CHAPTER 2

Iteration, Induction, and Recursion

The power of computers comes from their ability to execute the same task, or different versions of the same task, repeatedly. In computing, the theme of *iteration* is met in a number of guises. Many concepts in data models, such as lists, are forms of repetition, as "A list either is empty or is one element followed by another, then another, and so on." Programs and algorithms use iteration to perform repetitive jobs without requiring a large number of similar steps to be specified individually, as "Do the next step 1000 times." Programming languages use looping constructs, like the while- and for-statements of C, to implement iterative algorithms.

Closely related to repetition is *recursion,* a technique in which a concept is defined, directly or indirectly, in terms of itself. For example, we could have defined a list by saying "A list either is empty or is an element followed by a list." Recursion is supported by many programming languages. In C, a function F can call itself, either directly from within the body of F itself, or indirectly by calling some other function, which calls another, and another, and so on, until finally some function in the sequence calls F. Another important idea, *induction,* is closely related to "recursion" and is used in many mathematical proofs.

Iteration, induction, and recursion are fundamental concepts that appear in many forms in data models, data structures, and algorithms. The following list gives some examples of uses of these concepts; each will be covered in some detail in this book.

1. *Iterative techniques.* The simplest way to perform a sequence of operations repeatedly is to use an iterative construct such as the for-statement of C.

2. *Recursive programming.* C and many other languages permit recursive functions, which call themselves either directly or indirectly. Often, beginning programmers are more secure writing iterative programs than recursive ones, but an important goal of this book is to accustom the reader to thinking and programming recursively, when appropriate. Recursive programs can be simpler to write, analyze, and understand.

Notation: The Summation and Product Symbols

An oversized Greek capital letter sigma is often used to denote a summation, as in $\sum_{i=1}^{n} i$. This particular expression represents the sum of the integers from 1 to n; that is, it stands for the sum $1 + 2 + 3 + \cdots + n$. More generally, we can sum any function $f(i)$ of the summation index i. (Of course, the index could be some symbol other than i.) The expression $\sum_{i=a}^{b} f(i)$ stands for

$$f(a) + f(a+1) + f(a+2) + \cdots + f(b)$$

For example, $\sum_{j=2}^{m} j^2$ stands for the sum $4 + 9 + 16 + \cdots + m^2$. Here, the function f is "squaring," and we used index j instead of i.

As a special case, if $b < a$, then there are no terms in the sum $\sum_{i=a}^{b} f(i)$, and the value of the expression, by convention, is taken to be 0. If $b = a$, then there is exactly one term, that for $i = a$. Thus, the value of the sum $\sum_{i=a}^{a} f(i)$ is just $f(a)$.

The analogous notation for products uses an oversized capital pi. The expression $\prod_{i=a}^{b} f(i)$ stands for the product $f(a) \times f(a+1) \times f(a+2) \times \cdots \times f(b)$; if $b < a$, the product is taken to be 1.

3. *Proofs by induction.* An important technique for showing that a statement is true is "proof by induction." We shall cover inductive proofs extensively, starting in Section 2.3. The following is the simplest form of an inductive proof. We begin with a statement $S(n)$ involving a variable n; we wish to prove that $S(n)$ is true. We prove $S(n)$ by first proving a *basis,* that is, the statement $S(n)$ for a particular value of n. For example, we could let $n = 0$ and prove the statement $S(0)$. Second, we must prove an *inductive step,* in which we prove that the statement S, for one value of its argument, follows from the same statement S for the previous values of its argument; that is, $S(n)$ implies $S(n+1)$ for all $n \geq 0$. For example, $S(n)$ might be the familiar summation formula

Basis

Inductive step

$$\sum_{i=1}^{n} i = n(n+1)/2 \tag{2.1}$$

which says that the sum of the integers from 1 to n equals $n(n+1)/2$. The basis could be $S(1)$ — that is, Equation (2.1) with 1 in place of n — which is just the equality $1 = 1 \times 2/2$. The inductive step is to show that $\sum_{i=1}^{n} i = n(n+1)/2$ implies that $\sum_{i=1}^{n+1} i = (n+1)(n+2)/2$; the former is $S(n)$, which is Equation (2.1) itself, while the latter is $S(n+1)$, which is Equation (2.1) with $n+1$ replacing n everywhere n appears. Section 2.3 will show us how to construct proofs such as this.

4. *Proofs of program correctness.* In computer science, we often wish to prove, formally or informally, that a statement $S(n)$ about a program is true. The statement $S(n)$ might, for example, describe what is true on the nth iteration of some loop or what is true for the nth recursive call to some function. Proofs of this sort are generally inductive proofs.

5. *Inductive definitions.* Many important concepts of computer science, especially those involving data models, are best defined by an induction in which we give

a basis rule defining the simplest example or examples of the concept, and an inductive rule or rules, where we build larger instances of the concept from smaller ones. For instance, we noted that a list can be defined by a basis rule (an empty list is a list) together with an inductive rule (an element followed by a list is also a list).

6. *Analysis of running time.* An important criterion for the "goodness" of an algorithm is how long it takes to run on inputs of various sizes (its "running time"). When the algorithm involves recursion, we use a formula called a *recurrence equation,* which is an inductive definition that predicts how long the algorithm takes to run on inputs of different sizes.

Each of these subjects, except the last, is introduced in this chapter; the running time of a program is the topic of Chapter 3.

2.1 What This Chapter Is About

In this chapter we meet the following major concepts.

- Iterative programming (Section 2.2)
- Inductive proofs (Sections 2.3 and 2.4)
- Inductive definitions (Section 2.6)
- Recursive programming (Sections 2.7 and 2.8)
- Proving the correctness of a program (Sections 2.5 and 2.9)

In addition, we spotlight, through examples of these concepts, several interesting and important ideas from computer science. Among these are

- Sorting algorithms, including selection sort (Section 2.2) and merge sort (Section 2.8)
- Parity checking and detection of errors in data (Section 2.3)
- Arithmetic expressions and their transformation using algebraic laws (Sections 2.4 and 2.6)
- Balanced parentheses (Section 2.6)

2.2 Iteration

Each beginning programmer learns to use iteration, employing some kind of looping construct such as the for- or while-statement of C. In this section, we present an example of an iterative algorithm, called "selection sort." In Section 2.5 we shall prove by induction that this algorithm does indeed sort, and we shall analyze its running time in Section 3.6. In Section 2.8, we shall show how recursion can help us devise a more efficient sorting algorithm using a technique called "divide and conquer."

Common Themes: Self-Definition and Basis-Induction

As you study this chapter, you should be alert to two themes that run through the various concepts. The first is self-definition, in which a concept is defined, or built, in terms of itself. For example, we mentioned that a list can be defined as being empty or as being an element followed by a list.

The second theme is basis-induction. Recursive functions usually have some sort of test for a "basis" case where no recursive calls are made and an "inductive" case where one or more recursive calls are made. Inductive proofs are well known to consist of a basis and an inductive step, as do inductive definitions. This basis-induction pairing is so important that these words are highlighted in the text to introduce each occurrence of a basis case or an inductive step.

There is no paradox or circularity involved in properly used self-definition, because the self-defined subparts are always "smaller" than the object being defined. Further, after a finite number of steps to smaller parts, we arrive at the basis case, at which the self-definition ends. For example, a list L is built from an element and a list that is one element shorter than L. When we reach a list with zero elements, we have the basis case of the definition of a list: "The empty list is a list."

As another example, if a recursive function works, the arguments of the call must, in some sense, be "smaller" than the arguments of the calling copy of the function. Moreover, after some number of recursive calls, we must get to arguments that are so "small" that the function does not make any more recursive calls.

Sorting

To sort a list of n elements we need to permute the elements of the list so that they appear in nondecreasing order.

✦ **Example 2.1.** Suppose we are given the list of integers $(3, 1, 4, 1, 5, 9, 2, 6, 5)$. We sort this list by permuting it into the sequence $(1, 1, 2, 3, 4, 5, 5, 6, 9)$. Note that sorting not only orders the values so that each is either less than or equal to the one that follows, but it also preserves the number of occurrences of each value. Thus, the sorted list has two 1's, two 5's, and one each of the numbers that appear once in the original list. ✦

We can sort a list of elements of any type as long as the elements have a "less-than" order defined on them, which we usually represent by the symbol $<$. For example, if the values are real numbers or integers, then the symbol $<$ stands for the usual less-than relation on reals or integers, and if the values are character strings, we would use the lexicographic order on strings. (See the box on "Lexicographic Order.") Sometimes when the elements are complex, such as structures, we might use only a part of each element, such as one particular field, for the comparison.

Sorted list

The comparison $a \leq b$ means, as always, that either $a < b$ or a and b are the same value. A list $(a_1, a_2, \ldots, a_n)$ is said to be *sorted* if $a_1 \leq a_2 \leq \cdots \leq a_n$; that is, if the values are in nondecreasing order. *Sorting* is the operation of taking an arbitrary list $(a_1, a_2, \ldots, a_n)$ and producing a list $(b_1, b_2, \ldots, b_n)$ such that

Lexicographic Order

The usual way in which two character strings are compared is according to their *lexicographic order.* Let $c_1c_2\cdots c_k$ and $d_1d_2\cdots d_m$ be two strings, where each of the c's and d's represents a single character. The lengths of the strings, k and m, need not be the same. We assume that there is a $<$ ordering on characters; for example, in C characters are small integers, so character constants and variables can be used as integers in arithmetic expressions. Thus we can use the conventional $<$ relation on integers to tell which of two characters is "less than" the other. This ordering includes the natural notion that lower-case letters appearing earlier in the alphabet are "less than" lower-case letters appearing later in the alphabet, and the same holds for upper-case letters.

We may then define the ordering on character strings called the *lexicographic, dictionary,* or *alphabetic* ordering, as follows. We say $c_1c_2\cdots c_k < d_1d_2\cdots d_m$ if either of the following holds:

Proper prefix

1. The first string is a *proper prefix* of the second, which means that $k < m$ and for $i = 1, 2, \ldots, k$ we have $c_i = d_i$. According to this rule, `bat` < `batter`. As a special case of this rule, we could have $k = 0$, in which case the first string has no characters in it. We shall use ϵ, the Greek letter epsilon, to denote the *empty string,* the string with zero characters. When $k = 0$, rule (1) says that $\epsilon < s$ for any nonempty string s.

Empty string

2. For some value of $i > 0$, the first $i - 1$ characters of the two strings agree, but the ith character of the first string is less than the ith character of the second string. That is, $c_j = d_j$ for $j = 1, 2, \ldots, i - 1$, and $c_i < d_i$. According to this rule, `ball` < `base`, because the two words first differ at position 3, and at that position `ball` has an `l`, which precedes the character `s` found in the third position of `base`.

---

1. List $(b_1, b_2, \ldots, b_n)$ is sorted.

Permutation

2. List $(b_1, b_2, \ldots, b_n)$ is a *permutation* of the original list. That is, each value appears in list $(a_1, a_2, \ldots, a_n)$ exactly as many times as that value appears in list $(b_1, b_2, \ldots, b_n)$.

A *sorting algorithm* takes as input an arbitrary list and produces as output a sorted list that is a permutation of the input.

◆ **Example 2.2.** Consider the list of words

```
base, ball, mound, bat, glove, batter
```

Given this input, and using lexicographic order, a sorting algorithm would produce this output: `ball, base, bat, batter, glove, mound`. ◆

Selection Sort: An Iterative Sorting Algorithm

Suppose we have an array `A` of n integers that we wish to sort into nondecreasing

Convention Regarding Names and Values

We can think of a variable as a box with a name and a value. When we refer to a variable, such as `abc`, we use the constant-width, or "computer" font for its name, as we did in this sentence. When we refer to the value of the variable `abc`, we shall use italics, as *abc*. To summarize, `abc` refers to the name of the box, and *abc* to its contents.

order. We may do so by iterating a step in which a smallest element[1] not yet part of the sorted portion of the array is found and exchanged with the element in the first position of the unsorted part of the array. In the first iteration, we find ("select") a smallest element among the values found in the full array `A[0..n-1]` and exchange it with `A[0]`.[2] In the second iteration, we find a smallest element in `A[1..n-1]` and exchange it with `A[1]`. We continue these iterations. At the start of the $i+1$st iteration, `A[0..i-1]` contains the i smallest elements in `A` sorted in nondecreasing order, and the remaining elements of the array are in no particular order. A picture of `A` just before the $i+1$st iteration is shown in Fig. 2.1.

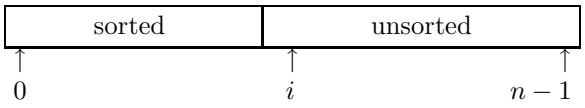


Fig. 2.1. Picture of array just before the $i+1$st iteration of selection sort.

In the $i+1$st iteration, we find a smallest element in `A[i..n-1]` and exchange it with `A[i]`. Thus, after the $i+1$st iteration, `A[0..i]` contains the $i+1$ smallest elements sorted in nondecreasing order. After the $(n-1)$st iteration, the entire array is sorted.

A C function for selection sort is shown in Fig. 2.2. This function, whose name is `SelectionSort`, takes an array `A` as the first argument. The second argument, n, is the length of array `A`.

Lines (2) through (5) select a smallest element in the unsorted part of the array, `A[i..n-1]`. We begin by setting the value of index `small` to i in line (2). The for-loop of lines (3) through (5) consider all higher indexes j in turn, and *small* is set to j if `A[j]` has a smaller value than any of the array elements in the range `A[i..j-1]`. As a result, we set the variable `small` to the index of the first occurrence of the smallest element in `A[i..n-1]`.

After choosing a value for the index `small`, we exchange the element in that position with the element in `A[i]`, in lines (6) to (8). If $small = i$, the exchange is performed, but has no effect on the array. Notice that in order to swap two elements, we need a temporary place to store one of them. Thus, we move the value

[1] We say "*a* smallest element" rather than "*the* smallest element" because there may be several occurrences of the smallest value. If so, we shall be happy with any of those occurrences.

[2] To describe a range of elements within an array, we adopt a convention from the language Pascal. If A is an array, then `A[i..j]` denotes those elements of A with indexes from i to j, inclusive.

```
     void SelectionSort(int A[], int n)
     {
         int i, j, small, temp;
(1)      for (i = 0; i < n-1; i++) {
             /* set small to the index of the first occur- */
             /* rence of the smallest element remaining */
(2)          small = i;
(3)          for (j = i+1; j < n; j++)
(4)              if (A[j] < A[small])
(5)                  small = j;
             /* when we reach here, small is the index of */
             /* the first smallest element in A[i..n-1]; */
             /* we now exchange A[small] with A[i] */
(6)          temp = A[small];
(7)          A[small] = A[i];
(8)          A[i] = temp;
         }
     }
```

Fig. 2.2. Iterative selection sort.

in `A[small]` to `temp` at line (6), move the value in `A[i]` to `A[small]` at line (7), and finally move the value originally in `A[small]` from `temp` to `A[i]` at line (8).

◆ **Example 2.3.** Let us study the behavior of `SelectionSort` on various inputs. First, let us look at what happens when we run `SelectionSort` on an array with no elements. When $n = 0$, the body of the for-loop of line (1) is not executed, so `SelectionSort` does "nothing" gracefully.

Now let us consider the case in which the array has only one element. Again, the body of the for-loop of line (1) is not executed. That response is satisfactory, because an array consisting of a single element is always sorted. The cases in which n is 0 or 1 are important boundary conditions, on which it is important to check the performance of any algorithm or program.

Finally, let us run `SelectionSort` on a small array with four elements, where `A[0]` through `A[3]` are

	0	1	2	3
A	40	30	20	10

We begin the outer loop with $i = 0$, and at line (2) we set `small` to 0. Lines (3) to (5) form an inner loop, in which `j` is set to 1, 2, and 3, in turn. With $j = 1$, the test of line (4) succeeds, since $A[1]$, which is 30, is less than $A[small]$, which is $A[0]$, or 40. Thus, we set `small` to 1 at line (5). At the second iteration of lines (3) to (5), with $j = 2$, the test of line (4) again succeeds, since $A[2] < A[1]$, and so we set `small` to 2 at line (5). At the last iteration of lines (3) to (5), with $j = 3$, the test of line (4) succeeds, since $A[3] < A[2]$, and we set `small` to 3 at line (5).

We now fall out of the inner loop to line (6). We set `temp` to 10, which is $A[small]$, then `A[3]` to $A[0]$, or 40, at line (7), and `A[0]` to 10 at line (8). Now, the

Sorting on Keys

When we sort, we apply a comparison operation to the values being sorted. Often the comparison is made only on specific parts of the values and the part used in the comparison is called the *key.*

For example, a course roster might be an array `A` of C structures of the form

```
struct STUDENT {
    int studentID;
    char *name;
    char grade;
} A[MAX];
```

We might want to sort by student ID, or name, or grade; each in turn would be the key. For example, if we wish to sort structures by student ID, we would use the comparison

```
A[j].studentID < A[small].studentID
```

at line (4) of `SelectionSort`. The type of array `A` and temporary `temp` used in the swap would be `struct STUDENT`, rather than `integer`. Note that entire structures are swapped, not just the key fields.

Since it is time-consuming to swap whole structures, a more efficient approach is to use a second array of pointers to `STUDENT` structures and sort only the pointers in the second array. The structures themselves remain stationary in the first array. We leave this version of selection sort as an exercise.

first iteration of the outer loop is complete, and array `A` appears as

	0	1	2	3
A	10	30	20	40

The second iteration of the outer loop, with $i = 1$, sets `small` to 1 at line (2). The inner loop sets `j` to 2 initially, and since $A[2] < A[1]$, line (5) sets `small` to 2. With $j = 3$, the test of line (4) fails, since $A[3] \geq A[2]$. Hence, $small = 2$ when we reach line (6). Lines (6) to (8) swap `A[1]` with `A[2]`, leaving the array

	0	1	2	3
A	10	20	30	40

Although the array now happens to be sorted, we still iterate the outer loop once more, with $i = 2$. We set `small` to 2 at line (2), and the inner loop is executed only with $j = 3$. Since the test of line (4) fails, *small* remains 2, and at lines (6) through (8), we "swap" `A[2]` with itself. The reader should check that the swapping has no effect when $small = i$. ❖

Figure 2.3 shows how the function `SelectionSort` can be used in a complete program to sort a sequence of n integers, provided that $n \leq 100$. Line (1) reads and stores n integers in an array `A`. If the number of inputs exceeds `MAX`, only the first `MAX` integers are put into `A`. A message warning the user that the number of inputs is too large would be useful here, but we omit it.

Line (3) calls `SelectionSort` to sort the array. Lines (4) and (5) print the integers in sorted order.

```
      #include <stdio.h>

      #define MAX 100
      int A[MAX];
      void SelectionSort(int A[], int n);

      main()
      {
          int i, n;
          /* read and store input in A */
(1)       for (n = 0; n < MAX && scanf("%d", &A[n]) != EOF; n++)
(2)           ;
(3)       SelectionSort(A,n); /* sort A */
(4)       for (i = 0; i < n; i++)
(5)           printf("%d\n", A[i]); /* print A */
          }

      void SelectionSort(int A[], int n)
      {
          int i, j, small, temp;
          for (i = 0; i < n-1; i++) {
              small = i;
              for (j = i+1; j < n; j++)
                      if (A[j] < A[small])
                          small = j;
              temp = A[small];
              A[small] = A[i];
              A[i] = temp;
          }
      }
```

Fig. 2.3. A sorting program using selection sort.

EXERCISES

2.2.1: Simulate the function `SelectionSort` on an array containing the elements

a) 6, 8, 14, 17, 23
b) 17, 23, 14, 6, 8
c) 23, 17, 14, 8, 6

How many comparisons and swaps of elements are made in each case?

2.2.2**: What are the minimum and maximum number of (a) comparisons and (b) swaps that `SelectionSort` can make in sorting a sequence of n elements?

2.2.3: Write a C function that takes two linked lists of characters as arguments and returns `TRUE` if the first string precedes the second in lexicographic order. *Hint*: Implement the algorithm for comparing character strings that was described in this section. Use recursion by having the function call itself on the tails of the character strings when it finds that the first characters of both strings are the same. Alternatively, one can develop an iterative algorithm to do the same.

2.2.4*: Modify your program from Exercise 2.2.3 to ignore the case of letters in comparisons.

2.2.5: What does selection sort do if all elements are the same?

2.2.6: Modify Fig. 2.3 to perform selection sort when array elements are not integers, but rather structures of type `struct STUDENT`, as defined in the box "Sorting on Keys." Suppose that the key field is `studentID`.

2.2.7*: Further modify Fig. 2.3 so that it sorts elements of an arbitrary type T. You may assume, however, that there is a function *key* that takes an element of type T as argument and returns the key for that element, of some arbitrary type K. Also assume that there is a function *lt* that takes two elements of type K as arguments and returns `TRUE` if the first is "less than" the second, and `FALSE` otherwise.

2.2.8: Instead of using integer indexes into the array `A`, we could use pointers to integers to indicate positions in the array. Rewrite the selection sort algorithm of Fig. 2.3 using pointers.

2.2.9*: As mentioned in the box on "Sorting on Keys," if the elements to be sorted are large structures such as type `STUDENT`, it makes sense to leave them stationary in an array and sort pointers to these structures, found in a second array. Write this variation of selection sort.

2.2.10: Write an iterative program to print the distinct elements of an integer array.

2.2.11: Use the $\sum$ and $\prod$ notations described at the beginning of this chapter to express the following.

a) The sum of the odd integers from 1 to 377

b) The sum of the squares of the even integers from 2 to n (assume that n is even)

c) The product of the powers of 2 from 8 to 2^k

2.2.12: Show that when $small = i$, lines (6) through (8) of Fig. 2.2 (the swapping steps) do not have any effect on array `A`.

2.3 Inductive Proofs

Mathematical induction is a useful technique for proving that a statement $S(n)$ is true for all nonnegative integers n, or, more generally, for all integers at or above some lower limit. For example, in the introduction to this chapter we suggested that the statement $\sum_{i=1}^{n} i = n(n+1)/2$ can be proved true for all $n \geq 1$ by an induction on n.

Now, let $S(n)$ be some arbitrary statement about an integer n. In the simplest form of an inductive proof of the statement $S(n)$, we prove two facts:

Naming the Induction Parameter

It is often useful to explain an induction by giving the intuitive meaning of the variable n in the statement $S(n)$ that we are proving. If n has no special meaning, as in Example 2.4, we simply say "The proof is by induction on n." In other cases, n may have a physical meaning, as in Example 2.6, where n is the number of bits in the code words. There we can say, "The proof is by induction on the number of bits in the code words."

1. The *basis* case, which is frequently taken to be $S(0)$. However, the basis can be $S(k)$ for any integer k, with the understanding that then the statement $S(n)$ is proved only for $n \geq k$.

2. The *inductive step,* where we prove that for all $n \geq 0$ [or for all $n \geq k$, if the basis is $S(k)$], $S(n)$ implies $S(n+1)$. In this part of the proof, we assume that the statement $S(n)$ is true. $S(n)$ is called the *inductive hypothesis,* and assuming it to be true, we must then prove that $S(n+1)$ is true.

Inductive hypothesis

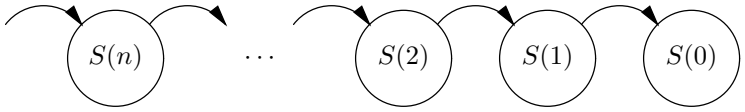


Fig. 2.4. In an inductive proof, each instance of the statement $S(n)$ is proved using the statement for the next lower value of n.

Figure 2.4 illustrates an induction starting at 0. For each integer n, there is a statement $S(n)$ to prove. The proof for $S(1)$ uses $S(0)$, the proof for $S(2)$ uses $S(1)$, and so on, as represented by the arrows. The way each statement depends on the previous one is uniform. That is, *by one proof of the inductive step, we prove each of the steps implied by the arrows in Fig. 2.4.*

◆ **Example 2.4.** As an example of mathematical induction, let us prove

STATEMENT $S(n)$: $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$ for any $n \geq 0$.

That is, the sum of the powers of 2, from the 0th power to the nth power, is 1 less than the $(n+1)$st power of 2.[3] For example, $1 + 2 + 4 + 8 = 16 - 1$. The proof proceeds as follows.

BASIS. To prove the basis, we substitute 0 for n in the equation $S(n)$. Then $S(n)$ becomes

[3] $S(n)$ can be proved without induction, using the formula for the sum of a geometric series. However, it will serve as a simple example of the technique of mathematical induction. Further, the proofs of the formulas for the sum of a geometric or arithmetic series that you have probably seen in high school are rather informal, and strictly speaking, mathematical induction should be used to prove those formulas.

$$\sum_{i=0}^{0} 2^i = 2^1 - 1 \tag{2.2}$$

There is only one term, for $i = 0$, in the summation on the left side of Equation (2.2), so that the left side of (2.2) sums to 2^0, or 1. The right side of Equation (2.2), which is $2^1 - 1$, or $2 - 1$, also has value 1. Thus we have proved the basis of $S(n)$; that is, we have shown that this equality is true for $n = 0$.

INDUCTION. Now we must prove the inductive step. We assume that $S(n)$ is true, and we prove the same equality with $n + 1$ substituted for n. The equation to be proved, $S(n + 1)$, is

$$\sum_{i=0}^{n+1} 2^i = 2^{n+2} - 1 \tag{2.3}$$

To prove Equation (2.3), we begin by considering the sum on the left side,

$$\sum_{i=0}^{n+1} 2^i$$

This sum is almost the same as the sum on the left side of $S(n)$, which is

$$\sum_{i=0}^{n} 2^i$$

except that (2.3) also has a term for $i = n + 1$, that is, the term 2^{n+1}.

Since we are allowed to assume that the inductive hypothesis $S(n)$ is true in our proof of Equation (2.3), we should contrive to use $S(n)$ to advantage. We do so by breaking the sum in (2.3) into two parts, one of which is the sum in $S(n)$. That is, we separate out the last term, where $i = n + 1$, and write

$$\sum_{i=0}^{n+1} 2^i = \sum_{i=0}^{n} 2^i + 2^{n+1} \tag{2.4}$$

Now we can make use of $S(n)$ by substituting its right side, $2^{n+1} - 1$, for $\sum_{i=0}^{n} 2^i$ in Equation (2.4):

$$\sum_{i=0}^{n+1} 2^i = 2^{n+1} - 1 + 2^{n+1} \tag{2.5}$$

When we simplify the right side of Equation (2.5), it becomes $2 \times 2^{n+1} - 1$, or $2^{n+2} - 1$. Now we see that the summation on the left side of (2.5) is the same as the left side of (2.3), and the right side of (2.5) is equal to the right side of (2.3). We have thus proved the validity of Equation (2.3) by using the equality $S(n)$; that proof is the inductive step. The conclusion we draw is that $S(n)$ holds for every nonnegative value of n. ✥

Why Does Proof by Induction Work?

In an inductive proof, we first prove that $S(0)$ is true. Next we show that if $S(n)$ is true, then $S(n + 1)$ holds. But why can we then conclude that $S(n)$ is true for all $n \geq 0$? We shall offer two "proofs." A mathematician would point out that

Substituting for Variables

People are often confused when they have to substitute for a variable such as n in $S(n)$, an expression involving the same variable. For example, we substituted $n+1$ for n in $S(n)$ to get Equation (2.3). To make the substitution, we must first mark every occurrence of n in S. One useful way to do so is to replace n by some new variable — say m — that does not otherwise appear in S. For example, $S(n)$ would become

$$\sum_{i=0}^{m} 2^i = 2^{m+1} - 1$$

We then literally substitute the desired expression, $n + 1$ in this case, for each occurrence of m. That gives us

$$\sum_{i=0}^{n+1} 2^i = 2^{(n+1)+1} - 1$$

When we simplify $(n + 1) + 1$ to $n + 2$, we have (2.3).

Note that we should put parentheses around the expression substituted, to avoid accidentally changing the order of operations. For example, had we substituted $n + 1$ for m in the expression $2 \times m$, and not placed the parentheses around $n+1$, we would have gotten $2 \times n + 1$, rather than the correct expression $2 \times (n+1)$, which equals $2 \times n + 2$.

each of our "proofs" that induction works requires an inductive proof itself, and therefore is no proof at all. Technically, induction must be accepted as axiomatic. Nevertheless, many people find the following intuition useful.

In what follows, we assume that the basis value is $n = 0$. That is, we know that $S(0)$ is true and that for all n greater than 0, if $S(n)$ is true, then $S(n + 1)$ is true. Similar arguments work if the basis value is any other integer.

First "proof": *Iteration of the inductive step.* Suppose we want to show that $S(a)$ is true for a particular nonnegative integer a. If $a = 0$, we just invoke the truth of the basis, $S(0)$. If $a > 0$, then we argue as follows. We know that $S(0)$ is true, from the basis. The statement "$S(n)$ implies $S(n + 1)$," with 0 in place of n, says "$S(0)$ implies $S(1)$." Since we know that $S(0)$ is true, we now know that $S(1)$ is true. Similarly, if we substitute 1 for n, we get "$S(1)$ implies $S(2)$," and so we also know that $S(2)$ is true. Substituting 2 for n, we have "$S(2)$ implies $S(3)$," so that $S(3)$ is true, and so on. No matter what the value of a is, we eventually get to $S(a)$, and we are done.

Second "proof": *Least counterexample.* Suppose $S(n)$ were not true for at least one value of n. Let a be the least nonnegative integer for which $S(a)$ is false. If $a = 0$, then we contradict the basis, $S(0)$, and so a must be greater than 0. But if $a > 0$, and a is the least nonnegative integer for which $S(a)$ is false, then $S(a - 1)$ must be true. Now, the inductive step, with n replaced by $a-1$, tells us that $S(a-1)$ implies $S(a)$. Since $S(a-1)$ is true, $S(a)$ must be true, another contradiction. Since we assumed there were nonnegative values of n for which $S(n)$ is false and derived a contradiction, $S(n)$ must therefore be true for any $n \geq 0$.

Error-Detecting Codes

We shall now begin an extended example of "error-detecting codes," a concept that is interesting in its own right and also leads to an interesting inductive proof. When we transmit information over a data network, we code characters (letters, digits, punctuation, and so on) into strings of bits, that is, 0's and 1's. For the moment let us assume that characters are represented by seven bits. However, it is normal to transmit more than seven bits per character, and an eighth bit can be used to help detect some simple errors in transmission. That is, occasionally, one of the 0's or 1's gets changed because of noise during transmission, and is received as the opposite bit; a 0 entering the transmission line emerges as a 1, or vice versa. It is useful if the communication system can tell when one of the eight bits has been changed, so that it can signal for a retransmission.

To detect changes in a single bit, we must be sure that no two characters are represented by sequences of bits that differ in only one position. For then, if that position were changed, the result would be the code for the other character, and we could not detect that an error had occurred. For example, if the code for one character is the sequence of bits 01010101, and the code for another is 01000101, then a change in the fourth position from the left turns the former into the latter.

Parity bit

One way to be sure that no characters have codes that differ in only one position is to precede the conventional 7-bit code for the character by a *parity bit.* If the total number of 1's in a group of bits is odd, the group is said to have *odd parity.* If the number of 1's in the group is even, then the group has *even parity.* The coding scheme we select is to represent each character by an 8-bit code with even parity; we could as well have chosen to use only the codes with odd parity. We force the parity to be even by selecting the parity bit judiciously.

ASCII

✦ **Example 2.5.** The conventional ASCII (pronounced "ask-ee"; it stands for "American Standard Code for Information Interchange") 7-bit code for the character `A` is 1000001. That sequence of seven bits already has an even number of 1's, and so we prefix it by 0 to get 01000001. The conventional code for `C` is 1000011, which differs from the 7-bit code for `A` only in the sixth position. However, this code has odd parity, and so we prefix a 1 to it, yielding the 8-bit code 11000011 with even parity. Note that after prefixing the parity bits to the codes for `A` and `C`, we have 01000001 and 11000011, which differ in two positions, namely the first and seventh, as seen in Fig. 2.5. ✦

`A`: 0 1 0 0 0 0 0 1

`C`: 1 1 0 0 0 0 1 1

Fig. 2.5. We can choose the initial parity bit so the 8-bit code always has even parity.

We can always pick a parity bit to attach to a 7-bit code so that the number of 1's in the 8-bit code is even. We pick parity bit 0 if the 7-bit code for the character at hand has even parity, and we pick parity bit 1 if the 7-bit code has odd parity. In either case, the number of 1's in the 8-bit code is even.

No two sequences of bits that each have even parity can differ in only one position. For if two such bit sequences differ in exactly one position, then one has exactly one more 1 than the other. Thus, one sequence must have odd parity and the other even parity, contradicting our assumption that both have even parity. We conclude that addition of a parity bit to make the number of 1's even serves to create an error-detecting code for characters.

The parity-bit scheme is quite "efficient," in the sense that it allows us to transmit many different characters. Note that there are 2^n different sequences of n bits, since we may choose either of two values (0 or 1) for the first position, either of two values for the second position, and so on, a total of $2 \times 2 \times \cdots \times 2$ (n factors) possible strings. Thus, we might expect to be able to represent up to $2^8 = 256$ characters with eight bits.

However, with the parity scheme, we can choose only seven of the bits; the eighth is then forced upon us. We can thus represent up to 2^7, or 128 characters, and still detect single errors. That is not so bad; we can use 128/256, or half, of the possible 8-bit codes as legal codes for characters, and still detect an error in one bit.

Similarly, if we use sequences of n bits, choosing one of them to be the parity bit, we can represent 2^{n-1} characters by taking sequences of $n-1$ bits and prefixing the suitable parity bit, whose value is determined by the other $n-1$ bits. Since there are 2^n sequences of n bits, we can represent $2^{n-1}/2^n$, or half the possible number of characters, and still detect an error in any one of the bits of a sequence.

Is it possible to detect errors and use more than half the possible sequences of bits as legal codes? Our next example tells us we cannot. The inductive proof uses a statement that is not true for 0, and for which we must choose a larger basis, namely 1.

◆ **Example 2.6.** We shall prove the following by induction on n.

Error-detecting code

STATEMENT $S(n)$: If C is any set of bit strings of length n that is *error detecting* (i.e., if there are no two strings that differ in exactly one position), then C contains at most 2^{n-1} strings.

This statement is not true for $n = 0$. $S(0)$ says that any error-detecting set of strings of length 0 has at most 2^{-1} strings, that is, half a string. Technically, the set C consisting of only the empty string (string with no positions) is an error-detecting set of length 0, since there are no two strings in C that differ in only one position. Set C has more than half a string; it has one string to be exact. Thus, $S(0)$ is false. However, for all $n \geq 1$, $S(n)$ is true, as we shall see.

BASIS. The basis is $S(1)$; that is, any error-detecting set of strings of length one has at most $2^{1-1} = 2^0 = 1$ string. There are only two bit strings of length one, the string 0 and the string 1. However, we cannot have both of them in an error-detecting set, because they differ in exactly one position. Thus, every error-detecting set for $n = 1$ must have at most one string.

INDUCTION. Let $n \geq 1$, and assume that the inductive hypothesis — an error-detecting set of strings of length n has at most 2^{n-1} strings — is true. We must

show, using this assumption, that any error-detecting set C of strings with length $n+1$ has at most 2^n strings. Thus, divide C into two sets, C_0, the set of strings in C that begin with 0, and C_1, the set of strings in C that begin with 1. For instance, suppose $n = 2$ and C is the code with strings of length $n+1 = 3$ constructed using a parity bit. Then, as shown in Fig. 2.6, C consists of the strings 000, 101, 110, and 011; C_0 consists of the strings 000 and 011, and C_1 has the other two strings, 101 and 110.

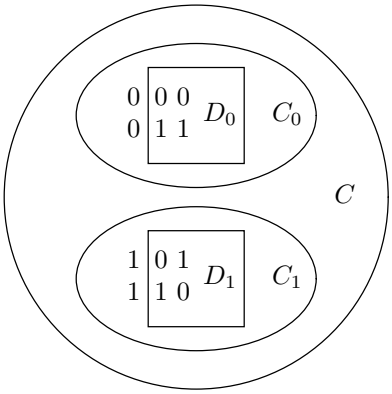


Fig. 2.6. The set C is split into C_0, the strings beginning with 0, and C_1, the strings beginning with 1. D_0 and D_1 are formed by deleting the leading 0's and 1's, respectively.

Consider the set D_0 consisting of those strings in C_0 with the leading 0 removed. In our example above, D_0 contains the strings 00 and 11. We claim that D_0 cannot have two strings differing in only one bit. The reason is that if there are two such strings — say $a_1a_2 \cdots a_n$ and $b_1b_2 \cdots b_n$ — then restoring their leading 0's gives us two strings in C_0, $0a_1a_2 \cdots a_n$ and $0b_1b_2 \cdots b_n$, and these strings would differ in only one position as well. But strings in C_0 are also in C, and we know that C does not have two strings that differ in only one position. Thus, neither does D_0, and so D_0 is an error detecting set.

Now we can apply the inductive hypothesis to conclude that D_0, being an error-detecting set with strings of length n, has at most 2^{n-1} strings. Thus, C_0 has at most 2^{n-1} strings.

We can reason similarly about the set C_1. Let D_1 be the set of strings in C_1, with their leading 1's deleted. D_1 is an error-detecting set with strings of length n, and by the inductive hypothesis, D_1 has at most 2^{n-1} strings. Thus, C_1 has at most 2^{n-1} strings. However, every string in C is in either C_0 or C_1. Therefore, C has at most $2^{n-1} + 2^{n-1}$, or 2^n strings.

We have proved that $S(n)$ implies $S(n+1)$, and so we may conclude that $S(n)$ is true for all $n \geq 1$. We exclude $n = 0$ from the claim, because the basis is $n = 1$, not $n = 0$. We now see that the error-detecting sets constructed by parity check are as large as possible, since they have exactly 2^{n-1} strings when strings of n bits are used. ✦

How to Invent Inductive Proofs

There is no "crank to turn" that is guaranteed to give you an inductive proof of any (true) statement $S(n)$. Finding inductive proofs, like finding proofs of any kind, or like writing programs that work, is a task with intellectual challenge, and we can only offer a few words of advice. If you examine the inductive steps in Examples 2.4 and 2.6, you will notice that in each case we had to rework the statement $S(n+1)$ that we were trying to prove so that it incorporated the inductive hypothesis, $S(n)$, plus something extra. In Example 2.4, we expressed the sum

$$1 + 2 + 4 + \cdots + 2^n + 2^{n+1}$$

as the sum

$$1 + 2 + 4 + \cdots + 2^n$$

which the inductive hypothesis tells us something about, plus the extra term, 2^{n+1}.

In Example 2.6, we expressed the set C, with strings of length $n+1$, in terms of two sets of strings (which we called D_0 and D_1) of length n, so that we could apply the inductive hypothesis to these sets and conclude that both of these sets were of limited size.

Of course, working with the statement $S(n+1)$ so that we can apply the inductive hypothesis is just a special case of the more universal problem-solving adage "Use what is given." The hard part always comes when we must deal with the "extra" part of $S(n+1)$ and complete the proof of $S(n+1)$ from $S(n)$. However, the following is a universal rule:

✦ An inductive proof must at some point say "$\cdots$and by the inductive hypothesis we know that$\cdots$." If it doesn't, then it isn't an inductive proof.

EXERCISES

2.3.1: Show the following formulas by induction on n starting at $n = 1$.

a) $\sum_{i=1}^{n} i = n(n+1)/2$.

b) $\sum_{i=1}^{n} i^2 = n(n+1)(2n+1)/6$.

c) $\sum_{i=1}^{n} i^3 = n^2(n+1)^2/4$.

d) $\sum_{i=1}^{n} 1/i(i+1) = n/(n+1)$.

Triangular number

2.3.2: Numbers of the form $t_n = n(n+1)/2$ are called *triangular numbers,* because marbles arranged in an equilateral triangle, n on a side, will total $\sum_{i=1}^{n} i$ marbles, which we saw in Exercise 2.3.1(a) is t_n marbles. For example, bowling pins are arranged in a triangle 4 on a side and there are $t_4 = 4 \times 5/2 = 10$ pins. Show by induction on n that $\sum_{j=1}^{n} t_j = n(n+1)(n+2)/6$.

2.3.3: Identify the parity of each of the following bit sequences as even or odd:

a) 01101

b) 111000111

c) 010101

2.3.4: Suppose we use three digits — say 0, 1, and 2 — to code symbols. A set of strings C formed from 0's, 1's, and 2's is *error detecting* if no two strings in C differ in only one position. For example, $\{00, 11, 22\}$ is an error-detecting set with strings of length two, using the digits 0, 1, and 2. Show that for any $n \geq 1$, an error-detecting set of strings of length n using the digits 0, 1, and 2, cannot have more than 3^{n-1} strings.

2.3.5*: Show that for any $n \geq 1$, there is an error-detecting set of strings of length n, using the digits 0, 1, and 2, that has 3^{n-1} strings.

2.3.6*: Show that if we use k symbols, for any $k \geq 2$, then there is an error-detecting set of strings of length n, using k different symbols as "digits," with k^{n-1} strings, but no such set of strings with more than k^{n-1} strings.

2.3.7*: If $n \geq 1$, the number of strings using the digits 0, 1, and 2, with no two consecutive places holding the same digit, is $3 \times 2^{n-1}$. For example, there are 12 such strings of length three: 010, 012, 020, 021, 101, 102, 120, 121, 201, 202, 210, and 212. Prove this claim by induction on the length of the strings. Is the formula true for $n = 0$?

2.3.8*: Prove that the ripple-carry addition algorithm discussed in Section 1.3 produces the correct answer. *Hint*: Show by induction on i that after considering the first i places from the right end, the sum of the tails of length i for the two addends equals the number whose binary representation is the carry bit followed by the i bits of answer generated so far.

2.3.9*: The formula for the sum of n terms of a geometric series $a, ar, ar^2, \ldots, ar^{n-1}$ is

$$\sum_{i=0}^{n-1} ar^i = \frac{(ar^n - a)}{(r-1)}$$

Prove this formula by induction on n. Note that you must assume $r \neq 1$ for the formula to hold. Where do you use that assumption in your proof?

2.3.10: The formula for the sum of an arithmetic series with first term a and increment b, that is, $a, (a+b), (a+2b), \ldots, \big(a+(n-1)b\big)$, is

$$\sum_{i=0}^{n-1} a + bi = n\big(2a + (n-1)b\big)/2$$

a) Prove this formula by induction on n.

b) Show how Exercise 2.3.1(a) is an example of this formula.

2.3.11: Give two informal proofs that induction starting at 1 "works," although the statement $S(0)$ may be false.

2.3.12: Show by induction on the length of strings that the code consisting of the odd-parity strings detects errors.

Arithmetic and Geometric Sums

There are two formulas from high-school algebra that we shall use frequently. They each have interesting inductive proofs, which we ask the reader to provide in Exercises 2.3.9 and 2.3.10.

Arithmetic series

An *arithmetic series* is a sequence of n numbers of the form

$$a, (a+b), (a+2b), \ldots, \big(a+(n-1)b\big)$$

The first term is a, and each term is b larger than the one before. The sum of these n numbers is n times the average of the first and last terms; that is:

$$\sum_{i=0}^{n-1} a+bi = n\big(2a+(n-1)b\big)/2$$

For example, consider the sum of $3+5+7+9+11$. There are $n=5$ terms, the first is 3 and the last 11. Thus, the sum is $5\times(3+11)/2 = 5\times 7 = 35$. You can check that this sum is correct by adding the five integers.

Geometric series

A *geometric series* is a sequence of n numbers of the form

$$a, ar, ar^2, ar^3, \ldots, ar^{n-1}$$

That is, the first term is a, and each successive term is r times the previous term. The formula for the sum of n terms of a geometric series is

$$\sum_{i=0}^{n-1} ar^i = \frac{(ar^n - a)}{(r-1)}$$

Here, r can be greater or less than 1. If $r=1$, the above formula does not work, but all terms are a so the sum is obviously an.

As an example of a geometric series sum, consider $1+2+4+8+16$. Here, $n=5$, the first term a is 1, and the ratio r is 2. Thus, the sum is

$$(1\times 2^5 - 1)/(2-1) = (32-1)/1 = 31$$

as you may check. For another example, consider $1+1/2+1/4+1/8+1/16$. Again $n=5$ and $a=1$, but $r=1/2$. The sum is

$$\big(1\times(\tfrac{1}{2})^5 - 1\big)/(\tfrac{1}{2}-1) = (-31/32)/(-1/2) = 1\tfrac{15}{16}$$

Error-correcting code

2.3.13**: If no two strings in a code differ in fewer than three positions, then we can actually correct a single error, by finding the unique string in the code that differs from the received string in only one position. It turns out that there is a code of 7-bit strings that corrects single errors and contains 16 strings. Find such a code. *Hint*: Reasoning it out is probably best, but if you get stuck, write a program that searches for such a code.

2.3.14*: Does the even parity code detect any "double errors," that is, changes in two different bits? Can it correct any single errors?

Template for Simple Inductions

Let us summarize Section 2.3 by giving a template into which the simple inductions of that section fit. Section 2.4 will cover a more general template.

1. Specify the statement $S(n)$ to be proved. Say you are going to prove $S(n)$ by induction on n, for all $n \geq i_0$. Here, i_0 is the constant of the basis; usually i_0 is 0 or 1, but it could be any integer. Explain intuitively what n means, e.g., the length of codewords.
2. State the basis case, $S(i_0)$.
3. Prove the basis case. That is, explain why $S(i_0)$ is true.
4. Set up the inductive step by stating that you are assuming $S(n)$ for some $n \geq i_0$, the "inductive hypothesis." Express $S(n+1)$ by substituting $n+1$ for n in the statement $S(n)$.
5. Prove $S(n+1)$, assuming the inductive hypothesis $S(n)$.
6. Conclude that $S(n)$ is true for all $n \geq i_0$ (but not necessarily for smaller n).

2.4 Complete Induction

Strong and weak induction

In the examples seen so far, we have proved that $S(n+1)$ is true using only $S(n)$ as an inductive hypothesis. However, since we prove our statement S for values of its parameter starting at the basis value and proceeding upward, we are entitled to use $S(i)$ for all values of i, from the basis value up to n. This form of induction is called *complete* (or sometimes *perfect* or *strong*) *induction*, while the simple form of induction of Section 2.3, where we used only $S(n)$ to prove $S(n+1)$ is sometimes called *weak* induction.

Let us begin by considering how to perform a complete induction starting with basis $n = 0$. We prove that $S(n)$ is true for all $n \geq 0$ in two steps:

1. We first prove the basis, $S(0)$.
2. As an inductive hypothesis, we assume all of $S(0), S(1), \ldots, S(n)$ to be true. From these statements we prove that $S(n+1)$ holds.

As for weak induction described in the previous section, we can also pick some value a other than 0 as the basis. Then, for the basis we prove $S(a)$, and in the inductive step we are entitled to assume only $S(a), S(a+1), \ldots, S(n)$. Note that weak induction is a special case of complete induction in which we elect not to use any of the previous statements except $S(n)$ to prove $S(n+1)$.

Figure 2.7 suggests how complete induction works. Each instance of the statement $S(n)$ can (optionally) use any of the lower-indexed instances to its right in its proof.

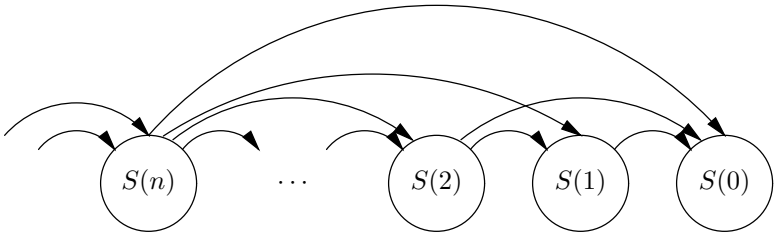


Fig. 2.7. Complete induction allows each instance to use one, some, or all of the previous instances in its proof.

Inductions With More Than One Basis Case

When performing a complete induction, there are times when it is useful to have more than one basis case. If we wish to prove a statement $S(n)$ for all $n \geq i_0$, then we could treat not only i_0 as a basis case, but also some number of consecutive integers above i_0, say $i_0, i_0+1, i_0+2, \ldots, j_0$. Then we must do the following two steps:

1. Prove each of the basis cases, the statements $S(i_0), S(i_0+1), \ldots, S(j_0)$.
2. As an inductive hypothesis, assume all of $S(i_0), S(i_0+1), \ldots, S(n)$ hold, for some $n \geq j_0$, and prove $S(n+1)$.

◆ **Example 2.7.** Our first example of a complete induction is a simple one that uses multiple basis cases. As we shall see, it is only "complete" in a limited sense. To prove $S(n+1)$ we do not use $S(n)$ but we use $S(n-1)$ only. In more general complete inductions to follow, we use $S(n)$, $S(n-1)$, and many other instances of the statement S.

Let us prove by induction on n the following statement for all $n \geq 0$.[4]

STATEMENT $S(n)$: There are integers a and b (positive, negative, or 0) such that $n = 2a + 3b$.

BASIS. We shall take both 0 and 1 as basis cases.

i) For $n = 0$ we may pick $a = 0$ and $b = 0$. Surely $0 = 2 \times 0 + 3 \times 0$.

ii) For $n = 1$, pick $a = -1$ and $b = 1$. Then $1 = 2 \times (-1) + 3 \times 1$.

INDUCTION. Now, we may assume $S(n)$ and prove $S(n+1)$, for any $n \geq 1$. Note that we may assume n is at least the largest of the consecutive values for which we have proved the basis: $n \geq 1$ here. Statement $S(n+1)$ says that $n+1 = 2a+3b$ for some integers a and b.

The inductive hypothesis says that all of $S(0), S(1), \ldots, S(n)$ are true. Note that we begin the sequence at 0 because that was the lowest of the consecutive basis cases. Since $n \geq 1$ can be assumed, we know that $n-1 \geq 0$, and therefore, $S(n-1)$ is true. This statement says that there are integers a and b such that $n-1 = 2a+3b$.

[4] Actually, this statement is true for all n, positive or negative, but the case of negative n requires a second induction which we leave as an exercise.

Since we need a in the statement $S(n+1)$, let us restate $S(n-1)$ to use different names for the integers and say there are integers a' and b' such that

$$n - 1 = 2a' + 3b' \tag{2.6}$$

If we add 2 to both sides of (2.6), we have $n + 1 = 2(a' + 1) + 3b'$. If we then let $a = a' + 1$ and $b = b'$, we have the statement $n + 1 = 2a + 3b$ for some integers a and b. This statement is $S(n + 1)$, so we have proved the induction. Notice that in this proof, we did not use $S(n)$, but we did use $S(n - 1)$. ◆

Justifying Complete Induction

Like the ordinary or "weak" induction discussed in Section 2.3, complete induction can be justified intuitively as a proof technique by a "least counterexample" argument. Let the basis cases be $S(i_0), S(i_0 + 1), \ldots, S(j_0)$, and suppose we have shown that for any $n \geq j_0$, $S(i_0), S(i_0+1), \ldots, S(n)$ together imply $S(n+1)$. Now, suppose $S(n)$ were not true for at least one value of $n \geq i_0$, and let b be the smallest integer equal to or greater than i_0 for which $S(b)$ is false. Then b cannot be between i_0 and j_0, or the basis is contradicted. Further, b cannot be greater than j_0. If it were, all of $S(i_0), S(i_0 + 1), \ldots, S(b - 1)$ would be true. But the inductive step would then tell us that $S(b)$ is true, yielding the contradiction.

Normal Forms for Arithmetic Expressions

We shall now explore an extended example concerning the transformation of arithmetic expressions to equivalent forms. It offers an illustration of a complete induction that takes full advantage of the fact that the statement S to be proved may be assumed for all arguments from n downward.

By way of motivation, a compiler for a programming language may take advantage of the algebraic properties of arithmetic operators to rearrange the order in which the operands of an arithmetic expression are evaluated. The goal of this rearrangement is to find a way for the computer to evaluate the expression using less time than the obvious evaluation order takes.

In this section we consider arithmetic expressions containing a single associative and commutative operator, like +, and examine what rearrangements of operands are possible. We shall prove that if we have any expression involving only the operator +, then the value of the expression is equal to the value of any other expression with + applied to the same operands, ordered and/or grouped in any arbitrary way. For example,

$$\big(a_3 + (a_4 + a_1)\big) + (a_2 + a_5) = a_1 + \Big(a_2 + \big(a_3 + (a_4 + a_5)\big)\Big)$$

We shall prove this claim by performing two separate inductions, the first of which is a complete induction.

◆ **Example 2.8.** We shall prove by complete induction on n (the number of operands in an expression) the statement

Associativity and Commutativity

Associative law

Recall that the *associative law* for addition says that we can add three values either by adding the first two and then adding the third to the result, or by adding the first to the result of adding the second and third; the result will be the same. Formally,

$$(E_1 + E_2) + E_3 = E_1 + (E_2 + E_3)$$

where E_1, E_2, and E_3 are any arithmetic expressions. For instance,

$$(1 + 2) + 3 = 1 + (2 + 3)$$

Here, $E_1 = 1$, $E_2 = 2$, and $E_3 = 3$. Also,

$$\big((xy) + (3z - 2)\big) + (y + z) = xy + \big((3z - 2) + (y + z)\big)$$

Here, $E_1 = xy$, $E_2 = 3z - 2$, and $E_3 = y + z$.

Commutative law

Also recall that the *commutative law* for addition says that we can sum two expressions in either order. Formally,

$$E_1 + E_2 = E_2 + E_1$$

For example, $1 + 2 = 2 + 1$, and $xy + (3z - 2) = (3z - 2) + xy$.

STATEMENT $S(n)$: If E is an expression involving the operator $+$ and n operands, and a is one of those operands, then E can be transformed, by using the associative and commutative laws, into an expression of the form $a + F$, where F is an expression involving all the operands of E except a, grouped in some order using the operator $+$.

Statement $S(n)$ only holds for $n \geq 2$, since there must be at least one occurrence of the operator $+$ in E. Thus, we shall use $n = 2$ as our basis.

BASIS. Let $n = 2$. Then E can be only $a + b$ or $b + a$, for some operand b other than a. In the first case, we let F be the expression b, and we are done. In the second case, we note that by the commutative law, $b + a$ can be transformed into $a + b$, and so we may again let $F = b$.

INDUCTION. Let E have $n + 1$ operands, and assume that $S(i)$ is true for $i = 2, 3, \ldots, n$. We need to prove the inductive step for $n \geq 2$, so we may assume that E has at least three operands and therefore at least two occurrences of $+$. We can write E as $E_1 + E_2$ for some expressions E_1 and E_2. Since E has exactly $n + 1$ operands, and E_1 and E_2 must each have at least one of these operands, it follows that neither E_1 nor E_2 can have more than n operands. Thus, the inductive hypothesis applies to E_1 and E_2, as long as they have more than one operand each (because we started with $n = 2$ as the basis). There are four cases we must consider, depending whether a is in E_1 or E_2, and on whether it is or is not the only operand in E_1 or E_2.

a) E_1 is a by itself. An example of this case occurs when E is $a + (b + c)$; here E_1 is a and E_2 is $b + c$. In this case, E_2 serves as F; that is, E is already of the form $a + F$.

b) E_1 has more than one operand, and a is among them. For instance,

$$E = \big(c + (d + a)\big) + (b + e)$$

where $E_1 = c + (d + a)$ and $E_2 = b + e$. Here, since E_1 has no more than n operands but at least two operands, we can apply the inductive hypothesis to tell us that E_1 can be transformed, using the commutative and associative laws, into $a+E_3$. Thus, E can be transformed into $(a+E_3)+E_2$. We apply the associative law and see that E can further be transformed into $a + (E_3 + E_2)$. Thus, we may choose F to be $E_3 + E_2$, which proves the inductive step in this case. For our example E above, we may suppose that $E_1 = c + (d + a)$ is transformed by the inductive hypothesis into $a + (c + d)$. Then E can be regrouped into $a + \big((c + d) + (b + e)\big)$.

c) E_2 is a alone. For instance, $E = (b+c)+a$. In this case, we use the commutative law to transform E into $a + E_1$, which is of the desired form if we let F be E_1.

d) E_2 has more than one operand, including a. An example is $E = b + (a + c)$. Apply the commutative law to transform E into $E_2 + E_1$. Then proceed as in case (b). If $E = b + (a + c)$, we transform E first into $(a + c) + b$. By the inductive hypothesis, $a + c$ can be put in the desired form; in fact, it is already there. The associative law then transforms E into $a + (c + b)$.

In all four cases, we have transformed E to the desired form. Thus, the inductive step is proved, and we conclude that $S(n)$ for all $n \geq 2$. ✥

✥ **Example 2.9.** The inductive proof of Example 2.8 leads directly to an algorithm that puts an expression into the desired form. As an example, consider the expression

$$E = \big(x + (z + v)\big) + (w + y)$$

and suppose that v is the operand we wish to "pull out," that is, to play the role of a in the transformation of Example 2.8. Initially, we have an example of case (b), with $E_1 = x + (z + v)$, and $E_2 = w + y$.

Next, we must work on the expression E_1 and "pull out" v. E_1 is an example of case (d), and so we first apply the commutative law to transform it into $(z+v)+x$. As an instance of case (b), we must work on the expression $z + v$, which is an instance of case (c). We thus transform it by the commutative law into $v + z$.

Now E_1 has been transformed into $(v+z)+x$, and a further use of the associative law transforms it to $v+(z+x)$. That, in turn, transforms E into $\big(v+(z+x)\big)+(w+y)$. By the associative law, E can be transformed into $v + \big((z + x) + (w + y)\big)$. Thus, $E = v + F$, where F is the expression $(z + x) + (w + y)$. The entire sequence of transformations is summarized in Fig. 2.8. ✥

Now, we can use the statement proved in Example 2.8 to prove our original contention, that any two expressions involving the operator + and the same list of distinct operands can be transformed one to the other by the associative and commutative laws. This proof is by weak induction, as discussed in Section 2.3, rather than complete induction.

$$\big(x + (z + v)\big) + (w + y)$$
$$\big((z + v) + x\big) + (w + y)$$
$$\big((v + z) + x\big) + (w + y)$$
$$\big(v + (z + x)\big) + (w + y)$$
$$v + \big((z + x) + (w + y)\big)$$

Fig. 2.8. Using the commutative and associative laws, we can "pull out" any operand, such as v.

◆ **Example 2.10.** Let us prove the following statement by induction on n, the number of operands in an expression.

STATEMENT $T(n)$: If E and F are expressions involving the operator $+$ and the same set of n distinct operands, then it is possible to transform E into F by a sequence of applications of the associative and commutative laws.

BASIS. If $n = 1$, then the two expressions must both be a single operand a. Since they are the same expression, surely E is "transformable" into F.

INDUCTION. Suppose $T(n)$ is true, for some $n \geq 1$. We shall now prove $T(n+1)$. Let E and F be expressions involving the same set of $n+1$ operands, and let a be one of these operands. Since $n+1 \geq 2$, $S(n+1)$ — the statement from Example 2.8 — must hold. Thus, we can transform E into $a + E_1$ for some expression E_1 involving the other n operands of E. Similarly, we can transform F into $a + F_1$, for some expression F_1 involving the same n operands as E_1. What is more important, in this case, is that we can also perform the transformations in the opposite direction, transforming $a + F_1$ into F by use of the associative and commutative laws.

Now we invoke the inductive hypothesis $T(n)$ on the expressions E_1 and F_1. Each has the same n operands, and so the inductive hypothesis applies. That tells us we can transform E_1 into F_1, and therefore we can transform $a + E_1$ into $a + F_1$. We may thus perform the transformations

$$\begin{array}{lll} E & \to \cdots \to a + E_1 & \text{Using } S(n+1) \\ & \to \cdots \to a + F_1 & \text{Using } T(n) \\ & \to \cdots \to F & \text{Using } S(n+1) \text{ in reverse} \end{array}$$

to turn E into F. ◆

◆ **Example 2.11.** Let us transform $E = (x+y)+(w+z)$ into $F = \big((w+z)+y\big)+x$. We begin by selecting an operand, say w, to "pull out." If we check the cases in Example 2.8, we see that for E we perform the sequence of transformations

$$(x + y) + (w + z) \to (w + z) + (x + y) \to w + \big(z + (x + y)\big) \tag{2.7}$$

while for F we do

$$\big((w + z) + y\big) + x \to \big(w + (z + y)\big) + x \to w + \big((z + y) + x\big) \tag{2.8}$$

We now have the subproblem of transforming $z + (x + y)$ into $(z + y) + x$. We shall do so by "pulling out" x. The sequences of transformations are

$$z + (x + y) \rightarrow (x + y) + z \rightarrow x + (y + z) \tag{2.9}$$

and

$$(z + y) + x \rightarrow x + (z + y) \tag{2.10}$$

That, in turn, gives us a subproblem of transforming $y + z$ into $z + y$. We do so by an application of the commutative law. Strictly speaking, we use the technique of Example 2.8 to "pull out" y for each, leaving $y + z$ for each expression. Then the basis case for Example 2.10 tells us that the expression z can be "transformed" into itself.

We can now transform $z + (x + y)$ into $(z + y) + x$ by the steps of line (2.9), then applying the commutative law to subexpression $y + z$, and finally using the transformation of line (2.10), in reverse. We use these transformations as the middle part of the transformation from $(x + y) + (w + z)$ to $\big((w + z) + y\big) + x$. First we apply the transformations of line (2.7), and then the transformations just discussed to change $z + (x + y)$ into $(z + y) + x$, and finally the transformations of line (2.8) in reverse. The entire sequence of transformations is summarized in Fig. 2.9. ❖

$(x + y) + (w + z)$	Expression E
$(w + z) + (x + y)$	Middle of (2.7)
$w + \big(z + (x + y)\big)$	End of (2.7)
$w + \big((x + y) + z\big)$	Middle of (2.9)
$w + \big(x + (y + z)\big)$	End of (2.9)
$w + \big(x + (z + y)\big)$	Commutative law
$w + \big((z + y) + x\big)$	(2.10) in reverse
$\big(w + (z + y)\big) + x$	Middle of (2.8) in reverse
$\big((w + z) + y\big) + x$	Expression F, end of (2.8) in reverse

Fig. 2.9. Transforming one expression into another using the commutative and associative laws.

EXERCISES

2.4.1: "Pull out" from the expression $E = (u + v) + \Big(\big(w + (x + y)\big) + z\Big)$ each of the operands in turn. That is, start from E in each of the six parts, and use the techniques of Example 2.8 to transform E into an expression of the form $u + E_1$. Then transform E_1 into an expression of the form $v + E_2$, and so on.

2.4.2: Use the technique of Example 2.10 to transform

a) $w + \big(x + (y + z)\big)$ into $\big((w + x) + y\big) + z$
b) $(v + w) + \big((x + y) + z\big)$ into $\big((y + w) + (v + z)\big) + x$

Binary operator

2.4.3*: Let E be an expression with operators $+$, $-$, $*$, and $/$; each operator is binary only; that is, it takes two operands. Show, using a complete induction on the number of occurrences of operators in E, that if E has n operator occurrences, then E has $n + 1$ operands.

A Template for All Inductions

The following organization of inductive proofs covers complete inductions with multiple basis cases. As a special case it includes the weak inductions of Section 2.3, and it includes the common situation where there is only one basis case.

1. Specify the statement $S(n)$ to be proved. Say that you are going to prove $S(n)$ by induction on n, for $n \geq i_0$. Specify what i_0 is; often it is 0 or 1, but i_0 could be any integer. Explain intuitively what n represents.
2. State the basis case(s). These will be all the integers from i_0 up to some integer j_0. Often $j_0 = i_0$, but j_0 could be larger.
3. Prove each of the basis cases $S(i_0), S(i_0 + 1), \ldots, S(j_0)$.
4. Set up the inductive step by stating that you are assuming

   $$S(i_0), S(i_0 + 1), \ldots, S(n)$$

   (the "inductive hypothesis") and that you want to prove $S(n + 1)$. State that you are assuming $n \geq j_0$; that is, n is at least as great as the highest basis case. Express $S(n + 1)$ by substituting $n + 1$ for n in the statement $S(n)$.
5. Prove $S(n + 1)$ under the assumptions mentioned in (4). If the induction is a weak, rather than complete, induction, then only $S(n)$ will be used in the proof, but you are free to use any or all of the statements of the inductive hypothesis.
6. Conclude that $S(n)$ is true for all $n \geq i_0$ (but not necessarily for smaller n).

2.4.4: Give an example of a binary operator that is commutative but not associative.

2.4.5: Give an example of a binary operator that is associative but not commutative.

2.4.6*: Consider an expression E whose operators are all binary. The *length* of E is the number of symbols in E, counting an operator or a left or right parenthesis as one symbol, and also counting any operand such as 123 or `abc` as one symbol. Prove that E must have an odd length. *Hint*: Prove the claim by complete induction on the length of the expression E.

2.4.7: Show that every negative integer can be written in the form $2a + 3b$ for some (not necessarily positive) integers a and b.

2.4.8*: Show that every integer (positive or negative) can be written in the form $5a + 7b$ for some (not necessarily positive) integers a and b.

2.4.9*: Is every proof by weak induction (as in Section 2.3) also a proof by complete induction? Is every proof by complete induction also a proof by weak induction?

2.4.10*: We showed in this section how to justify complete induction by a least counterexample argument. Show how complete induction can also be justified by an iteration.

Truth in Advertising

There are many difficulties, both theoretical and practical, in proving programs correct. An obvious question is "What does it mean for a program to be 'correct'?" As we mentioned in Chapter 1, most programs in practice are written to satisfy some informal specification. The specification itself may be incomplete or inconsistent. Even if there were a precise formal specification, we can show that no algorithm exists to prove that an arbitrary program is equivalent to a given specification.

However, in spite of these difficulties, it is beneficial to state and prove assertions about programs. The loop invariants of a program are often the most useful short explanation one can give of how the program works. Further, the programmer should have a loop invariant in mind while writing a piece of code. That is, there must be a reason why a program works, and this reason often has to do with an inductive hypothesis that holds each time the program goes around a loop or each time it performs a recursive call. The programmer should be able to envision a proof, even though it may be impractical to write out such a proof line by line.

2.5 Proving Properties of Programs

In this section we shall delve into an area where inductive proofs are essential: proving that a program does what it is claimed to do. We shall see a technique for explaining what an iterative program does as it goes around a loop. If we understand what the loops do, we generally understand what we need to know about an iterative program. In Section 2.9, we shall consider what is needed to prove properties of recursive programs.

Loop Invariants

Inductive assertion

The key to proving a property of a loop in a program is selecting a *loop invariant,* or *inductive assertion,* which is a statement S that is true each time we enter a particular point in the loop. The statement S is then proved by induction on a parameter that in some way measures the number of times we have gone around the loop. For example, the parameter could be the number of times we have reached the test of a while-loop, it could be the value of the loop index in a for-loop, or it could be some expression involving the program variables that is known to increase by 1 for each time around the loop.

◆ **Example 2.12.** As an example, let us consider the inner loop of `SelectionSort` from Section 2.2. These lines, with the original numbering from Fig. 2.2, are

```
(2)          small = i;
(3)          for (j = i+1; j < n; j++)
(4)              if (A[j] < A[small])
(5)                  small = j;
```

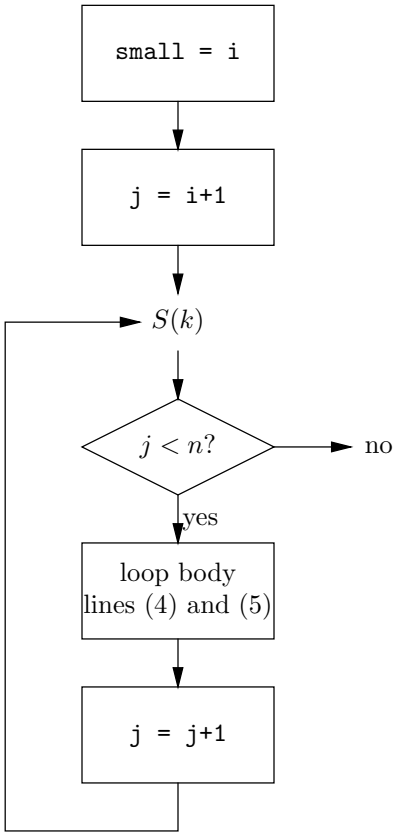


Fig. 2.10. Flowchart for the inner loop of `SelectionSort`.

Recall that the purpose of these lines is to make `small` equal to the index of an element of `A[i..n-1]` with the smallest value. To see why that claim is true, consider the flowchart for our loop shown in Fig. 2.10. This flowchart shows the five steps necessary to execute the program:

1. First, we need to initialize `small` to i, as we do in line (2).
2. At the beginning of the for-loop of line (3), we need to initialize `j` to $i + 1$.
3. Then, we need to test whether $j < n$.
4. If so, we execute the body of the loop, which consists of lines (4) and (5).
5. At the end of the body, we need to increment `j` and go back to the test.

In Fig. 2.10 we see a point just before the test that is labeled by a loop-invariant statement we have called $S(k)$; we shall discover momentarily what this statement must be. The first time we reach the test, `j` has the value $i + 1$ and `small` has the value i. The second time we reach the test, `j` has the value $i+2$, because `j` has been incremented once. Because the body (lines 4 and 5) sets `small` to $i + 1$ if $A[i + 1]$ is less than $A[i]$, we see that `small` is the index of whichever of $A[i]$ and $A[i + 1]$ is smaller.[5]

[5] In case of a tie, *small* will be i. In general, we shall pretend that no ties occur and talk about "the smallest element" when we really mean "the first occurrence of the smallest element."

Similarly, the third time we reach the test, the value of `j` is $i+3$ and `small` is the index of the smallest of `A[i..i+2]`. We shall thus try to prove the following statement, which appears to be the general rule.

STATEMENT $S(k)$: If we reach the test for $j < n$ in the for-statement of line (3) with k as the value of loop index `j`, then the value of `small` is the index of the smallest of `A[i..k-1]`.

Note that we are using the letter k to stand for one of the values that the variable `j` assumes, as we go around the loop. That is less cumbersome than trying to use j as the value of `j`, because we sometimes need to keep k fixed while the value of `j` changes. Also notice that $S(k)$ has the form "if we reach $\cdots$," because for some values of k we may never reach the loop test, as we broke out of the loop for a smaller value of the loop index `j`. If k is one of those values, then $S(k)$ is surely true, because any statement of the form "if A then B" is true when A is false.

BASIS. The basis case is $k = i+1$, where i is the value of the variable `i` at line (3).[6] Now $j = i+1$ when we begin the loop. That is, we have just executed line (2), which gives `small` the value i, and we have initialized `j` to $i+1$ to begin the loop. $S(i+1)$ says that `small` is the index of the smallest element in `A[i..i]`, which means that the value of `small` must be i. But we just observed that line (2) causes `small` to have the value i. Technically, we must also show that `j` can never have value $i+1$ except the first time we reach the test. The reason, intuitively, is that each time around the loop, we increment `j`, so it will never again be as low as $i+1$. (To be perfectly precise, we should give an inductive proof of the assumption that $j > i+1$ except the first time through the test.) Thus, the basis, $S(i+1)$, has been shown to be true.

INDUCTION. Now let us assume as our inductive hypothesis that $S(k)$ holds, for some $k \geq i+1$, and prove $S(k+1)$. First, if $k \geq n$, then we break out of the loop when `j` has the value k, or earlier, and so we are sure never to reach the loop test with the value of `j` equal to $k+1$. In that case, $S(k+1)$ is surely true.

Thus, let us assume that $k < n$, so that we actually make the test with `j` equal to $k+1$. $S(k)$ says that `small` indexes the smallest of `A[i..k-1]`, and $S(k+1)$ says that `small` indexes the smallest of `A[i..k]`. Consider what happens in the body of the loop (lines 4 and 5) when `j` has the value k; there are two cases, depending on whether the test of line (4) is true or not.

1. If $A[k]$ is not smaller than the smallest of `A[i..k-1]`, then the value of `small` does not change. In that case, however, `small` also indexes the smallest of `A[i..k]`, since $A[k]$ is not the smallest. Thus, the conclusion of $S(k+1)$ is true in this case.

2. If $A[k]$ is smaller than the smallest of $A[i]$ through $A[k-1]$, then `small` is set to k. Again, the conclusion of $S(k+1)$ now holds, because k is the index of the smallest of `A[i..k]`.

---

[6] As far as the loop of lines (3) to (5) is concerned, `i` does not change. Thus, $i+1$ is an appropriate constant to use as the basis value.

Thus, in either case, `small` is the index of the smallest of `A[i..k]`. We go around the for-loop by incrementing the variable `j`. Thus, just before the loop test, when `j` has the value $k+1$, the conclusion of $S(k+1)$ holds. We have now shown that $S(k)$ implies $S(k+1)$. We have completed the induction and conclude that $S(k)$ holds for all values $k \geq i+1$.

Next, we apply $S(k)$ to make our claim about the inner loop of lines (3) through (5). We exit the loop when the value of `j` reaches n. Since $S(n)$ says that `small` indexes the smallest of `A[i..n-1]`, we have an important conclusion about the working of the inner loop. We shall see how it is used in the next example. ✦

```
(1)          for (i = 0; i < n-1; i++) {
(2)              small = i;
(3)              for (j = i+1; j < n; j++)
(4)                  if (A[j] < A[small])
(5)                      small = j;
(6)              temp = A[small];
(7)              A[small] = A[i];
(8)              A[i] = temp;
             }
```

Fig. 2.11. The body of the `SelectionSort` function.

✦ **Example 2.13.** Now, let us consider the entire `SelectionSort` function, the heart of which we reproduce in Fig. 2.11. A flowchart for this code is shown in Fig. 2.12, where "body" refers to lines (2) through (8) of Fig. 2.11. Our inductive assertion, which we refer to as $T(m)$, is again a statement about what must be true just before the test for termination of the loop. Informally, when `i` has the value m, we have selected m of the smallest elements and sorted them at the beginning of the array. More precisely, we prove the following statement $T(m)$ by induction on m.

STATEMENT $T(m)$: If we reach the loop test $i < n-1$ of line (1) with the value of variable `i` equal to m, then

a) `A[0..m-1]` are in sorted order; that is, $A[0] \leq A[1] \leq \cdots \leq A[m-1]$.

b) All of `A[m..n-1]` are at least as great as any of `A[0..m-1]`.

BASIS. The basis case is $m = 0$. The basis is true for trivial reasons. If we look at the statement $T(0)$, part (a) says that `A[0..-1]` are sorted. But there are no elements in the range $A[0], \ldots, A[-1]$, and so (a) must be true. Similarly, part (b) of $T(0)$ says that all of `A[0..n-1]` are at least as large as any of `A[0..-1]`. Since there are no elements of the latter description, part (b) is also true.

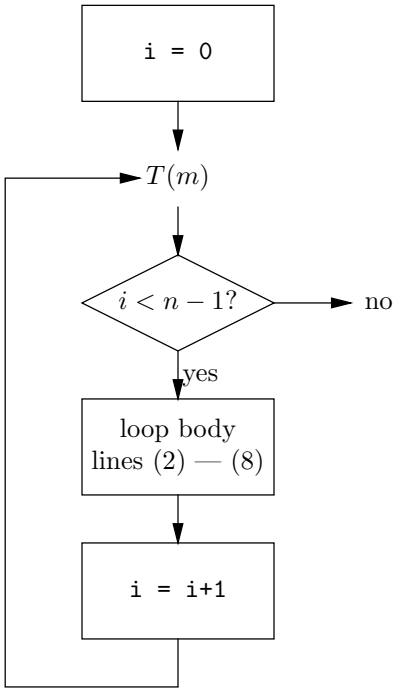


Fig. 2.12. Flow-chart for the entire selection sort function.

INDUCTION. For the inductive step, we assume that $T(m)$ is true for some $m \geq 0$, and we show that $T(m+1)$ holds. As in Example 2.12, we are trying to prove a statement of the form "if A then B," and such a statement is true whenever A is false. Thus, $T(m+1)$ is true if the assumption that we reach the for-loop test with `i` equal to $m+1$ is false. Thus, we may assume that we actually reach the test with `i` having the value $m+1$; that is, we may assume $m < n-1$.

When `i` has the value m, the body of the loop finds a smallest element in `A[m..n-1]` (as proved by the statement $S(m)$ of Example 2.12). This element is swapped with $A[m]$ in lines (6) through (8). Part (b) of the inductive hypothesis, $T(m)$, tells us the element chosen must be at least as large as any of `A[0..m-1]`. Moreover, those elements were sorted, so now all of `A[i..m]` are sorted. That proves part (a) of statement $T(m+1)$.

To prove part (b) of $T(m+1)$, we see that $A[m]$ was just selected to be as small as any of `A[m+1..n-1]`. Part (a) of $T(m)$ tells us that `A[0..m-1]` were already as small as any of `A[m+1..n-1]`. Thus, after executing the body of lines (2) through (8) and incrementing `i`, we know that all of `A[m+1..n-1]` are at least as large as any of `A[0..m]`. Since now the value of `i` is $m+1$, we have shown the truth of the statement $T(m+1)$ and thus have proved the inductive step.

Now, let $m = n-1$. We know that we exit the outer for-loop when `i` has the value $n-1$, so $T(n-1)$ will hold after we finish this loop. Part (a) of $T(n-1)$ says that all of `A[0..n-2]` are sorted, and part (b) says that $A[n-1]$ is as large as any of the other elements. Thus, after the program terminates the elements in `A` are in nonincreasing order; that is, they are sorted. ✦

Loop Invariants for While-Loops

When we have a while-loop of the form

while (<condition>)
    <body>

it usually makes sense to find the appropriate loop invariant for the point just before the test of the condition. Generally, we try to prove the loop invariant holds by induction on the number of times around the loop. Then, when the condition becomes false, we can use the loop invariant, together with the falsehood of the condition, to conclude something useful about what is true after the while-loop terminates.

However, unlike for-loops, there may not be a variable whose value counts the number of times around the while-loop. Worse, while the for-loop is guaranteed to iterate only up to the limit of the loop (for example, up to $n-1$ for the inner loop of the `SelectionSort` program), there is no reason to believe that the condition of the while-loop will ever become false. Thus, part of the proof of correctness for a while-loop is a proof that it eventually terminates. We usually prove termination by identifying some expression E, involving the variables of the program, such that

While-loop termination

1. The value of E decreases by at least 1 each time around the loop, and
2. The loop condition is false if E is as low as some specified constant, such as 0.

◆ **Example 2.14.** The factorial function, written $n!$, is defined as the product of the integers $1 \times 2 \times \cdots \times n$. For example, $1! = 1$, $2! = 1 \times 2 = 2$, and

Factorial

$$5! = 1 \times 2 \times 3 \times 4 \times 5 = 120$$

Figure 2.13 shows a simple program fragment to compute $n!$ for integers $n \geq 1$.

```
(1)     scanf("%d", &n);
(2)     i = 2;
(3)     fact = 1;
(4)     while (i <= n) {
(5)         fact = fact*i;
(6)         i++;
        }
(7)     printf("%d\n", fact);
```

Fig. 2.13. Factorial program fragment.

To begin, let us prove that the while-loop of lines (4) to (6) in Fig. 2.13 must terminate. We shall choose E to be the expression $n - i$. Notice that each time around the while-loop, `i` is increased by 1 at line (6) and `n` remains unchanged. Therefore, E decreases by 1 each time around the loop. Moreover, when E is -1 or less, we have $n - i \leq -1$, or $i \geq n + 1$. Thus, when E becomes negative, the loop condition $i \leq n$ will be false and the loop will terminate. We don't know how large E is initially, since we don't know what value of `n` will be read. Whatever that value is, however, E will eventually reach as low as -1, and the loop will terminate.

Now we must prove that the program of Fig. 2.13 does what it is intended to do. The appropriate loop-invariant statement, which we prove by induction on the value of the variable `i`, is

STATEMENT $S(j)$: If we reach the loop test $i \leq n$ with the variable `i` having the value j, then the value of the variable `fact` is $(j-1)!$.

BASIS. The basis is $S(2)$. We reach the test with `i` having value 2 only when we enter the loop from the outside. Prior to the loop, lines (2) and (3) of Fig. 2.13 set `fact` to 1 and `i` to 2. Since $1 = (2-1)!$, the basis is proved.

INDUCTION. Assume $S(j)$, and prove $S(j+1)$. If $j > n$, then we break out of the while-loop when `i` has the value j or earlier, and thus we never reach the loop test with `i` having the value $j+1$. In that case, $S(j+1)$ is trivially true, because it is of the form "If we reach $\cdots$."

Thus, assume $j \leq n$, and consider what happens when we execute the body of the while-loop with `i` having the value j. By the inductive hypothesis, before line (5) is executed, `fact` has value $(j-1)!$, and `i` has the value j. Thus, after line (5) is executed, `fact` has the value $j \times (j-1)!$, which is $j!$.

At line (6), `i` is incremented by 1 and so attains the value $j+1$. Thus, when we reach the loop test with `i` having value $j+1$, the value of `fact` is $j!$. The statement $S(j+1)$ says that when `i` equals $j+1$, `fact` equals $((j+1)-1)!$, or $j!$. Thus, we have proved statement $S(j+1)$, and completed the inductive step.

We already have shown that the while-loop will terminate. Evidently, it terminates when `i` first attains a value greater than n. Since `i` is an integer and is incremented by 1 each time around the loop, `i` must have the value $n+1$ when the loop terminates. Thus, when we reach line (7), statement $S(n+1)$ must hold. But that statement says that `fact` has the value $n!$. Thus, the program prints $n!$, as we wished to prove.

As a practical matter, we should point out that on any computer the factorial program in Fig. 2.13 will print $n!$ as an answer for very few values of n. The problem is that the factorial function grows so rapidly that the size of the answer quickly exceeds the maximum size of an integer on any real computer. ✦

EXERCISES

2.5.1: What is an appropriate loop invariant for the following program fragment, which sets `sum` equal to the sum of the integers from 1 to n?

```
scanf("%d",&n);
sum = 0;
for (i = 1; i <= n; i++)
    sum = sum + i;
```

Prove your loop invariant by induction on i, and use it to prove that the program works as intended.

2.5.2: The following fragment computes the sum of the integers in array `A[0..n-1]`:

```
sum = 0;
for (i = 0; i < n; i++)
    sum = sum + A[i];
```

What is an appropriate loop invariant? Use it to show that the fragment works as intended.

2.5.3*: Consider the following fragment:

```
scanf("%d", &n);
x = 2;
for (i = 1; i <= n; i++)
    x = x * x;
```

An appropriate loop invariant for the point just before the test for $i \leq n$ is that if we reach that point with the value k for variable `i`, then $x = 2^{2^{k-1}}$. Prove that this invariant holds, by induction on k. What is the value of `x` after the loop terminates?

```
sum = 0;
scanf("%d", &x);
while (x >= 0) {
    sum = sum + x;
    scanf("%d", &x);
}
```

Fig. 2.14. Summing a list of integers terminated by a negative integer.

2.5.4*: The fragment in Fig. 2.14 reads integers until it finds a negative integer, and then prints the accumulated sum. What is an appropriate loop invariant for the point just before the loop test? Use the invariant to show that the fragment performs as intended.

2.5.5: Find the largest value of n for which the program in Fig. 2.13 works on your computer. What are the implications of fixed-length integers for proving programs correct?

2.5.6: Show by induction on the number of times around the loop of Fig. 2.10 that $j > i + 1$ after the first time around.

2.6 Recursive Definitions

Inductive definition

In a *recursive*, or *inductive*, definition, we define one or more classes of closely related objects (or facts) in terms of the objects themselves. The definition must not be meaningless, like "a widget is a widget of some color," or paradoxical, like "something is a glotz if and only if it is not a glotz." Rather, a recursive definition involves

1. One or more *basis rules*, in which some simple objects are defined, and
2. One or more *inductive rules*, whereby larger objects are defined in terms of smaller ones in the collection.

◆ **Example 2.15.** In the previous section we defined the factorial function by an iterative algorithm: multiply $1 \times 2 \times \cdots \times n$ to get $n!$. However, we can also define the value of $n!$ recursively, as follows.

BASIS. $1! = 1$.

INDUCTION. $n! = n \times (n-1)!$.

For example, the basis tells us that $1! = 1$. We can use this fact in the inductive step with $n = 2$ to find

$$2! = 2 \times 1! = 2 \times 1 = 2$$

With $n = 3,\ 4$, and 5, we get

$$3! = 3 \times 2! = 3 \times 2 = 6$$
$$4! = 4 \times 3! = 4 \times 6 = 24$$
$$5! = 5 \times 4! = 5 \times 24 = 120$$

and so on. Notice that, although it appears that the term "factorial" is defined in terms of itself, in practice, we can get the value of $n!$ for progressively higher values of n in terms of the factorials for lower values of n only. Thus, we have a meaningful definition of "factorial."

Strictly speaking, we should prove that our recursive definition of $n!$ gives the same result as our original definition,

$$n! = 1 \times 2 \times \cdots \times n$$

To do so, we shall prove the following statement:

STATEMENT $S(n)$: $n!$, as defined recursively above, equals $1 \times 2 \times \cdots \times n$.

The proof will be by induction on n.

BASIS. $S(1)$ clearly holds. The basis of the recursive definition tells us that $1! = 1$, and the product $1 \times \cdots \times 1$ (i.e., the product of the integers "from 1 to 1") is evidently 1 as well.

INDUCTION. Assume that $S(n)$ holds; that is, $n!$, as given by the recursive definition, equals $1 \times 2 \times \cdots \times n$. Then the recursive definition tells us that

$$(n+1)! = (n+1) \times n!$$

If we use the commutative law for multiplication, we see that

$$(n+1)! = n! \times (n+1) \tag{2.11}$$

By the inductive hypothesis,

$$n! = 1 \times 2 \times \cdots \times n$$

Thus, we may substitute $1 \times 2 \times \cdots \times n$ for $n!$ in Equation (2.11) to get

$$(n+1)! = 1 \times 2 \times \cdots \times n \times (n+1)$$

which is the statement $S(n+1)$. We have thereby proved the inductive hypothesis and shown that our recursive definition of $n!$ is the same as our iterative definition.✥

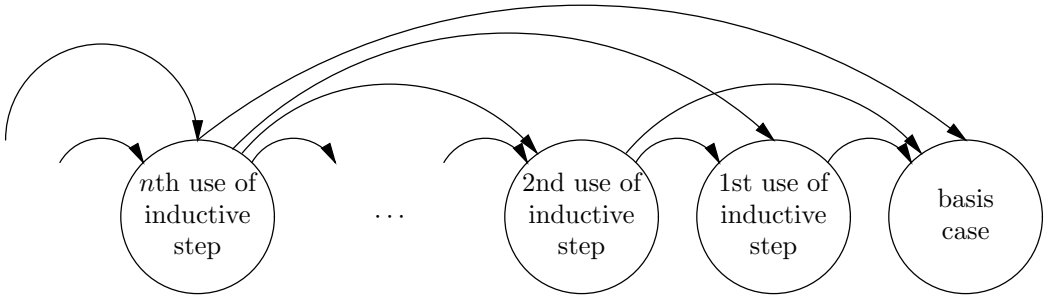


Fig. 2.15. In a recursive definition, we construct objects in rounds, where the objects constructed in one round may depend on objects constructed in all previous rounds.

Figure 2.15 suggests the general nature of a recursive definition. It is similar in structure to a complete induction, in that there is an infinite sequence of cases, each of which can depend on any or all of the previous cases. We start by applying the basis rule or rules. On the next round, we apply the inductive rule or rules to what we have already obtained, to construct new facts or objects. On the following round, we again apply the inductive rules to what we have, obtaining new facts or objects, and so on.

In Example 2.15, where we were defining the factorial, we discovered the value of 1! by the basis case, 2! by one application of the inductive step, 3! by two applications, and so on. Here, the induction had the form of an "ordinary" induction, where we used in each round only what we had discovered in the previous round.

✥ **Example 2.16.** In Section 2.2 we defined the notion of lexicographic order of strings, and our definition was iterative in nature. Roughly, we test whether string $c_1 \cdots c_n$ precedes string $d_1 \cdots d_m$ by comparing corresponding symbols c_i and d_i from the left, until we either find an i for which $c_i \neq d_i$ or come to the end of one of the strings. The following recursive definition defines those pairs of strings w and x such that w precedes x in lexicographic order. Intuitively, the induction is on the number of pairs of equal characters at the beginnings of the two strings involved.

Lexicographic order

BASIS. The basis covers those pairs of strings for which we can immediately resolve the question of which comes first in lexicographic order. There are two parts of the basis.

1. $\epsilon < w$ for any string w other than ϵ itself. Recall that ϵ is the empty string, or the string with no characters.
2. If $c < d$, where c and d are characters, then for any strings w and x, we have $cw < dx$.

INDUCTION. If $w < x$ for strings w and x, then for any character c we have $cw < cx$.

For instance, we can use the above definition to show that `base` $<$ `batter`. By rule (2) of the basis, with $c =$ `s`, $d =$ `t`, $w =$ `e`, and $x =$ `ter`, we have `se` $<$ `tter`. If we apply the recursive rule once, with $c =$ `a`, $w =$ `se`, and $x =$ `tter`, we infer that `ase` $<$ `atter`. Finally, applying the recursive rule a second time with $c =$ `b`, $w =$ `ase`, and $x =$ `atter`, we find `base` $<$ `batter`. That is, the basis and inductive steps appear as follows:

`se`	$<$	`tter`
`ase`	$<$	`atter`
`base`	$<$	`batter`

We can also show that `bat` $<$ `batter` as follows. Part (1) of the basis tells us that $\epsilon <$ `ter`. If we apply the recursive rule three times — with c equal to `t`, `a`, and `b`, in turn — we make the following sequence of inferences:

ϵ	$<$	`ter`
`t`	$<$	`tter`
`at`	$<$	`atter`
`bat`	$<$	`batter`

Now we should prove, by induction on the number of characters that two strings have in common at their left ends, that one string precedes the other according to the definition in Section 2.2 if and only if it precedes according to the recursive definition just given. We leave these two inductive proofs as exercises. ✥

In Example 2.16, the groups of facts suggested by Fig. 2.15 are large. The basis case gives us all facts $w < x$ for which either $w = \epsilon$ or w and x begin with different letters. One use of the inductive step gives us all $w < x$ facts where w and x have exactly one initial letter in common, the second use covers those cases where w and x have exactly two initial letters in common, and so on.

Expressions

Arithmetic expressions of all kinds are naturally defined recursively. For the basis of the definition, we specify what the atomic operands can be. For example, in C, atomic operands are either variables or constants. Then, the induction tells us what operators may be applied, and to how many operands each is applied. For instance, in C, the operator $<$ can be applied to two operands, the operator symbol $-$ can be applied to one or two operands, and the function application operator, represented by a pair of parentheses with as many commas inside as necessary, can be applied to one or more operands, as $f(a_1, \ldots, a_n)$.

✥ **Example 2.17.** It is common to refer to the following set of expressions as "arithmetic expressions."

BASIS. The following types of atomic operands are arithmetic expressions:

1. Variables
2. Integers
3. Real numbers

INDUCTION. If E_1 and E_2 are arithmetic expressions, then the following are also arithmetic expressions:

1. $(E_1 + E_2)$
2. $(E_1 - E_2)$
3. $(E_1 \times E_2)$
4. $(E_1 \ / \ E_2)$

Infix operator

The operators $+$, $-$, $\times$, and / are said to be binary operators, because they take two arguments. They are also said to be *infix* operators, because they appear between their two arguments.

Additionally, we allow a minus sign to imply negation (change of sign), as well as subtraction. That possibility is reflected in the fifth and last recursive rule:

5. If E is an arithmetic expression, then so is $(-E)$.

Unary, prefix operator

An operator like $-$ in rule (5), which takes only one operand, is said to be a *unary* operator. It is also said to be a *prefix* operator, because it appears before its argument.

Figure 2.16 illustrates some arithmetic expressions and explains why each is an expression. Note that sometimes parentheses are not needed, and we can omit them. In the final expression (vi) of Fig. 2.16, the outer parentheses and the parentheses around $-(x + 10)$ can be omitted, and we could write $y \times -(x + 10)$. However, the remaining parentheses are essential, since $y \times -x + 10$ is conventionally interpreted as $(y \times -x) + 10$, which is not an equivalent expression (try $y = 1$ and $x = 0$, for instance).[7] ✦

$i)$	x	Basis rule (1)
$ii)$	10	Basis rule (2)
$iii)$	$(x + 10)$	Recursive rule (1) on (i) and (ii)
$iv)$	$\big(-(x + 10)\big)$	Recursive rule (5) on (iii)
$v)$	y	Basis rule (1)
$vi)$	$\Big(y \times \big(-(x + 10)\big)\Big)$	Recursive rule (3) on (v) and (iv)

Fig. 2.16. Some sample arithmetic expressions.

[7] Parentheses are redundant when they are implied by the conventional precedences of operators (unary minus highest, then multiplication and division, then addition and subtraction) and by the convention of "left associativity," which says that we group operators at the same precedence level (e.g., a string of plusses and minuses) from the left. These conventions should be familiar from C, as well as from ordinary arithmetic.

More Operator Terminology

Postfix operator

A unary operator that appears after its argument, as does the factorial operator ! in expressions like $n!$, is said to be a *postfix* operator. Operators that take more than one operand can also be prefix or postfix operators, if they appear before or after all their arguments, respectively. There are no examples in C or ordinary arithmetic of operators of these types, although in Section 5.4 we shall discuss notations in which all operators are prefix or postfix operators.

Ternary operator

An operator that takes three arguments is a *ternary* operator. In C, `?:` is a ternary operator, as in the expression `c?x:y` meaning "if c then x else y." In general, if an operator takes k arguments, it is said to be *k-ary.*

Balanced Parentheses

Strings of parentheses that can appear in expressions are called *balanced parentheses.* For example, the pattern `(((())))` appears in expression (vi) of Fig. 2.16, and the expression

$$\Big((a+b)\times\big((c+d)-e\big)\Big)$$

has the pattern `(()(()))`. The empty string, ϵ, is also a string of balanced parentheses; it is the pattern of the expression x, for example. In general, what makes a string of parentheses balanced is that it is possible to match each left parenthesis with a right parenthesis that appears somewhere to its right. Thus, a common definition of "balanced parenthesis strings" consists of two rules:

1. A balanced string has an equal number of left and right parentheses.

Profile

2. As we move from left to right along the string, the profile of the string never becomes negative, where the *profile* is the running total of the number of left parentheses seen minus the number of right parentheses seen.

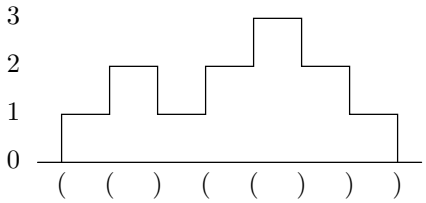
Note that the profile must begin and end at 0. For example, Fig. 2.17(a) shows the profile of `(()(()))`, and Fig. 2.17(b) shows the profile of `()(())()`.

There are a number of recursive definitions for the notion of "balanced parentheses." The following is a bit subtle, but we shall prove that it is equivalent to the preceding, nonrecursive definition involving profiles.

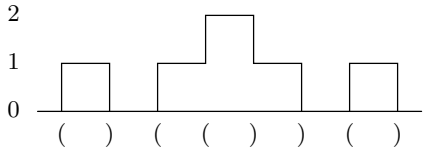
BASIS. The empty string is a string of balanced parentheses.

INDUCTION. If x and y are strings of balanced parentheses, then `(`x`)`y is also a string of balanced parentheses.

◆ **Example 2.18.** By the basis, ϵ is a balanced-parenthesis string. If we apply the recursive rule, with x and y both equal to ϵ, then we infer that `()` is balanced. Notice that when we substitute the empty string for a variable, such as x or y, that variable "disappears." Then we may apply the recursive rule with:



(a) Profile of `((()(()))`.



(b) Profile of `()((())()`.

Fig. 2.17. Profiles of two strings of parentheses.

1. $x =$ `()` and $y = \epsilon$, to discover that `(())` is balanced.
2. $x = \epsilon$ and $y =$ `()`, to find that `()()` is balanced.
3. $x = y =$ `()` to infer that `(())()` is balanced.

As a final example, since we now know that `(())` and `()()` are balanced, we may let these be x and y in the recursive rule, respectively, and show that `((()))()()` is balanced. ✥

Profile-balanced

We can show that the two definitions of "balanced" specify the same sets of strings. To make things clearer, let us refer to strings that are balanced according to the recursive definition simply as *balanced* and refer to those balanced according to the nonrecursive definition as *profile-balanced.* That is, the profile-balanced strings are those whose profile ends at 0 and never goes negative. We need to show two things:

1. Every balanced string is profile-balanced.
2. Every profile-balanced string is balanced.

These are the aims of the inductive proofs in the next two examples.

✥ **Example 2.19.** First, let us prove part (1), that every balanced string is profile-balanced. The proof is a complete induction that mirrors the induction by which the class of balanced strings is defined. That is, we prove

STATEMENT $S(n)$: If string w is defined to be balanced by n applications of the recursive rule, then w is profile-balanced.

BASIS. The basis is $n = 0$. The only string that can be shown to be balanced without any application of the recursive rule is ϵ, which is balanced according to the basis rule. Evidently, the profile of the empty string ends at 0 and does not go negative, so ϵ is profile-balanced.

INDUCTION. Assume that $S(i)$ is true for $i = 0, 1, \ldots, n$, and consider an instance of $S(n+1)$, that is, a string w whose proof of balance requires $n+1$ uses of the recursive rule. Consider the last such use, in which we took two strings x and y, already known to be balanced, and formed w as $(x)y$. We used the recursive rule $n+1$ times to form w, and one use was the last step, which helped form neither x nor y. Thus, neither x nor y requires more than n uses of the recursive rule. Therefore, the inductive hypothesis applies to both x and y, and we can conclude that x and y are profile-balanced.

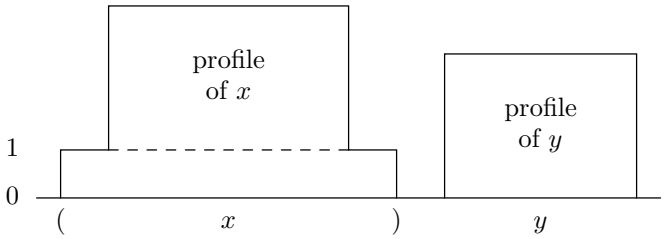


Fig. 2.18. Constructing the profile of $w = (x)y$.

The profile of w is as suggested in Fig. 2.18. It first goes up one level, in response to the first left parenthesis. Then comes the profile of x, raised one level, as indicated by the dashed line. We used the inductive hypothesis to conclude that x is profile-balanced; therefore, its profile begins and ends at level 0 and never goes negative. As the x portion of w's profile is raised one level in Fig. 2.18, that portion begins and ends at level 1 and never goes below level 1.

The explicitly shown right parenthesis between x and y lowers the profile of w to 0. Then comes the profile of y. By the inductive hypothesis, y is profile-balanced. Thus, the y portion of w's profile does not go below 0, and it ends the profile of w at 0.

We have now constructed the profile of w and see that it meets the condition for a profile-balanced string. That is, it begins and ends at 0, and it never becomes negative. Thus, we have proved that if a string is balanced, it is profile-balanced. ✥

Now we shall address the second direction of the equivalence between the two definitions of "balanced parentheses." We show in the next example that a profile-balanced string is balanced.

✥ **Example 2.20.** We prove part (2), that "profile-balanced" implies "balanced," by complete induction on the length of the string of parentheses. The formal statement is

Proofs About Recursive Definitions

Notice that Example 2.19 proves an assertion about a class of recursively defined objects (the balanced strings of parentheses) by induction on the number of times the recursive rule is used to establish that the object is in the defined class. That is a very common way to deal with recursively defined concepts; in fact, it is one of the reasons recursive definitions are useful. As another illustration, in Example 2.15, we showed a property of the recursively defined factorial values (that $n!$ is the product of the integers from 1 to n) by induction on n. But n is also 1 plus the number of times we used the recursive rule in the definition of $n!$, so the proof could also be considered an induction on the number of applications of the recursive rule.

STATEMENT $S(n)$: If a string w of length n is profile-balanced, then it is balanced.

BASIS. If $n = 0$, then the string must be ϵ. We know that ϵ is balanced by the basis rule of the recursive definition.

INDUCTION. Suppose that profile-balanced strings of length equal to or less than n are balanced. We must prove $S(n+1)$, that profile-balanced strings of length $n+1$ are also balanced.[8] Consider such a string w. Since w is profile-balanced, it cannot start with a right parenthesis, or its profile would immediately go negative. Thus, w begins with a left parenthesis.

Let us break w into two parts. The first part starts at the beginning of w and ends where the profile of w first becomes 0. The second part is the remainder of w. For example, the profile of Fig. 2.17(a) first becomes 0 at the end, so if $w =$ `(()(()))`, then the first part is the entire string and the second part is ϵ. In Fig. 2.17(b), where $w =$ `()(())()`, the first part is `()`, and the second part is `(())()`.

The first part can never end in a left parenthesis, because then the profile would be negative at the position just before. Thus, the first part begins with a left parenthesis and ends with a right parenthesis. We can therefore write w as $(x)y$, where (x) is the first part and y is the second part. Both x and y are shorter than w, so if we can show they are profile-balanced, then we can use the inductive hypothesis to infer that they are balanced. Then we can use the recursive rule in the definition of "balanced" to show that $w = (x)y$ is balanced.

It is easy to see that y is profile-balanced. Figure 2.18 also illustrates the relationship between the profiles of w, x, and y here. That is, the profile of y is a tail of the profile of w, beginning and ending at height 0. Since w is profile-balanced, we can conclude that y is also. Showing that x is profile-balanced is almost the same. The profile of x is a part of the profile of w; it begins and ends at level 1 in the profile of w, but we can lower it by one level to get the profile of x. We know that the profile of w never reaches 0 during the extent of x, because we picked (x) to be the shortest prefix of w that ends with the profile of w at level 0. Hence, the profile of x within w never reaches level 0, and the profile of x itself never becomes negative.

We have now shown both x and y to be profile-balanced. Since they are each

[8] Note that all profile-balanced strings happen to be of even length, so if $n+1$ is odd, we are not saying anything. However, we do not need the evenness of n for the proof.

shorter than w, the inductive hypothesis applies to them, and they are each balanced. The recursive rule defining "balanced" says that if x and y are balanced, then so is $(x)y$. But $w = (x)y$, and so w is balanced. We have now completed the inductive step and shown statement $S(n)$ to be true for all $n \geq 0$. ✦

EXERCISES

2.6.1*: Prove that the definitions of lexicographic order given in Example 2.16 and in Section 2.2 are the same. *Hint*: The proof consists of two parts, and each is an inductive proof. For the first part, suppose that $w < x$ according to the definition in Example 2.16. Prove the following statement $S(i)$ by induction on i: "If it is necessary to apply the recursive rule i times to show that $w < x$, then w precedes x according to the definition of 'lexicographic order' in Section 2.2." The basis is $i = 0$. The second part of the exercise is to show that if w precedes x in lexicographic order according to the definition in Section 2.2, then $w < x$ according to the definition in Example 2.16. Now the induction is on the number of initial positions that w and x have in common.

2.6.2: Draw the profiles of the following strings of parentheses:

a) `(()(()))`
b) `()())(()`
c) `(((())()()))`
d) `(()(()(())))`

Which are profile-balanced? For those that are profile-balanced, use the recursive definition in Section 2.6 to show that they are balanced.

2.6.3*: Show that every string of balanced parentheses (according to the recursive definition in Section 2.6) is the string of parentheses in some arithmetic expression (see Example 2.17 for a definition of arithmetic expressions). *Hint*: Use a proof by induction on the number of times the recursive rule of the definition of "balanced parentheses" is used to construct the given string of balanced parentheses.

2.6.4: Tell whether each of the following C operators is prefix, postfix, or infix, and whether they are unary, binary, or k-ary for some $k > 2$:

a) `<`
b) `&`
c) `%`

2.6.5: If you are familiar with the UNIX file system or a similar system, give a recursive definition of the possible directory/file structures.

2.6.6*: A certain set S of integers is defined recursively by the following rules.

BASIS. 0 is in S.

INDUCTION. If i is in S, then $i + 5$ and $i + 7$ are in S.

a) What is the largest integer *not* in S?

b) Let j be your answer to part (a). Prove that all integers $j+1$ and greater are in S. *Hint*: Note the similarity to Exercise 2.4.8 (although here we are dealing with only nonnegative integers).

2.6.7*: Define recursively the set of even-parity strings, by induction on the length of the string. *Hint*: It helps to define two concepts simultaneously, both the even-parity strings and the odd-parity strings.

2.6.8*: We can define sorted lists of integers as follows.

BASIS. A list consisting of a single integer is sorted.

INDUCTION. If L is a sorted list in which the last element is a, and if $b \geq a$, then L followed by b is a sorted list.

Prove that this recursive definition of "sorted list" is equivalent to our original, nonrecursive definition, which is that the list consist of integers

$$a_1 \leq a_2 \leq \cdots \leq a_n$$

Remember, you need to prove two parts: (a) If a list is sorted by the recursive definition, then it is sorted by the nonrecursive definition, and (b) if a list is sorted by the nonrecursive definition, then it is sorted by the recursive definition. Part (a) can use induction on the number of times the recursive rule is used, and (b) can use induction on the length of the list.

2.6.9**: As suggested by Fig. 2.15, whenever we have a recursive definition, we can classify the objects defined according to the "round" on which each is generated, that is, the number of times the inductive step is applied before we obtain each object. In Examples 2.15 and 2.16, it was fairly easy to describe the results generated on each round. Sometimes it is more challenging to do so. How do you characterize the objects generated on the nth round for each of the following?

a) Arithmetic expressions like those described in Example 2.17. *Hint*: If you are familiar with trees, which are the subject of Chapter 5, you might consider the tree representation of expressions.

b) Balanced parenthesis strings. Note that the "number of applications used," as discussed in Example 2.19, is not the same as the round on which a string is discovered. For example, `(())()` uses the inductive rule three times but is discovered on round 2.

2.7 Recursive Functions

Direct and indirect recursion

A recursive function is one that is called from within its own body. Often, the call is *direct;* for example, a function F has a call to F within itself. Sometimes, however, the call is *indirect*: some function F_1 calls a function F_2 directly, which calls F_3 directly, and so on, until some function F_k in the sequence calls F_1.

There is a common belief that it is easier to learn to program iteratively, or to use nonrecursive function calls, than it is to learn to program recursively. While we cannot argue conclusively against that point of view, we do believe that recursive programming is easy once one has had the opportunity to practice the style.

More Truth in Advertising

A potential disadvantage of using recursion is that function calls on some machines are time-consuming, so that a recursive program may take more time to run than an iterative program for the same problem. However, on many modern machines function calls are quite efficient, and so this argument against using recursive programs is becoming less important.

Profiling

Even on machines with slow function-calling mechanisms, one can *profile* a program to find how much time is spent on each part of the program. One can then rewrite the parts of the program in which the bulk of its time is spent, replacing recursion by iteration if necessary. That way, one gets the advantages of recursion throughout most of the program, except for a small fraction of the code where speed is most critical.

Recursive programs are often more succinct or easier to understand than their iterative counterparts. More importantly, some problems are more easily attacked by recursive programs than by iterative programs.[9]

Often, we can develop a recursive algorithm by mimicking a recursive definition in the specification of a program we are trying to implement. A recursive function that implements a recursive definition will have a basis part and an inductive part. Frequently, the basis part checks for a simple kind of input that can be solved by the basis of the definition, with no recursive call needed. The inductive part of the function requires one or more recursive calls to itself and implements the inductive part of the definition. Some examples should clarify these points.

◆ **Example 2.21.** Figure 2.19 gives a recursive function that computes $n!$ given a positive integer n. This function is a direct transcription of the recursive definition of $n!$ in Example 2.15. That is, line (1) of Fig. 2.19 distinguishes the basis case from the inductive case. We assume that $n \geq 1$, so the test of line (1) is really asking whether $n = 1$. If so, we apply the basis rule, $1! = 1$, at line (2). If $n > 1$, then we apply the inductive rule, $n! = n \times (n-1)!$, at line (3).

```
            int fact(int n)
            {
(1)             if (n <= 1)
(2)                 return 1; /* basis */
                else
(3)                 return n*fact(n-1); /* induction */
            }
```

Fig. 2.19. Recursive function to compute $n!$ for $n \geq 1$.

For instance, if we call `fact(4)`, the result is a call to `fact(3)`, which calls

[9] Such problems often involve some kind of search. For instance, in Chapter 5 we shall see some recursive algorithms for searching trees, algorithms that have no convenient iterative analog (although there are equivalent iterative algorithms using stacks).

Defensive Programming

The program of Fig. 2.19 illustrates an important point about writing recursive programs so that they do not run off into infinite sequences of calls. We tacitly assumed that `fact` would never be called with an argument less than 1. Best, of course, is to begin `fact` with a test that $n \geq 1$, printing an error message and returning some particular value such as 0 if it is not. However, even if we believe very strongly that `fact` will never be called with $n < 1$, we shall be wise to include in the basis case all these "error cases." Then, the function `fact` called with erroneous input will simply return the value 1, which is wrong, but not a disaster (in fact, 1 is even correct for $n = 0$, since 0! is conventionally defined to be 1).

However, suppose we were to ignore the error cases and write line (1) of Fig. 2.19 as

```
if (n == 1)
```

Then if we called `fact(0)`, it would look like an instance of the inductive case, and we would next call `fact(-1)`, then `fact(-2)`, and so on, terminating with failure when the computer ran out of space to record the recursive calls.

`fact(2)`, which calls `fact(1)`. At that point, `fact(1)` applies the basis rule, because $n \leq 1$, and returns the value 1 to `fact(2)`. That call to `fact` completes line (3), returning 2 to `fact(3)`. In turn, `fact(3)` returns 6 to `fact(4)`, which completes line (3) by returning 24 as the answer. Figure 2.20 suggests the pattern of calls and returns. ✥

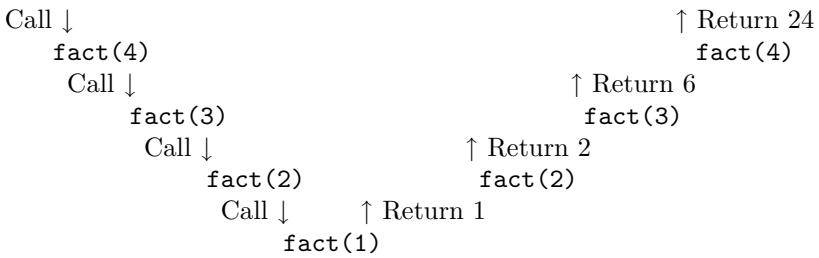


Fig. 2.20. Calls and returns resulting from call to `fact(4)`.

Size of arguments

We can picture a recursion much as we have pictured inductive proofs and definitions. In Fig. 2.21 we have assumed that there is a notion of the "size" of arguments for a recursive function. For example, for the function *fact* in Example 2.21 the value of the argument n itself is the appropriate size. We shall say more about the matter of size in Section 2.9. However, let us note here that it is essential for a recursion to make only calls involving arguments of smaller size. Also, we must reach the basis case — that is, we must terminate the recursion — when we reach some particular size, which in Fig. 2.21 is size 0.

In the case of the function `fact`, the calls are not as general as suggested by Fig. 2.21. A call to `fact(n)` results in a direct call to `fact(n-1)`, but `fact(n)` does

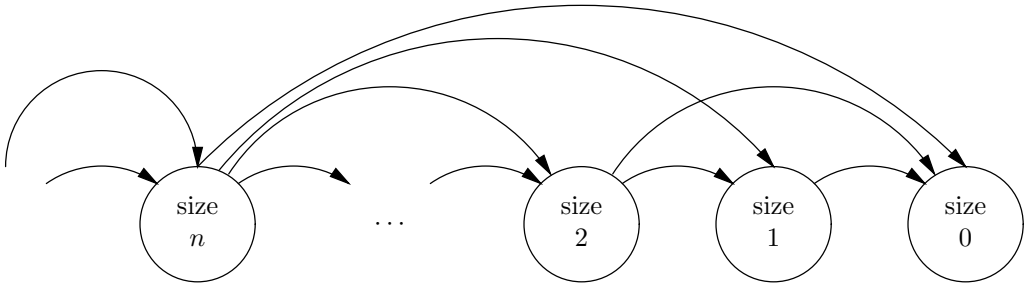


Fig. 2.21. A recursive function calls itself with arguments of smaller size.

not call `fact` with any smaller argument directly.

✦ **Example 2.22.** We can turn the function `SelectionSort` of Fig. 2.2 into a recursive function `recSS`, if we express the underlying algorithm as follows. Assume the data to be sorted is in `A[0..n-1]`.

1. Pick a smallest element from the tail of the array `A`, that is, from `A[i..n-1]`.
2. Swap the element selected in step (1) with `A[i]`.
3. Sort the remainder of the array, `A[i+1..n-1]`.

We can express selection sort as the following recursive algorithm.

BASIS. If $i = n - 1$, then only the last element of the array remains to be sorted. Since any one element is already sorted, we need not do anything.

INDUCTION. If $i < n - 1$, then find the smallest element in `A[i..n-1]`, swap it with `A[i]`, and recursively sort `A[i+1..n-1]`.

The entire algorithm is to perform the above recursion starting with $i = 0$.

Backward induction

If we see i as the parameter in the preceding induction, it is a case of *backward induction,* where we start with a high basis and by the inductive rule solve instances with smaller values of the parameter in terms of instances with higher values. That is a perfectly good style of induction, although we have not previously mentioned its possibility. However, we can also see this induction as an ordinary, or "forward" induction on a parameter $k = n - i$ that represents the number of elements in the tail of the array waiting to be sorted.

In Fig. 2.22, we see the program for `recSS(A,i,n)`. The second parameter `i` is the index of the first element in the unsorted tail of the array `A`. The third parameter `n` is the total number of elements in the array `A` to be sorted. Presumably, n is less than or equal to the maximum size of `A`. Thus, a call to `recSS(A,0,n)` will sort the entire array `A[0..n-1]`.

In terms of Fig. 2.21, $s = n - i$ is the appropriate notion of "size" for arguments of the function `recSS`. The basis is $s = 1$ — that is, sorting one element, in which case no recursive calls occur. The inductive step tells us how to sort s elements by picking the smallest and then sorting the remaining $s - 1$ elements.

At line (1) we test for the basis case, in which there is only one element remaining to be sorted (again, we are being defensive, so that if we somehow make a

```
     void recSS(int A[], int i, int n)
     {
         int j, small, temp;
(1)      if (i < n-1) {/* basis is when i = n-1, in which case */
                 /* the function returns without changing A */
                 /* induction follows */
(2)              small = i;
(3)              for (j = i+1; j < n; j++)
(4)                  if (A[j] < A[small])
(5)                      small = j;
(6)              temp = A[small];
(7)              A[small] = A[i];
(8)              A[i] = temp;
(9)              recSS(A, i+1, n);
             }
     }
```

Fig. 2.22. Recursive selection sort.

call with $i \geq n$, we shall not go into an infinite sequence of calls). In the basis case, we have nothing to do, so we just return.

The remainder of the function is the inductive case. Lines (2) through (8) are copied directly from the iterative version of selection sort. Like that program, these lines set `small` to the index of the array `A[i..n-1]` that holds a smallest element and then swap this element with `A[i]`. Finally, line (9) is the recursive call, which sorts the remainder of the array. ✦

EXERCISES

2.7.1: We can define n^2 recursively as follows.

BASIS. For $n = 1$, $1^2 = 1$.

INDUCTION. If $n^2 = m$, then $(n+1)^2 = m + 2n + 1$.

a) Write a recursive C function to implement this recursion.

b) Prove by induction on n that this definition correctly computes n^2.

2.7.2: Suppose that we are given array `A[0..4]`, with elements 10, 13, 4, 7, 11, in that order. What are the contents of the array `A` just before each recursive call to `recSS`, according to the recursive function of Fig. 2.22?

2.7.3: Suppose we define cells for a linked list of integers, as discussed in Section 1.3, using the macro `DefCell(int, CELL, LIST)` of Section 1.6. Recall, this macro expands to be the following type definition:

Divide-and-Conquer

One way of attacking a problem is to try to break it into subproblems and then solve the subproblems and combine their solutions into a solution for the problem as a whole. The term *divide-and-conquer* is used to describe this problem-solving technique. If the subproblems are similar to the original, then we may be able to use the same function to solve the subproblems recursively.

There are two requirements for this technique to work. The first is that the subproblems must be simpler than the original problem. The second is that after a finite number of subdivisions, we must encounter a subproblem that can be solved outright. If these criteria are not met, a recursive algorithm will continue subdividing the problem forever, without finding a solution.

Note that the recursive function `recSS` in Fig. 2.22 satisfies both criteria. Each time it is invoked, it is on a subarray that has one fewer element, and when it is invoked on a subarray containing a single element, it returns without invoking itself again. Similarly, the factorial program of Fig. 2.19 involves calls with a smaller integer value at each call, and the recursion stops when the argument of the call reaches 1. Section 2.8 discusses a more powerful use of the divide-and-conquer technique, called "merge sort." There, the size of the arrays being sorted diminishes very rapidly, because merge sort works by dividing the size in half, rather than subtracting 1, at each recursive call.

```
typedef struct CELL *LIST;
struct CELL {
    int element;
    LIST next;
};
```

Write a recursive function `find` that takes an argument of type `LIST` and returns `TRUE` if some cell of the list contains the integer 1698 as its element and returns `FALSE` if not.

2.7.4: Write a recursive function `add` that takes an argument of type `LIST`, as defined in Exercise 2.7.3, and returns the sum of the elements on the list.

2.7.5: Write a version of recursive selection sort that takes as argument a list of integers, using the cells mentioned in Exercise 2.7.3.

2.7.6: In Exercise 2.2.8 we suggested that one could generalize selection sort to use arbitrary *key* and *lt* functions to compare elements. Rewrite the recursive selection sort algorithm to incorporate this generality.

2.7.7*: Give a recursive algorithm that takes an integer i and produces the binary representation of i as a sequence of 0's and 1's, low-order bit first.

GCD

2.7.8*: The *greatest common divisor* (GCD) of two integers i and j is the largest integer that divides both i and j evenly. For example, $gcd(24, 30) = 6$, and $gcd(24, 35) = 1$. Write a recursive function that takes two integers i and j, with $i > j$, and returns $gcd(i, j)$. *Hint*: You may use the following recursive definition of *gcd*. It assumes that $i > j$.

BASIS. If j divides i evenly, then j is the GCD of i and j.

INDUCTION. If j does not divide i evenly, let k be the remainder when i is divided by j. Then $gcd(i, j)$ is the same as $gcd(j, k)$.

2.7.9**: Prove that the recursive definition of GCD given in Exercise 2.7.8 gives the same result as the nonrecursive definition (largest integer dividing both i and j evenly).

2.7.10: Often, a recursive definition can be turned into an algorithm fairly directly. For example, consider the recursive definition of "less than" on strings given in Example 2.16. Write a recursive function that tests whether the first of two given strings is "less than" the other. Assume that strings are represented by linked lists of characters.

2.7.11*: From the recursive definition of a sorted list given in Exercise 2.6.8, create a recursive sorting algorithm. How does this algorithm compare with the recursive selection sort of Example 2.22?

2.8 Merge Sort: A Recursive Sorting Algorithm

Divide and conquer

We shall now consider a sorting algorithm, called *merge sort,* which is radically different from selection sort. Merge sort is best described recursively, and it illustrates a powerful use of the *divide-and-conquer* technique, in which we sort a list $(a_1, a_2, \ldots, a_n)$ by "dividing" the problem into two similar problems of half the size. In principle, we could begin by dividing the list into two arbitrarily chosen equal-sized lists, but in the program we develop, we shall make one list out of the odd-numbered elements, $(a_1, a_3, a_5, \ldots)$ and the other out of the even-numbered elements, $(a_2, a_4, a_6, \ldots)$.[10] We then sort each of the half-sized lists separately. To complete the sorting of the original list of n elements, we merge the two sorted, half-sized lists by an algorithm to be described in the next example.

In the next chapter, we shall see that the time required for merge sort grows much more slowly, as a function of the length n of the list to be sorted, than does the time required by selection sort. Thus, even if recursive calls take some extra time, merge sort is greatly preferable to selection sort when n is large. In Chapter 3 we shall examine the relative performance of these two sorting algorithms.

Merging

To "merge" means to produce from two sorted lists a single sorted list containing all the elements of the two given lists and no other elements. For example, given the lists $(1, 2, 7, 7, 9)$ and $(2, 4, 7, 8)$, the merger of these lists is $(1, 2, 2, 4, 7, 7, 7, 8, 9)$. Note that it does not make sense to talk about "merging" lists that are not already sorted.

One simple way to merge two lists is to examine them from the front. At each step, we find the smaller of the two elements at the current fronts of the lists, choose that element as the next element on the combined list, and remove the chosen element from its list, exposing a new "first" element on that list. Ties can be broken

[10] Remember that "odd-numbered" and "even-numbered" refer to the positions of the elements on the list, and not to the values of these elements.

L_1	L_2	M
$1, 2, 7, 7, 9$	$2, 4, 7, 8$	empty
$2, 7, 7, 9$	$2, 4, 7, 8$	1
$7, 7, 9$	$2, 4, 7, 8$	$1, 2$
$7, 7, 9$	$4, 7, 8$	$1, 2, 2$
$7, 7, 9$	$7, 8$	$1, 2, 2, 4$
$7, 9$	$7, 8$	$1, 2, 2, 4, 7$
9	$7, 8$	$1, 2, 2, 4, 7, 7$
9	8	$1, 2, 2, 4, 7, 7, 7$
9	empty	$1, 2, 2, 4, 7, 7, 7, 8$
empty	empty	$1, 2, 2, 4, 7, 7, 7, 8, 9$

Fig. 2.23. Example of merging.

arbitrarily, although we shall take from the first list when the leading elements of both lists are the same.

◆ **Example 2.23.** Consider merging the two lists

$$L_1 = (1, 2, 7, 7, 9) \text{ and } L_2 = (2, 4, 7, 8)$$

The first elements of the lists are 1 and 2, respectively. Since 1 is smaller, we choose it as the first element of the merged list M and remove 1 from L_1. The new L_1 is thus $(2, 7, 7, 9)$. Now, both L_1 and L_2 have 2 as their first elements. We can pick either. Suppose we adopt the policy that we always pick the element from L_1 in case of a tie. Then merged list M becomes $(1, 2)$, list L_1 becomes $(7, 7, 9)$, and L_2 remains $(2, 4, 7, 8)$. The table in Fig. 2.23 shows the merging steps until lists L_1 and L_2 are both exhausted. ◆

We shall find it easier to design a recursive merging algorithm if we represent lists in the linked form suggested in Section 1.3. Linked lists will be reviewed in more detail in Chapter 6. In what follows, we shall assume that list elements are integers. Thus, each element can be represented by a "cell," or structure of the type `struct CELL`, and the list by a type `LIST`, which is a pointer to a `CELL`. These definitions are provided by the macro `DefCell(int, CELL, LIST)`, which we discussed in Section 1.6. This use of macro `DefCell` expands into:

```
typedef struct CELL *LIST;
struct CELL {
    int element;
    LIST next;
};
```

The `element` field of each cell contains an integer, and the `next` field contains a pointer to the next cell on the list. If the element at hand is the last on the list, then the `next` field contains the value `NULL`, which represents a null pointer. A list of integers is then represented by a pointer to the first cell on the list, that is, by a variable of type `LIST`. An empty list is represented by a variable with the value

```
     LIST merge(LIST list1, LIST list2)
     {
(1)      if (list1 == NULL) return list2;
(2)      else if (list2 == NULL) return list1;
(3)      else if (list1->element <= list2->element) {
             /* Here, neither list is empty, and the first list
                has the smaller first element. The answer is the
                first element of the first list followed by the
                merge of the remaining elements. */
(4)          list1->next = merge(list1->next, list2);
(5)          return list1;
         }
         else { /* list2 has smaller first element */
(6)          list2->next = merge(list1, list2->next);
(7)          return list2;
         }
     }
```

Fig. 2.24. Recursive merge.

`NULL`, in place of a pointer to the first element.

Figure 2.24 is a C implementation of a recursive merging algorithm. The function `merge` takes two lists as arguments and returns the merged list. That is, the formal parameters `list1` and `list2` are pointers to the two given lists, and the return value is a pointer to the merged list. The recursive algorithm can be described as follows.

BASIS. If either list is empty, then the other list is the desired result. This rule is implemented by lines (1) and (2) of Fig. 2.24. Note that if both lists are empty, then `list2` will be returned. But that is correct, since the value of `list2` is then `NULL` and the merger of two empty lists is an empty list.

INDUCTION. If neither list is empty, then each has a first element. We can refer to the two first elements as `list1->element` and `list2->element`, that is, the `element` fields of the cells pointed to by `list1` and `list2`, respectively. Fig 2.25 is a picture of the data structure. The list to be returned begins with the cell of the smallest element. The remainder of the list is formed by merging all but that element.

For example, lines (4) and (5) handle the case in which the first element of list 1 is smallest. Line (4) is a recursive call to `merge`. The first argument of this call is `list1->next`, that is, a pointer to the second element on the first list (or `NULL` if the first list only has one element). Thus, the recursive call is passed the list consisting of all but the first element of the first list. The second argument is the entire second list. As a consequence, the recursive call to `merge` at line (4) will return a pointer to the merged list of all the remaining elements and store a pointer to this merged list in the `next` field of the first cell on list 1. At line (5), we return a pointer to that cell, which is now the first cell on the merged list of all the elements.

Figure 2.25 illustrates the changes. Dotted arrows are present when `merge` is

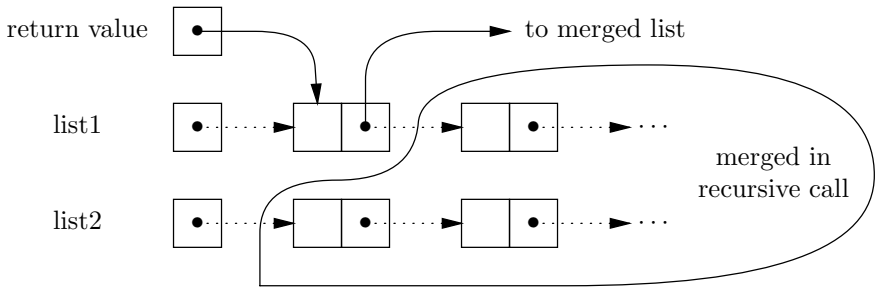


Fig. 2.25. Inductive step of merging algorithm.

called. Solid arrows are created by `merge`. Specifically, the return value of `merge` is a pointer to the cell of the smallest element, and the `next` field of that element is shown pointing to the list returned by the recursive call to `merge` at line (4).

Finally, lines (6) and (7) handle the case where the second list has the smallest element. The behavior of the algorithm is exactly as in lines (4) and (5), but the roles of the two lists are reversed.

◆ **Example 2.24.** Suppose we call `merge` on the lists $(1, 2, 7, 7, 9)$ and $(2, 4, 7, 8)$ of Example 2.23. Figure 2.26 illustrates the sequence of calls made to `merge`, if we read the first column downward. We omit the commas separating list elements, but commas are used to separate the arguments of `merge`.

CALL	RETURN
`merge(12779, 2478)`	`122477789`
`merge(2779, 2478)`	`22477789`
`merge(779, 2478)`	`2477789`
`merge(779, 478)`	`477789`
`merge(779, 78)`	`77789`
`merge(79, 78)`	`7789`
`merge(9, 78)`	`789`
`merge(9, 8)`	`89`
`merge(9, NULL)`	`9`

Fig. 2.26. Recursive calls to `merge`.

For instance, since the first element of list 1 is less than the first element of list 2, line (4) of Fig. 2.24 is executed and we recursively merge all but the first element of list 1. That is, the first argument is the tail of list 1, or $(2, 7, 7, 9)$, and the second argument is the full list 2, or $(2, 4, 7, 8)$. Now the leading elements of both lists are the same. Since the test of line (3) in Fig. 2.24 favors the first list, we remove the 2 from list 1, and our next call to `merge` has first argument $(7, 7, 9)$ and second argument $(2, 4, 7, 8)$.

The returned lists are indicated in the second column, read upward. Notice that, unlike the iterative description of merging suggested by Fig. 2.23, the recursive

algorithm assembles the merged list from the rear, whereas the iterative algorithm assembles it from the front. ◆

Splitting Lists

Another important task required for merge sort is splitting a list into two equal parts, or into parts whose lengths differ by 1 if the original list is of odd length. One way to do this job is to count the number of elements on the list, divide by 2, and break the list at the midpoint. Instead, we shall give a simple recursive function `split` that "deals" the elements into two lists, one consisting of the first, third, and fifth elements, and so on, and the other consisting of the elements at the even positions. More precisely, the function `split` removes the even-numbered elements from the list it is given as an argument and returns a new list consisting of the even-numbered elements.

The C code for function `split` is shown in Fig. 2.27. Its argument is a list of the type `LIST` that was defined in connection with the `merge` function. Note that the local variable `pSecondCell` is defined to be of type `LIST`. We really use `pSecondCell` as a pointer to the second cell on a list, rather than as a list itself; but of course type `LIST` is, in fact, a pointer to a cell.

It is important to observe that `split` is a function with a side effect. It removes the cells in the even positions from the list it is given as an argument, and it assembles these cells into a new list, which becomes the return value of the function.

```
     LIST split(LIST list)
     {
         LIST pSecondCell;

(1)      if (list == NULL) return NULL;
(2)      else if (list->next == NULL) return NULL;
         else { /* there are at least two cells */
(3)          pSecondCell = list->next;
(4)          list->next = pSecondCell->next;
(5)          pSecondCell->next = split(pSecondCell->next);
(6)          return pSecondCell;
         }
     }
```

Fig. 2.27. Splitting a list into two equal pieces.

The splitting algorithm can be described inductively, as follows. It uses an induction on the length of a list; that induction has a multiple basis case.

BASIS. If the list is of length 0 or 1, then we do nothing. That is, an empty list is "split" into two empty lists, and a list of a single element is split by leaving the element on the given list and returning an empty list of the even-numbered elements, of which there are none. The basis is handled by lines (1) and (2) of Fig. 2.27. Line (1) handles the case where `list` is empty, and line (2) handles the case where it is a single element. Notice that we are careful not to examine `list->next` in line (2) unless we have previously determined, at line (1), that `list` is not `NULL`.

INDUCTION. The inductive step applies when there are at least two elements on `list`. At line (3), we keep a pointer to the second cell of the list in the local variable `pSecondCell`. Line (4) makes the `next` field of the first cell skip over the second cell and point to the third cell (or become `NULL` if there are only two cells on the list). At line (5), we call `split` recursively, on the list consisting of all but the first two elements. The value returned by that call is a pointer to the fourth element (or `NULL` if the list is shorter than four elements), and we place this pointer in the `next` field of the second cell, to complete the linking of the even-numbered elements. A pointer to the second cell is returned by `split` at line (6); that pointer gives us access to the linked list of all the even-numbered elements of the original list.

The changes made by `split` are suggested in Fig. 2.28. Original pointers are dotted, and new pointers are solid. We also indicate the number of the line that creates each of the new pointers.

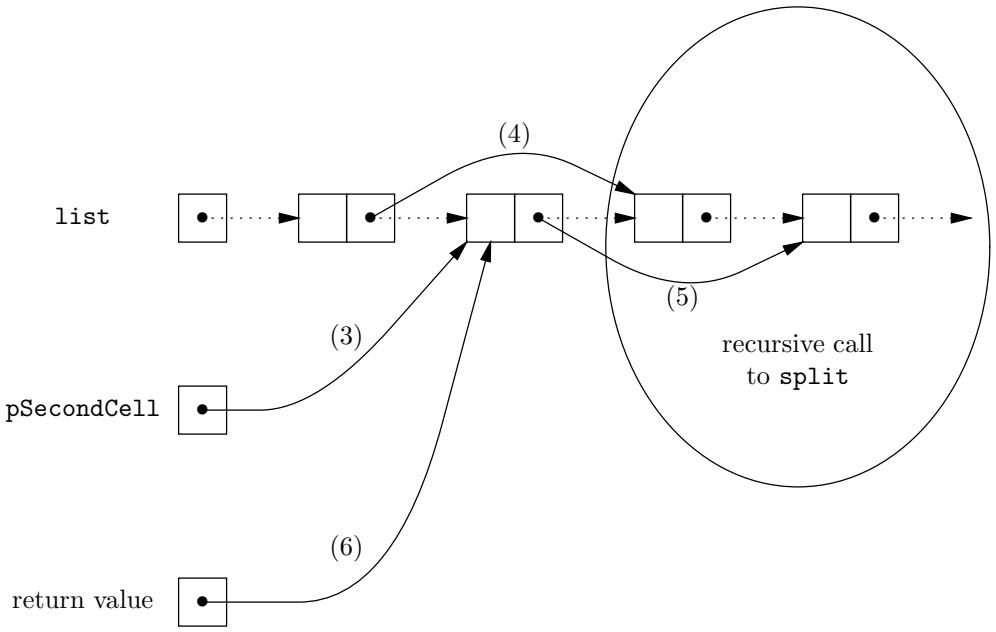


Fig. 2.28. Action of function `split`.

The Sorting Algorithm

The recursive sorting algorithm is shown in Fig. 2.29. The algorithm can be described by the following basis and inductive step.

BASIS. If the list to be sorted is empty or of length 1, just return the list; it is already sorted. The basis is taken care of by lines (1) and (2) of Fig. 2.29.

INDUCTION. If the list is of length at least 2, use the function `split` at line (3) to remove the even-numbered elements from `list` and use them to form another list, pointed to by local variable `SecondList`. Line (4) recursively sorts the half-sized lists, and returns the merger of the two lists.

```
    LIST MergeSort(LIST list)
    {
        LIST SecondList;

(1)     if (list == NULL) return NULL;
(2)     else if (list->next == NULL) return list;
        else {
            /* at least two elements on list */
(3)         SecondList = split(list);
            /* Note that as a side effect, half
               the elements are removed from list */
(4)         return merge(MergeSort(list), MergeSort(SecondList));
        }
    }
```

Fig. 2.29. The merge sort algorithm.

◆ **Example 2.25.** Let us use merge sort on the list of single-digit numbers

742897721

We again omit commas between digits for succinctness. First, the list is split into two, by the call to `split` at line (3) of `MergeSort`. One of the resulting lists consists of the odd positions, and the other the evens; that is, `list` = 72971 and `SecondList` = 4872. At line (4), these lists are sorted, resulting in lists 12779 and 2478, and then merged to produce the sorted list 122477789.

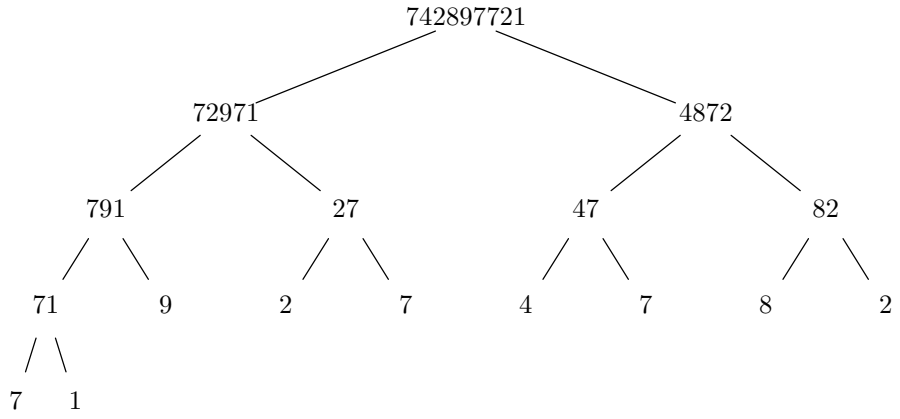
However, the sorting of the two half-sized lists does not occur by magic, but rather by the methodical application of the recursive algorithm. Initially, `MergeSort` splits the list on which it is called, if the list has length greater than 1. Figure 2.30(a) shows the recursive splitting of the lists until each list is of length 1. Then the split lists are merged, in pairs, going up the tree, until the entire list is sorted. This process is suggested in Fig. 2.30(b). However, it is worth noting that the splits and merges occur in a mixed order; not all splits are followed by all merges. For example, the first half list, 72971, is completely split and merged before we begin on the second half list, 4872. ◆

The Complete Program

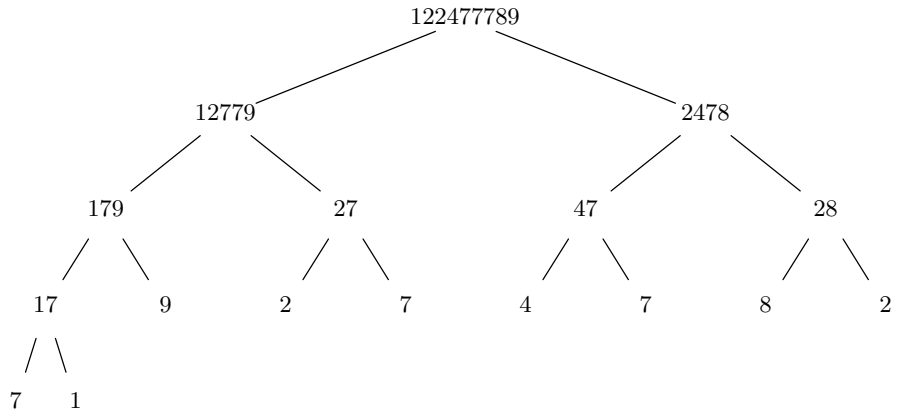
Figure 2.31 contains the complete merge sort program. It is analogous to the program in Fig. 2.3 that was based on selection sort. The function `MakeList` on line (1) reads each integer from the input and puts it into a linked list by a simple recursive algorithm, which we shall describe in detail in the next section. Line (2) of the main program contains the call to `MergeSort`, which returns a sorted list to `PrintList`. The function `PrintList` marches down the sorted list, printing each element.

EXERCISES

2.8.1: Show the result of applying the function `merge` to the lists $(1, 2, 3, 4, 5)$ and $(2, 4, 6, 8, 10)$.



(a) Splitting.



(b) Merging.

Fig. 2.30. Recursive splitting and merging.

2.8.2: Suppose we start with the list $(8, 7, 6, 5, 4, 3, 2, 1)$. Show the sequence of calls to `merge`, `split`, and `MergeSort` that result.

Multiway merge sort

2.8.3*: A *multiway* merge sort divides a list into k pieces of equal (or approximately equal) size, sorts them recursively, and then merges all k lists by comparing all their respective first elements and taking the smallest. The merge sort described in this section is for the case $k = 2$. Modify the program in Fig. 2.31 so that it becomes a multiway merge sort for the case $k = 3$.

2.8.4*: Rewrite the merge sort program to use the functions *lt* and *key*, described in Exercise 2.2.7, to compare elements of arbitrary type.

2.8.5: Relate each of the functions (a) `merge` (b) `split` (c) `MakeList` to Fig. 2.21. What is the appropriate notion of size for each of these functions?

```
     #include <stdio.h>
     #include <stdlib.h>

     typedef struct CELL *LIST;
     struct CELL {
         int element;
         LIST next;
     };

     LIST merge(LIST list1, LIST list2);
     LIST split(LIST list);
     LIST MergeSort(LIST list);
     LIST MakeList();
     void PrintList(LIST list);

     main()
     {
         LIST list;

(1)      list = MakeList();
(2)      PrintList(MergeSort(list));
     }

     LIST MakeList()
     {
         int x;
         LIST pNewCell;

(3)      if (scanf("%d", &x) == EOF) return NULL;
         else {
(4)          pNewCell = (LIST) malloc(sizeof(struct CELL));
(5)          pNewCell->next = MakeList();
(6)          pNewCell->element = x;
(7)          return pNewCell;
         }
     }

     void PrintList(LIST list)
     {
(8)      while (list != NULL) {
(9)          printf("%d\n", list->element);
(10)         list = list->next;
         }
     }
```

Fig. 2.31(a). A sorting program using merge sort (start).

```
LIST MergeSort(LIST list)
{
    LIST SecondList;

    if (list == NULL) return NULL;
    else if (list->next == NULL) return list;
    else {
        SecondList = split(list);
        return merge(MergeSort(list), MergeSort(SecondList));
    }
}

LIST merge(LIST list1, LIST list2)
{
    if (list1 == NULL) return list2;
    else if (list2 == NULL) return list1;
    else if (list1->element <= list2->element) {
        list1->next = merge(list1->next, list2);
        return list1;
    }
    else {
        list2->next = merge(list1, list2->next);
        return list2;
    }
}

LIST split(LIST list)
{
    LIST pSecondCell;

    if (list == NULL) return NULL;
    else if (list->next == NULL) return NULL;
    else {
        pSecondCell = list->next;
        list->next = pSecondCell->next;
        pSecondCell->next = split(pSecondCell->next);
        return pSecondCell;
    }
}
```

Fig. 2.31(b). A sorting program using merge sort (conclusion).

2.9 Proving Properties of Recursive Programs

If we want to prove a certain property of a recursive function, we generally need to prove a statement about the effect of one call to that function. For example, that effect might be a relationship between the arguments and the return value, such as "the function, called with argument i, returns $i!$." Frequently, we define a notion of the "size" of the arguments of a function and perform a proof by induction on this

Size of arguments

size. Some of the many possible ways in which size of arguments could be defined are

1. The value of some argument. For instance, for the recursive factorial program of Fig. 2.19, the appropriate size is the value of the argument n.

2. The length of a list pointed to by some argument. The recursive function `split` of Fig. 2.27 is an example where the length of the list is the appropriate size.

3. Some function of the arguments. For instance, we mentioned that the recursive selection sort of Fig. 2.22 performs an induction on the number of elements in the array that remain to be sorted. In terms of the arguments n and i, this function is $n - i + 1$. As another example, the appropriate size for the `merge` function of Fig. 2.24 is the sum of the lengths of the lists pointed to by the two arguments of the function.

Whatever notion of size we pick, it is essential that when a function is called with arguments of size s, it makes only function calls with arguments of size $s-1$ or less. That requirement is so we can do an induction on the size to prove a property of the program. Further, when the size falls to some fixed value — say 0 — the function must make no recursive calls. This condition is so we can start off our inductive proof with a basis case.

✦ **Example 2.26.** Consider the factorial program of Fig. 2.19 in Section 2.7. The statement to prove by induction on i, for $i \geq 1$, is

STATEMENT $S(i)$: When called with the value i for the argument `n`, `fact` returns $i!$.

BASIS. For $i = 1$, the test at line (1) of Fig. 2.19 causes the basis, line (2), to be executed. That results in the return value 1, which is 1!.

INDUCTION. Assume $S(i)$ to be true, that is, when called with some argument $i \geq 1$, `fact` returns $i!$. Now, consider what happens when `fact` is called with $i + 1$ as the value of variable `n`. If $i \geq 1$, then $i+1$ is at least 2, so the inductive case, line (3), applies. The return value is thus $n \times fact(n-1)$; or, since the variable `n` has the value $i+1$, the result $(i+1) \times fact(i)$ is returned. By the inductive hypothesis, `fact(i)` returns $i!$. Since $(i+1) \times i! = (i+1)!$, we have proved the inductive step, that `fact`, with argument $i+1$, returns $(i+1)!$. ✦

✦ **Example 2.27.** Now, let us examine the function `MakeList`, one of the auxiliary routines in Fig. 2.31(a), in Section 2.8. This function creates a linked list to hold the input elements and returns a pointer to this list. We shall prove the following statement by induction on $n \geq 0$, the number of elements in the input sequence.

STATEMENT $S(n)$: If $x_1, x_2, ..., x_n$ is the sequence of input elements, `Makelist` creates a linked list that contains $x_1, x_2, ..., x_n$ and returns a pointer to this list.

BASIS. The basis is $n = 0$, that is, when the input sequence is empty. The test for `EOF` in line (3) of `MakeList` causes the return value to be set to `NULL`. Thus, `MakeList` correctly returns an empty list.

INDUCTION. Suppose that $S(n)$ is true for $n \geq 0$, and consider what happens when `MakeList` is called on an input sequence of $n + 1$ elements. Suppose we have just read the first element x_1.

Line (4) of `MakeList` creates a pointer to a new cell c. Line (5) recursively calls `Makelist` to create, by the inductive hypothesis, a pointer to a linked list for the remaining n elements, $x_2, x_3, \ldots, x_{n+1}$. This pointer is put into the next field of c at line (5). Line (6) puts x_1 into the element field of c. Line (7) returns the pointer created by line (4). This pointer points to a linked list for the $n+1$ input elements, $x_1, x_2, \ldots, x_{n+1}$.

We have proved the inductive step and conclude that `MakeList` works correctly on all inputs. ✦

✦ **Example 2.28.** For our last example, let us prove the correctness of the merge-sort program of Fig. 2.29, assuming that the functions `split` and `merge` perform their respective tasks correctly. The induction will be on the length of the list that `MergeSort` is given as an argument. The statement to be proved by complete induction on $n \geq 0$ is

STATEMENT $S(n)$: If `list` is a list of length n when `MergeSort` is called, then `MergeSort` returns a sorted list of the same elements.

BASIS. We take the basis to be both $S(0)$ and $S(1)$. When `list` is of length 0, its value is `NULL`, and so the test of line (1) in Fig. 2.29 succeeds and the entire function returns `NULL`. Likewise, if `list` is of length 1, the test of line (2) succeeds, and the function returns `list`. Thus, `MergeSort` returns `list` when n is 0 or 1. This observation proves statements $S(0)$ and $S(1)$, because a list of length 0 or 1 is already sorted.

INDUCTION. Suppose $n \geq 1$ and $S(i)$ is true for all $i = 0, 1, \ldots, n$. We must prove $S(n + 1)$. Thus, consider a list of length $n + 1$. Since $n \geq 1$, this list is of length at least 2, so we reach line (3) in Fig. 2.29. There, `split` divides the list into two lists of length $(n + 1)/2$ if $n + 1$ is even, and of lengths $(n/2) + 1$ and $n/2$ if $n + 1$ is odd. Since $n \geq 1$, none of these lists can be as long as $n + 1$. Thus, the inductive hypothesis applies to them, and we can conclude that the half-sized lists are correctly sorted by the recursive calls to `MergeSort` at line (4). Finally, the two sorted lists are merged into one list, which becomes the return value. We have assumed that `merge` works correctly, and so the resulting returned list is sorted. ✦

```
DefCell(int, CELL, LIST);

int sum(LIST L)
{
    if (L == NULL) return 0;
    else return(L->element + sum(L->next));
}

int find0(LIST L)
{
    if (L == NULL) return FALSE;
    else if (L->element == 0) return TRUE;
    else return find0(L->next);
}
```

Fig. 2.32. Two recursive functions, `sum` and `find0`.

EXERCISES

2.9.1: Prove that the function `PrintList` in Fig. 2.31(b) prints the elements on the list that it is passed as an argument. What statement $S(i)$ do you prove inductively? What is the basis value for i?

2.9.2: The function `sum` in Fig. 2.32 computes the sum of the elements on its given list (whose cells are of the usual type as defined by the macro `DefCell` of Section 1.6 and used in the merge-sort program of Section 2.8) by adding the first element to the sum of the remaining elements; the latter sum is computed by a recursive call on the remainder of the list. Prove that `sum` correctly computes the sum of the list elements. What statement $S(i)$ do you prove inductively? What is the basis value for i?

2.9.3: The function `find0` in Fig. 2.32 returns `TRUE` if at least one of the elements on its list is 0, and returns `FALSE` otherwise. It returns `FALSE` if the list is empty, returns `TRUE` if the first element is 0 and otherwise, makes a recursive call on the remainder of the list, and returns whatever answer is produced for the remainder. Prove that `find0` correctly determines whether 0 is present on the list. What statement $S(i)$ do you prove inductively? What is the basis value for i?

2.9.4*: Prove that the functions (a) `merge` of Fig. 2.24 and (b) `split` of Fig. 2.27 perform as claimed in Section 2.8.

2.9.5: Give an intuitive "least counterexample" proof of why induction starting from a basis including both 0 and 1 is valid.

2.9.6**: Prove the correctness of (your C implementation of) the recursive GCD algorithm of Exercise 2.7.8.

2.10 Summary of Chapter 2

Here are the important ideas we should take from Chapter 2.

- Inductive proofs, recursive definitions, and recursive programs are closely related ideas. Each depends on a basis and an inductive step to "work."
- In "ordinary" or "weak" inductions, successive steps depend only on the previous step. We frequently need to perform a proof by complete induction, in which each step depends on all the previous steps.
- There are several different ways to sort. Selection sort is a simple but slow sorting algorithm, and merge sort is a faster but more complex algorithm.
- Induction is essential to prove that a program or program fragment works correctly.
- Divide-and-conquer is a useful technique for designing some good algorithms, such as merge sort. It works by dividing the problem into independent subparts and then combining the results.
- Expressions are defined in a natural, recursive way in terms of their operands and operators. Operators can be classified by the number of arguments they take: unary (one argument), binary (two arguments), and k-ary (k arguments). Also, a binary operator appearing between its operands is infix, an operator appearing before its operands is prefix, and one appearing after its operands is postfix.

2.11 Bibliographic Notes for Chapter 2

An excellent treatment of recursion is Roberts [1986]. For more on sorting algorithms, the standard source is Knuth [1973]. Berlekamp [1968] tells about techniques — of which the error detection scheme in Section 2.3 is the simplest — for detecting and correcting errors in streams of bits.

Berlekamp, E. R. [1968]. *Algebraic Coding Theory*, McGraw-Hill, New York.

Knuth, D. E. [1973]. *The Art of Computer Programming*, Vol. III: *Sorting and Searching*, Addison-Wesley, Reading, Mass.

Roberts, E. [1986]. *Thinking Recursively*, Wiley, New York.