n + 1 \le 60n^2 + 5n^2 + n^2 = 66n^2 \qquad \text{for } n \ge 1,$$

we may take $C_1 = 66$ in Definition 3.5.2 to obtain

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

Since

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

we may take $C_2 = 60$ in Definition 3.5.2 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 3.5.3 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 3.5.3 has to be modified.]

THEOREM 3.5.4 *Let*

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

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

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

Proof. Let

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

Then

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

Therefore,

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

Since

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

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

Thus

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

■

EXAMPLE 3.5.5 In this book we let $\lg n$ denote $\log_2 n$ (the logarithm of n to the base 2). Since $\lg n < n$ for $n \ge 1$ (see Figure 3.5.1),

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

thus

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

Also,

$$2n + 3\lg n \ge 2n \qquad \text{for } n \ge 1,$$

so

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

Therefore,

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

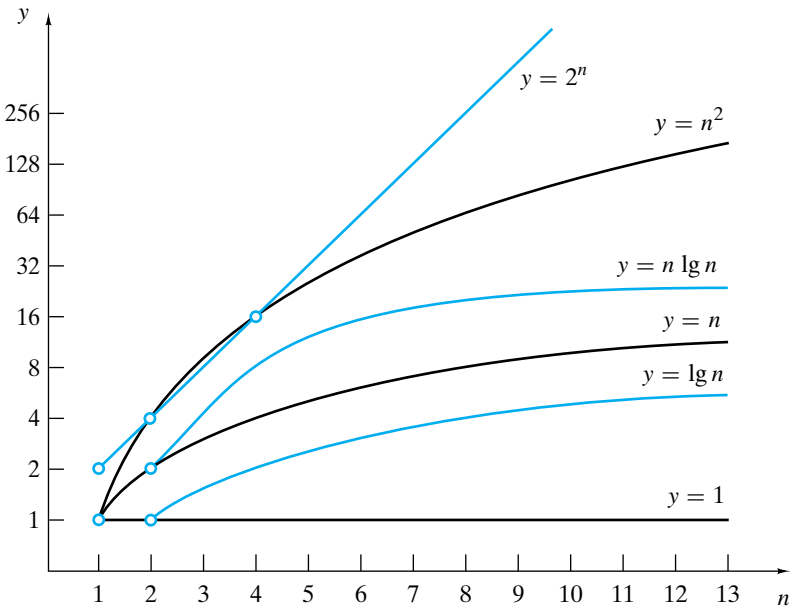


FIGURE 3.5.1
Growth of some common functions.

EXAMPLE 3.5.6 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

$$1 + 2 + \cdots + n \leq n + n + \cdots + n = n \cdot n = n^2 \tag{3.5.2}$$

for $n \geq 1$. 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

$$1 + 2 + \cdots + n \geq 1 + 1 + \cdots + 1 = n.$$

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 before, but first throw away the first half of the terms, to obtain

$$\begin{aligned} 1 + 2 + \cdots + n &\geq \lceil n/2 \rceil + \cdots + (n-1) + n \\ &\geq \lceil n/2 \rceil + \cdots + \lceil n/2 \rceil + \lceil n/2 \rceil \\ &= \lceil (n+1)/2 \rceil \lceil n/2 \rceil \geq (n/2)(n/2) = n^2/4. \end{aligned} \tag{3.5.3}$$

We can now conclude that

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

Therefore,

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

Example 3.5.7 If k is a positive integer and, as in Example 3.5.6, we replace each integer $1, 2, \ldots, n$ by n, we have

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

for $n \geq 1$; hence

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

We can also obtain a lower bound as in Example 3.5.6:

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

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 3.5.4 and the expression

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

in Example 3.5.7. A polynomial has a fixed number of terms, whereas the number of terms in the expression in Example 3.5.7 is dependent on the value of n. Furthermore, the polynomial in Theorem 3.5.4 is $\Theta(n^k)$, but the expression in Example 3.5.7 is $\Theta(n^{k+1})$.

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

Example 3.5.8 Using an argument similar to that in Example 3.5.6, we show that

$$\lg n! = \Theta(n \lg n).$$

By properties of logarithms, we have

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

Since lg is an increasing function,

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

We conclude that

$$\lg n! = O(n \lg n).$$

Now

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

A proof by mathematical induction (see Exercise 52) shows that if $n \geq 4$,

$$(n/2) \lg(n/2) \geq (n \lg n)/4.$$

The last inequalities combine to give

$$\lg n + \lg(n-1) + \cdots + \lg 2 + \lg 1 \geq (n \lg n)/4$$

for $n \geq 4$. Therefore,

$$\lg n! = \Omega(n \lg n).$$

It follows that

$$\lg n! = \Theta(n \lg n).$$ ■

EXAMPLE 3.5.9 Show that if $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$, then $f(n) = \Theta(h(n))$.

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 3.5.10 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 3.5.10, 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 3.5.11 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 3.5.3 that

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

Thus the worst-case time required by this algorithm is $\Theta(n^2)$. ■

Example 3.5.12 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 do
2.    for j := 1 to i do
3.      x := x + 1
```

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 3.5.6)

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

Thus a theta notation for the number of times the statement $x := x+1$ is executed is $\Theta(n^2)$. ■

Example 3.5.13 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 do
3.    begin
4.    x := x + 1
5.    i := ⌊i/2⌋
6.    end
```

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 4, we execute the statement $x := x + 1$ the first time. At line 5, i is reset to 4 and we return to line 2.

At line 2, the condition $i \geq 1$ is again true. At line 4, we execute the statement $x := x + 1$ the second time. At line 5, i is reset to 2 and we return to line 2.

At line 2, the condition $i \geq 1$ is again true. At line 4, we execute the statement $x := x + 1$ the third time. At line 5, i is reset to 1 and we return to line 2.

At line 2, the condition $i \geq 1$ is again true. At line 4, we execute the statement $x := x + 1$ the fourth time. At line 5, 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 4, we execute the statement $x := x + 1$ the first time. At line 5, 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

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

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

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

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

Suppose that n satisfies

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

At line 1, i is set to n. At line 2, the condition $i \geq 1$ is true. At line 4, we execute the statement $x := x + 1$ the first time. At line 5, 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, 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

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

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

Suppose that n satisfies

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

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 (for example, tromino tiling, Algorithm 3.4.4, is based on this idea). Example 3.5.13 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 3.5.14 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 do
3.    begin
4.    for i := 1 to j do
5.      x := x + 1
6.    j := ⌊j/2⌋
7.    end
```

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 $t(n) \geq n$ 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 6, 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 1.6.4) is equal to

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

Now

$$t(n) \leq \frac{n\left(1 - \frac{1}{2^k}\right)}{1 - \frac{1}{2}} = 2n\left(1 - \frac{1}{2^k}\right) \leq 2n,$$

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

EXAMPLE 3.5.15 Determine, in theta notation, the best-case, worst-case, and average-case times required to execute Algorithm 3.5.16, 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.

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 3.5.16 is

$$\Theta(1).$$

The worst-case time of Algorithm 3.5.16 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 3.5.16 is

$$\Theta(n).$$

Finally, consider the average-case time of Algorithm 3.5.16. 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

$$\frac{(1+2+\cdots+n)+n}{n+1} \le \frac{n^2+n}{n+1} \qquad \text{by (3.5.2)}$$
$$= \frac{n(n+1)}{n+1} = n.$$

Therefore, the average-case time of Algorithm 3.5.16 is

$$O(n).$$

Also,

$$\frac{(1+2+\cdots+n)+n}{n+1} \ge \frac{n^2/4+n}{n+1} \qquad \text{by (3.5.3)}$$
$$\ge \frac{n^2/4+n/4}{n+1} = \frac{n}{4}.$$

Therefore the average-case time of Algorithm 3.5.16 is

$$\Omega(n).$$

Thus the average-case time of Algorithm 3.5.16 is

$$\Theta(n).$$

For this algorithm, the worst-case and average-case times are both $\Theta(n)$. ■

ALGORITHM 3.5.16 SEARCHING AN UNORDERED SEQUENCE

Given the sequence

$$s_1, \quad s_2, \quad \cdots, \quad s_n$$

and a value *key*, this algorithm finds the location of *key*. If *key* is not found, the algorithm outputs 0.

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

```
1.  procedure linear_search(s, n, key)
2.    for i := 1 to n do
3.      if key = s_i then
4.        return(i)    // successful search
5.    return(0)    // unsuccessful search
6.  end linear_search
```

■

TABLE 3.5.3
Common growth functions

Theta Form†	*Name*
$\Theta(1)$	Constant
$\Theta(\lg\lg n)$	Log log
$\Theta(\lg n)$	Logarithmic
$\Theta(n)$	Linear
$\Theta(n \lg n)$	n log n
$\Theta(n^2)$	Quadratic
$\Theta(n^3)$	Cubic
$\Theta(n^m)$	Polynomial
$\Theta(m^n), m \ge 2$	Exponential
$\Theta(n!)$	Factorial

† lg = log to the base 2; m is a fixed, nonnegative integer.

In Section 3.6 we consider a more involved example, the worst-case time of the Euclidean algorithm (Algorithm 3.3.7).

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.

Certain forms occur so often that they are given special names, as shown in Table 3.5.3. The forms in Table 3.5.3, with the exception of $\Theta(n^m)$, are arranged so that if $\Theta(f(n))$ is above $\Theta(g(n))$, then $f(n) \le g(n)$ 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 3.5.3, 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 3.5.3. In Figure 3.5.1 we have graphed some of these functions. Another way to develop some appreciation for the relative sizes of the functions $f(n)$

in Table 3.5.3 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 3.5.1 shows the execution times, under this assumption, for various input sizes. Notice that it is feasible 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 infeasible, 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. 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 intractable. (These problems belong to the class NP; see [Hopcroft] for details.) An example of a solvable problem thought to be intractable, but not known to be intractable, 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 solvable problems thought to be intractable, but not known to be intractable, include the traveling salesperson problem and the Hamiltonian cycle problem (see Section 6.3).

Section Review Exercises

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

Exercises

Select a theta notation from Table 3.5.3 for each expression in Exercises 1–12.

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. $\dfrac{\left(n^2 + \lg n\right)(n + 1)}{n + n^2}$
12. $2 + 4 + 8 + 16 + \cdots + 2^n$

In Exercises 13–15, select a theta notation for $f(n) + g(n)$.

13. $f(n) = \Theta(1), \quad g(n) = \Theta(n^2)$

14. $f(n) = 6n^3 + 2n^2 + 4, \quad g(n) = \Theta(n \lg n)$

15. $f(n) = \Theta(n^{3/2}), \quad g(n) = \Theta(n^{5/2})$

In Exercises 16–26, 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.*

16.
```
for i := 1 to 2n do
    x := x + 1
```

17.
```
i := 1
while i ≤ 2n do
  begin
  x := x + 1
  i := i + 2
  end
```

18.
```
for i := 1 to n do
  for j := 1 to n do
    x := x + 1
```

19.
```
for i := 1 to 2n do
  for j := 1 to n do
    x := x + 1
```

20.
```
for i := 1 to n do
  for j := 1 to ⌊i/2⌋ do
    x := x + 1
```

21.
```
for i := 1 to n do
  for j := 1 to n do
    for k := 1 to n do
      x := x + 1
```

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

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

24.
```
j := n
while j ≥ 1 do
  begin
  for i := 1 to j do
    x := x + 1
  j := ⌊j/3⌋
  end
```

25.
```
i := n
while i ≥ 1 do
  begin
  for j := 1 to n do
    x := x + 1
  i := ⌊i/2⌋
  end
```

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

```
i := 2
while i < n do
  begin
  i := i^2
  x := x + 1
  end
```

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

```
i := 1
j := n
while i < j do
  begin
  while i < j and a_i < 0 do
    i := i + 1
  while i < j and a_j ≥ 0 do
    j := j − 1
  if i < j then
    swap(a_i, a_j)
  end
```

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

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

```
procedure iskey(s, n, key)
for i := 1 to n − 1 do
  for j := i + 1 to n do
    if s_i + s_j = key then
      return(1)
    else
      return(0)
end iskey
```

29. In addition to finding a theta notation in Exercises 1–28, prove that it is correct.

30. Find the exact number of comparisons (lines 12, 18, 20, 28, and 30) 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.  procedure large_small(s, n, large, small)
2.    if n = 1 then
3.      begin
4.      large := s_1
5.      small := s_1
6.      return
7.      end
8.    m := 2⌊n/2⌋
9.    i := 1
10.   while i ≤ m − 1 do
11.     begin
12.     if s_i > s_{i+1} then
13.       swap(s_i, s_{i+1})
14.     i := i + 2
15.     end
16.   if n > m then
17.     begin
18.     if s_{m−1} > s_n then
19.       swap(s_{m−1}, s_n)
20.     if s_n > s_m then
21.       swap(s_m, s_n)
22.     end
23.   small := s_1
24.   large := s_2
25.   i := 3
26.   while i ≤ m − 1 do
27.     begin
28.     if s_i < small then
29.       small := s_i
30.     if s_{i+1} > large then
31.       large := s_{i+1}
32.     i := i + 2
33.     end
34.   end large_small
```

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

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

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

34. By using an argument like that in Examples 3.5.6–3.5.8 or otherwise, prove that $\sum_{i=1}^{n} i \lg i = \Theta(n^2 \lg n)$.

35. 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 $n \geq 1$.

36. State and prove a result for Ω similar to that for Exercise 35.

37. State and prove a result for Θ similar to that for Exercises 35 and 36.

Determine whether each statement in Exercises 38–46 is true or false. If the statement is false, give a counterexample. Assume that the functions f, g, and h take on only positive values.

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

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

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

41. 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 $n = 1, 2, \ldots$.

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

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

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

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

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

★ **47.** Find functions f and g satisfying

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

★ **48.** Give an example of strictly increasing positive functions f and g defined on the positive integers [i.e., $0 < f(n) < f(n+1)$ and $0 < g(n) < g(n+1)$ for $n = 1, 2, \ldots$] for which

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

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

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

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

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

51. Where is the error in the following reasoning? Suppose that the worst-case time of an algorithm is $\Theta(n)$. 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.

52. Show that if $n \geq 4$,

$$\frac{n}{2} \lg \frac{n}{2} \geq \frac{n \lg n}{4}.$$

53. Does

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

define an equivalence relation on the set of real-valued functions on $\{1, 2, \ldots\}$?

54. Does

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

define an equivalence relation on the set of real-valued functions on $\{1, 2, \ldots\}$?

55. [Requires the integral]

(a) Show, by consulting the figure, that

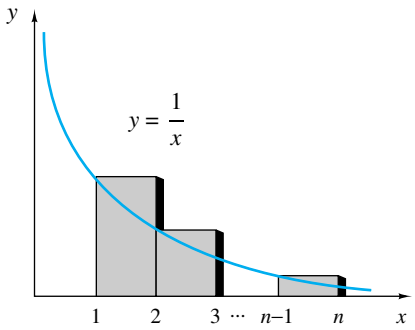
$$\frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} < \log_e n.$$

(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

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



56. [Requires the integral] Use an argument like that in Exercise 55 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.

57. 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 Cn. Therefore, the run time is $O(n)$.

58. [Requires calculus] Determine whether each statement is true or false. If the statement is false, give a counterexample. It is assumed that f and g are real-valued functions defined on the set of positive integers and that $g(n) \neq 0$ for $n \geq 1$.

(a) If
$$\lim_{n\to\infty} \frac{f(n)}{g(n)}$$
exists and is equal to some real number, then $f(n) = O(g(n))$.

(b) If $f(n) = O(g(n))$, then
$$\lim_{n\to\infty} \frac{f(n)}{g(n)}$$
exists and is equal to some real number.

(c) If
$$\lim_{n\to\infty} \frac{f(n)}{g(n)}$$
exists and is equal to some real number, then $f(n) = \Theta(g(n))$.

(d) If
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1,$$
then $f(n) = \Theta(g(n))$.

(e) If $f(n) = \Theta(g(n))$, then
$$\lim_{n\to\infty} \frac{f(n)}{g(n)}$$
exists and is equal to some real number.

★ **59.** Use induction to prove that
$$\lg n! \geq \frac{n}{2} \lg \frac{n}{2}.$$

60. [Requires calculus] Let $\ln x$ denote the natural logarithm ($\log_e x$) of x. Use the integral to obtain the estimate
$$n \ln n - n \leq \sum_{k=1}^{n} \ln k = \ln n!, \qquad n \geq 1.$$

61. Use the result of Exercise 60 and the change-of-base formula for logarithms to obtain the formula
$$n \lg n - n \lg e \leq \lg n!, \qquad n \geq 1.$$

62. Deduce
$$\lg n! \geq \frac{n}{2} \lg \frac{n}{2}$$
from the inequality of Exercise 61.

PROBLEM-SOLVING CORNER

DESIGN AND ANALYSIS OF AN ALGORITHM

Problem

Develop and analyze an algorithm that outputs 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 were

27 6 −50 21 −3 14 16 −8
42 33 −21 9,

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

Input: $s_1, \ldots, s_n$
Output: max

```
procedure max_sum1(s, n)
 // sum_ji is the sum s_j + ··· + s_i.
 for i := 1 to n do
   begin
   for j := 1 to i − 1 do
     sum_ji := sum_j,i−1 + s_i
   sum_ii := s_i
   end

 // step through sum_ji and find the maximum
 max := 0
 for i := 1 to n do
   for j := 1 to i do
     if sum_ji > max then
       max := sum_ji
 return(max)
end max_sum1
```

The first nested for loops compute the sums

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

The computation relies on the fact that

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

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 complexity of the algorithm, by computing the maximum within the same nested for loops in which we compute sum_{ji}:

Input: $s_1, \ldots, s_n$
Output: max

```
procedure max_sum2(s, n)
 // sum_ji is the sum s_j + ··· + s_i
 max := 0
 for i := 1 to n do
   begin
   for j := 1 to i − 1 do
     begin
     sum_ji := sum_j,i−1 + s_i
     if sum_ji > max then
       max := sum_ji
     end
   sum_ii := s_i
   if sum_ii > max then
     max := sum_ii
   end
 return(max)
end max_sum2
```

Since the nested for loops take time $\Theta(n^2)$, *max_sum2*'s time is $\Theta(n^2)$. To reduce the time complexity, 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 the ith term 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 then
    sum := sum +s_i
else
    sum := 0
```

Formal Solution

Input: $s_1, \ldots, s_n$
Output: max

```
procedure 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 position i.
   max := 0
   sum := 0
```

```
    for i := 1 to n do
      begin
      if sum + s_i > 0 then
        sum := sum + s_i
      else
        sum := 0
      if sum > max then
        max := sum
      end
    return(max)
end max_sum3
```

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

EXERCISE

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.

3.6 ANALYSIS OF THE EUCLIDEAN ALGORITHM

In this section we analyze the worst-case performance of the Euclidean algorithm for finding the greatest common divisor of two nonnegative integers, not both zero (Algorithm 3.3.7). For reference, we summarize the algorithm:

Input: a and b (nonnegative integers, not both zero)
Output: Greatest common divisor of a and b

```
1.  procedure gcd(a, b)
2.    // make a largest
3.    if a < b then
4.      swap(a, b)
5.    while b ≠ 0 do
6.      begin
7.      r := a mod b
8.      a := b
9.      b := r
10.     end
11.   return(a)
12. end gcd
```

We define the time required by the Euclidean algorithm as the number of modulus operations executed at line 7. Table 3.6.1 lists the number of modulus operations required for some small input values.

The worst case for the Euclidean algorithm occurs when the number of modulus operations is as large as possible. By referring to Table 3.6.1, we can determine the input pair $a, b, a > b$, with a as small as possible, that requires n modulus operations for $n = 0, \ldots, 5$. The results are given in Table 3.6.2.

Recall that the Fibonacci sequence $\{f_n\}$ (see Example 3.4.6) is defined by the equations

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

TABLE 3.6.1
Number of modulus operations required by the Euclidean algorithm for various values of the input

a \ *b*	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*
0	—	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1	1	1	1	1	1	1	1
2	0	1	1	2	1	2	1	2	1	2	1	2	1	2
3	0	1	2	1	2	3	1	2	3	1	2	3	1	2
4	0	1	1	2	1	2	2	3	1	2	2	3	1	2
5	0	1	2	3	2	1	2	3	4	3	1	2	3	4
6	0	1	1	1	2	2	1	2	2	2	3	3	1	2
7	0	1	2	2	3	3	2	1	2	3	3	4	4	3
8	0	1	1	3	1	4	2	2	1	2	2	4	2	5
9	0	1	2	1	2	3	2	3	2	1	2	3	2	3
10	0	1	1	2	2	1	3	3	2	2	1	2	2	3
11	0	1	2	3	3	2	3	4	4	3	2	1	2	3
12	0	1	1	1	1	3	1	4	2	2	2	2	1	2
13	0	1	2	2	2	4	2	3	5	3	3	3	2	1

TABLE 3.6.2
Smallest input pair that requires n modulus operations in the Euclidean algorithm

a	*b*	*n* (= *number of modulus operations*)
1	0	0
2	1	1
3	2	2
5	3	3
8	5	4
13	8	5

The Fibonacci sequence begins

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

A surprising pattern develops in Table 3.6.2: The a column is the beginning of the Fibonacci sequence and, except for the first value, the b column is the beginning of the Fibonacci sequence! We are led to conjecture that if the pair $a, b, a > b$, when input to the Euclidean algorithm requires $n \geq 1$ modulus operations, then $a \geq f_{n+1}$ and $b \geq f_n$. As further evidence of our conjecture, if we compute the smallest input pair that requires six modulus operations, we obtain $a = 21$ and $b = 13$. Our next theorem confirms that our conjecture is correct. The proof of this theorem is illustrated in Figure 3.6.1.

$34 = 91 \bmod 57$	(1 modulus operation)
34, 57 requires 4 modulus operations	(to make a total of 5)
$57 \geq f_5$ and $34 \geq f_4$	(by inductive assumption)
$\therefore\ 91 = 57 \cdot 1 + 34 \geq 57 + 34 \geq f_5 + f_4 = f_6$	

FIGURE 3.6.1
The proof of Theorem 3.6.1. The pair 57, 91, which requires $n + 1 = 5$ modulus operations, is input to the Euclidean algorithm.

THEOREM 3.6.1

Suppose that the pair $a, b, a > b$, requires $n \geq 1$ modulus operations when input to the Euclidean algorithm. Then $a \geq f_{n+1}$ and $b \geq f_n$, where $\{f_n\}$ denotes the Fibonacci sequence.

Proof. The proof is by induction on n.

BASIS STEP ($n = 1$). We have already observed that the theorem is true if $n = 1$.

INDUCTIVE STEP. Assume that the theorem is true for $n \geq 1$. We must show that the theorem is true for $n + 1$.

Suppose that the pair $a, b, a > b$, requires $n + 1$ modulus operations when input to the Euclidean algorithm. At line 7, we compute $r = a \bmod b$. Thus

$$a = bq + r, \qquad 0 \leq r < b. \tag{3.6.1}$$

The algorithm then repeats using the values b and r, $b > r$. These values require n additional modulus operations. By the inductive assumption,

$$b \geq f_{n+1} \qquad \text{and} \qquad r \geq f_n. \tag{3.6.2}$$

Combining (3.6.1) and (3.6.2), we obtain

$$a = bq + r \geq b + r \geq f_{n+1} + f_n = f_{n+2}. \tag{3.6.3}$$

[The first inequality in (3.6.3) holds because $q > 0$; q cannot equal 0, because $a > b$.] Inequalities (3.6.2) and (3.6.3) give

$$a \geq f_{n+2} \qquad \text{and} \qquad b \geq f_{n+1}.$$

The Inductive Step is finished and the proof is complete. ■

We may use Theorem 3.6.1 to analyze the worst-case performance of the Euclidean algorithm.

THEOREM 3.6.2

If integers in the range 0 to m, $m \geq 8$, not both zero, are input to the Euclidean algorithm, then at most

$$\log_{3/2} \frac{2m}{3}$$

modulus operations are required.

Proof. Let n be the maximum number of modulus operations required by the Euclidean algorithm for integers in the range 0 to m, $m \geq 8$. Let a, b be an input pair in the range 0 to m that requires n modulus operations. Table 3.6.1 shows that $n \geq 4$ and that $a \neq b$. We may assume that $a > b$. (Interchanging the values of a and b does not alter the number of modulus operations required.) By Theorem 3.6.1, $a \geq f_{n+1}$. Thus

$$f_{n+1} \leq m.$$

By Exercise 27, Section 3.4, since $n + 1 \geq 5$,

$$\left(\frac{3}{2}\right)^{n+1} < f_{n+1}.$$

Combining these last inequalities, we obtain

$$\left(\frac{3}{2}\right)^{n+1} < m.$$

Taking the logarithm to the base $\frac{3}{2}$, we obtain

$$n + 1 < \log_{3/2} m.$$

Therefore,

$$n < \log_{3/2} m - 1 = \log_{3/2} m - \log_{3/2} \frac{3}{2} = \log_{3/2} \frac{2m}{3}.$$ ■

Because the logarithm function grows so slowly, Theorem 3.6.2 tells us that the Euclidean algorithm is quite efficient, even for large values of the input. For example, since

$$\log_{3/2} \frac{2(1{,}000{,}000)}{3} = 33.07\ldots,$$

the Euclidean algorithm requires at most 33 modulus operations to compute the greatest common divisor of any pair of integers, not both zero, in the range 0 to 1,000,000.

SECTION REVIEW EXERCISES

1. If the pair $a, b, a > b$, requires $n \geq 1$ modulus operations when input to the Euclidean algorithm, how are a and b related to the Fibonacci sequence?
2. Integers in the range 0 to m, $m \geq 8$, not both zero, are input to the Euclidean algorithm. Give an upper bound for the number of modulus operations required.

EXERCISES

1. Extend Tables 3.6.1 and 3.6.2 to the range 0 to 21.
2. Exactly how many modulus operations are required by the Euclidean algorithm in the worst case for numbers in the range 0 to 1,000,000?
3. How many subtractions are required by the algorithm of Exercise 22, Section 3.3, in the worst case for numbers in the range 0 to m? (This algorithm finds the greatest common divisor by using subtraction instead of the modulus operation.)
4. Prove that when the pair f_{n+1}, f_n is input to the Euclidean algorithm, $n \geq 1$, exactly n modulus operations are required.
5. Show that for any integer $k > 1$, the number of modulus operations required by the Euclidean algorithm to compute $\gcd(a, b)$ is the same as the number of modulus operations required to compute $\gcd(ka, kb)$.
6. Show that $\gcd(f_n, f_{n+1}) = 1, n \geq 1$.

†3.7 THE RSA PUBLIC-KEY CRYPTOSYSTEM

Cryptology is the study of systems, called **cryptosystems**, for secure communications. In a cryptosystem, the sender transforms the message before transmitting it so that, hopefully, only authorized recipients can reconstruct the original message (i.e., the message before it was transformed). The sender is said to **encrypt** the message, and the recipient is said to **decrypt** the message. If the cryptosystem is secure, unauthorized persons will be unable to discover the decryption technique, so even if they read the encrypted message, they will be unable to decrypt it. Cryptosystems are important for large organizations (e.g., government and military), all Internet-based businesses, and individuals. For example, if a credit card number is sent over the Internet, it is important for the number to be read only by the intended recipient. In this section, we look at some algorithms that support secure communication.

In one of the oldest and simplest systems, the sender and receiver each have a key that defines a substitute character for each potential character to be sent. Moreover, the sender and receiver do not disclose the key. Such keys are said to be *private*.

---

† This section can be omitted without loss of continuity.

EXAMPLE 3.7.1 If a key is defined as

character:	ABCDEFGHIJKLMNOPQRSTUVWXYZ
replaced by:	EIJFUAXVHWP GSRKOBTQYDMLZNC

the message SEND MONEY would be encrypted as QARUESKRAN. The encrypted message SKRANEKRELIN would be decrypted as MONEY ON WAY. ■

Simple systems such as that in Example 3.7.1 are easily broken since certain letters (e.g., E in English) and letter combinations (e.g., ER in English) appear more frequently than others. Also, a problem with private keys in general is that the keys have to be securely sent to the sender and recipient before messages can be sent. We devote the remainder of this section to the **RSA public-key cryptosystem**, named after its inventors, Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman, that is believed to be secure. In the RSA system, each participant makes public an encryption key and hides a decryption key. To send a message, all one needs to do is look up the recipient's encryption key in a publicly distributed table. The recipient then decrypts the message using the hidden decryption key.

In the RSA system, messages are represented as numbers. For example, each character might be represented as a number. If a blank space is represented as 1, A as 2, B as 3, and so on, the message SEND MONEY would be represented as 20, 6, 15, 5, 1, 14, 16, 15, 6, 26. If desired, the integers could be combined into the single integer

20061505011416150626

(note that leading zeros have been added to all single-digit numbers).

We next describe how the RSA system works, present a concrete example, and then discuss why it works. Each prospective recipient chooses two primes p and q and computes $z = pq$. Since the security of the RSA system rests primarily on the inability of anyone knowing the value of z to discover the numbers p and q, p and q are typically chosen so that each has 100 or more digits. Next, the prospective recipient computes $\phi = (p-1)(q-1)$ and chooses an integer n such that $\gcd(n, \phi) = 1$. In practice, n is often chosen to be a prime. The pair z, n is then made public. Finally, the prospective recipient computes the unique number $s, 0 < s < \phi$, satisfying $ns \bmod \phi = 1$. The number s is kept secret and used to decrypt messages.

To send the integer $a, 0 \leq a \leq z - 1$, to the holder of public key z, n, the sender computes $c = a^n \bmod z$ and sends c. To decrypt the message, the recipient computes $c^s \bmod z$, which can be shown to be equal to a.

EXAMPLE 3.7.2 Suppose that we choose $p = 23$, $q = 31$, and $n = 29$. Then $z = pq = 713$ and $\phi = (p-1)(q-1) = 660$. Now $s = 569$ since $ns \bmod \phi = 29 \cdot 569 \bmod 660 = 16501 \bmod 660 = 1$. The pair $z, n = 713, 29$ is made publicly available.

To transmit $a = 572$ to the holder of public key 713, 29, the sender computes $c = a^n \bmod z = 572^{29} \bmod 713 = 113$ and sends 113. The receiver computes $c^s \bmod z = 113^{569} \bmod 713 = 572$ in order to decrypt the message. ■

It may appear that huge numbers must be computed in order to encrypt and decrypt messages using the RSA system. For example, the number 572^{29} in Example 3.7.2 has 80 digits, and if p and q have 100 or more digits, the numbers would be far larger. The key to simplifying the computation is to note that the arithmetic is done mod z. It can be shown that

$$ab \bmod z = [(a \bmod z)(b \bmod z)] \bmod z \tag{3.7.1}$$

(see Exercise 10). We show how to use (3.7.1) to compute $572^{29} \bmod 713$.

EXAMPLE 3.7.3 We use (3.7.1) to compute $572^{29} \bmod 713$. We note that

$$29 = 16 + 8 + 4 + 1$$

(which is just the base 2 representation of 29), so we compute 572 to each of the powers 16, 8, 4, and 1, mod 713, by repeated squaring and then multiply them, mod 713:

$$\begin{aligned}
572^2 \bmod 713 &= 327184 \bmod 713 = 630 \\
572^4 \bmod 713 &= 630^2 \bmod 713 = 396900 \bmod 713 = 472 \\
572^8 \bmod 713 &= 472^2 \bmod 713 = 222784 \bmod 713 = 328 \\
572^{16} \bmod 713 &= 328^2 \bmod 713 = 107584 \bmod 713 = 634 \\
572^{24} \bmod 713 &= 572^{16} \cdot 572^8 \bmod 713 = 634 \cdot 328 \bmod 713 \\
&= 207952 \bmod 713 = 469 \\
572^{28} \bmod 713 &= 572^{24} \cdot 572^4 \bmod 713 = 469 \cdot 472 \bmod 713 \\
&= 221368 \bmod 713 = 338 \\
572^{29} \bmod 713 &= 572^{28} \cdot 572^1 \bmod 713 = 338 \cdot 572 \bmod 713 \\
&= 193336 \bmod 713 = 113.
\end{aligned}$$

The method is readily converted to an algorithm (see Exercise 11). ■

The Euclidean algorithm may be used by a prospective recipient to compute efficiently the unique number $s, 0 < s < \phi$, satisfying $ns \bmod \phi = 1$ (see Exercise 12). The main result that makes encryption and decryption work is that

$$a^u \bmod z = a \quad \text{for all } 0 \le a < z \text{ and } u \bmod \phi = 1$$

(for a proof, see [Cormen: Theorem 33.36, page 834]). Using this result and (3.7.1), we may show that decryption produces the correct result. Since $ns \bmod \phi = 1$,

$$c^s \bmod z = (a^n \bmod z)^s \bmod z = (a^n)^s \bmod z = a^{ns} \bmod z = a.$$

The security of the RSA encryption system relies mainly on the fact that at present there is no efficient algorithm known for factoring integers; that is, currently no algorithm is known for factoring d-digit integers in polynomial time, $O(d^k)$. Thus if the primes p and q are chosen large enough, it is impractical to compute the factorization $z = pq$. If the factorization could be found by a person who intercepts a message, the message could be decrypted just as the authorized recipient does. At this time, no practical method is known for factoring integers with 200 or more digits, so if p and q are chosen so that each has 100 or more digits, pq would then have about 200 or more digits, which seems to make RSA secure.

The first description of the RSA encryption system was in Martin Gardner's February 1977 *Scientific American* column (see [Gardner, 1977]). Included in this column were an encoded message using the key z, n, where z was the product of 64- and 65-digit primes, and $n = 9007$, and an offer of $100 to the first person to crack the code. At the time the article was written, it was estimated that it would take 40 quadrillion years to factor z. In fact, in April 1994, Arjen Lenstra, Paul Leyland, Michael Graff, and Derek Atkins, with the assistance of 600 volunteers from 25 countries using over 1600 computers, factored z (see [Taubes]). The work was coordinated on the Internet.

Another possible way a message could be intercepted and decrypted would be to take the nth root of $c \bmod z$, where c is the encrypted value sent. Since $c = a^n \bmod z$, the nth root of $c \bmod z$ would give a, the decrypted value. Again, at

present there is no polynomial-time algorithm known for computing nth roots mod z. It is also conceivable that a message could be decrypted by some means other than factoring integers or taking nth roots mod z. For example, in the mid–1990s Paul Kocher proposed a way to break RSA based on the time it takes to decrypt messages (see [English]). The idea is that different secret keys require different amounts of time to decrypt messages and, by using this timing information, an unauthorized person might be able to unveil the secret key and thus decrypt the message. Implementors of RSA have taken steps to alter the observed time to decrypt messages to thwart such attacks.

SECTION REVIEW EXERCISES

1. To what does "cryptology" refer?
2. What is a cryptosystem?
3. What does it mean to "encrypt a message"?
4. What does it mean to "decrypt a message"?
5. In the RSA public-key cryptosystem, how does one encrypt a and send it to the holder of public key z, n?
6. In the RSA public-key cryptosystem, how does one decrypt c?
7. On what does the security of the RSA encryption system rest?

EXERCISES

1. Encrypt the message `COOL BEAVIS` using the key of Example 3.7.1.
2. Decrypt the message `UTWR ENKDTEKMIGYWRA` using the key of Example 3.7.1.
3. Encrypt 333 using the public key 713, 29 of Example 3.7.2.
4. Decrypt 411 using $s = 569$ as in Example 3.7.2.

In Exercises 5–9, assume that we choose primes $p = 17$, $q = 23$, *and* $n = 31$.

5. Compute z.
6. Compute ϕ.
7. Verify that $s = 159$.
8. Encrypt 101 using the public key z, n.
9. Decrypt 250.
10. Prove equation (3.7.1).
11. Give an efficient algorithm to compute $a^n \bmod z$.
12. Show how to compute efficiently the value of s given n and ϕ; that is, given positive integers n and ϕ, with $\gcd(n, \phi) = 1$, give an efficient algorithm to compute positive integers s and t, with $0 < s < \phi$, such that $ns - t\phi = 1$ and, in particular, $ns \bmod \phi = 1$. *Hint*: Use the method of Exercise 17, Section 3.3, to compute efficiently integers s' and t' such that $s'n + t'\phi = 1$. If $s' > 0$, take $s = s'$. If $s' < 0$, take
$$s = -s'(\phi - 1) \bmod \phi.$$
13. Show that the number s of Exercise 12 is unique.
14. Show how to use the method of Exercise 12 to compute the value s of Example 3.7.2.
15. Show how to use the method of Exercise 12 to compute the value s of Exercise 7.

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; 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.

Full details of the RSA cryptosystem may be found in [Cormen]. [Pfleeger] is devoted to computer security.

CHAPTER REVIEW

Section 3.1

1. Algorithm
2. Properties of an algorithm: Precision, uniqueness, finiteness, input, output, generality
3. Assignment statement: $x := y$
4. Trace

Section 3.2

5. Pseudocode
6. Procedure
7. If-then structure:

   if *p* **then**
   action

8. If-then-else structure:

   if *p* **then**
   action 1
   else
   action 2

9. Comment: nonexecutable information. A comment starts with // and continues to the end of the line.
10. Return statements: **return** or **return**(x)
11. While loop:

    while *p* **do**
    action

12. For loop:

    for *var* := *init* **to** *limit* **do**
    action

13. Call statement: **call** $proc(p_1, p_2, \ldots, p_k)$

Section 3.3

14. b divides a : $b \mid a$
15. b is a divisor of a
16. Common divisor
17. Greatest common divisor
18. Euclidean algorithm

Section 3.4

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

Section 3.5

24. Analysis of algorithms
25. Complexity of algorithms
26. Worst-case time of an algorithm
27. Best-case time of an algorithm
28. Average-case time of an algorithm
29. Big oh notation: $f(n) = O(g(n))$
30. Omega notation: $f(n) = \Omega(g(n))$
31. Theta notation: $f(n) = \Theta(g(n))$

Section 3.6

32. If the pair $a, b, a > b$, requires $n \geq 1$ modulus operations when input to the Euclidean algorithm, then $a \geq f_{n+1}$ and $b \geq f_n$, where $\{f_n\}$ denotes the Fibonacci sequence.
33. If integers in the range 0 to $m, m \geq 8$, not both zero, are input to the Euclidean algorithm, then at most
$$\log_{3/2} \frac{2m}{3}$$
modulus operations are required.

Section 3.7

34. Cryptology
35. Cryptosystem
36. Encrypt a message
37. Decrypt a message
38. RSA public key cryptosystem: To encrypt a and send it to the holder of public key z, n, compute $c = a^n \bmod z$, and send c. To decrypt the message, compute $c^s \bmod z$, which can be shown to be equal to a.
39. $ab \bmod z = [(a \bmod z)(b \bmod z)] \bmod z$
40. The security of the RSA encryption system relies mainly on the fact that at present there is no efficient algorithm known for factoring integers.

CHAPTER SELF-TEST

Section 3.1

1. Trace the "find max" algorithm in Section 3.1 for the values $a = 12, b = 3, c = 0$.
2. 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
$$x < y < z.$$
3. Write an algorithm that outputs "Yes" if the values of a, b, and c are distinct, and "No" otherwise.
4. Which of the algorithm properties—precision, uniqueness, finiteness, input, output, and generality—if any, are lacking in the following? Explain.

   Input: S, a set of integers; m, an integer

   Output: All subsets of S that sum to m

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

Section 3.2

5. Trace Algorithm 3.2.2 for the input
$$s_1 = 7, \quad s_2 = 9, \quad s_3 = 17, \quad s_4 = 7.$$
6. Write an algorithm that receives as input the matrix of a relation R and tests whether R is symmetric.
7. Write an algorithm that receives as input the $n \times n$ matrix A and outputs the transpose A^T.
8. Write an algorithm that receives as input the sequence
$$s_1, \ldots, s_n$$
sorted in increasing order and prints all values that appear more than once. *Example*: If the sequence were
$$1 \quad 1 \quad 1 \quad 5 \quad 8 \quad 8 \quad 9 \quad 12$$
the output would be
$$1 \quad 8.$$

Section 3.3

9. If $a = 333$ and $b = 24$, find integers q and r so that $a = bq + r$, with $0 \leq r < b$.

10. Using the Euclidean algorithm, find the greatest common divisor of the integers 396 and 480.

11. Using the Euclidean algorithm, find the greatest common divisor of the integers 2390 and 4326.

12. Fill in the blank to make a true statement: If a and b are integers satisfying $a > b > 0$ and $r = a \bmod b$, then $\gcd(a, b) =$ _____.

Section 3.4

13. Trace Algorithm 3.4.4 (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 defined by the equations

$$t_1 = t_2 = t_3 = 1; \qquad t_n = t_{n-1} + t_{n-2} + t_{n-3}, \qquad n \geq 4.$$

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.

Section 3.5

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 17 and 18.*

17. $4n^3 + 2n - 5$

18. $1^3 + 2^3 + \cdots + n^3$

19. 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 do
  for j := 1 to n do
    x := x + 1
```

20. Write an algorithm that tests whether two $n \times n$ matrices are equal and find a theta notation for its worst-case time.

Section 3.6

21. Exactly how many modulus operations are required by the Euclidean algorithm in the worst case for numbers in the range 0 to 1000?

22. Exactly how many modulus operations are required by the Euclidean algorithm to compute gcd(2, 76652913)?

23. Exactly how many modulus operations are required by the Euclidean algorithm to compute $\gcd(f_{324}, f_{323})$? ($\{f_n\}$ denotes the Fibonacci sequence.)

24. Given that $\log_{3/2} 100 = 11.357747$, provide an upper bound for the number of modulus operations required by the Euclidean algorithm for integers in the range 0 to 100,000,000.

Section 3.7

In Exercises 25–28, assume that we choose primes $p = 13$, $q = 17$, *and* $n = 19$.

25. Compute z and ϕ.

26. Verify that $s = 91$.

27. Encrypt 144 using public key z, n.

28. Decrypt 28.

COMPUTER EXERCISES

1. Implement Algorithm 3.2.2, finding the largest element in a finite sequence, as a program.
2. Implement Algorithm 3.2.4, testing whether a positive integer is prime, as a program.
3. Implement Algorithm 3.2.5, finding a prime larger than a given integer, as a program.
4. Write recursive and nonrecursive programs to compute the greatest common divisor. Compare the times required by the programs.
5. Write recursive and nonrecursive programs to compute $n!$. Compare the times required by the programs.
6. 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.
7. 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.
8. 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.
9. Write recursive and nonrecursive programs to compute the Fibonacci sequence. Compare the times required by the programs.
10. 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.
11. 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.
12. Implement the RSA public-key cryptosystem.