

## 6.2 Recursion-Reading

**Notebook:** Discrete Mathematics [CM1020]

**Created:** 2019-10-07 2:31 PM

**Updated:** 2019-12-07 12:32 PM

**Author:** SUKHJIT MANN

Cornell Notes	Topic: 6.2 Recursion-Reading	Course: BSc Computer Science
		Class: Discrete Mathematics-Reading
		Date: December 06, 2019
Essential Question:		
What are recursion, recurrence, and recurrence relations?		
Questions/Cues:		
<ul style="list-style-type: none"><li>• What is recursion?</li><li>• What is a Recursively defined function?</li><li>• What is a recursively defined set?</li><li>• What is a tree?</li><li>• What is recursive algorithm?</li><li>• How is it that we prove a recursive algorithm is correct?</li><li>• What is the process of iteration when dealing with recursive algorithms?</li></ul>		
Notes		
<ul style="list-style-type: none"><li>• Recursion = Sometimes difficult to define object explicitly; it may be easier to define object in terms of itself. The process called recursion.</li></ul>		
<h3><u>Recursively Defined Functions</u></h3>		
<p>We use two steps to define a function with the set of nonnegative integers as its domain:</p>		
<p><b>BASIS STEP:</b> Specify the value of the function at zero.</p>		
<p><b>RECURSIVE STEP:</b> Give a rule for finding its value at an integer from its values at smaller integers.</p>		
<p>Such a definition is called a <b>recursive or inductive definition</b>. Note that a function <math>f(n)</math> from the set of nonnegative integers to the set of a real numbers is the same as a sequence <math>a_0, a_1, \dots</math> where <math>a_i</math> is a real number for every nonnegative integer <math>i</math>. So, defining a real-valued sequence <math>a_0, a_1, \dots</math> using a recurrence relation, as was done in Section 2.4, is the same as defining a function from the set of nonnegative integers to the set of real numbers.</p>		

**EXAMPLE 1** Suppose that  $f$  is defined recursively by

$$\begin{aligned}f(0) &= 3, \\f(n+1) &= 2f(n) + 3.\end{aligned}$$

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

*Solution:* From the recursive definition it follows that

$$\begin{aligned}f(1) &= 2f(0) + 3 = 2 \cdot 3 + 3 = 9, \\f(2) &= 2f(1) + 3 = 2 \cdot 9 + 3 = 21, \\f(3) &= 2f(2) + 3 = 2 \cdot 21 + 3 = 45, \\f(4) &= 2f(3) + 3 = 2 \cdot 45 + 3 = 93.\end{aligned}$$

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.

**EXAMPLE 2** 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  $n = 0, 1, 2, 3, \dots$ , is given. These two equations uniquely define  $a^n$  for all nonnegative integers  $n$ .

**EXAMPLE 3** 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).

Recall from Section 2.4 that the Fibonacci numbers,  $f_0, f_1, f_2, \dots$ , are defined by the equations  $f_0 = 0$ ,  $f_1 = 1$ , and

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

for  $n = 2, 3, 4, \dots$  [We can think of the Fibonacci number  $f_n$  either as the  $n$ th term of the sequence of Fibonacci numbers  $f_0, f_1, \dots$  or as the value at the integer  $n$  of a function  $f(n)$ .]

We can use the recursive definition of the Fibonacci numbers to prove many properties of these numbers. We give one such property in Example 4.

**EXAMPLE 4** Show that whenever  $n \geq 3$ ,  $f_n > \alpha^{n-2}$ , where  $\alpha = (1 + \sqrt{5})/2$ .



**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

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

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  $\alpha$  is a solution of  $x^2 - x - 1 = 0$  (as the quadratic formula shows), it follows that  $\alpha^2 = \alpha + 1$ . Therefore,


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

By the inductive hypothesis, because  $k \geq 4$ , we have

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

Therefore, it follows that

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

Hence,  $P(k+1)$  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.


- Recursively-defined Sets = Also have a basis step & recursive step. In basis step, initial collection of elements specified. In recursive step, rules for forming new elements in the set from elements already known in 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 basis step or generated by applications of the recursive step.

**EXAMPLE 5** Consider the subset  $S$  of the set of integers recursively defined by

**BASIS STEP:**  $3 \in S$ .

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




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 in Example 10 that  $S$  is the set of all positive multiples of 3. 

**EXAMPLE 8** **Well-Formed Formulae in Propositional Logic** We can define the set of well-formed formulae in propositional logic 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  $(p \vee q)$ ,  $(p \rightarrow F)$ ,  $(F \rightarrow q)$ , and  $(q \wedge F)$  are well-formed formulae. A second application of the recursive step shows that  $((p \vee q) \rightarrow (q \wedge F))$ ,  $(q \vee (p \vee q))$ , and  $((p \rightarrow F) \rightarrow T)$  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 9 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 a 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 from the basis step and one or more applications of the recursive step. ◀

We will study trees extensively in Chapter 11. 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 10. We will briefly introduce them here to illustrate how they can be defined recursively.

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.

Basis step



Step 1



Step 2



**FIGURE 2** Building Up Rooted Trees.

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 11 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.

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

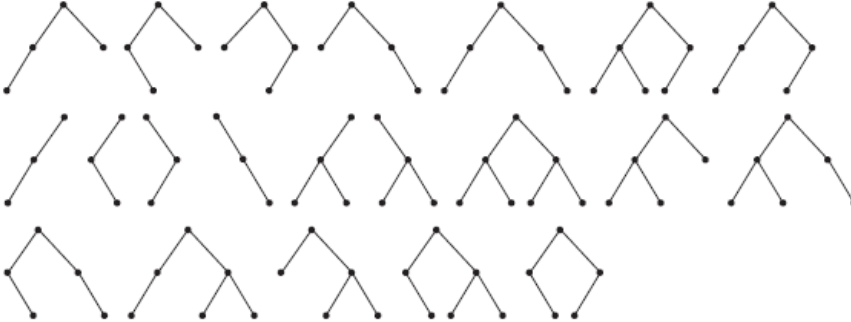
**BASIS STEP:** The empty set is an extended binary tree.

**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.

Basis step  $\emptyset$

Step 1 •

Step 2 

Step 3 

**FIGURE 3** Building Up Extended Binary Trees.


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  $T_2$ .

Basis step •

Step 1 

Step 2 

**FIGURE 4** Building Up Full Binary Trees.

- To prove results about recursively defined sets, we generally use some form of mathematical induction.



**EXAMPLE 10** Show that the set  $S$  defined in Example 5 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.

*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 4.1, it follows that  $3 \mid x + y$ , completing the proof. ◀

In Example 10 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 recursive 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.

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

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



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

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

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

```

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

```

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

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

**ALGORITHM 2** A Recursive Algorithm for Computing  $a^n$ .

```

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

```

**EXAMPLE 4** Devise a recursive algorithm for computing  $b^n \bmod m$ , 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

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

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

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

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

when  $n$  is even and

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

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

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

#### ALGORITHM 4 Recursive Modular Exponentiation.

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

### 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  $\text{power}(a, 0) = 1$ . 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  $\text{power}(a, k) = a^k$  for all  $a \neq 0$  for an arbitrary nonnegative integer  $k$ . That is, the inductive hypothesis is the statement that the algorithm correctly computes  $a^k$ . To complete the inductive step, we show that if the inductive hypothesis is true, then the algorithm correctly computes  $a^{k+1}$ . Because  $k + 1$  is a positive integer, when the algorithm computes  $a^{k+1}$ , the algorithm sets  $\text{power}(a, k + 1) = a \cdot \text{power}(a, k)$ . By the inductive hypothesis, we have  $\text{power}(a, k) = a^k$ , so  $\text{power}(a, k + 1) = a \cdot \text{power}(a, k) = a \cdot a^k = a^{k+1}$ . 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** Prove that Algorithm 4, which computes modular powers, is correct.

*Solution:* We use strong induction on the exponent  $n$ .

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



**INDUCTIVE STEP:** For the inductive hypothesis we assume that  $mpower(b, j, m) = b^j \bmod m$  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  $mpower(b, k, m) = b^k \bmod m$ . 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


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

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

When  $k$  is odd, we have

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

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

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

## Recursion and Iteration

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

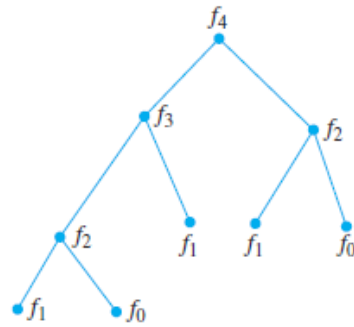
### ALGORITHM 7 A Recursive Algorithm for Fibonacci Numbers.

```

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

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

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



**FIGURE 1** Evaluating  $f_4$  Recursively.

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

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

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

```

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

## Summary

In this week, we learned what recursion is, what is recursively-defined set & function are and what a recursive algorithm is. Also we explored what it means to prove a recursive algorithm is correct and what the iteration process is when comes to dealing with recursive algorithms.

