three-cent and four-cent stamps. We can then form postage of $k + 1$ cents by replacing one three-cent stamp with a four-cent stamp or by replacing two four-cent stamps by three three-cent stamps.

33. Show that we can prove that $P(n, k)$ is true for all pairs of positive integers n and k if we show

a) $P(1, 1)$ is true and $P(n, k) \to [P(n + 1, k) \wedge P(n, k + 1)]$ is true for all positive integers n and k.

b) $P(1, k)$ is true for all positive integers k, and $P(n, k) \to P(n + 1, k)$ is true for all positive integers n and k.

c) $P(n, 1)$ is true for all positive integers n, and $P(n, k) \to P(n, k + 1)$ is true for all positive integers n and k.

34. Prove that $\sum_{j=1}^{n} j(j + 1)(j + 2) \cdots (j + k - 1) = n(n + 1)(n + 2) \cdots (n + k)/(k + 1)$ for all positive integers k and n. [*Hint*: Use a technique from Exercise 33.]

***35.** Show that if $a_1, a_2, \ldots, a_n$ are n distinct real numbers, exactly $n - 1$ multiplications are used to compute the product of these n numbers no matter how parentheses are inserted into their product. [*Hint:* Use strong induction and consider the last multiplication.]

***36.** The well-ordering property can be used to show that there is a unique greatest common divisor of two positive integers. Let a and b be positive integers, and let S be the set of positive integers of the form $as + bt$, where s and t are integers.

a) Show that S is nonempty.

b) Use the well-ordering property to show that S has a smallest element c.

c) Show that if d is a common divisor of a and b, then d is a divisor of c.

d) Show that $c \mid a$ and $c \mid b$. [*Hint:* First, assume that $c \nmid a$. Then $a = qc + r$, where $0 < r < c$. Show that $r \in S$, contradicting the choice of c.]

e) Conclude from (c) and (d) that the greatest common divisor of a and b exists. Finish the proof by showing that this greatest common divisor is unique.

37. Let a be an integer and d be a positive integer. Show that the integers q and r with $a = dq + r$ and $0 \leq r < d$, which were shown to exist in Example 5, are unique.

38. Use mathematical induction to show that a rectangular checkerboard with an even number of cells and two squares missing, one white and one black, can be covered by dominoes.

****39.** Can you use the well-ordering property to prove the statement: "Every positive integer can be described using no more than fifteen English words"? Assume the words come from a particular dictionary of English. [*Hint:* Suppose that there are positive integers that cannot be described using no more than fifteen English words. By well ordering, *the smallest positive integer that cannot be described using no more than fifteen English words* would then exist.]

40. Use the well-ordering principle to show that if x and y are real numbers with $x < y$, then there is a rational number r with $x < r < y$. [*Hint:* Use the Archimedean property, given in Appendix 1, to find a positive integer A with $A > 1/(y - x)$. Then show that there is a rational number r with denominator A between x and y by looking at the numbers $\lfloor x \rfloor + j/A$, where j is a positive integer.]

***41.** Show that the well-ordering property can be proved when the principle of mathematical induction is taken as an axiom.

***42.** Show that the principle of mathematical induction and strong induction are equivalent; that is, each can be shown to be valid from the other.

***43.** Show that we can prove the well-ordering property when we take either the principle of mathematical induction or strong induction as an axiom instead of taking the well-ordering property as an axiom.

4.3 Recursive Definitions and Structural Induction

Introduction

Sometimes it is difficult to define an object explicitly. However, it may be easy to define this object in terms of itself. This process is called **recursion.** For instance, the picture shown in Figure 1 is produced recursively. First, an original picture is given. Then a process of successively superimposing centered smaller pictures on top of the previous pictures is carried out.

We can use recursion to define sequences, functions, and sets. In previous discussions, we specified the terms of a sequence using an explicit formula. For instance, the sequence of powers of 2 is given by $a_n = 2^n$ for $n = 0, 1, 2, \ldots$. However, this sequence can also be defined by giving the first term of the sequence, namely, $a_0 = 1$, and a rule for finding a term of the

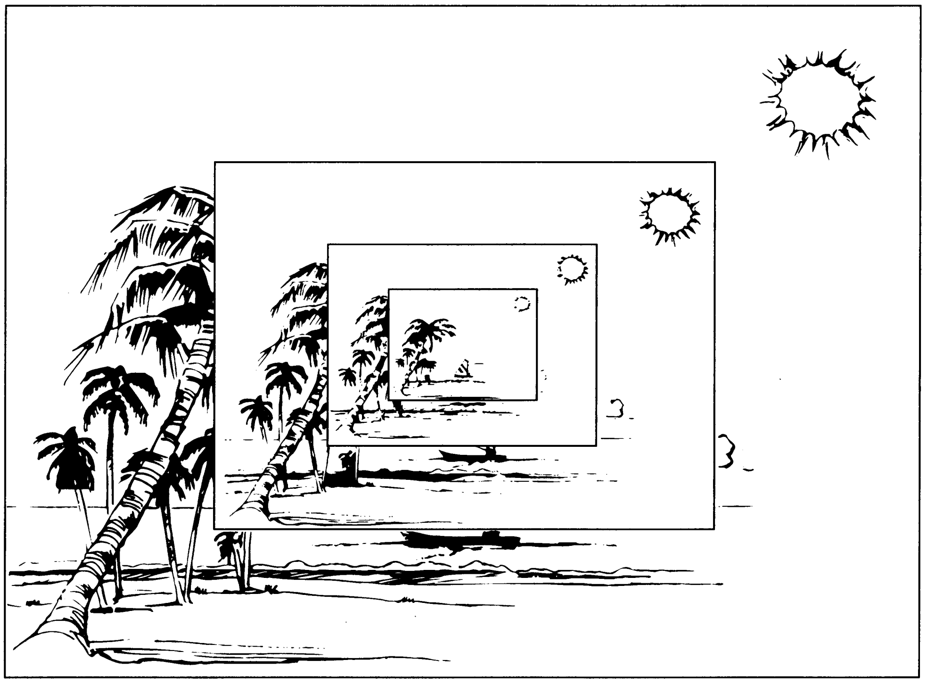


FIGURE 1 A Recursively Defined Picture.

sequence from the previous one, namely, $a_{n+1} = 2a_n$ for $n = 0, 1, 2, \ldots$. When we define a sequence *recursively* by specifying how terms of the sequence are found from previous terms, we can use induction to prove results about the sequence.

When we define a set recursively, we specify some initial elements in a basis step and provide a rule for constructing new elements from those we already have in the recursive step. To prove results about recursively defined sets we use a method called *structural induction.*

Recursively Defined Functions

We use two steps to define a function with the set of nonnegative integers as its domain:

BASIS STEP: Specify the value of the function at zero.

Assessment

RECURSIVE STEP: Give a rule for finding its value at an integer from its values at smaller integers.

Such a definition is called a **recursive** or **inductive definition.**

EXAMPLE 1 Suppose that f is defined recursively by

Extra Examples

$$f(0) = 3,$$
$$f(n+1) = 2f(n) + 3.$$

Find $f(1)$, $f(2)$, $f(3)$, and $f(4)$.

Solution: From the recursive definition it follows that

$$f(1) = 2f(0) + 3 = 2 \cdot 3 + 3 = 9,$$
$$f(2) = 2f(1) + 3 = 2 \cdot 9 + 3 = 21,$$
$$f(3) = 2f(2) + 3 = 2 \cdot 21 + 3 = 45,$$
$$f(4) = 2f(3) + 3 = 2 \cdot 45 + 3 = 93.$$ ◀

Many functions can be studied using their recursive definitions. The factorial function is one such example.

EXAMPLE 2 Give an inductive definition of the factorial function $F(n) = n!$.

Solution: We can define the factorial function by specifying the initial value of this function, namely, $F(0) = 1$, and giving a rule for finding $F(n+1)$ from $F(n)$. This is obtained by noting that $(n+1)!$ is computed from $n!$ by multiplying by $n+1$. Hence, the desired rule is

$$F(n+1) = (n+1)F(n).$$ ◀

To determine a value of the factorial function, such as $F(5) = 5!$, from the recursive definition found in Example 2, it is necessary to use the rule that shows how to express $F(n+1)$ in terms of $F(n)$ several times:

$$F(5) = 5F(4) = 5 \cdot 4F(3) = 5 \cdot 4 \cdot 3F(2) = 5 \cdot 4 \cdot 3 \cdot 2F(1)$$
$$= 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \cdot F(0) = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \cdot 1 = 120.$$

Once $F(0)$ is the only value of the function that occurs, no more reductions are necessary. The only thing left to do is to insert the value of $F(0)$ into the formula.

Recursively defined functions are **well defined.** That is, for every positive integer, the value of the function at this integer is determined in an unambiguous way. This means that given any positive integer, we can use the two parts of the definition to find the value of the function at that integer, and that we obtain the same value no matter how we apply the two parts of the definition. This is a consequence of the principle of mathematical induction. (See Exercise 56 at the end of this section.) Additional examples of recursive definitions are given in Examples 3 and 4.

EXAMPLE 3 Give a recursive definition of a^n, where a is a nonzero real number and n is a nonnegative integer.

Solution: The recursive definition contains two parts. First a^0 is specified, namely, $a^0 = 1$. Then the rule for finding a^{n+1} from a^n, namely, $a^{n+1} = a \cdot a^n$, for $n = 0, 1, 2, 3, \ldots$, is given. These two equations uniquely define a^n for all nonnegative integers n. ◀

EXAMPLE 4 Give a recursive definition of

$$\sum_{k=0}^{n} a_k .$$

Solution: The first part of the recursive definition is

$$\sum_{k=0}^{0} a_k = a_0.$$

The second part is

$$\sum_{k=0}^{n+1} a_k = \left(\sum_{k=0}^{n} a_k\right) + a_{n+1}.$$

◀

In some recursive definitions of functions, the values of the function at the first k positive integers are specified, and a rule is given for determining the value of the function at larger integers from its values at some or all of the preceding k integers. That recursive definitions defined in this way produce well-defined functions follows from strong induction (see Exercise 57 at the end of this section).

DEFINITION 1 The *Fibonacci numbers,* $f_0, f_1, f_2, \ldots$, are defined by the equations $f_0 = 0$, $f_1 = 1$, and

Links

$$f_n = f_{n-1} + f_{n-2}$$

for $n = 2, 3, 4, \ldots$.

EXAMPLE 5 Find the Fibonacci numbers f_2, f_3, f_4, f_5, and f_6.

Solution: Because the first part of the definition states that $f_0 = 0$ and $f_1 = 1$, it follows from the second part of the definition that

$$\begin{aligned} f_2 &= f_1 + f_0 = 1 + 0 = 1, \\ f_3 &= f_2 + f_1 = 1 + 1 = 2, \\ f_4 &= f_3 + f_2 = 2 + 1 = 3, \\ f_5 &= f_4 + f_3 = 3 + 2 = 5, \\ f_6 &= f_5 + f_4 = 5 + 3 = 8. \end{aligned}$$

◀

We can use the recursive definition of the Fibonacci numbers to prove many properties of these numbers. We give one such property in Example 6.

EXAMPLE 6 Show that whenever $n \geq 3$, $f_n > \alpha^{n-2}$, where $\alpha = (1 + \sqrt{5})/2$.

Extra Examples

Solution: We can use strong induction to prove this inequality. Let $P(n)$ be the statement $f_n > \alpha^{n-2}$. We want to show that $P(n)$ is true whenever n is an integer greater than or equal to 3.

BASIS STEP: First, note that

$$\alpha < 2 = f_3, \qquad \alpha^2 = (3 + \sqrt{5})/2 < 3 = f_4,$$

so $P(3)$ and $P(4)$ are true.

INDUCTIVE STEP: Assume that $P(j)$ is true, namely, that $f_j > \alpha^{j-2}$, for all integers j with $3 \leq j \leq k$, where $k \geq 4$. We must show that $P(k+1)$ is true, that is, that $f_{k+1} > \alpha^{k-1}$. Because α is a solution of $x^2 - x - 1 = 0$ (as the quadratic formula shows), it follows that $\alpha^2 = \alpha + 1$. Therefore,

$$\alpha^{k-1} = \alpha^2 \cdot \alpha^{k-3} = (\alpha + 1)\alpha^{k-3} = \alpha \cdot \alpha^{k-3} + 1 \cdot \alpha^{k-3} = \alpha^{k-2} + \alpha^{k-3}.$$

By the inductive hypothesis, if $k \geq 4$, it follows that

$$f_{k-1} > \alpha^{k-3}, \qquad f_k > \alpha^{k-2}.$$

Therefore, we have

$$f_{k+1} = f_k + f_{k-1} > \alpha^{k-2} + \alpha^{k-3} = \alpha^{k-1}.$$

It follows that $P(k+1)$ is true. This completes the proof. ◀

Remark: The inductive step shows that whenever $k \geq 4$, $P(k+1)$ follows from the assumption that $P(j)$ is true for $3 \leq j \leq k$. Hence, the inductive step does *not* show that $P(3) \rightarrow P(4)$. Therefore, we had to show that $P(4)$ is true separately.

We can now show that the Euclidean algorithm uses $O(\log b)$ divisions to find the greatest common divisor of the positive integers a and b, where $a \geq b$.

THEOREM 1 **LAMÉ'S THEOREM** Let a and b be positive integers with $a \geq b$. Then the number of divisions used by the Euclidean algorithm to find gcd(a, b) is less than or equal to five times the number of decimal digits in b.

Proof: Recall that when the Euclidean algorithm is applied to find gcd(a, b) with $a \geq b$, this sequence of equations (where $a = r_0$ and $b = r_1$) is obtained.

$$\begin{aligned} r_0 &= r_1 q_1 + r_2 & 0 &\leq r_2 < r_1 \\ r_1 &= r_2 q_2 + r_3 & 0 &\leq r_3 < r_2 \\ &\vdots \\ r_{n-2} &= r_{n-1} q_{n-1} + r_n & 0 &\leq r_n < r_{n-1} \\ r_{n-1} &= r_n q_n. \end{aligned}$$

Links

Here n divisions have been used to find $r_n = \gcd(a, b)$. Note that the quotients $q_1, q_2, \ldots, q_{n-1}$ are all at least 1. Moreover, $q_n \geq 2$, because $r_n < r_{n-1}$. This implies that

FIBONACCI (1170–1250) Fibonacci (short for *filius Bonacci,* or "son of Bonacci") was also known as Leonardo of Pisa. He was born in the Italian commercial center of Pisa. Fibonacci was a merchant who traveled extensively throughout the Mideast, where he came into contact with Arabian mathematics. In his book *Liber Abaci,* Fibonacci introduced the European world to Arabic notation for numerals and algorithms for arithmetic. It was in this book that his famous rabbit problem (described in Section 7.1) appeared. Fibonacci also wrote books on geometry and trigonometry and on Diophantine equations, which involve finding integer solutions to equations.

$$
\begin{aligned}
r_n &\geq 1 = f_2,\\
r_{n-1} &\geq 2r_n \geq 2f_2 = f_3,\\
r_{n-2} &\geq r_{n-1} + r_n \geq f_3 + f_2 = f_4,\\
&\vdots\\
r_2 &\geq r_3 + r_4 \geq f_{n-1} + f_{n-2} = f_n,\\
b = r_1 &\geq r_2 + r_3 \geq f_n + f_{n-1} = f_{n+1}.
\end{aligned}
$$

It follows that if n divisions are used by the Euclidean algorithm to find $\gcd(a, b)$ with $a \geq b$, then $b \geq f_{n+1}$. From Example 6 we know that $f_{n+1} > \alpha^{n-1}$ for $n > 2$, where $\alpha = (1 + \sqrt{5})/2$. Therefore, it follows that $b > \alpha^{n-1}$. Furthermore, because $\log_{10} \alpha \sim 0.208 > 1/5$, we see that

$$\log_{10} b > (n-1)\log_{10} \alpha > (n-1)/5.$$

Hence, $n - 1 < 5 \cdot \log_{10} b$. Now suppose that b has k decimal digits. Then $b < 10^k$ and $\log_{10} b < k$. It follows that $n - 1 < 5k$, and because k is an integer, it follows that $n \leq 5k$. This finishes the proof. ◁

Because the number of decimal digits in b, which equals $\lfloor \log_{10} b \rfloor + 1$, is less than or equal to $\log_{10} b + 1$, Theorem 1 tells us that the number of divisions required to find $\gcd(a, b)$ with $a > b$ is less than or equal to $5(\log_{10} b + 1)$. Because $5(\log_{10} b + 1)$ is $O(\log b)$, we see that $O(\log b)$ divisions are used by the Euclidean algorithm to find $\gcd(a, b)$ whenever $a > b$.

Recursively Defined Sets and Structures

Assessment

We have explored how functions can be defined recursively. We now turn our attention to how sets can be defined recursively. Just as in the recursive definition of functions, recursive definitions of sets have two parts, a **basis step** and a **recursive step.** In the basis step, an initial collection of elements is specified. In the recursive step, rules for forming new elements in the set from those already known to be in the set are provided. Recursive definitions may also include an **exclusion rule,** which specifies that a recursively defined set contains nothing other than those elements specified in the basis step or generated by applications of the recursive step. In our discussions, we will always tacitly assume that the exclusion rule holds and no element belongs to a recursively defined set unless it is in the initial collection specified in the basis step or can

Links

GABRIEL LAMÉ (1795–1870) Gabriel Lamé entered the École Polytechnique in 1813, graduating in 1817. He continued his education at the École des Mines, graduating in 1820.

In 1820 Lamé went to Russia, where he was appointed director of the Schools of Highways and Transportation in St. Petersburg. Not only did he teach, but he also planned roads and bridges while in Russia. He returned to Paris in 1832, where he helped found an engineering firm. However, he soon left the firm, accepting the chair of physics at the École Polytechnique, which he held until 1844. While holding this position, he was active outside academia as an engineering consultant, serving as chief engineer of mines and participating in the building of railways.

Lamé contributed original work to number theory, applied mathematics, and thermodynamics. His best-known work involves the introduction of curvilinear coordinates. His work on number theory includes proving Fermat's Last Theorem for $n = 7$, as well as providing the upper bound for the number of divisions used by the Euclidean algorithm given in this text.

In the opinion of Gauss, one of the most important mathematicians of all time, Lamé was the foremost French mathematician of his time. However, French mathematicians considered him too practical, whereas French scientists considered him too theoretical.

be generated using the recursive step one or more times. Later we will see how we can use a technique known as structural induction to prove results about recursively defined sets.

Examples 7, 8, 10, and 11 illustrate the recursive definition of sets. In each example, we show those elements generated by the first few applications of the recursive step.

EXAMPLE 7 Consider the subset S of the set of integers defined by

BASIS STEP: $3 \in S$.

RECURSIVE STEP: If $x \in S$ and $y \in S$, then $x + y \in S$.

Extra Examples

The new elements found to be in S are 3 by the basis step, $3 + 3 = 6$ at the first application of the recursive step, $3 + 6 = 6 + 3 = 9$ and $6 + 6 = 12$ at the second application of the recursive step, and so on. We will show later that S is the set of all positive multiples of 3. ◀

Recursive definitions play an important role in the study of strings. (See Chapter 12 for an introduction to the theory of formal languages, for example.) Recall from Section 2.4 that a string over an alphabet Σ is a finite sequence of symbols from Σ. We can define Σ^*, the set of strings over Σ, recursively, as Definition 2 shows.

DEFINITION 2 The set Σ^* of *strings* over the alphabet Σ can be defined recursively by

BASIS STEP: $\lambda \in \Sigma^*$ (where λ is the empty string containing no symbols).

RECURSIVE STEP: If $w \in \Sigma^*$ and $x \in \Sigma$, then $wx \in \Sigma^*$.

The basis step of the recursive definition of strings says that the empty string belongs to Σ^*. The recursive step states that new strings are produced by adding a symbol from Σ to the end of strings in Σ^*. At each application of the recursive step, strings containing one additional symbol are generated.

EXAMPLE 8 If $\Sigma = \{0, 1\}$, the strings found to be in Σ^*, the set of all bit strings, are λ, specified to be in Σ^* in the basis step, 0 and 1 formed during the first application of the recursive step, 00, 01, 10, and 11 formed during the second application of the recursive step, and so on. ◀

Recursive definitions can be used to define operations or functions on the elements of recursively defined sets. This is illustrated in Definition 3 of the concatenation of two strings and Example 9 concerning the length of a string.

DEFINITION 3 Two strings can be combined via the operation of *concatenation.* Let Σ be a set of symbols and Σ^* the set of strings formed from symbols in Σ. We can define the concatenation of two strings, denoted by $\cdot$, recursively as follows.

BASIS STEP: If $w \in \Sigma^*$, then $w \cdot \lambda = w$, where λ is the empty string.

RECURSIVE STEP: If $w_1 \in \Sigma^*$ and $w_2 \in \Sigma^*$ and $x \in \Sigma$, then $w_1 \cdot (w_2 x) = (w_1 \cdot w_2)x$.

The concatenation of the strings w_1 and w_2 is often written as $w_1 w_2$ rather than $w_1 \cdot w_2$. By repeated application of the recursive definition, it follows that the concatenation of two strings

w_1 and w_2 consists of the symbols in w_1 followed by the symbols in w_2. For instance, the concatenation of $w_1 = abra$ and $w_2 = cadabra$ is $w_1w_2 = abracadabra$.

EXAMPLE 9 **Length of a String** Give a recursive definition of $l(w)$, the length of the string w.

Solution: The length of a string can be defined by

$$l(\lambda) = 0;$$
$$l(wx) = l(w) + 1 \text{ if } w \in \Sigma^* \text{ and } x \in \Sigma.$$ ◀

Another important use of recursive definitions is to define **well-formed formulae** of various types. This is illustrated in Examples 10 and 11.

EXAMPLE 10 **Well-Formed Formulae for Compound Statement Forms** We can define the set of well-formed formulae for compound statement forms involving **T**, **F**, propositional variables, and operators from the set $\{\neg, \wedge, \vee, \rightarrow, \leftrightarrow\}$.

BASIS STEP: **T**, **F**, and s, where s is a propositional variable, are well-formed formulae.

RECURSIVE STEP: If E and F are well-formed formulae, then $(\neg E)$, $(E \wedge F)$, $(E \vee F)$, $(E \rightarrow F)$, and $(E \leftrightarrow F)$ are well-formed formulae.

For example, by the basis step we know that **T**, **F**, p, and q are well-formed formulae, where p and q are propositional variables. From an initial application of the recursive step, we know that $(p \vee q)$, $(p \rightarrow \mathbf{F})$, $(\mathbf{F} \rightarrow q)$, and $(q \wedge \mathbf{F})$ are well-formed formulae. A second application of the recursive step shows that $((p \vee q) \rightarrow (q \wedge \mathbf{F}))$, $(q \vee (p \vee q))$, and $((p \rightarrow \mathbf{F}) \rightarrow \mathbf{T})$ are well-formed formulae. We leave it to the reader to show that $p\neg \wedge q$, $pq\wedge$, and $\neg \wedge pq$ are *not* well-formed formulae, by showing that none can be obtained using the basis step and one or more applications of the recursive step. ◀

EXAMPLE 11 **Well-Formed Formulae of Operators and Operands** We can define the set of well-formed formulae consisting of variables, numerals, and operators from the set $\{+, -, *, /, \uparrow\}$ (where $*$ denotes multiplication and $\uparrow$ denotes exponentiation) recursively.

BASIS STEP: x is a well-formed formula if x is a numeral or variable.

RECURSIVE STEP: If F and G are well-formed formulae, then $(F + G)$, $(F - G)$, $(F * G)$, (F/G), and $(F \uparrow G)$ are well-formed formulae.

For example, by the basis step we see that x, y, 0, and 3 are well-formed formulae (as is any variable or numeral). Well-formed formulae generated by applying the recursive step once include $(x + 3)$, $(3 + y)$, $(x - y)$, $(3 - 0)$, $(x * 3)$, $(3 * y)$, $(3/0)$, (x/y), $(3 \uparrow x)$, and $(0 \uparrow 3)$. Applying the recursive step twice shows that formulae such as $((x + 3) + 3)$ and $(x - (3 * y))$ are well-formed formulae. [Note that $(3/0)$ is a well-formed formula because we are concerned only with syntax matters here.] We leave it to the reader to show that each of the formulae $x3\,+$, $y * + x$, and $* \, x/y$ is *not* a well-formed formula by showing that none of them can be obtained using the basis step and one or more applications of the recursive step. ◀

We will study trees extensively in Chapter 10. A tree is a special type of a graph; a graph is made up of vertices and edges connecting some pairs of vertices. We will study graphs in Chapter 9. We will briefly introduce them here to illustrate how they can be defined recursively.

DEFINITION 4 The set of *rooted trees,* where a rooted tree consists of a set of vertices containing a distinguished vertex called the *root,* and edges connecting these vertices, can be defined recursively by these steps:

BASIS STEP: A single vertex r is a rooted tree.

RECURSIVE STEP: Suppose that $T_1, T_2, \dots, T_n$ are disjoint rooted trees with roots $r_1, r_2, \dots, r_n$, respectively. Then the graph formed by starting with a root r, which is not in any of the rooted trees $T_1, T_2, \dots, T_n$, and adding an edge from r to each of the vertices $r_1, r_2, \dots, r_n$, is also a rooted tree.

In Figure 2 we illustrate some of the rooted trees formed starting with the basis step and applying the recursive step one time and two times. Note that infinitely many rooted trees are formed at each application of the recursive definition.

Binary trees are a special type of rooted trees. We will provide recursive definitions of two types of binary trees—full binary trees and extended binary trees. In the recursive step of the definition of each type of binary tree, two binary trees are combined to form a new tree with one of these trees designated the left subtree and the other the right subtree. In extended binary trees, the left subtree or the right subtree can be empty, but in full binary trees this is not possible. Binary trees are one of the most important types of structures in computer science. In Chapter 10 we will see how they can be used in searching and sorting algorithms, in algorithms for compressing data, and in many other applications. We first define extended binary trees.

DEFINITION 5 The set of *extended binary trees* can be defined recursively by these steps:

BASIS STEP: The empty set is an extended binary tree.

RECURSIVE STEP: If T_1 and T_2 are disjoint extended binary trees, there is an extended binary tree, denoted by $T_1 \cdot T_2$, consisting of a root r together with edges connecting the root to each of the roots of the left subtree T_1 and the right subtree T_2 when these trees are nonempty.

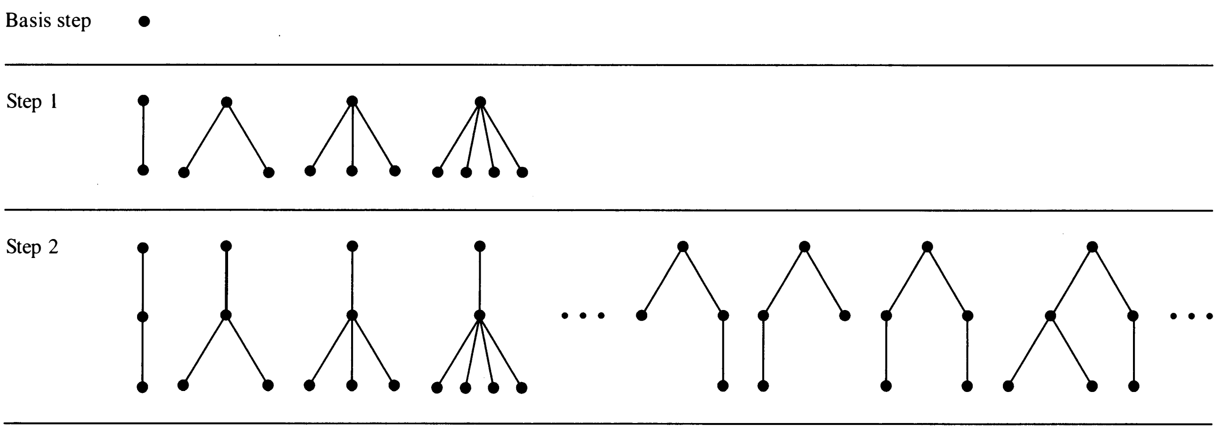


FIGURE 2 Building Up Rooted Trees.

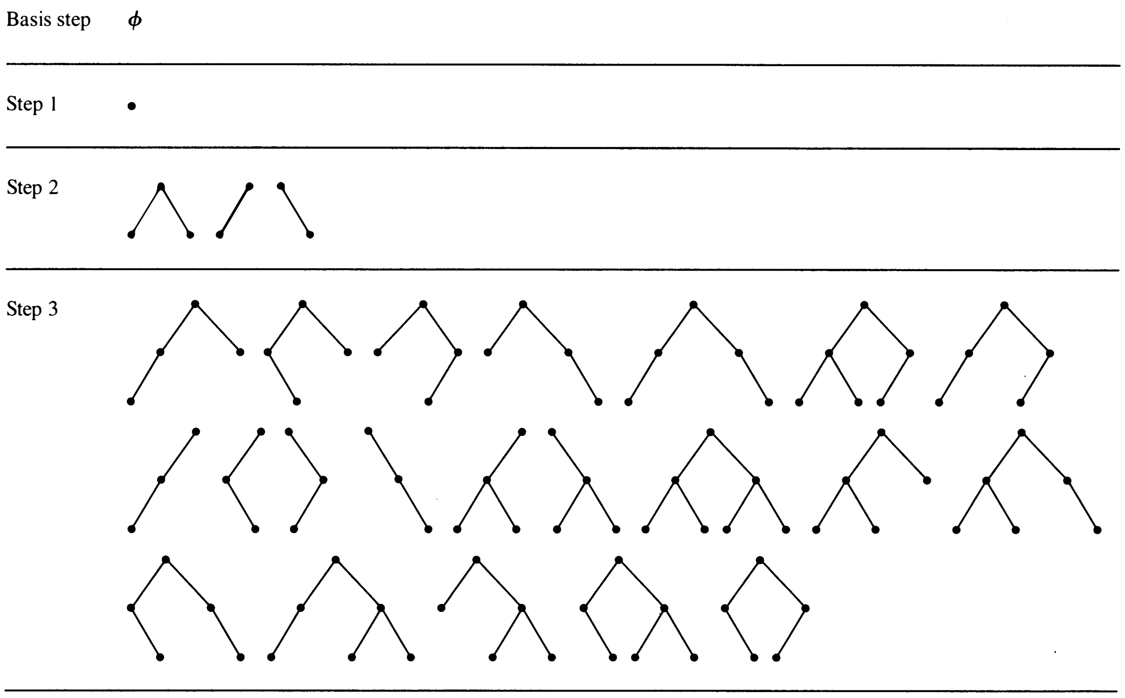


FIGURE 3 Building Up Extended Binary Trees.

Figure 3 shows how extended binary trees are built up by applying the recursive step from one to three times.

We now show how to define the set of full binary trees. Note that the difference between this recursive definition and that of extended binary trees lies entirely in the basis step.

DEFINITION 6 The set of full binary trees can be defined recursively by these steps:

BASIS STEP: There is a full binary tree consisting only of a single vertex r.

RECURSIVE STEP: If T_1 and T_2 are disjoint full binary trees, there is a full binary tree, denoted by $T_1 \cdot T_2$, consisting of a root r together with edges connecting the root to each of the roots of the left subtree T_1 and the right subtree T_2.

Figure 4 shows how full binary trees are built up by applying the recursive step one and two times.

Structural Induction

To prove results about recursively defined sets we generally use some form of mathematical induction. Example 12 illustrates the connection between recursively defined sets and mathematical induction.

EXAMPLE 12 Show that the set S defined in Example 7 by specifying that $3 \in S$ and that if $x \in S$ and $y \in S$, then $x + y \in S$, is the set of all positive integers that are multiples of 3.

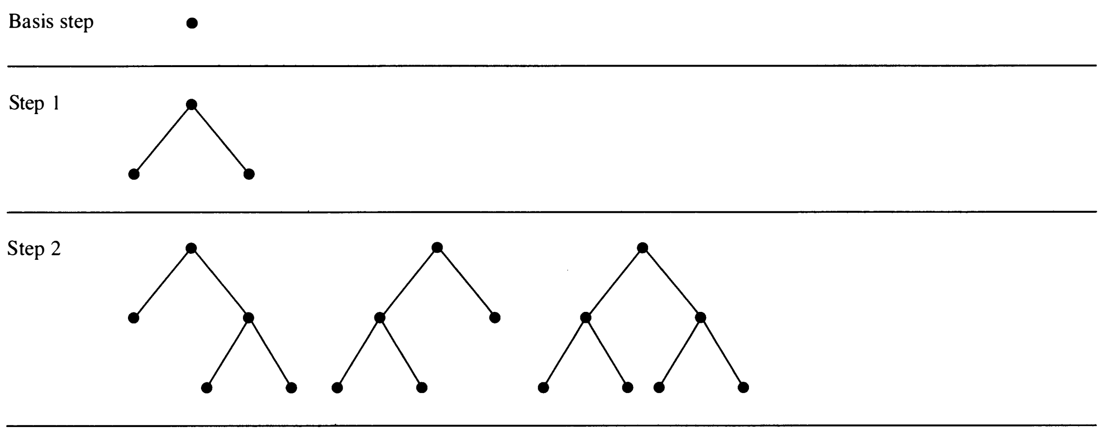


FIGURE 4 Building Up Full Binary Trees.

Solution: Let A be the set of all positive integers divisible by 3. To prove that $A = S$, we must show that A is a subset of S and that S is a subset of A. To prove that A is a subset of S, we must show that every positive integer divisible by 3 is in S. We will use mathematical induction to prove this.

Let $P(n)$ be the statement that $3n$ belongs to S. The basis step holds because by the first part of the recursive definition of S, $3 \cdot 1 = 3$ is in S. To establish the inductive step, assume that $P(k)$ is true, namely, that $3k$ is in S. Because $3k$ is in S and because 3 is in S, it follows from the second part of the recursive definition of S that $3k + 3 = 3(k + 1)$ is also in S.

To prove that S is a subset of A, we use the recursive definition of S. First, the basis step of the definition specifies that 3 is in S. Because $3 = 3 \cdot 1$, all elements specified to be in S in this step are divisible by 3 and are therefore in A. To finish the proof, we must show that all integers in S generated using the second part of the recursive definition are in A. This consists of showing that $x + y$ is in A whenever x and y are elements of S also assumed to be in A. Now if x and y are both in A, it follows that $3 \mid x$ and $3 \mid y$. By part (i) of Theorem 1 of Section 3.4, it follows that $3 \mid x + y$, completing the proof. ◀

In Example 12 we used mathematical induction over the set of positive integers and a recursive definition to prove a result about a recursively defined set. However, instead of using mathematical induction directly to prove results about recursively defined sets, we can use a more convenient form of induction known as **structural induction.** A proof by structural induction consists of two parts. These parts are

BASIS STEP: Show that the result holds for all elements specified in the basis step of the recursive definition to be in the set.

RECURSIVE STEP: Show that if the statement is true for each of the elements used to construct new elements in the recursive step of the definition, the result holds for these new elements.

The validity of structural induction follows from the principle of mathematical induction for the nonnegative integers. To see this, let $P(n)$ state that the claim is true for all elements of the set that are generated by n or fewer applications of the rules in the recursive step of a recursive definition. We will have established that the principle of mathematical induction implies the principle of structural induction if we can show that $P(n)$ is true whenever n is a positive integer. In the basis step of a proof by structural induction we show that $P(0)$ is true. That is, we show that the result is true of all elements specified to be in the set in the basis step of the definition. A consequence of the inductive step is that if we assume $P(k)$ is true, it follows that $P(k + 1)$ is true. When we have completed a proof using structural induction, we have shown that $P(0)$ is true and that $P(k)$ implies $P(k + 1)$. By mathematical induction it follows that $P(n)$ is true for all nonnegative integers n. This also shows that the result is true for all elements generated by the recursive definition, and shows that structural induction is a valid proof technique.

EXAMPLES OF PROOFS USING STRUCTURAL INDUCTION Structural induction can be used to prove that all members of a set constructed recursively have a particular property. We will illustrate this idea by using structural induction to prove results about well-formed formulae, strings, and binary trees. For each proof, we have to carry out the appropriate basis step and the appropriate recursive step. For example, to use structural induction to prove a result about the set of well-formed formulae defined in Example 10, where we specify that **T**, **F**, and every propositional variable s are well-formed formulae and where we specify that if E and F are well-formed formulae, then $(\neg E)$, $(E \wedge F)$, $(E \vee F)$, $(E \rightarrow F)$, and $(E \leftrightarrow F)$ are well-formed formulae, we need to complete this basis step and this recursive step.

BASIS STEP: Show that the result is true for **T**, **F**, and s whenever s is a propositional variable.

RECURSIVE STEP: Show that if the result is true for the compound propositions p and q, it is also true for $(\neg p)$, $(p \vee q)$, $(p \wedge q)$, $(p \rightarrow q)$, and $(p \leftrightarrow q)$.

Example 13 illustrates how we can prove results about well-formed formulae using structural induction.

EXAMPLE 13 Show that every well-formed formulae for compound propositions, as defined in Example 10, contains an equal number of left and right parentheses.

Solution:

BASIS STEP: Each of the formulae **T**, **F**, and s contains no parentheses, so clearly they contain an equal number of left and right parentheses.

RECURSIVE STEP: Assume p and q are well-formed formulae each containing an equal number of left and right parentheses. That is, if l_p and l_q are the number of left parentheses in p and q, respectively, and r_p and r_q are the number of right parentheses in p and q, respectively, then $l_p = r_p$ and $l_q = r_q$. To complete the inductive step, we need to show that each of $(\neg p)$, $(p \vee q)$, $(p \wedge q)$, $(p \rightarrow q)$, and $(p \leftrightarrow q)$ also contains an equal number of left and right parentheses. The number of left parentheses in the first of these compound propositions equals $l_p + 1$ and in each of the other compound propositions equals $l_p + l_q + 1$. Similarly, the number of right parentheses in the first of these compound propositions equals $r_p + 1$ and in each of the other compound propositions equals $r_p + r_q + 1$. Because $l_p = r_p$ and $l_q = r_q$, it follows that each of these compound expressions contains the same number of left and right parentheses. This completes the inductive proof. ◀

Suppose that $P(w)$ is a propositional function over the set of strings $w \in \Sigma^*$. To use structural induction to prove that $P(w)$ holds for all strings $w \in \Sigma^*$, we need to complete both a basis step and a recursive step. These steps are:

BASIS STEP: Show that $P(\lambda)$ is true.

RECURSIVE STEP: Assume that $P(w)$ is true, where $w \in \Sigma^*$. Show that if $x \in \Sigma$, then $P(wx)$ must also be true.

Example 14 illustrates how structural induction can be used in proofs about strings.

EXAMPLE 14 Use structural induction to prove that $l(xy) = l(x) + l(y)$, where x and y belong to Σ^*, the set of strings over the alphabet Σ.

Solution: We will base our proof on the recursive definition of the set Σ^* given in Definition 2 and the definition of the length of a string in Example 9, which specifies that $l(\lambda) = 0$ and $l(wx) = l(w) + 1$ when $w \in \Sigma^*$ and $x \in \Sigma$. Let $P(y)$ be the statement that $l(xy) = l(x) + l(y)$ whenever x belongs to Σ^*.

BASIS STEP: To complete the basis step, we must show that $P(\lambda)$ is true. That is, we must show that $l(x\lambda) = l(x) + l(\lambda)$ for all $x \in \Sigma^*$. Because $l(x\lambda) = l(x) = l(x) + 0 = l(x) + l(\lambda)$ for every string x, it follows that $P(\lambda)$ is true.

RECURSIVE STEP: To complete the inductive step, we assume that $P(y)$ is true and show that this implies that $P(ya)$ is true whenever $a \in \Sigma$. What we need to show is that $l(xya) = l(x) + l(ya)$ for every $a \in \Sigma$. To show this, note that by the recursive definition of $l(w)$ (given in Example 9), we have $l(xya) = l(xy) + 1$ and $l(ya) = l(y) + 1$. And, by the inductive hypothesis, $l(xy) = l(x) + l(y)$. We conclude that $l(xya) = l(x) + l(y) + 1 = l(x) + l(ya)$. ◀

We can prove results about trees or special classes of trees using structural induction. For example, to prove a result about full binary trees using structural induction we need to complete this basis step and this recursive step.

BASIS STEP: Show that the result is true for the tree consisting of a single vertex.

RECURSIVE STEP: Show that if the result is true for the trees T_1 and T_2, then it is true for tree $T_1 \cdot T_2$ consisting of a root r, which has T_1 as its left subtree and T_2 as its right subtree.

Before we provide an example showing how structural induction can be used to prove a result about full binary trees, we need some definitions. We will recursively define the height $h(T)$ and the number of vertices $n(T)$ of a full binary tree T. We begin by defining the height of a full binary tree.

DEFINITION 7 We define the height $h(T)$ of a full binary tree T recursively.

BASIS STEP: The height of the full binary tree T consisting of only a root r is $h(T) = 0$.

RECURSIVE STEP: If T_1 and T_2 are full binary trees, then the full binary tree $T = T_1 \cdot T_2$ has height $h(T) = 1 + \max(h(T_1), h(T_2))$.

If we let $n(T)$ denote the number of vertices in a full binary tree, we observe that $n(T)$ satisfies the following recursive formula:

BASIS STEP: The number of vertices $n(T)$ of the full binary tree T consisting of only a root r is $n(T) = 1$.

RECURSIVE STEP: If T_1 and T_2 are full binary trees, then the number of vertices of the full binary tree $T = T_1 \cdot T_2$ is $n(T) = 1 + n(T_1) + n(T_2)$.

We now show how structural induction can be used to prove a result about full binary trees.

THEOREM 2 If T is a full binary tree T, then $n(T) \leq 2^{h(T)+1} - 1$.

Proof: We prove this inequality using structural induction.

BASIS STEP: For the full binary tree consisting of just the root r the result is true because $n(T) = 1$ and $h(T) = 0$, so that $n(T) = 1 \leq 2^{0+1} - 1 = 1$.

INDUCTIVE STEP: For the inductive hypothesis we assume that $n(T_1) \leq 2^{h(T_1)+1} - 1$ and $n(T_2) \leq 2^{h(T_2)+1} - 1$ whenever T_1 and T_2 are full binary trees. By the recursive formulae for $n(T)$ and $h(T)$ we have $n(T) = 1 + n(T_1) + n(T_2)$ and $h(T) = 1 + \max(h(T_1), h(T_2))$.

We find that

$$\begin{aligned}
n(T) &= 1 + n(T_1) + n(T_2) && \text{by the recursive formula for } n(T)\\
&\leq 1 + (2^{h(T_1)+1} - 1) + (2^{h(T_2)+1} - 1) && \text{by the inductive hypothesis}\\
&\leq 2 \cdot \max(2^{h(T_1)+1}, 2^{h(T_2)+1}) - 1 && \text{because the sum of two terms is at most 2 times the larger}\\
&= 2 \cdot 2^{\max(h(T_1),h(T_2))+1} - 1 && \text{because } \max(2^x, 2^y) = 2^{\max(x,y)}\\
&= 2 \cdot 2^{h(T)} - 1 && \text{by the recursive definition of } h(T)\\
&= 2^{h(T)+1} - 1.
\end{aligned}$$

This completes the inductive step. ◁

Generalized Induction

We can extend mathematical induction to prove results about other sets that have the well-ordering property besides the set of integers. Although we will discuss this concept in detail in Section 8.6, we provide an example here to illustrate the usefulness of such an approach.

As an example, note that we can define an ordering on $\mathbf{N} \times \mathbf{N}$, the ordered pairs of nonnegative integers, by specifying that (x_1, y_1) is less than or equal to (x_2, y_2) if either $x_1 < x_2$, or $x_1 = x_2$ and $y_1 < y_2$; this is called the **lexicographic ordering.** The set $\mathbf{N} \times \mathbf{N}$ with this ordering has the property that every subset of $\mathbf{N} \times \mathbf{N}$ has a least element (see Supplementary Exercise 53 in Section 8.6). This implies that we can recursively define the terms $a_{m,n}$, with $m \in \mathbf{N}$ and $n \in \mathbf{N}$, and prove results about them using a variant of mathematical induction, as illustrated in Example 15.

EXAMPLE 15 Suppose that $a_{m,n}$ is defined recursively for $(m, n) \in \mathbf{N} \times \mathbf{N}$ by $a_{0,0} = 0$ and

$$a_{m,n} = \begin{cases} a_{m-1,n} + 1 & \text{if } n = 0 \text{ and } m > 0\\ a_{m,n-1} + n & \text{if } n > 0. \end{cases}$$

Show that $a_{m,n} = m + n(n + 1)/2$ for all $(m, n) \in \mathbf{N} \times \mathbf{N}$, that is, for all pairs of nonnegative integers.

Solution: We can prove that $a_{m,n} = m + n(n + 1)/2$ using a generalized version of mathematical induction. The basis step requires that we show that this formula is valid when $(m, n) = (0, 0)$. The induction step requires that we show that if the formula holds for all pairs smaller than (m, n) in the lexicographic ordering of $\mathbf{N} \times \mathbf{N}$, then it also holds for (m, n).

BASIS STEP: Let $(m, n) = (0, 0)$. Then by the basis case of the recursive definition of $a_{m,n}$ we have $a_{0,0} = 0$. Furthermore, when $m = n = 0$, $m + n(n + 1)/2 = 0 + (0 \cdot 1)/2 = 0$. This completes the basis step.

INDUCTIVE STEP: Suppose that $a_{m',n'} = m' + n'(n' + 1)/2$ whenever (m', n') is less than (m, n) in the lexicographic ordering of $\mathbf{N} \times \mathbf{N}$. By the recursive definition, if $n = 0$, then $a_{m,n} = a_{m-1,n} + 1$. Because $(m - 1, n)$ is smaller than (m, n), the inductive hypothesis tells us that $a_{m-1,n} = m - 1 + n(n + 1)/2$, so that $a_{m,n} = m - 1 + n(n + 1)/2 + 1 = m + n(n + 1)/2$, giving us the desired equality. Now suppose that $n > 0$, so $a_{m,n} = a_{m,n-1} + n$. Because $(m, n-1)$ is smaller than (m, n), the inductive hypothesis tells us that $a_{m,n-1} = m + (n-1)n/2$, so $a_{m,n} = m + (n-1)n/2 + n = m + (n^2 - n + 2n)/2 = m + n(n+1)/2$. This finishes the inductive step. ◀

As mentioned, we will justify this proof technique in Section 8.6.

Exercises

1. Find $f(1)$, $f(2)$, $f(3)$, and $f(4)$ if $f(n)$ is defined recursively by $f(0) = 1$ and for $n = 0, 1, 2, \ldots$

a) $f(n+1) = f(n) + 2$.
b) $f(n+1) = 3f(n)$.
c) $f(n+1) = 2^{f(n)}$.
d) $f(n+1) = f(n)^2 + f(n) + 1$.

2. Find $f(1)$, $f(2)$, $f(3)$, $f(4)$, and $f(5)$ if $f(n)$ is defined recursively by $f(0) = 3$ and for $n = 0, 1, 2, \ldots$

a) $f(n+1) = -2f(n)$.
b) $f(n+1) = 3f(n) + 7$.
c) $f(n+1) = f(n)^2 - 2f(n) - 2$.
d) $f(n+1) = 3^{f(n)/3}$.

3. Find $f(2)$, $f(3)$, $f(4)$, and $f(5)$ if f is defined recursively by $f(0) = -1$, $f(1) = 2$ and for $n = 1, 2, \ldots$

a) $f(n+1) = f(n) + 3f(n-1)$.
b) $f(n+1) = f(n)^2 f(n-1)$.
c) $f(n+1) = 3f(n)^2 - 4f(n-1)^2$.
d) $f(n+1) = f(n-1)/f(n)$.

4. Find $f(2)$, $f(3)$, $f(4)$, and $f(5)$ if f is defined recursively by $f(0) = f(1) = 1$ and for $n = 1, 2, \ldots$

a) $f(n+1) = f(n) - f(n-1)$.
b) $f(n+1) = f(n)f(n-1)$.
c) $f(n+1) = f(n)^2 + f(n-1)^3$.
d) $f(n+1) = f(n)/f(n-1)$.

5. Determine whether each of these proposed definitions is a valid recursive definition of a function f from the set of nonnegative integers to the set of integers. If f is well defined, find a formula for $f(n)$ when n is a nonnegative integer and prove that your formula is valid.

a) $f(0) = 0$, $f(n) = 2f(n-2)$ for $n \geq 1$
b) $f(0) = 1$, $f(n) = f(n-1) - 1$ for $n \geq 1$
c) $f(0) = 2$, $f(1) = 3$, $f(n) = f(n-1) - 1$ for $n \geq 2$
d) $f(0) = 1$, $f(1) = 2$, $f(n) = 2f(n-2)$ for $n \geq 2$
e) $f(0) = 1$, $f(n) = 3f(n-1)$ if n is odd and $n \geq 1$ and $f(n) = 9f(n-2)$ if n is even and $n \geq 2$

6. Determine whether each of these proposed definitions is a valid recursive definition of a function f from the set of nonnegative integers to the set of integers. If f is well defined, find a formula for $f(n)$ when n is a nonnegative integer and prove that your formula is valid.

a) $f(0) = 1$, $f(n) = -f(n-1)$ for $n \geq 1$
b) $f(0) = 1$, $f(1) = 0$, $f(2) = 2$, $f(n) = 2f(n-3)$ for $n \geq 3$
c) $f(0) = 0$, $f(1) = 1$, $f(n) = 2f(n+1)$ for $n \geq 2$
d) $f(0) = 0$, $f(1) = 1$, $f(n) = 2f(n-1)$ for $n \geq 1$
e) $f(0) = 2$, $f(n) = f(n-1)$ if n is odd and $n \geq 1$ and $f(n) = 2f(n-2)$ if $n \geq 2$

7. Give a recursive definition of the sequence $\{a_n\}$, $n = 1, 2, 3, \ldots$ if

a) $a_n = 6n$. **b)** $a_n = 2n + 1$.
c) $a_n = 10^n$. **d)** $a_n = 5$.

8. Give a recursive definition of the sequence $\{a_n\}$, $n = 1, 2, 3, \ldots$ if

a) $a_n = 4n - 2$. **b)** $a_n = 1 + (-1)^n$.
c) $a_n = n(n+1)$. **d)** $a_n = n^2$.

9. Let F be the function such that $F(n)$ is the sum of the first n positive integers. Give a recursive definition of $F(n)$.

10. Give a recursive definition of $S_m(n)$, the sum of the integer m and the nonnegative integer n.

11. Give a recursive definition of $P_m(n)$, the product of the integer m and the nonnegative integer n.

In Exercises 12–19 f_n is the nth Fibonacci number.

12. Prove that $f_1^2 + f_2^2 + \cdots + f_n^2 = f_n f_{n+1}$ when n is a positive integer.

13. Prove that $f_1 + f_3 + \cdots + f_{2n-1} = f_{2n}$ when n is a positive integer.

***14.** Show that $f_{n+1}f_{n-1} - f_n^2 = (-1)^n$ when n is a positive integer.

***15.** Show that $f_0 f_1 + f_1 f_2 + \cdots + f_{2n-1} f_{2n} = f_{2n}^2$ when n is a positive integer.

***16.** Show that $f_0 - f_1 + f_2 - \cdots - f_{2n-1} + f_{2n} = f_{2n-1} - 1$ when n is a positive integer.

17. Determine the number of divisions used by the Euclidean algorithm to find the greatest common divisor of the Fibonacci numbers f_n and f_{n+1}, where n is a nonnegative integer. Verify your answer using mathematical induction.

18. Let

$$\mathbf{A} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

Show that

$$\mathbf{A}^n = \begin{bmatrix} f_{n+1} & f_n \\ f_n & f_{n-1} \end{bmatrix}$$

when n is a positive integer.

19. By taking determinants of both sides of the equation in Exercise 18, prove the identity given in Exercise 14. (Recall that the determinant of the matrix $\begin{vmatrix} a & b \\ c & d \end{vmatrix}$ is $ad - bc$.)

***20.** Give a recursive definition of the functions max and min so that $\max(a_1, a_2, \ldots, a_n)$ and $\min(a_1, a_2, \ldots, a_n)$ are the maximum and minimum of the n numbers $a_1, a_2, \ldots, a_n$, respectively.

***21.** Let $a_1, a_2, \ldots, a_n$, and $b_1, b_2, \ldots, b_n$ be real numbers. Use the recursive definitions that you gave in Exercise 20 to prove these.

a) $\max(-a_1, -a_2, \ldots, -a_n) = -\min(a_1, a_2, \ldots, a_n)$

b) $\max(a_1 + b_1, a_2 + b_2, \ldots, a_n + b_n) \le \max(a_1, a_2, \ldots, a_n) + \max(b_1, b_2, \ldots, b_n)$

c) $\min(a_1 + b_1, a_2 + b_2, \ldots, a_n + b_n) \ge \min(a_1, a_2, \ldots, a_n) + \min(b_1, b_2, \ldots, b_n)$

22. Show that the set S defined by $1 \in S$ and $s + t \in S$ whenever $s \in S$ and $t \in S$ is the set of positive integers.

23. Give a recursive definition of the set of positive integers that are multiples of 5.

24. Give a recursive definition of

a) the set of odd positive integers.

b) the set of positive integer powers of 3.

c) the set of polynomials with integer coefficients.

25. Give a recursive definition of

a) the set of even integers.

b) the set of positive integers congruent to 2 modulo 3.

c) the set of positive integers not divisible by 5.

26. Let S be the subset of the set of ordered pairs of integers defined recursively by

Basis step: $(0, 0) \in S$.

Recursive step: If $(a, b) \in S$, then $(a + 2, b + 3) \in S$ and $(a + 3, b + 2) \in S$.

a) List the elements of S produced by the first five applications of the recursive definition.

b) Use strong induction on the number of applications of the recursive step of the definition to show that $5 \mid a + b$ when $(a, b) \in S$.

c) Use structural induction to show that $5 \mid a + b$ when $(a, b) \in S$.

27. Let S be the subset of the set of ordered pairs of integers defined recursively by

Basis step: $(0, 0) \in S$.

Recursive step: If $(a, b) \in S$, then $(a, b + 1) \in S$, $(a + 1, b + 1) \in S$, and $(a + 2, b + 1) \in S$.

a) List the elements of S produced by the first four applications of the recursive definition.

b) Use strong induction on the number of applications of the recursive step of the definition to show that $a \le 2b$ whenever $(a, b) \in S$.

c) Use structural induction to show that $a \le 2b$ whenever $(a, b) \in S$.

28. Give a recursive definition of each of these sets of ordered pairs of positive integers. [*Hint:* Plot the points in the set in the plane and look for lines containing points in the set.]

a) $S = \{(a, b) \mid a \in \mathbf{Z}^+, b \in \mathbf{Z}^+, \text{ and } a + b \text{ is odd}\}$

b) $S = \{(a, b) \mid a \in \mathbf{Z}^+, b \in \mathbf{Z}^+, \text{ and } a \mid b\}$

c) $S = \{(a, b) \mid a \in \mathbf{Z}^+, b \in \mathbf{Z}^+, \text{ and } 3 \mid a + b\}$

29. Give a recursive definition of each of these sets of ordered pairs of positive integers. Use structural induction to prove that the recursive definition you found is correct. [*Hint:* To find a recursive definition, plot the points in the set in the plane and look for patterns.]

a) $S = \{(a, b) \mid a \in \mathbf{Z}^+, b \in \mathbf{Z}^+, \text{ and } a + b \text{ is even}\}$

b) $S = \{(a, b) \mid a \in \mathbf{Z}^+, b \in \mathbf{Z}^+, \text{ and } a \text{ or } b \text{ is odd}\}$

c) $S = \{(a, b) \mid a \in \mathbf{Z}^+, b \in \mathbf{Z}^+, a + b \text{ is odd, and } 3 \mid b\}$

30. Prove that in a bit string, the string 01 occurs at most one more time than the string 10.

31. Define well-formed formulae of sets, variables representing sets, and operators from $\{^-, \cup, \cap, -\}$.

32. a) Give a recursive definition of the function *ones*(s), which counts the number of ones in a bit string s.

b) Use structural induction to prove that *ones*(st) = *ones*(s) + *ones*(t).

33. a) Give a recursive definition of the function $m(s)$, which equals the smallest digit in a nonempty string of decimal digits.

b) Use structural induction to prove that $m(st) = \min(m(s), m(t))$.

The **reversal** of a string is the string consisting of the symbols of the string in reverse order. The reversal of the string w is denoted by w^R.

34. Find the reversal of the following bit strings.

a) 0101 **b)** 1 1011 **c)** 1000 1001 0111

35. Give a recursive definition of the reversal of a string. [*Hint:* First define the reversal of the empty string. Then write a string w of length $n + 1$ as xy, where x is a string of length n, and express the reversal of w in terms of x^R and y.]

***36.** Use structural induction to prove that $(w_1w_2)^R = w_2^Rw_1^R$.

37. Give a recursive definition of w^i, where w is a string and i is a nonnegative integer. (Here w^i represents the concatenation of i copies of the string w.)

***38.** Give a recursive definition of the set of bit strings that are palindromes.

39. When does a string belong to the set A of bit strings defined recursively by

$$\lambda \in A$$
$$0x1 \in A \text{ if } x \in A,$$

where λ is the empty string?

***40.** Recursively define the set of bit strings that have more zeros than ones.

41. Use Exercise 37 and mathematical induction to show that $l(w^i) = i \cdot l(w)$, where w is a string and i is a nonnegative integer.

***42.** Show that $(w^R)^i = (w^i)^R$ whenever w is a string and i is a nonnegative integer; that is, show that the ith power of the reversal of a string is the reversal of the ith power of the string.

43. Use structural induction to show that $n(T) \geq 2h(T) + 1$, where T is a full binary tree, $n(T)$ equals the number of vertices of T, and $h(T)$ is the height of T.

The set of leaves and the set of internal vertices of a full binary tree can be defined recursively.

Basis step: The root r is a leaf of the full binary tree with exactly one vertex r. This tree has no internal vertices.

Recursive step: The set of leaves of the tree $T = T_1 \cdot T_2$ is the union of the set of leaves of T_1 and the set of leaves of T_2. The internal vertices of T are the root r of T and the union of the set of internal vertices of T_1 and the set of internal vertices of T_2.

44. Use structural induction to show that $l(T)$, the number of leaves of a full binary tree T, is 1 more than $i(T)$, the number of internal vertices of T.

45. Use generalized induction as was done in Example 15 to show that if $a_{m,n}$ is defined recursively by $a_{0,0} = 0$ and

$$a_{m,n} = \begin{cases} a_{m-1,n} + 1 & \text{if } n = 0 \text{ and } m > 0 \\ a_{m,n-1} + 1 & \text{if } n > 0, \end{cases}$$

then $a_{m,n} = m + n$ for all $(m, n) \in \mathbf{N} \times \mathbf{N}$.

46. Use generalized induction as was done in Example 15 to show that if $a_{m,n}$ is defined recursively by $a_{1,1} = 5$ and

$$a_{m,n} = \begin{cases} a_{m-1,n} + 2 & \text{if } n = 1 \text{ and } m > 1 \\ a_{m,n-1} + 2 & \text{if } n > 1, \end{cases}$$

then $a_{m,n} = 2(m + n) + 1$ for all $(m, n) \in \mathbf{Z}^+ \times \mathbf{Z}^+$.

***47.** A **partition** of a positive integer n is a way to write n as a sum of positive integers where the order of terms in the sum does not matter. For instance, $7 = 3 + 2 + 1 + 1$ is a partition of 7. Let P_m equal the number of different partitions of m, and let $P_{m,n}$ be the number of different ways to express m as the sum of positive integers not exceeding n.

a) Show that $P_{m,m} = P_m$.

b) Show that the following recursive definition for $P_{m,n}$ is correct:

$$P_{m,n} = \begin{cases} 1 & \text{if } m = 1 \\ 1 & \text{if } n = 1 \\ P_{m,m} & \text{if } m < n \\ 1 + P_{m,m-1} & \text{if } m = n > 1 \\ P_{m,n-1} + P_{m-n,n} & \text{if } m > n > 1. \end{cases}$$

c) Find the number of partitions of 5 and of 6 using this recursive definition.

Consider an inductive definition of a version of **Ackermann's function.** This function was named after Wilhelm Ackermann, a German mathematician who was a student of the great mathematician David Hilbert. Ackermann's function plays an important role in the theory of recursive functions and in the study of the complexity of certain algorithms involving set unions. (There are several different variants of this function. All are called Ackermann's function and have similar properties even though their values do not always agree.)

$$A(m, n) = \begin{cases} 2n & \text{if } m = 0 \\ 0 & \text{if } m \geq 1 \text{ and } n = 0 \\ 2 & \text{if } m \geq 1 \text{ and } n = 1 \\ A(m-1, A(m, n-1)) & \\ & \text{if } m \geq 1 \text{ and } n \geq 2 \end{cases}$$

Exercises 48–55 involve this version of Ackermann's function.

48. Find these values of Ackermann's function.

a) $A(1, 0)$ b) $A(0, 1)$
c) $A(1, 1)$ d) $A(2, 2)$

49. Show that $A(m, 2) = 4$ whenever $m \geq 1$.

50. Show that $A(1, n) = 2^n$ whenever $n \geq 1$.

51. Find these values of Ackermann's function.

a) $A(2, 3)$ ***b)** $A(3, 3)$

***52.** Find $A(3, 4)$.

****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 $\log^{(k)} n \le 1$.

60. Find each of 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 each of 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 $f^k(n) \le c$.

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?

4.4 Recursive Algorithms

Introduction

Sometimes we can reduce the solution to a problem with a particular set of input 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 first 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** *factorial*($n$) := 1
**else** *factorial*($n$) := $n \cdot$ *factorial*($n - 1$)
```

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** *power*($a, n$) := 1
**else** *power*($a, n$) := $a \cdot$ *power*($a, n - 1$)
```

EXAMPLE 3 Devise a recursive algorithm for computing b^n **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 3.4, and the initial condition b^0 **mod** $m = 1$. We leave this as Exercise 12 for the reader at the end of the section.

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

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

when n is odd, which we describe in pseudocode as Algorithm 3.

We use a trace of Algorithm 3 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 3 Recursive Modular Exponentiation.

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

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

EXAMPLE 4 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 4, which is a recursive version of the Euclidean algorithm.

We illustrate the workings of Algorithm 4 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 4 A Recursive Algorithm for Computing gcd(a, b).

```
procedure gcd(a, b: nonnegative integers with $a < b$)
if $a = 0$ then $gcd(a, b) := b$
else $gcd(a, b) := gcd(b \textbf{ mod } a, a)$
```

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 x in the search 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 i is the location of x. Otherwise, the search for x is reduced to a search in a sequence with one fewer element, namely, the sequence $a_{i+1}, \ldots, a_n$. 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 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 procedure 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. ◀

ALGORITHM 5 **A Recursive Linear Search Algorithm.**

```
procedure search($i, j, x$: $i, j, x$ integers, $1 \le i \le n, 1 \le j \le n$)
if $a_i = x$ then
    location := $i$
else if $i = j$ then
    location := 0
else
    search($i + 1, j, x$)
```

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

Solution: Suppose we want to locate x in the sequence $a_1, a_2, \ldots, a_n$, 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. 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 approximately half as long. 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, $1 \le i \le n, 1 \le j \le n$)
$m := \lfloor (i + j)/2 \rfloor$
if $x = a_m$ then
    location := $m$
else if ($x < a_m$ and $i < m$) then
    binary search($x, i, m - 1$)
else if ($x > a_m$ and $j > m$) then
    binary search($x, m + 1, j$)
else location := 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 $a \neq 0$ for the 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 $a \neq 0$ 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 3 is correct.

EXAMPLE 8 Prove that Algorithm 3, which computes modular powers, is correct.

Extra Examples

Solution: We use strong induction on the exponent n.

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

INDUCTIVE STEP: For the inductive hypothesis we assume that $mpower(b, j, m) = b^j \ \mathbf{mod}\ m$ 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 $mpower(b, k, m) = b^k \ \mathbf{mod}\ m$. 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

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

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

When k is odd, we have

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

using Corollary 2 in Section 3.4, 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 $b^{\lfloor k/2 \rfloor} \ \mathbf{mod}\ m$. This completes the inductive step.

We have completed the basis step and the inductive step, so by strong induction we know that Algorithm 3 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 fibonacci(0) := 0
else if $n = 1$ then fibonacci(1) := 1
else fibonacci(n) := fibonacci(n − 1) + fibonacci(n − 2)
```

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 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.

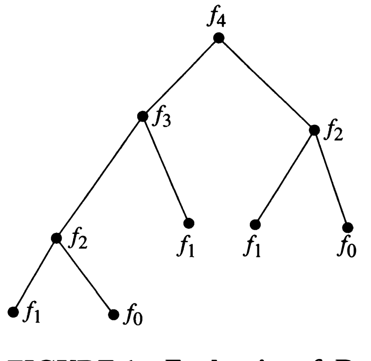


FIGURE 1 Evaluating f_4 Recursively.

ALGORITHM 8 **An Iterative Algorithm for Computing Fibonacci Numbers.**

```
procedure iterative fibonacci($n$: nonnegative integer)
if $n = 0$ then $y := 0$
else
begin
    $x := 0$
    $y := 1$
    for $i := 1$ to $n - 1$
    begin
        $z := x + y$
        $x := y$
        $y := z$
    end
end
{$y$ is the $n$th 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.)

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 Sort the list 8, 2, 4, 6, 9, 7, 10, 1, 5, 3 using the merge sort.

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

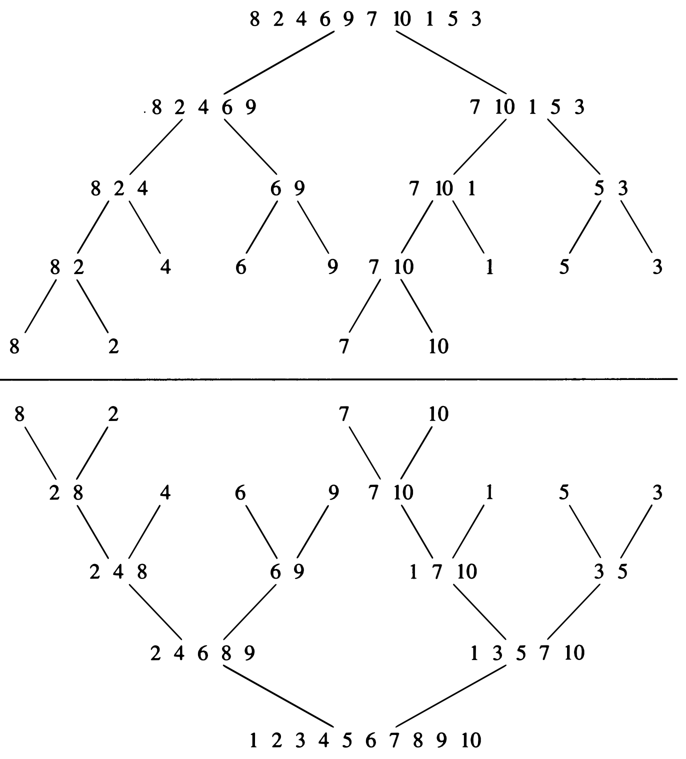


FIGURE 2 The Merge Sort of 8, 2, 4, 6, 9, 7, 10, 1, 5, 3.

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}
```

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	

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 left 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.

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
begin
   remove smaller of first element of $L_1$ and $L_2$ from the list it is
        in and put it at the right end of $L$
   if removal of this element makes one list empty then remove
        all elements from the other list and append them to $L$
end {$L$ 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 at the end of this section.)

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.)

This analysis shows that the merge sort achieves the best possible big-O estimate for the number of comparisons needed by sorting algorithms, as stated in the following theorem.

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

We describe another efficient algorithm, the quick sort, in the exercises.

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 is given $m = 5$, $n = 11$, and $b = 3$ as input. That is, show all the steps Algorithm 3 uses to find 3^{11} **mod** 5.
4. Trace Algorithm 3 when it is given $m = 7$, $n = 10$, and $b = 2$ as input. That is, show all the steps Algorithm 3 uses to find 2^{10}**mod** 7.
5. Trace Algorithm 4 when it finds gcd(8, 13). That is, show all the steps used by Algorithm 4 to find gcd(8, 13).
6. Trace Algorithm 4 when it finds gcd(12, 17). That is, show all the steps used by Algorithm 4 to find gcd(12, 17).
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 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.
17. Describe a recursive algorithm for multiplying two nonnegative integers x and y based on the fact that $xy = 2(x \cdot (y/2))$ when y is even and $xy = 2(x \cdot \lfloor y/2 \rfloor) + x$ when y is odd, together with the initial condition $xy = 0$ when $y = 0$.
18. Prove that Algorithm 1 for computing $n!$ when n is a nonnegative integer is correct.
19. Prove that Algorithm 4 for computing gcd(a, b) when a and b are positive integers with $a < b$ is correct.
20. Prove that the algorithm you devised in Exercise 11 is correct.
21. Prove that the recursive algorithm that you found in Exercise 7 is correct.
22. Prove that the recursive algorithm that you found in Exercise 10 is correct.
23. Devise a recursive algorithm for computing n^2 where n is a nonnegative integer using the fact that $(n + 1)^2 = n^2 + 2n + 1$. Then prove that this algorithm is correct.
24. Devise a recursive algorithm to find a^{2^n}, where a is a real number and n is a positive integer. [*Hint:* Use the equality $a^{2^{n+1}} = (a^{2^n})^2$.]
25. How does the number of multiplications used by the algorithm in Exercise 24 compare to the number of multiplications used by Algorithm 2 to evaluate a^{2^n}?

*26. Use the algorithm in Exercise 24 to devise an algorithm for evaluating a^n when n is a nonnegative integer. [*Hint:* Use the binary expansion of n.]

*27. How does the number of multiplications used by the algorithm in Exercise 26 compare to the number of multiplications used by Algorithm 2 to evaluate a^n?

28. How many additions are used by the recursive and iterative algorithms given in Algorithms 7 and 8, respectively, to find the Fibonacci number f_7?
29. Devise a recursive algorithm to find the nth term of the sequence defined by $a_0 = 1$, $a_1 = 2$, and $a_n = a_{n-1} \cdot a_{n-2}$, for $n = 2, 3, 4, \ldots$.
30. Devise an iterative algorithm to find the nth term of the sequence defined in Exercise 29.

31. Is the recursive or the iterative algorithm for finding the sequence in Exercise 29 more efficient?

32. Devise a recursive algorithm to find the nth term of the sequence defined by $a_0 = 1$, $a_1 = 2$, $a_2 = 3$, and $a_n = a_{n-1} + a_{n-2} + a_{n-3}$, for $n = 3, 4, 5, \ldots$.

33. Devise an iterative algorithm to find the nth term of the sequence defined in Exercise 32.

34. Is the recursive or the iterative algorithm for finding the sequence in Exercise 32 more efficient?

35. Give iterative and recursive algorithms for finding the nth term of the sequence defined by $a_0 = 1$, $a_1 = 3$, $a_2 = 5$, and $a_n = a_{n-1} \cdot a_{n-2}^2 \cdot a_{n-3}^3$. Which is more efficient?

36. Give a recursive algorithm to find the number of partitions of a positive integer based on the recursive definition given in Exercise 47 in Section 4.3.

37. Give a recursive algorithm for finding the reversal of a bit string. (See the definition of the reversal of a bit string in the preamble of Exercise 34 in Section 4.3.)

38. Give a recursive algorithm for finding the string w^i, the concatenation of i copies of w, when w is a bit string.

39. Prove that the recursive algorithm for finding the reversal of a bit string that you gave in Exercise 37 is correct.

40. Prove that the recursive algorithm for finding the concatenation of i copies of a bit string that you gave in Exercise 38 is correct.

***41.** Give a recursive algorithm for tiling a $2^n \times 2^n$ checkerboard with one square missing using right triominoes.

42. Give a recursive algorithm for triangulating a simple polygon with n sides, using Lemma 1 in Section 4.2.

43. Give a recursive algorithm for computing values of the Ackermann function. [*Hint:* See the preamble to Exercise 48 in Section 4.3.]

44. Use a merge sort to sort 4, 3, 2, 5, 1, 8, 7, 6. Show all the steps used by the algorithm.

45. Use a merge sort to sort *b, d, a, f, g, h, z, p, o, k*. Show all the steps used by the algorithm.

46. How many comparisons are required to merge these pairs of lists using Algorithm 10?

a) 1, 3, 5, 7, 9; 2, 4, 6, 8, 10
b) 1, 2, 3, 4, 5; 6, 7, 8, 9, 10
c) 1, 5, 6, 7, 8; 2, 3, 4, 9, 10

47. Show that for all positive integers m and n there are lists with m elements and n elements, respectively, such that Algorithm 10 uses $m + n - 1$ comparisons to merge them into one sorted list.

***48.** What is the least number of comparisons needed to merge any two lists in increasing order into one list in increasing order when the number of elements in the two lists are

a) 1, 4? **b)** 2, 4? **c)** 3, 4? **d)** 4, 4?

***49.** Prove that the merge sort algorithm is correct.

The **quick sort** is an efficient algorithm. To sort $a_1, a_2, \ldots, a_n$, this algorithm begins by taking the first element a_1 and forming two sublists, the first containing those elements that are less than a_1, in the order they arise, and the second containing those elements greater than a_1, in the order they arise. Then a_1 is put at the end of the first sublist. This procedure is repeated recursively for each sublist, until all sublists contain one item. The ordered list of n items is obtained by combining the sublists of one item in the order they occur.

50. Sort 3, 5, 7, 8, 1, 9, 2, 4, 6 using the quick sort.

51. Let $a_1, a_2, \ldots, a_n$ be a list of n distinct real numbers. How many comparisons are needed to form two sublists from this list, the first containing elements less than a_1 and the second containing elements greater than a_1?

52. Describe the quick sort algorithm using pseudocode.

53. What is the largest number of comparisons needed to order a list of four elements using the quick sort algorithm?

54. What is the least number of comparisons needed to order a list of four elements using the quick sort algorithm?

55. Determine the worst-case complexity of the quick sort algorithm in terms of the number of comparisons used.

4.5 Program Correctness

Introduction

Suppose that we have designed an algorithm to solve a problem and have written a program to implement it. How can we be sure that the program always produces the correct answer? After all the bugs have been removed so that the syntax is correct, we can test the program with sample input. It is not correct if an incorrect result is produced for any sample input. But even if the program gives the correct answer for all sample input, it may not always produce the correct answer (unless all possible input has been tested). We need a proof to show that the program *always* gives the correct output.

Program verification, the proof of correctness of programs, uses the rules of inference and proof techniques described in this chapter, including mathematical induction. Because an incorrect program can lead to disastrous results, a large amount of methodology has been constructed for verifying programs. Efforts have been devoted to automating program verification so that it can be carried out using a computer. However, only limited progress has been made toward this goal. Indeed, some mathematicians and theoretical computer scientists argue that it will never be realistic to mechanize the proof of correctness of complex programs.

Some of the concepts and methods used to prove that programs are correct will be introduced in this section. Many different methods have been devised for proving that programs are correct. We will discuss a method for program verification introduced by Tony Hoare in this section; many other methods have been devised. Furthermore, we will not develop a complete methodology for program verification in this book. This section is meant to be a brief introduction to the area of program verification, which ties together the rules of logic, proof techniques, and the concept of an algorithm.

Program Verification

A program is said to be **correct** if it produces the correct output for every possible input. A proof that a program is correct consists of two parts. The first part shows that the correct answer is obtained if the program terminates. This part of the proof establishes the **partial correctness** of the program. The second part of the proof shows that the program always terminates.

To specify what it means for a program to produce the correct output, two propositions are used. The first is the **initial assertion,** which gives the properties that the input values must have. The second is the **final assertion,** which gives the properties that the output of the program should have, if the program did what was intended. The appropriate initial and final assertions must be provided when a program is checked.

DEFINITION 1 A program, or program segment, S is said to be *partially correct with respect to* the initial assertion p and the final assertion q if whenever p is true for the input values of S and S terminates, then q is true for the output values of S. The notation $p\{S\}q$ indicates that the program, or program segment, S is partially correct with respect to the initial assertion p and the final assertion q.

Note: The notation $p\{S\}q$ is known as a *Hoare triple.* Tony Hoare introduced the concept of partial correctness.

Links

Note that the notion of partial correctness has nothing to do with whether a program terminates; it focuses only on whether the program does what it is expected to do if it terminates.

A simple example illustrates the concepts of initial and final assertions.

EXAMPLE 1 Show that the program segment

$$y := 2$$
$$z := x + y$$

Extra Examples

is correct with respect to the initial assertion p: $x = 1$ and the final assertion q: $z = 3$.

Solution: Suppose that p is true, so that $x = 1$ as the program begins. Then y is assigned the value 2, and z is assigned the sum of the values of x and y, which is 3. Hence, S is correct with respect to the initial assertion p and the final assertion q. Thus, $p\{S\}q$ is true. ◀

Rules of Inference

A useful rule of inference proves that a program is correct by splitting the program into a series of subprograms and then showing that each subprogram is correct.

Suppose that the program S is split into subprograms S_1 and S_2. Write $S = S_1; S_2$ to indicate that S is made up of S_1 followed by S_2. Suppose that the correctness of S_1 with respect to the initial assertion p and final assertion q, and the correctness of S_2 with respect to the initial assertion q and the final assertion r, have been established. It follows that if p is true and S_1 is executed and terminates, then q is true; and if q is true, and S_2 executes and terminates, then r is true. Thus, if p is true and $S = S_1; S_2$ is executed and terminates, then r is true. This rule of inference, called the **composition rule,** can be stated as

$$\begin{array}{l} p\{S_1\}q \\ \underline{q\{S_2\}r} \\ \therefore p\{S_1; S_2\}r. \end{array}$$

This rule of inference will be used later in this section.

Next, some rules of inference for program segments involving conditional statements and loops will be given. Because programs can be split into segments for proofs of correctness, this will let us verify many different programs.

Conditional Statements

First, rules of inference for conditional statements will be given. Suppose that a program segment has the form

```
if condition then
    S
```

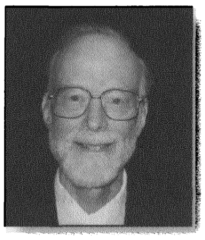
where S is a block of statements. Then S is executed if *condition* is true, and it is not executed when *condition* is false. To verify that this segment is correct with respect to the initial assertion p and final assertion q, two things must be done. First, it must be shown that when p is true and *condition* is also true, then q is true after S terminates. Second, it must be shown that when p is true and *condition* is false, then q is true (because in this case S does not execute).

This leads to the following rule of inference:

$$\begin{array}{l} (p \wedge \mathit{condition})\{S\}q \\ \underline{(p \wedge \neg \mathit{condition}) \to q} \\ \therefore p\{\textbf{if } \mathit{condition} \textbf{ then } S\}q. \end{array}$$

Example 2 illustrates how this rule of inference is used.

Links



C. ANTHONY R. HOARE (BORN 1934) Tony Hoare is currently Professor of Computer Science at Oxford University, England, and is a Fellow of the Royal Society. Hoare has made many important contributions to the theory of programming languages and to programming methodology. He was the first person to define a programming language based on how programs could be proved to be correct with respect to their specifications. Hoare is also the creator of the quick sort, one of the most commonly used and studied sorting algorithms (see the preamble to Exercise 50 in Section 4.4). Hoare is a noted writer in the technical and social aspects of computer science.

EXAMPLE 2 Verify that the program segment

```
if $x > y$ then
    $y := x$
```

is correct with respect to the initial assertion **T** and the final assertion $y \geq x$.

Solution: When the initial assertion is true and $x > y$, the assignment $y := x$ is carried out. Hence, the final assertion, which asserts that $y \geq x$, is true in this case. Moreover, when the initial assertion is true and $x > y$ is false, so that $x \leq y$, the final assertion is again true. Hence, using the rule of inference for program segments of this type, this program is correct with respect to the given initial and final assertions. ◀

Similarly, suppose that a program has a statement of the form

```
if condition then
    $S_1$
else
    $S_2$
```

If *condition* is true, then S_1 executes; if *condition* is false, then S_2 executes. To verify that this program segment is correct with respect to the initial assertion p and the final assertion q, two things must be done. First, it must be shown that when p is true and *condition* is true, then q is true after S_1 terminates. Second, it must be shown that when p is true and *condition* is false, then q is true after S_2 terminates. This leads to the following rule of inference:

$$\begin{array}{l} (p \wedge \mathit{condition})\{S_1\}q \\ (p \wedge \neg\mathit{condition})\{S_2\}q \\ \hline \therefore p\{\textbf{if } \mathit{condition} \textbf{ then } S_1 \textbf{ else } S_2\}q. \end{array}$$

Example 3 illustrates how this rule of inference is used.

EXAMPLE 3 Verify that the program segment

```
if $x < 0$ then
    $abs := -x$
else
    $abs := x$
```

is correct with respect to the initial assertion **T** and the final assertion $abs = |x|$.

Solution: Two things must be demonstrated. First, it must be shown that if the initial assertion is true and $x < 0$, then $abs = |x|$. This is correct, because when $x < 0$ the assignment statement $abs := -x$ sets $abs = -x$, which is $|x|$ by definition when $x < 0$. Second, it must be shown that if the initial assertion is true and $x < 0$ is false, so that $x \geq 0$, then $abs = |x|$. This is also correct, because in this case the program uses the assignment statement $abs := x$, and x is $|x|$

by definition when $x \geq 0$, so $abs := x$. Hence, using the rule of inference for program segments of this type, this segment is correct with respect to the given initial and final assertions. ◀

Loop Invariants

Links

Next, proofs of correctness of **while** loops will be described. To develop a rule of inference for program segments of the type

```
while condition
    S
```

note that S is repeatedly executed until *condition* becomes false. An assertion that remains true each time S is executed must be chosen. Such an assertion is called a **loop invariant.** In other words, p is a loop invariant if $(p \wedge condition)\{S\}p$ is true.

Suppose that p is a loop invariant. It follows that if p is true before the program segment is executed, p and $\neg condition$ are true after termination, if it occurs. This rule of inference is

$$\frac{(p \wedge condition)\{S\}p}{\therefore p\{\textbf{while}\ condition\quad S\}(\neg\ condition\ \wedge p).}$$

The use of a loop invariant is illustrated in Example 4.

EXAMPLE 4 A loop invariant is needed to verify that the program segment

Extra Examples

```
i := 1
factorial := 1
while i < n
begin
    i := i + 1
    factorial := factorial · i
end
```

terminates with $factorial = n!$ when n is a positive integer.

Let p be the assertion "$factorial = i!$ and $i \leq n$." We first prove that p is a loop invariant. Suppose that, at the beginning of one execution of the **while** loop, p is true and the condition of the **while** loop holds; in other words, assume that $factorial = i!$ and that $i < n$. The new values i_{new} and $factorial_{\text{new}}$ of i and $factorial$ are $i_{\text{new}} = i + 1$ and $factorial_{\text{new}} = factorial \cdot (i + 1) = (i + 1)! = i_{\text{new}}!$. Because $i < n$, we also have $i_{\text{new}} = i + 1 \leq n$. Thus, p is true at the end of the execution of the loop. This shows that p is a loop invariant.

Now we consider the program segment. Just before entering the loop, $i = 1 \leq n$ and $factorial = 1 = 1! = i!$ both hold, so p is true. Because p is a loop invariant, the rule of inference just introduced implied that if the **while** loop terminates, it terminates with p true and with $i < n$ false. In this case, at the end, $factorial = i!$ and $i \leq n$ are true, but $i < n$ is false; in other words, $i = n$ and $factorial = i! = n!$, as desired.

Finally, we need to check that the **while** loop actually terminates. At the beginning of the program i is assigned the value 1, so after $n - 1$ traversals of the loop, the new value of i will be n, and the loop terminates at that point. ◀

A final example will be given to show how the various rules of inference can be used to verify the correctness of a longer program.

EXAMPLE 5 We will outline how to verify the correctness of the program S for computing the product of two integers.

```
procedure multiply(m, n: integers)

S_1 { if n < 0 then a := −n
    { else a := n

S_2 { k := 0
    { x := 0

    { while k < a
    { begin
S_3 {    x := x + m
    {    k := k + 1
    { end

S_4 { if n < 0 then product := −x
    { else product := x
```

The goal is to prove that after S is executed, *product* has the value mn. The proof of correctness can be carried out by splitting S into four segments, with $S = S_1; S_2; S_3; S_4$, as shown in the listing of S. The rule of composition can be used to build the correctness proof. Here is how the argument proceeds. The details will be left as an exercise for the reader.

Let p be the initial assertion "m and n are integers." Then, it can be shown that $p\{S_1\}q$ is true, when q is the proposition $p \wedge (a = |n|)$. Next, let r be the proposition $q \wedge (k = 0) \wedge (x = 0)$. It is easily verified that $q\{S_2\}r$ is true. It can be shown that "$x = mk$ and $k \leq a$" is an invariant for the loop in S_3. Furthermore, it is easy to see that the loop terminates after a iterations, with $k = a$, so $x = ma$ at this point. Because r implies that $x = m \cdot 0$ and $0 \leq a$, the loop invariant is true before the loop is entered. Because the loop terminates with $k = a$, it follows that $r\{S_3\}s$ is true where s is the proposition "$x = ma$ and $a = |n|$." Finally, it can be shown that S_4 is correct with respect to the initial assertion s and final assertion t, where t is the proposition "*product* $= mn$."

Putting all this together, because $p\{S_1\}q$, $q\{S_2\}r$, $r\{S_3\}s$, and $s\{S_4\}t$ are all true, it follows from the rule of composition that $p\{S\}t$ is true. Furthermore, because all four segments terminate, S does terminate. This verifies the correctness of the program. ◀

Exercises

1. Prove that the program segment

```
y := 1
z := x + y
```

is correct with respect to the initial assertion $x = 0$ and the final assertion $z = 1$.

2. Verify that the program segment

```
if x < 0 then x := 0
```

is correct with respect to the initial assertion **T** and the final assertion $x \geq 0$.

3. Verify that the program segment

```
x := 2
z := x + y
**if** y > 0 **then**
  z := z + 1
**else**
  z := 0
```

is correct with respect to the initial assertion $y = 3$ and the final assertion $z = 6$.

4. Verify that the program segment

```
if x < y then
  min := x
else
  min := y
```

is correct with respect to the initial assertion **T** and the final assertion $(x \le y \wedge min = x) \vee (x > y \wedge min = y)$.

***5.** Devise a rule of inference for verification of partial correctness of statements of the form

```
if condition 1 then
  S_1
else if condition 2 then
  S_2
    .
    .
    .
else
  S_n
```

where $S_1, S_2, \ldots, S_n$ are blocks.

6. Use the rule of inference developed in Exercise 5 to verify that the program

```
if x < 0 then
  y := -2|x|/x
else if x > 0 then
  y := 2|x|/x
else if x = 0 then
  y := 2
```

is correct with respect to the initial assertion **T** and the final assertion $y = 2$.

7. Use a loop invariant to prove that the following program segment for computing the nth power, where n is a positive integer, of a real number x is correct.

```
power := 1
i := 1
while i ≤ n
begin
  power := power * x
  i := i + 1
end
```

***8.** Prove that the iterative program for finding f_n given in Section 4.4 is correct.

9. Provide all the details in the proof of correctness given in Example 5.

10. Suppose that both the conditional statement $p_0 \to p_1$ and the program assertion $p_1\{S\}q$ are true. Show that $p_0\{S\}q$ also must be true.

11. Suppose that both the program assertion $p\{S\}q_0$ and the conditional statement $q_0 \to q_1$ are true. Show that $p\{S\}q_1$ also must be true.

12. This program computes quotients and remainders.

```
r := a
q := 0
while r ≥ d
begin
  r := r - d
  q := q + 1
end
```

Verify that it is partially correct with respect to the initial assertion "a and d are positive integers" and the final assertion "q and r are integers such that $a = dq + r$ and $0 \le r < d$."

13. Use a loop invariant to verify that the Euclidean algorithm (Algorithm 6 in Section 3.6) is partially correct with respect to the initial assertion "a and b are positive integers" and the final assertion "$x = \gcd(a, b)$."

Key Terms and Results

TERMS

sequence: a function with domain that is a subset of the set of integers

geometric progression: a sequence of the form $a, ar, ar^2, \ldots,$ where a and r are real numbers

arithmetic progression: a sequence of the form $a, a + d, a + 2d, \ldots,$ where a and d are real numbers

the principle of mathematical induction: The statement $\forall n\, P(n)$ is true if $P(1)$ is true and $\forall k[P(k) \to P(k+1)]$ is true.

basis step: the proof of $P(1)$ in a proof by mathematical induction of $\forall n\, P(n)$

inductive step: the proof of $P(k) \to P(k+1)$ for all positive integers k in a proof by mathematical induction of $\forall n\, P(n)$

strong induction: The statement $\forall n\, P(n)$ is true if $P(1)$ is true and $\forall k[(P(1) \wedge \cdots \wedge P(k)) \to P(k+1)]$ is true.

well-ordering property: Every nonempty set of nonnegative integers has a least element.

recursive definition of a function: a definition of a function that specifies an initial set of values and a rule for obtaining values of this function at integers from its values at smaller integers