****53.** Prove that $A(m, n+1) > A(m, n)$ whenever m and n are nonnegative integers.

***54.** Prove that $A(m+1, n) \geq A(m, n)$ whenever m and n are nonnegative integers.

55. Prove that $A(i, j) \geq j$ whenever i and j are nonnegative integers.

56. Use mathematical induction to prove that a function F defined by specifying $F(0)$ and a rule for obtaining $F(n+1)$ from $F(n)$ is well defined.

57. Use strong induction to prove that a function F defined by specifying $F(0)$ and a rule for obtaining $F(n+1)$ from the values $F(k)$ for $k = 0, 1, 2, \ldots, n$ is well defined.

58. Show that each of these proposed recursive definitions of a function on the set of positive integers does not produce a well-defined function.

a) $F(n) = 1 + F(\lfloor n/2 \rfloor)$ for $n \geq 1$ and $F(1) = 1$.
b) $F(n) = 1 + F(n-3)$ for $n \geq 2$, $F(1) = 2$, and $F(2) = 3$.
c) $F(n) = 1 + F(n/2)$ for $n \geq 2$, $F(1) = 1$, and $F(2) = 2$.
d) $F(n) = 1 + F(n/2)$ if n is even and $n \geq 2$, $F(n) = 1 - F(n-1)$ if n is odd, and $F(1) = 1$.
e) $F(n) = 1 + F(n/2)$ if n is even and $n \geq 2$, $F(n) = F(3n-1)$ if n is odd and $n \geq 3$, and $F(1) = 1$.

59. Show that each of these proposed recursive definitions of a function on the set of positive integers does not produce a well-defined function.

a) $F(n) = 1 + F(\lfloor (n+1)/2 \rfloor)$ for $n \geq 1$ and $F(1) = 1$.
b) $F(n) = 1 + F(n-2)$ for $n \geq 2$ and $F(1) = 0$.
c) $F(n) = 1 + F(n/3)$ for $n \geq 3$, $F(1) = 1$, $F(2) = 2$, and $F(3) = 3$.
d) $F(n) = 1 + F(n/2)$ if n is even and $n \geq 2$, $F(n) = 1 + F(n-2)$ if n is odd, and $F(1) = 1$.
e) $F(n) = 1 + F(F(n-1))$ if $n \geq 2$ and $F(1) = 2$.

Exercises 60–62 deal with iterations of the logarithm function. Let $\log n$ denote the logarithm of n to the base 2, as usual. The function $\log^{(k)} n$ is defined recursively by

$$\log^{(k)} n = \begin{cases} n & \text{if } k = 0 \\ \log(\log^{(k-1)} n) & \text{if } \log^{(k-1)} n \text{ is defined and positive} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

The **iterated logarithm** is the function $\log^* n$ whose value at n is the smallest nonnegative integer k such that $\log^{(k)} n \leq 1$.

60. Find these values.

a) $\log^{(2)} 16$ **b)** $\log^{(3)} 256$
c) $\log^{(3)} 2^{65536}$ **d)** $\log^{(4)} 2^{2^{65536}}$

61. Find the value of $\log^* n$ for these values of n.

a) 2 **b)** 4 **c)** 8 **d)** 16
e) 256 **f)** 65536 **g)** 2^{2048}

62. Find the largest integer n such that $\log^* n = 5$. Determine the number of decimal digits in this number.

Exercises 63–65 deal with values of iterated functions. Suppose that $f(n)$ is a function from the set of real numbers, or positive real numbers, or some other set of real numbers, to the set of real numbers such that $f(n)$ is monotonically increasing [that is, $f(n) < f(m)$ when $n < m$) and $f(n) < n$ for all n in the domain of f.] The function $f^{(k)}(n)$ is defined recursively by

$$f^{(k)}(n) = \begin{cases} n & \text{if } k = 0 \\ f(f^{(k-1)}(n)) & \text{if } k > 0. \end{cases}$$

Furthermore, let c be a positive real number. The **iterated function** f_c^* is the number of iterations of f required to reduce its argument to c or less, so $f_c^*(n)$ is the smallest nonnegative integer k such that $f^k(n) \leq c$.

63. Let $f(n) = n - a$, where a is a positive integer. Find a formula for $f^{(k)}(n)$. What is the value of $f_0^*(n)$ when n is a positive integer?

64. Let $f(n) = n/2$. Find a formula for $f^{(k)}(n)$. What is the value of $f_1^*(n)$ when n is a positive integer?

65. Let $f(n) = \sqrt{n}$. Find a formula for $f^{(k)}(n)$. What is the value of $f_2^*(n)$ when n is a positive integer?

5.4 Recursive Algorithms

Introduction

Here's a famous humorous quote: "To understand recursion, you must first understand recursion."

Sometimes we can reduce the solution to a problem with a particular set of input values to the solution of the same problem with smaller input values. For instance, the problem of finding the greatest common divisor of two positive integers a and b, where $b > a$, can be reduced to finding the greatest common divisor of a pair of smaller integers, namely, b **mod** a and a, because $\gcd(b \textbf{ mod } a, a) = \gcd(a, b)$. When such a reduction can be done, the solution to the original problem can be found with a sequence of reductions, until the problem has been reduced to some initial case for which the solution is known. For instance, for finding the greatest common divisor, the reduction continues until the smaller of the two numbers is zero, because $\gcd(a, 0) = a$ when $a > 0$.

We will see that algorithms that successively reduce a problem to the same problem with smaller input are used to solve a wide variety of problems.

DEFINITION 1 An algorithm is called *recursive* if it solves a problem by reducing it to an instance of the same problem with smaller input.

Links

We will describe a variety of different recursive algorithms in this section.

EXAMPLE 1 Give a recursive algorithm for computing $n!$, where n is a nonnegative integer.

Extra Examples

Solution: We can build a recursive algorithm that finds $n!$, where n is a nonnegative integer, based on the recursive definition of $n!$, which specifies that $n! = n \cdot (n-1)!$ when n is a positive integer, and that $0! = 1$. To find $n!$ for a particular integer, we use the recursive step n times, each time replacing a value of the factorial function with the value of the factorial function at the next smaller integer. At this last step, we insert the value of $0!$. The recursive algorithm we obtain is displayed as Algorithm 1.

To help understand how this algorithm works, we trace the steps used by the algorithm to compute $4!$. First, we use the recursive step to write $4! = 4 \cdot 3!$. We then use the recursive step repeatedly to write $3! = 3 \cdot 2!$, $2! = 2 \cdot 1!$, and $1! = 1 \cdot 0!$. Inserting the value of $0! = 1$, and working back through the steps, we see that $1! = 1 \cdot 1 = 1$, $2! = 2 \cdot 1! = 2$, $3! = 3 \cdot 2! = 3 \cdot 2 = 6$, and $4! = 4 \cdot 3! = 4 \cdot 6 = 24$. ◀

ALGORITHM 1 A Recursive Algorithm for Computing $n!$.

```
procedure factorial(n: nonnegative integer)
if $n = 0$ then return 1
else return $n \cdot factorial(n-1)$
{output is $n!$}
```

Example 2 shows how a recursive algorithm can be constructed to evaluate a function from its recursive definition.

EXAMPLE 2 Give a recursive algorithm for computing a^n, where a is a nonzero real number and n is a nonnegative integer.

Solution: We can base a recursive algorithm on the recursive definition of a^n. This definition states that $a^{n+1} = a \cdot a^n$ for $n > 0$ and the initial condition $a^0 = 1$. To find a^n, successively use the recursive step to reduce the exponent until it becomes zero. We give this procedure in Algorithm 2. ◀

ALGORITHM 2 A Recursive Algorithm for Computing a^n.

```
procedure power(a: nonzero real number, n: nonnegative integer)
if $n = 0$ then return 1
else return $a \cdot power(a, n-1)$
{output is $a^n$}
```

Next we give a recursive algorithm for finding greatest common divisors.

EXAMPLE 3 Give a recursive algorithm for computing the greatest common divisor of two nonnegative integers a and b with $a < b$.

Solution: We can base a recursive algorithm on the reduction $\gcd(a, b) = \gcd(b \textbf{ mod } a, a)$ and the condition $\gcd(0, b) = b$ when $b > 0$. This produces the procedure in Algorithm 3, which is a recursive version of the Euclidean algorithm.

We illustrate the workings of Algorithm 3 with a trace when the input is $a = 5, b = 8$. With this input, the algorithm uses the "else" clause to find that $\gcd(5, 8) = \gcd(8 \textbf{ mod } 5, 5) = \gcd(3, 5)$. It uses this clause again to find that $\gcd(3, 5) = \gcd(5 \textbf{ mod } 3, 3) = \gcd(2, 3)$, then to get $\gcd(2, 3) = \gcd(3 \textbf{ mod } 2, 2) = \gcd(1, 2)$, then to get $\gcd(1, 2) = \gcd(2 \textbf{ mod } 1, 1) = \gcd(0, 1)$. Finally, to find $\gcd(0, 1)$ it uses the first step with $a = 0$ to find that $\gcd(0, 1) = 1$. Consequently, the algorithm finds that $\gcd(5, 8) = 1$. ◀

ALGORITHM 3 A Recursive Algorithm for Computing gcd(a, b).

```
procedure gcd(a, b: nonnegative integers with a < b)
if a = 0 then return b
else return gcd(b mod a, a)
{output is gcd(a, b)}
```

EXAMPLE 4 Devise a recursive algorithm for computing $b^n \textbf{ mod } m$, where b, n, and m are integers with $m \geq 2, n \geq 0$, and $1 \leq b < m$.

Solution: We can base a recursive algorithm on the fact that

$$b^n \textbf{ mod } m = (b \cdot (b^{n-1} \textbf{ mod } m)) \textbf{ mod } m,$$

which follows by Corollary 2 in Section 4.1, and the initial condition $b^0 \textbf{ mod } m = 1$. We leave this as Exercise 12 for the reader.

However, we can devise a much more efficient recursive algorithm based on the observation that

$$b^n \textbf{ mod } m = (b^{n/2} \textbf{ mod } m)^2 \textbf{ mod } m$$

when n is even and

$$b^n \textbf{ mod } m = \left((b^{\lfloor n/2 \rfloor} \textbf{ mod } m)^2 \textbf{ mod } m \cdot b \textbf{ mod } m\right) \textbf{ mod } m$$

when n is odd, which we describe in pseudocode as Algorithm 4.

We trace the execution of Algorithm 4 with input $b = 2, n = 5$, and $m = 3$ to illustrate how it works. First, because $n = 5$ is odd we use the "else" clause to see that $mpower(2, 5, 3) = (mpower(2, 2, 3)^2 \textbf{ mod } 3 \cdot 2 \textbf{ mod } 3) \textbf{ mod } 3$. We next use the "else if" clause to see that $mpower(2, 2, 3) = mpower(2, 1, 3)^2 \textbf{ mod } 3$. Using the "else" clause again, we see that $mpower(2, 1, 3) = (mpower(2, 0, 3)^2 \textbf{ mod } 3 \cdot 2 \textbf{ mod } 3) \textbf{ mod } 3$. Finally, using the "if" clause, we see that $mpower(2, 0, 3) = 1$. Working backwards, it follows that $mpower(2, 1, 3) = (1^2 \textbf{ mod } 3 \cdot 2 \textbf{ mod } 3) \textbf{ mod } 3 = 2$, so $mpower(2, 2, 3) = 2^2 \textbf{ mod } 3 = 1$, and finally $mpower(2, 5, 3) = (1^2 \textbf{ mod } 3 \cdot 2 \textbf{ mod } 3) \textbf{ mod } 3 = 2$. ◀

ALGORITHM 4 Recursive Modular Exponentiation.

```
procedure mpower(b, n, m: integers with $b > 0$ and $m \geq 2, n \geq 0$)
if $n = 0$ then
    return 1
else if $n$ is even then
    return $mpower(b, n/2, m)^2$ mod $m$
else
    return $(mpower(b, \lfloor n/2 \rfloor, m)^2$ mod $m \cdot b$ mod $m)$ mod $m$
{output is $b^n$ mod $m$}
```

We will now give recursive versions of searching algorithms that were introduced in Section 3.1.

EXAMPLE 5 Express the linear search algorithm as a recursive procedure.

Solution: To *search* for the first occurrence of x in the sequence $a_1, a_2, \ldots, a_n$, at the ith step of the algorithm, x and a_i are compared. If x equals a_i, then the algorithm returns i, the location of x in the sequence. Otherwise, the search for the first occurrence of x is reduced to a search in a sequence with one fewer element, namely, the sequence $a_{i+1}, \ldots, a_n$. The algorithm returns 0 when x is never found in the sequence after all terms have been examined. We can now give a recursive procedure, which is displayed as pseudocode in Algorithm 5.

Let *search* (i, j, x) be the procedure that searches for the first occurrence of x in the sequence $a_i, a_{i+1}, \ldots, a_j$. The input to the procedure consists of the triple $(1, n, x)$. The algorithm terminates at a step if the first term of the remaining sequence is x or if there is only one term of the sequence and this is not x. If x is not the first term and there are additional terms, the same procedure is carried out but with a search sequence of one fewer term, obtained by deleting the first term of the search sequence. If the algorithm terminates without x having been found, the algorithm returns the value 0. ◀

ALGORITHM 5 A Recursive Linear Search Algorithm.

```
procedure search(i, j, x: $i, j, x$ integers, $1 \leq i \leq j \leq n$)
if $a_i = x$ then
    return $i$
else if $i = j$ then
    return 0
else
    return search($i + 1, j, x$)
{output is the location of $x$ in $a_1, a_2, \ldots, a_n$ if it appears; otherwise it is 0}
```

EXAMPLE 6 Construct a recursive version of a binary search algorithm.

Solution: Suppose we want to locate x in the sequence $a_1, a_2, \ldots, a_n$ of integers in increasing order. To perform a binary search, we begin by comparing x with the middle term, $a_{\lfloor (n+1)/2 \rfloor}$. Our algorithm will terminate if x equals this term and return the location of this term in the sequence. Otherwise, we reduce the search to a smaller search sequence, namely, the first half of the sequence if x is smaller than the middle term of the original sequence, and the second half otherwise. We have reduced the solution of the search problem to the solution of the same problem with a sequence at most half as long. If have we never encountered the search term x, our algorithm returns the value 0. We express this recursive version of a binary search algorithm as Algorithm 6. ◀

ALGORITHM 6 A Recursive Binary Search Algorithm.

```
**procedure** *binary search*$(i, j, x$: $i, j, x$ integers, $1 \le i \le j \le n)$
$m := \lfloor (i + j)/2 \rfloor$
**if** $x = a_m$ **then**
    **return** $m$
**else if** $(x < a_m$ and $i < m)$ **then**
    **return** *binary search*$(i, m - 1, x)$
**else if** $(x > a_m$ and $j > m)$ **then**
    **return** *binary search*$(m + 1, j, x)$
**else return** 0
{output is location of $x$ in $a_1, a_2, \ldots, a_n$ if it appears; otherwise it is 0}
```

Proving Recursive Algorithms Correct

Mathematical induction, and its variant strong induction, can be used to prove that a recursive algorithm is correct, that is, that it produces the desired output for all possible input values. Examples 7 and 8 illustrate how mathematical induction or strong induction can be used to prove that recursive algorithms are correct. First, we will show that Algorithm 2 is correct.

EXAMPLE 7 Prove that Algorithm 2, which computes powers of real numbers, is correct.

Solution: We use mathematical induction on the exponent n.

BASIS STEP: If $n = 0$, the first step of the algorithm tells us that *power* $(a, 0) = 1$. This is correct because $a^0 = 1$ for every nonzero real number a. This completes the basis step.

INDUCTIVE STEP: The inductive hypothesis is the statement that *power* $(a, k) = a^k$ for all $a \ne 0$ for an arbitrary nonnegative integer k. That is, the inductive hypothesis is the statement that the algorithm correctly computes a^k. To complete the inductive step, we show that if the inductive hypothesis is true, then the algorithm correctly computes a^{k+1}. Because $k + 1$ is a positive integer, when the algorithm computes a^{k+1}, the algorithm sets *power* $(a, k + 1) = a \cdot$ *power* (a, k). By the inductive hypothesis, we have *power* $(a, k) = a^k$, so *power* $(a, k + 1) = a \cdot$ *power* $(a, k) = a \cdot a^k = a^{k+1}$. This completes the inductive step.

We have completed the basis step and the inductive step, so we can conclude that Algorithm 2 always computes a^n correctly when $a \ne 0$ and n is a nonnegative integer. ◀

Generally, we need to use strong induction to prove that recursive algorithms are correct, rather than just mathematical induction. Example 8 illustrates this; it shows how strong induction can be used to prove that Algorithm 4 is correct.

EXAMPLE 8 Prove that Algorithm 4, which computes modular powers, is correct.

Solution: We use strong induction on the exponent n.

BASIS STEP: Let b be an integer and m an integer with $m \ge 2$. When $n = 0$, the algorithm sets *mpower*(b, n, m) equal to 1. This is correct because b^0 **mod** $m = 1$. The basis step is complete.

INDUCTIVE STEP: For the inductive hypothesis we assume that $mpower(b, j, m) = b^j \textbf{ mod } m$ for all integers $0 \leq j < k$ whenever b is a positive integer and m is an integer with $m \geq 2$. To complete the inductive step, we show that if the inductive hypothesis is correct, then $mpower(b, k, m) = b^k \textbf{ mod } m$. Because the recursive algorithm handles odd and even values of k differently, we split the inductive step into two cases.

When k is even, we have

$$mpower(b, k, m) = (mpower(b, k/2, m))^2 \textbf{ mod } m = (b^{k/2} \textbf{ mod } m)^2 \textbf{ mod } m = b^k \textbf{ mod } m,$$

where we have used the inductive hypothesis to replace $mpower(b, k/2, m)$ by $b^{k/2} \textbf{ mod } m$.

When k is odd, we have

$$\begin{aligned} mpower(b, k, m) &= ((mpower(b, \lfloor k/2 \rfloor, m))^2 \textbf{ mod } m \cdot b \textbf{ mod } m) \textbf{ mod } m \\ &= ((b^{\lfloor k/2 \rfloor} \textbf{ mod } m)^2 \textbf{ mod } m \cdot b \textbf{ mod } m) \textbf{ mod } m \\ &= b^{2\lfloor k/2 \rfloor + 1} \textbf{ mod } m = b^k \textbf{ mod } m, \end{aligned}$$

using Corollary 2 in Section 4.1, because $2\lfloor k/2 \rfloor + 1 = 2(k-1)/2 + 1 = k$ when k is odd. Here we have used the inductive hypothesis to replace $mpower(b, \lfloor k/2 \rfloor, m)$ by $b^{\lfloor k/2 \rfloor} \textbf{ mod } m$. This completes the inductive step.

We have completed the basis step and the inductive step, so by strong induction we know that Algorithm 4 is correct. ◀

Recursion and Iteration

A recursive definition expresses the value of a function at a positive integer in terms of the values of the function at smaller integers. This means that we can devise a recursive algorithm to evaluate a recursively defined function at a positive integer. Instead of successively reducing the computation to the evaluation of the function at smaller integers, we can start with the value of the function at one or more integers, the base cases, and successively apply the recursive definition to find the values of the function at successive larger integers. Such a procedure is called **iterative**. Often an iterative approach for the evaluation of a recursively defined sequence requires much less computation than a procedure using recursion (unless special-purpose recursive machines are used). This is illustrated by the iterative and recursive procedures for finding the nth Fibonacci number. The recursive procedure is given first.

ALGORITHM 7 A Recursive Algorithm for Fibonacci Numbers.

```
procedure fibonacci(n: nonnegative integer)
if $n = 0$ then return 0
else if $n = 1$ then return 1
else return fibonacci($n - 1$) + fibonacci($n - 2$)
{output is fibonacci($n$)}
```

When we use a recursive procedure to find f_n, we first express f_n as $f_{n-1} + f_{n-2}$. Then we replace both of these Fibonacci numbers by the sum of two previous Fibonacci numbers, and so on. When f_1 or f_0 arises, it is replaced by its value.

Note that at each stage of the recursion, until f_1 or f_0 is obtained, the number of Fibonacci numbers to be evaluated has doubled. For instance, when we find f_4 using this recursive algorithm, we must carry out all the computations illustrated in the tree diagram in Figure 1. This

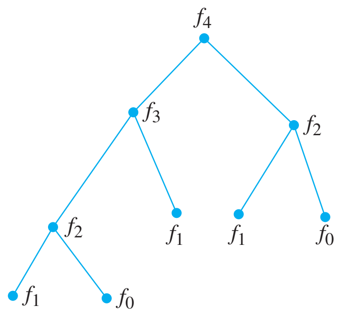

FIGURE 1 Evaluating f_4 Recursively.

tree consists of a root labeled with f_4, and branches from the root to vertices labeled with the two Fibonacci numbers f_3 and f_2 that occur in the reduction of the computation of f_4. Each subsequent reduction produces two branches in the tree. This branching ends when f_0 and f_1 are reached. The reader can verify that this algorithm requires $f_{n+1} - 1$ additions to find f_n.

Now consider the amount of computation required to find f_n using the iterative approach in Algorithm 8.

ALGORITHM 8 An Iterative Algorithm for Computing Fibonacci Numbers.

```
procedure iterative fibonacci(n: nonnegative integer)
if $n = 0$ then return 0
else
    $x := 0$
    $y := 1$
    for $i := 1$ to $n - 1$
        $z := x + y$
        $x := y$
        $y := z$
    return $y$
{output is the nth Fibonacci number}
```

This procedure initializes x as $f_0 = 0$ and y as $f_1 = 1$. When the loop is traversed, the sum of x and y is assigned to the auxiliary variable z. Then x is assigned the value of y and y is assigned the value of the auxiliary variable z. Therefore, after going through the loop the first time, it follows that x equals f_1 and y equals $f_0 + f_1 = f_2$. Furthermore, after going through the loop $n - 1$ times, x equals f_{n-1} and y equals f_n (the reader should verify this statement). Only $n - 1$ additions have been used to find f_n with this iterative approach when $n > 1$. Consequently, this algorithm requires far less computation than does the recursive algorithm.

We have shown that a recursive algorithm may require far more computation than an iterative one when a recursively defined function is evaluated. It is sometimes preferable to use a recursive procedure even if it is less efficient than the iterative procedure. In particular, this is true when the recursive approach is easily implemented and the iterative approach is not. (Also, machines designed to handle recursion may be available that eliminate the advantage of using iteration.)

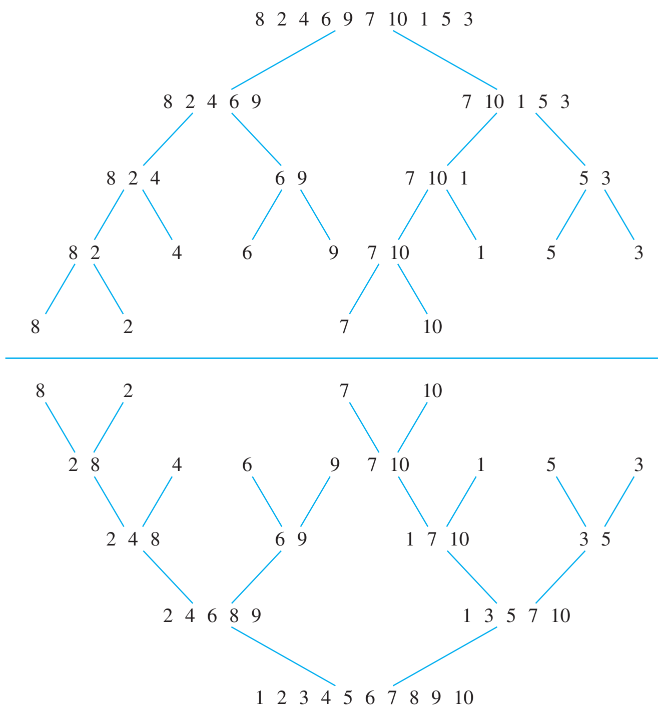

FIGURE 2 **The Merge Sort of 8, 2, 4, 6, 9, 7, 10, 1, 5, 3.**

The Merge Sort

Links

We now describe a recursive sorting algorithm called the **merge sort** algorithm. We will demonstrate how the merge sort algorithm works with an example before describing it in generality.

EXAMPLE 9 Use the merge sort to put the terms of the list 8, 2, 4, 6, 9, 7, 10, 1, 5, 3 in increasing order.

Solution: A merge sort begins by splitting the list into individual elements by successively splitting lists in two. The progression of sublists for this example is represented with the balanced binary tree of height 4 shown in the upper half of Figure 2.

Sorting is done by successively merging pairs of lists. At the first stage, pairs of individual elements are merged into lists of length two in increasing order. Then successive merges of pairs of lists are performed until the entire list is put into increasing order. The succession of merged lists in increasing order is represented by the balanced binary tree of height 4 shown in the lower half of Figure 2 (note that this tree is displayed "upside down"). ◀

In general, a merge sort proceeds by iteratively splitting lists into two sublists of equal length (or where one sublist has one more element than the other) until each sublist contains one element. This succession of sublists can be represented by a balanced binary tree. The procedure continues by successively merging pairs of lists, where both lists are in increasing order, into a larger list with elements in increasing order, until the original list is put into increasing order. The succession of merged lists can be represented by a balanced binary tree.

Demo

We can also describe the merge sort recursively. To do a merge sort, we split a list into two sublists of equal, or approximately equal, size, sorting each sublist using the merge sort algorithm, and then merging the two lists. The recursive version of the merge sort is given in Algorithm 9. This algorithm uses the subroutine *merge*, which is described in Algorithm 10.

ALGORITHM 9 A Recursive Merge Sort.

```
procedure mergesort($L = a_1, \dots, a_n$)
if $n > 1$ then
    $m := \lfloor n/2 \rfloor$
    $L_1 := a_1, a_2, \dots, a_m$
    $L_2 := a_{m+1}, a_{m+2}, \dots, a_n$
    $L := merge(mergesort(L_1),\ mergesort(L_2))$
{$L$ is now sorted into elements in nondecreasing order}
```

An efficient algorithm for merging two ordered lists into a larger ordered list is needed to implement the merge sort. We will now describe such a procedure.

EXAMPLE 10 Merge the two lists 2, 3, 5, 6 and 1, 4.

Solution: Table 1 illustrates the steps we use. First, compare the smallest elements in the two lists, 2 and 1, respectively. Because 1 is the smaller, put it at the beginning of the merged list and remove it from the second list. At this stage, the first list is 2, 3, 5, 6, the second is 4, and the combined list is 1.

Next, compare 2 and 4, the smallest elements of the two lists. Because 2 is the smaller, add it to the combined list and remove it from the first list. At this stage the first list is 3, 5, 6, the second is 4, and the combined list is 1, 2.

Continue by comparing 3 and 4, the smallest elements of their respective lists. Because 3 is the smaller of these two elements, add it to the combined list and remove it from the first list. At this stage the first list is 5, 6, and the second is 4. The combined list is 1, 2, 3.

Then compare 5 and 4, the smallest elements in the two lists. Because 4 is the smaller of these two elements, add it to the combined list and remove it from the second list. At this stage the first list is 5, 6, the second list is empty, and the combined list is 1, 2, 3, 4.

Finally, because the second list is empty, all elements of the first list can be appended to the end of the combined list in the order they occur in the first list. This produces the ordered list 1, 2, 3, 4, 5, 6. ◀

We will now consider the general problem of merging two ordered lists L_1 and L_2 into an ordered list L. We will describe an algorithm for solving this problem. Start with an empty list L. Compare the smallest elements of the two lists. Put the smaller of these two elements at the right end of L, and remove it from the list it was in. Next, if one of L_1 and L_2 is empty, append the other (nonempty) list to L, which completes the merging. If neither L_1 nor L_2 is empty, repeat this process. Algorithm 10 gives a pseudocode description of this procedure.

TABLE 1 Merging the Two Sorted Lists 2, 3, 5, 6 and 1, 4.

First List	*Second List*	*Merged List*	*Comparison*
2 3 5 6	1 4		$1 < 2$
2 3 5 6	4	1	$2 < 4$
3 5 6	4	1 2	$3 < 4$
5 6	4	1 2 3	$4 < 5$
5 6		1 2 3 4	
		1 2 3 4 5 6	

We will need estimates for the number of comparisons used to merge two ordered lists in the analysis of the merge sort. We can easily obtain such an estimate for Algorithm 10. Each time a comparison of an element from L_1 and an element from L_2 is made, an additional element is added to the merged list L. However, when either L_1 or L_2 is empty, no more comparisons are needed. Hence, Algorithm 10 is least efficient when $m + n - 2$ comparisons are carried out, where m and n are the number of elements in L_1 and L_2, respectively, leaving one element in each of L_1 and L_2. The next comparison will be the last one needed, because it will make one of these lists empty. Hence, Algorithm 10 uses no more than $m + n - 1$ comparisons. Lemma 1 summarizes this estimate.

ALGORITHM 10 Merging Two Lists.

```
procedure merge($L_1$, $L_2$: sorted lists)
$L$ := empty list
while $L_1$ and $L_2$ are both nonempty
    remove smaller of first elements of $L_1$ and $L_2$ from its list; put it at the right end of $L$
    if this removal makes one list empty then remove all elements from the other list and
        append them to $L$
return $L$ {$L$ is the merged list with elements in increasing order}
```

LEMMA 1 Two sorted lists with m elements and n elements can be merged into a sorted list using no more than $m + n - 1$ comparisons.

Sometimes two sorted lists of length m and n can be merged using far fewer than $m + n - 1$ comparisons. For instance, when $m = 1$, a binary search procedure can be applied to put the one element in the first list into the second list. This requires only $\lceil \log n \rceil$ comparisons, which is much smaller than $m + n - 1 = n$, for $m = 1$. On the other hand, for some values of m and n, Lemma 1 gives the best possible bound. That is, there are lists with m and n elements that cannot be merged using fewer than $m + n - 1$ comparisons. (See Exercise 47.)

We can now analyze the complexity of the merge sort. Instead of studying the general problem, we will assume that n, the number of elements in the list, is a power of 2, say 2^m. This will make the analysis less complicated, but when this is not the case, various modifications can be applied that will yield the same estimate.

At the first stage of the splitting procedure, the list is split into two sublists, of 2^{m-1} elements each, at level 1 of the tree generated by the splitting. This process continues, splitting the two sublists with 2^{m-1} elements into four sublists of 2^{m-2} elements each at level 2, and so on. In general, there are 2^{k-1} lists at level $k - 1$, each with 2^{m-k+1} elements. These lists at level $k - 1$ are split into 2^k lists at level k, each with 2^{m-k} elements. At the end of this process, we have 2^m lists each with one element at level m.

We start merging by combining pairs of the 2^m lists of one element into 2^{m-1} lists, at level $m - 1$, each with two elements. To do this, 2^{m-1} pairs of lists with one element each are merged. The merger of each pair requires exactly one comparison.

The procedure continues, so that at level k $(k = m, m - 1, m - 2, \ldots, 3, 2, 1)$, 2^k lists each with 2^{m-k} elements are merged into 2^{k-1} lists, each with 2^{m-k+1} elements, at level $k - 1$. To do this a total of 2^{k-1} mergers of two lists, each with 2^{m-k} elements, are needed. But, by Lemma 1, each of these mergers can be carried out using at most $2^{m-k} + 2^{m-k} - 1 = 2^{m-k+1} - 1$ comparisons. Hence, going from level k to $k - 1$ can be accomplished using at most $2^{k-1}(2^{m-k+1} - 1)$ comparisons.

Summing all these estimates shows that the number of comparisons required for the merge sort is at most

$$\sum_{k=1}^{m} 2^{k-1}(2^{m-k+1} - 1) = \sum_{k=1}^{m} 2^m - \sum_{k=1}^{m} 2^{k-1} = m2^m - (2^m - 1) = n \log n - n + 1,$$

because $m = \log n$ and $n = 2^m$. (We evaluated $\sum_{k=1}^{m} 2^m$ by noting that it is the sum of m identical terms, each equal to 2^m. We evaluated $\sum_{k=1}^{m} 2^{k-1}$ using the formula for the sum of the terms of a geometric progression from Theorem 1 of Section 2.4.)

Theorem 1 summarizes what we have discovered about the worst-case complexity of the merge sort algorithm.

THEOREM 1 The number of comparisons needed to merge sort a list with n elements is $O(n \log n)$.

In Chapter 11 we will show that the fastest comparison-based sorting algorithm have $O(n \log n)$ time complexity. (A comparison-based sorting algorithm has the comparison of two elements as its basic operation.) Theorem 1 tells us that the merge sort achieves this best possible big-O estimate for the complexity of a sorting algorithm. We describe another efficient algorithm, the quick sort, in the preamble to Exercise 50.

Exercises

1. Trace Algorithm 1 when it is given $n = 5$ as input. That is, show all steps used by Algorithm 1 to find 5!, as is done in Example 1 to find 4!.

2. Trace Algorithm 1 when it is given $n = 6$ as input. That is, show all steps used by Algorithm 1 to find 6!, as is done in Example 1 to find 4!.

3. Trace Algorithm 3 when it finds $\gcd(8, 13)$. That is, show all the steps used by Algorithm 3 to find $\gcd(8, 13)$.

4. Trace Algorithm 3 when it finds $\gcd(12, 17)$. That is, show all the steps used by Algorithm 3 to find $\gcd(12, 17)$.

5. Trace Algorithm 4 when it is given $m = 5$, $n = 11$, and $b = 3$ as input. That is, show all the steps Algorithm 4 uses to find 3^{11} **mod** 5.

6. Trace Algorithm 4 when it is given $m = 7$, $n = 10$, and $b = 2$ as input. That is, show all the steps Algorithm 4 uses to find 2^{10} **mod** 7.

7. Give a recursive algorithm for computing nx whenever n is a positive integer and x is an integer, using just addition.

8. Give a recursive algorithm for finding the sum of the first n positive integers.

9. Give a recursive algorithm for finding the sum of the first n odd positive integers.

10. Give a recursive algorithm for finding the maximum of a finite set of integers, making use of the fact that the maximum of n integers is the larger of the last integer in the list and the maximum of the first $n - 1$ integers in the list.

11. Give a recursive algorithm for finding the minimum of a finite set of integers, making use of the fact that the minimum of n integers is the smaller of the last integer in the list and the minimum of the first $n - 1$ integers in the list.

12. Devise a recursive algorithm for finding x^n **mod** m whenever n, x, and m are positive integers based on the fact that x^n **mod** $m = (x^{n-1}$ **mod** $m \cdot x$ **mod** $m)$ **mod** m.

13. Give a recursive algorithm for finding $n!$ **mod** m whenever n and m are positive integers.

14. Give a recursive algorithm for finding a **mode** of a list of integers. (A **mode** is an element in the list that occurs at least as often as every other element.)

15. Devise a recursive algorithm for computing the greatest common divisor of two nonnegative integers a and b with $a < b$ using the fact that $\gcd(a, b) = \gcd(a, b - a)$.

16. Prove that the recursive algorithm for finding the sum of the first n positive integers you found in Exercise 8 is correct.