CHAPTER 3

Recursion: The Mirrors

The goal of this chapter is to ensure that you have a basic understanding of recursion, which is one of the most powerful techniques of solution available to the computer scientist. This chapter assumes that you have had little or no previous introduction to recursion. If, however, you already have studied recursion, you can review this chapter as necessary.

By presenting several relatively simple problems, the chapter demonstrates the thought processes that lead to recursive solutions. These problems are diverse and include examples of counting, searching, and organizing data. In addition to presenting recursion from a conceptual viewpoint, this chapter discusses techniques that will help you to understand the mechanics of recursion. These techniques are particularly useful for tracing and debugging recursive methods.

Some recursive solutions are far more elegant and concise than the best of their nonrecursive counterparts. For example, the classic Towers of Hanoi problem appears to be quite difficult, yet it has an extremely simple recursive solution. On the other hand, some recursive solutions are terribly inefficient, as you will see, and should not be used.

Chapter 6 continues the formal discussion of recursion by examining more-difficult problems. Recursion will play a major role in many of the solutions that appear throughout the remainder of this book.

3.1 Recursive Solutions

Recursion is an extremely powerful problem-solving technique. Problems that at first appear to be quite difficult often have simple recursive solutions. Like top-down design, recursion breaks a problem into several smaller problems. What is striking about recursion is that these smaller problems are of *exactly the same type* as the original problem—mirror images, so to speak.

Recursion breaks a problem into smaller identical problems

Did you ever hold a mirror in front of another mirror so that the two mirrors face each other? You will see many images of yourself, each behind and slightly smaller than the other. Recursion is like these mirror images. That is, a recursive solution solves a problem by solving a smaller instance of the same problem! It solves this new problem by solving an even smaller instance of the same problem. Eventually, the new problem will be so small that its solution will be either obvious or known. This solution will lead to the solution of the original problem.

For example, suppose that you could solve problem P_1 if you had the solution to problem P_2, which is a smaller instance of P_1. Suppose further that you could solve problem P_2 if you had the solution to problem P_3, which is a smaller instance of P_2. If you knew the solution to P_3 because it was small enough to be trivial, you would be able to solve P_2. You could then use the solution to P_2 to solve the original problem P_1.

Some recursion solutions are inefficient and impractical

Recursion can seem like magic, especially at first, but as you will see, it is a very real and important problem-solving approach that is an alternative to **iteration.** An iterative solution involves loops. You should know at the outset that not all recursive solutions are better than iterative solutions. In fact, some recursive solutions are impractical because they are so inefficient. Recursion, however, can provide elegantly simple solutions to problems of great complexity.

Complex problems can have simple recursive solutions

As an illustration of the elements in a recursive solution, consider the problem of looking up a word in a dictionary. Suppose you wanted to look up the word "vademecum." Imagine starting at the beginning of the dictionary and looking at every word in order until you found "vademecum." That is precisely what a **sequential search** does, and, for obvious reasons, you want a faster way to perform the search.

One such method is the **binary search,** which in spirit is similar to the way in which you actually use a dictionary. You open the dictionary—probably to a point near its middle—and by glancing at the page, determine which "half" of the dictionary contains the desired word. The following pseudocode is a first attempt to formalize this process:

A binary search of a dictionary

```
// Search a dictionary for a word by using a recursive
// binary search

  if (the dictionary contains only one page) {
    Scan the page for the word
  }
  else {
    Open the dictionary to a point near the middle
    Determine which half of the dictionary contains the
      word
```

```
    if (the word is in the first half of the dictionary) {
      Search the first half of the dictionary for the word
    }
    else {
      Search the second half of the dictionary for the word
    }  // end if
  }  // end if
```

Parts of this solution are intentionally vague: How do you scan a single page? How do you find the middle of the dictionary? Once the middle is found, how do you determine which half contains the word? The answers to these questions are not difficult, but they will only obscure the solution strategy right now.

The previous search strategy reduces the problem of searching the dictionary for a word to a problem of searching half of the dictionary for the word, as Figure 3-1 illustrates. Notice two important points. First, once you have divided the dictionary in half, you already know how to search the appropriate half: You can use exactly the same strategy that you employed to search the original dictionary. Second, note that there is a special case that is different from all the other cases: After you have divided the dictionary so many times that you are left with only a single page, the halving ceases. At this point, the problem is sufficiently small that you can solve it directly by scanning the single page that remains for the word. This special case is called the **base case** (or **basis** or **degenerate case**).

A base case is a special case whose solution you know

This strategy is one of **divide and conquer.** You solve the dictionary search problem by first *dividing* the dictionary into two halves and then *conquering* the appropriate half. You solve the smaller problem by using the same divide-and-conquer strategy. The dividing continues until you reach the base case. As you will see, this strategy is inherent in many recursive solutions.

A binary search uses a divide-and-conquer strategy

To further explore the nature of the solution to the dictionary problem, consider a slightly more rigorous formulation.

```
search(in theDictionary:Dictionary, in aWord: string)

  if (theDictionary is one page in size) {
    Scan the page for aWord
```

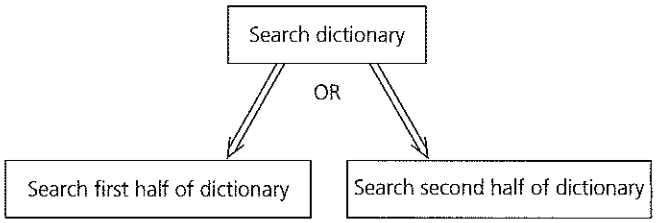


FIGURE 3-1

A recursive solution

```
}
else {
  Open theDictionary to a point near the middle
  Determine which half of theDictionary contains aWord

  if (aWord is in the first half of theDictionary) {
    search(first half of theDictionary, aWord)
  }
  else {
    search(second half of theDictionary, aWord)
  } // end if
} // end if
```

Writing the solution as a method allows several important observations:

A recursive method calls itself

1. One of the actions of the method is to call itself; that is, the method `search` calls the method `search`. This action is what makes the solution recursive. The solution strategy is to split `theDictionary` in half, determine which half contains `aWord`, and apply the same strategy to the appropriate half.

Each recursive call solves an identical, but smaller, problem

2. Each call to the method `search` made from within the method `search` passes a dictionary that is one-half the size of the previous dictionary. That is, at each successive call to `search (theDictionary, aWord)`, the size of `theDictionary` is cut in half. The method solves the search problem by solving another search problem that is identical in nature but smaller in size.

A test for the base case enables the recursive calls to stop

3. There is one search problem that you handle differently from all of the others. When `theDictionary` contains only a single page, you use another technique: You scan the page directly. Searching a one-page dictionary is the base case of the search problem. When you reach the base case, the recursive calls stop and you solve the problem directly.

Eventually, one of the smaller problems must be the base case

4. The manner in which the size of the problem diminishes ensures that you will eventually reach the base case.

These facts describe the general form of a recursive solution. Though not all recursive solutions fit these criteria as nicely as this solution does, the similarities are far greater than the differences. As you attempt to construct a new recursive solution, you should keep in mind the following four questions:

KEY CONCEPTS

Four Questions for Constructing Recursive Solutions

1. How can you define the problem in terms of a smaller problem of the same type?
2. How does each recursive call diminish the size of the problem?
3. What instance of the problem can serve as the base case?
4. As the problem size diminishes, will you reach this base case?

Now consider two relatively simple problems: computing the factorial of a number and writing a string backward. Their recursive solutions further illustrate the points raised by the solution to the dictionary search problem. These examples also illustrate the difference between a recursive valued method and a recursive `void` method.

A Recursive Valued Method: The Factorial of *n*

Consider a recursive solution to the problem of computing the factorial of an integer *n*. This problem is a good first example because its recursive solution is easy to understand and neatly fits the mold described earlier. However, because the problem has a simple and efficient iterative solution, you should not use the recursive solution in practice.

Do not use recursion if a problem has a simple, efficient iterative solution

To begin, consider the familiar iterative definition of *factorial*(*n*) (more commonly written *n*!):

$$factorial(n) = n * (n-1) * (n-2) * \cdots * 1 \text{ for any integer } n > 0$$

$$factorial(0) = 1$$

An iterative definition of factorial

The factorial of a negative integer is undefined. You should have no trouble writing an iterative factorial method based on this definition.

To define *factorial(n)* recursively, you first need to define *factorial(n)* in terms of the factorial of a smaller number. To do so, simply observe that the factorial of *n* is equal to the factorial of *(n – 1)* multiplied by *n*; that is,

$$\begin{aligned} factorial(n) &= n * [(n-1) * (n-2) * \cdots * 1] \\ &= n * factorial(n-1) \end{aligned}$$

A recurrence relation

The definition of *factorial*(*n*) in terms of *factorial*(*n* – 1), which is an example of a **recurrence relation,** implies that you can also define *factorial*(*n* – 1) in terms of *factorial*(*n* – 2), and so on. This process is analogous to the dictionary search solution, in which you search a dictionary by searching a smaller dictionary in exactly the same way.

The definition of *factorial*(*n*) lacks one key element: the base case. As was done in the dictionary search solution, here you must define one case differently from all the others, or else the recursion will never stop. The base case for the factorial method is *factorial*(0), which you know is 1. Because *n* originally is greater than or equal to zero and each call to *factorial* decrements *n* by 1, you will always reach the base case. With the addition of the base case, the complete recursive definition of the factorial method is

$$factorial(n) = \begin{cases} 1 & \text{if } n = 0 \\ n * factorial(n-1) & \text{if } n > 0 \end{cases}$$

A recursive definition of factorial

To be sure that you understand this recursive definition, apply it to the computation of *factorial*(4). Since 4 > 0, the recursive definition states that

$$factorial(4) = 4 * factorial(3)$$

Similarly,

$$factorial(3) = 3 * factorial(2)$$

$$factorial(2) = 2 * factorial(1)$$

$$factorial(1) = 1 * factorial(0)$$

You have reached the base case, and the definition directly states that

$$factorial(0) = 1$$

At this point, the application of the recursive definition stops and you still do not know the answer to the original question: What is *factorial*(4)? However, the information to answer this question is now available:

Since $factorial(0) = 1$, then $factorial(1) = 1 * 1 = 1$

Since $factorial(1) = 1$, then $factorial(2) = 2 * 1 = 2$

Since $factorial(2) = 2$, then $factorial(3) = 3 * 2 = 6$

Since $factorial(3) = 6$, then $factorial(4) = 4 * 6 = 24$

You can think of recursion as a process that divides a problem into a task that you can do and a task that a friend can do for you. For example, if I ask you to compute *factorial*(4), you could first determine whether you know the answer immediately. You know immediately that *factorial*(0) is 1—that is, you know the base case—but you do not know the value of *factorial*(4) immediately. However, if your friend computes *factorial*(3) for you, you could compute *factorial*(4) by multiplying *factorial*(3) by 4. Thus, your task will be to do this multiplication, and your friend's task will be to compute *factorial*(3).

Your friend now uses the same process to compute *factorial*(3) as you are using to compute *factorial*(4). Thus, your friend determines that *factorial*(3) is not the base case, and so asks another friend to compute *factorial*(2). Knowing *factorial*(2) enables your friend to compute *factorial*(3), and when you learn the value of *factorial*(3) from your friend, you can compute *factorial*(4).

Notice that the recursive definition of *factorial*(4) yields the same result as the iterative definition, which gives 4 * 3 * 2 * 1 = 24. To prove that the two definitions of *factorial* are equivalent for all nonnegative integers, you would use **mathematical induction.** (See Appendix D.) Chapter 5 discusses the close tie between recursion and mathematical induction.

The recursive definition of the factorial method has illustrated two points: (1) *Intuitively*, you can define *factorial*(n) in terms of *factorial*($n - 1$), and (2) *mechanically*, you can apply the definition to determine the value of a given factorial. Even in this simple example, applying the recursive definition required quite a bit of work. That, of course, is where the computer comes in.

Once you have a recursive definition of *factorial*(n), it is easy to construct a Java method that implements the definition:

```
public static int fact(int n) {
// ------------------------------------------------
```

```
// Computes the factorial of a nonnegative integer.
// Precondition: n must be greater than or equal to 0.
// Postcondition: Returns the factorial of n.
// ------------------------------------------------
   if (n == 0) {
     return 1;
   }
   else {
     return n * fact(n-1);
   }  // end if
}  // end fact
```

Suppose that you use the statement

```
System.out.println(fact(3));
```

to call the method. Figure 3-2 depicts the sequence of computations that this call would require.

Note that the *fact* method is defined as **static.** This means that *fact* is a class method; it is invoked independently of any instance of the class that contains *fact*. Instances of the class share the *fact* method, and if the static method is public, other objects can access it through the class name. For example, *java.lang.Math.sqrt* provides access to the static method *sqrt* contained in the class *java.lang.Math*. Methods that don't need access to

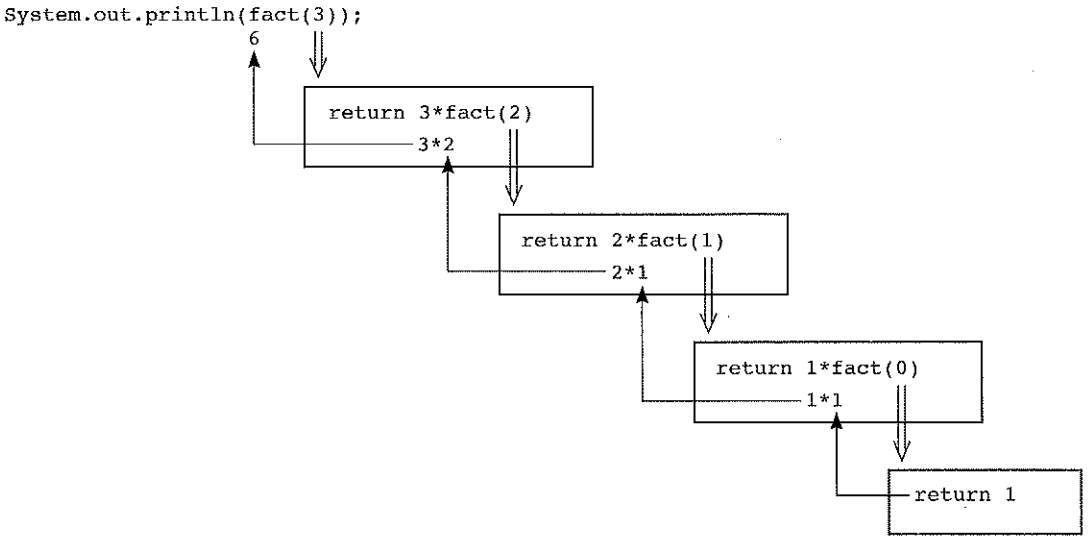


FIGURE 3-2

fact(3)

instance variables and are self-contained (except for parameter input) are good candidates to be designated as static methods. For this reason, all of the recursive methods in this chapter are declared `static`.

The `fact` method fits the model of a recursive solution given earlier in this chapter as follows:

`fact` satisfies the four criteria of a recursive solution

1. One action of `fact` is to *call itself.*
2. At each recursive call to `fact`, the integer whose factorial you need to compute is *diminished by 1.*
3. The method handles the factorial of 0 differently from all the other factorials: It does not generate a recursive call. Rather, you know that `fact(0)` is 1. Thus, the *base case* occurs when `n` is 0.
4. Given that `n` is nonnegative, item 2 of this list assures you that you will always *reach the base case.*

At an intuitive level, it should be clear that the method `fact` implements the recursive definition of *factorial.* Now consider the mechanics of executing this recursive method. The logic of `fact` is straightforward except perhaps for the expression in the `else` clause. This expression has the following effect:

1. Each operand of the product `n * fact(n-1)` is evaluated.
2. The second operand—`fact(n-1)`—is a call to the method `fact`. Although this is a recursive call (the method `fact` calls the method `fact`), there really is nothing special about it. Imagine substituting a call to another method—the Java API method `java.lang.Math.abs`, for example—for the recursive call to `fact`. The principle is the same: Simply evaluate the method.

In theory, evaluating a recursive method is no more difficult than evaluating a nonrecursive method. In practice, however, the bookkeeping can quickly get out of hand. The **box trace** is a systematic way to trace the actions of a recursive method. You can use the box trace both to help you to understand recursion and to debug recursive methods. However, such a mechanical device is no substitute for an intuitive understanding of recursion. The box trace illustrates how compilers frequently implement recursion. As you read the following description of the method, realize that each box roughly corresponds to an **activation record,** which a compiler typically uses in its implementation of a method call. Chapter 7 will discuss this implementation further.

An activation record is created for each method call

The box trace. The box trace is illustrated here for the recursive method `fact`. As you will see in the next section, this trace is somewhat simpler for a `void` method, as no value needs to be returned.

1. Label each recursive call in the body of the recursive method. Several recursive calls might occur within a method, and it will be important to distinguish among them. These labels help you to keep track of the correct place

to which you must return after a method call completes. For example, mark the expression *fact(n-1)* within the body of the method with the letter A:

```
if (n == 0) {
  return 1;
}
else {
  return n * fact(n-1);
}  // end if        (A)
```

Label each recursive call in the method

You return to point A after each recursive call, substitute the computed value for *fact(n-1)*, and continue execution by evaluating the expression *n * fact(n-1)*.

2. Represent each call to the method during the course of execution by a new box in which you note the method's **local environment.** More specifically, each box will contain

   Each time a method is called, a new box represents its local environment

   a. The values of the references and primitive types of the method's arguments.

   b. The method's local variables.

   c. A placeholder for the value returned by each recursive call from the current box. Label this placeholder to correspond to the labeling in Step 1.

   d. The value of the method itself.

   When you first create a box, you will know only the values of the input arguments. You fill in the values of the other items as you determine them from the method's execution. For example, you would create the box in Figure 3-3 for the call *fact(3)*. (You will see in later examples that you must handle reference arguments [objects] somewhat differently from value arguments [primitive types] and local variables.)

3. Draw an arrow from the statement that initiates the recursive process to the first box. Then, when you create a new box after a recursive call, as described in Step 2, you draw an arrow from the box that makes the call to the newly created box. Label each arrow to correspond to the label (from Step 1) of the recursive call; this label indicates exactly where to return after the call completes. For example, Figure 3-4 shows the first two boxes generated by the call to *fact* in the statement *System.out.println(fact(3))*.

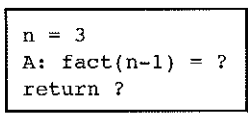


FIGURE 3-3

A box

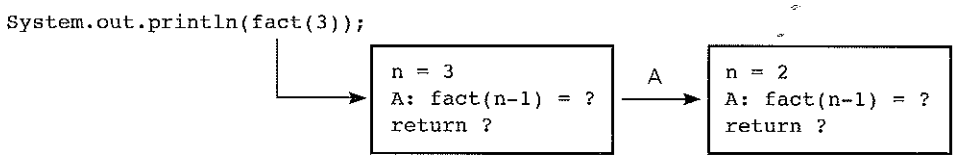


FIGURE 3-4

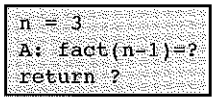
The beginning of the box trace

4. After you create the new box and arrow as described in Steps 2 and 3, start executing the body of the method. Each reference to an item in the method's local environment references the corresponding value in the current box, regardless of how you generated the current box.

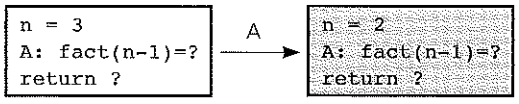
5. On exiting the method, cross off the current box and follow its arrow back to the box that called the method. This box now becomes the current box, and the label on the arrow specifies the exact location at which execution of the method should continue. Substitute the value returned by the just-terminated method call into the appropriate item in the current box.

Figure 3-5 is a complete box trace for the call *fact(3)*. In the sequence of diagrams in this figure, the current box is the deepest along the path of arrows and is shaded, whereas crossed-off boxes have a dashed outline.

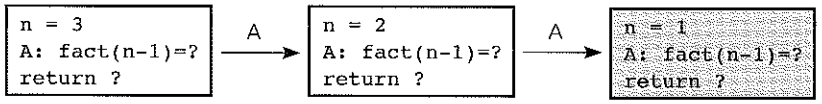
The initial call is made, and method `fact` begins execution:



At point A a recursive call is made, and the new invocation of the method `fact` begins execution:



At point A a recursive call is made, and the new invocation of the method `fact` begins execution:



At point A a recursive call is made, and the new invocation of the method `fact` begins execution:

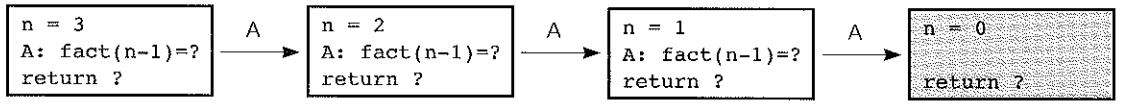


FIGURE 3-5

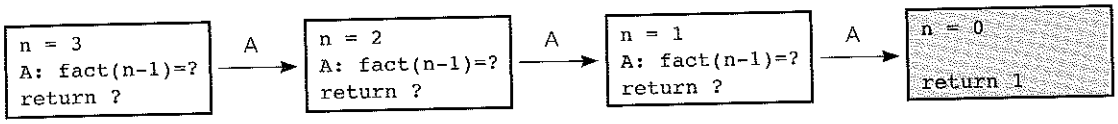
Box trace of *fact(3)*

(continues)

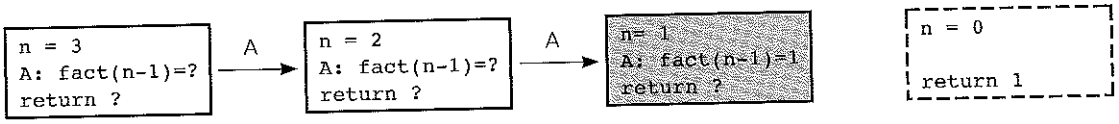
FIGURE 3-5

(continued)

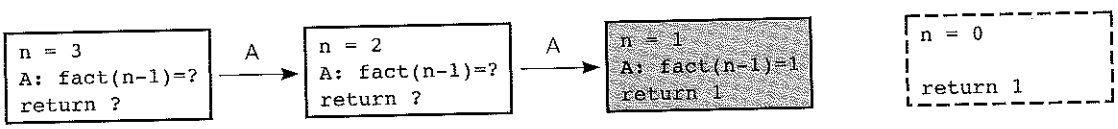
This is the base case, so this invocation of `fact` completes:



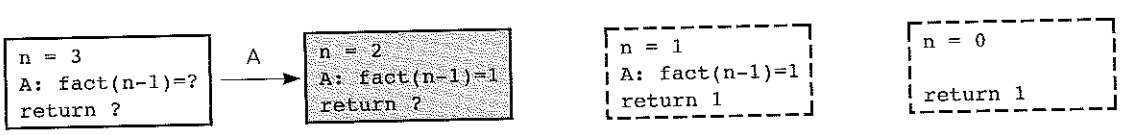
The method value is returned to the calling box, which continues execution:



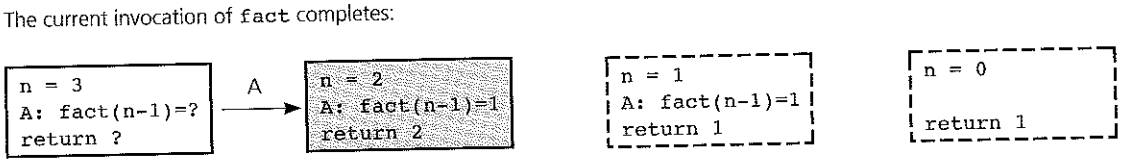
The current invocation of `fact` completes:



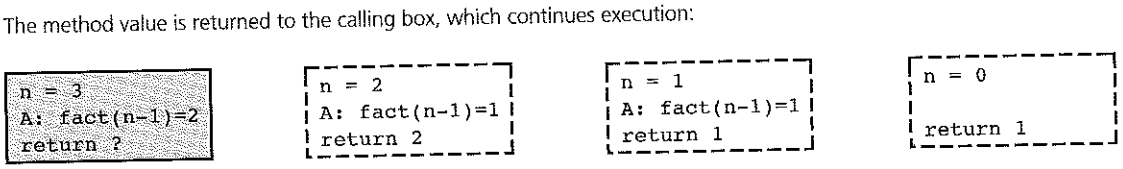
The method value is returned to the calling box, which continues execution:



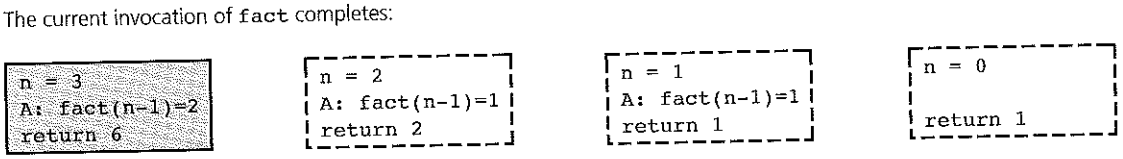
The current invocation of `fact` completes:



The method value is returned to the calling box, which continues execution:



The current invocation of `fact` completes:



The value 6 is returned to the initial call.

Invariants. Writing invariants for recursive methods is as important as writing them for iterative methods, and is often simpler. For example, consider the recursive method *fact*:

```
public static int fact(int n) {
// Precondition: n must be greater than or equal to 0.
// Postcondition: Returns the factorial of n.
  if (n == 0) {
   return 1;
  }
  else { // Invariant: n > 0, so n-1 >= 0.
         // Thus, fact(n-1) returns (n-1)!
     return n * fact(n-1);  // n * (n-1)! is n!
  } // end if
}  // end fact
```

Expect a recursive call's postcondition to be true if the precondition is true

The method requires as its precondition a nonnegative value of n. At the time of the recursive call `fact(n-1)`, n is positive, so $n - 1$ is nonnegative. Since the recursive call satisfies *fact*'s precondition, you can expect from the postcondition that `fact(n-1)` will return the factorial of $n - 1$. Therefore, `n * fact(n-1)` is the factorial of n. Chapter 6 uses mathematical induction to prove formally that `fact(n)` returns the factorial of n.

Violating **fact**'s precondition causes "infinite" recursion

If you ever violated *fact*'s precondition, the method would not behave correctly. That is, if the calling program ever passed a negative value to *fact*, an infinite sequence of recursive calls, terminated only by a system-defined limit, would occur because the method would never reach the base case. For example, `fact(-4)` would call `fact(-5)`, which would call `fact(-6)`, and so on.

The method ideally should protect itself by testing for a negative n. If $n < 0$, the method could, for example, either return zero to indicate an error or throw an exception. Chapter 2 discussed error checking in the two sections "Fail-Safe Programming" and "Style"; you might want to review that discussion at this time.

A Recursive *void* Method: Writing a String Backward

Now consider a problem that is slightly more difficult: Given a string of characters, write it in reverse order. For example, write the string "cat" as "tac". To construct a recursive solution, you should ask the four questions in the Key Concepts box on page 116.

You can construct a solution to the problem of writing a string of length n backward in terms of the problem of writing a string of length $n - 1$ backward. That is, each recursive step of the solution diminishes by 1 the length of the string to be written backward. The fact that the strings get shorter and shorter suggests that the problem of writing some very short strings backward can serve as the base case. One very short string is the empty string, the string of length zero. Thus, you can choose for the base case the problem

The base case

Write the empty string backward

The solution to this problem is to do nothing at all—a very straightforward solution indeed! (Alternatively, you could use the string of length 1 as the base case.)

How can you write an n-character string backward, if you can write an $(n-1)$-character string backward?

Exactly how can you use the solution to the problem of writing a string of length $n-1$ backward to solve the problem of writing a string of length n backward? This approach is analogous to the one used to construct the solution to the factorial problem, where you specified how to use *factorial*$(n-1)$ in the computation of *factorial*(n). Unlike the factorial problem, however, the string problem does not suggest an immediately clear way to proceed. Obviously, not any string of length $n-1$ will do. For example, there is no relation between writing "apple" (a string of length 5) backward and writing "pear" (a string of length 4) backward. You must choose the smaller problem carefully so that you can use its solution in the solution to the original problem.

The string of length $n-1$ that you choose must be a substring (part) of the original string. Suppose that you strip away one character from the original string, leaving a substring of length $n-1$. For the recursive solution to be valid, the ability to write the substring backward, combined with the ability to perform some minor task, must result in the ability to write the original string backward. Compare this approach with the way you computed *factorial* recursively: The ability to compute *factorial*$(n-1)$, combined with the ability to multiply this value by n, resulted in the ability to compute *factorial*(n).

You need to decide which character to strip away and which minor task to perform. Consider the minor task first. Since you are writing characters, a likely candidate for the minor task is writing a single character. As for the character that you should strip away from the string, there are several possible alternatives. Two of the more intuitive alternatives are

```
Strip away the last character
```

or

```
Strip away the first character
```

Consider the first of these alternatives, stripping away the last character, as Figure 3-6 illustrates.

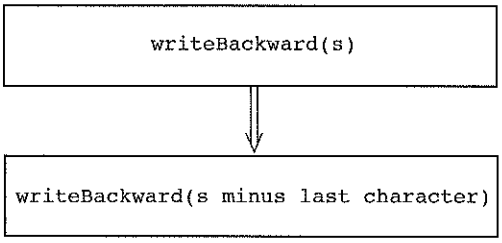


FIGURE 3-6

A recursive solution

For the solution to be valid, you must write the last character in the string first. Therefore, you must write the last character before you write the remainder of the string backward. A high-level recursive solution, given the string *s*, is

writeBackward writes a string backward

```
writeBackward(in s:string)

  if (the string s is empty) {
    Do nothing -- this is the base case
  }
  else {
    Write the last character of s
    writeBackward(s minus its last character)
  }  // end if
```

This solution to the problem is conceptual. To obtain a Java method, you must resolve a few implementation issues. Suppose that the method will receive two arguments: a string *s* to be written backward and an integer *size* that specifies the length of the string. To simplify matters, you can assume that the string begins at position 0 and ends at position *size* – 1. That is, all characters, including blanks, in that range are part of the string. The Java method *writeBackward* appears as follows:

```
public static void writeBackward(String s, int size) {
// ------------------------------------------------
// Writes a character string backward.
// Precondition: The string s contains size
// characters, where size >= 0.
// Postcondition: s is written backward, but remains
// unchanged.
// ------------------------------------------------
  if (size > 0) {
    // write the last character
    System.out.println(s.substring(size-1, size));

    // write the rest of the string backward
    writeBackward(s, size-1);  // Point A
  }  // end if
  // size == 0 is the base case - do nothing
}  // end writeBackward
```

Notice that the recursive calls to `writeBackward` use successively smaller values of `size`. This decrease in `size` has the effect of stripping away the last character of the string and ensures that the base case will be reached.

You can trace the execution of `writeBackward` by using the box trace. As was true for the method `fact`, each box contains the local environment of the recursive call—in this case, the input arguments `s` and `size`. The trace will differ some-

what from the trace of *fact* shown in Figure 3-5 because, as a *void* method, *writeBackward* does not use a *return* statement to return a computed value. Figure 3-7 traces the call to the method *writeBackward* with the string "cat".

writeBackward does not return a computed value

Now consider a slightly different approach to the problem. Recall the two alternatives for the character that you could strip away from the string: the last character or the first character. The solution just given strips away the last character of the string. It will now be interesting to construct a solution based on the second alternative:

```
Strip away the first character
```

To begin, consider a simple modification of the previous pseudocode solution that replaces each occurrence of "last" with "first." Thus, the method writes the first character rather than the last and then recursively writes the remainder of the string backward.

```
writeBackward1(in s:string)

   if (the string s is empty) {
     Do nothing -- this is the base case
   }
   else {
     Write the first character of s
     writeBackward1(s minus its first character)
   }  // end if
```

Does this solution do what you want it to? If you think about this method, you will realize that it writes the string in its normal left-to-right direction instead of backward. After all, the steps in the pseudocode are

```
Write the first character of s
Write the rest of s
```

These steps simply write the string *s*. Naming the method *writeBackward1* does not guarantee that it will actually write the string backward—recursion really is not magic!

You can write *s* backward correctly by using the following recursive formulation:

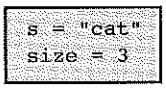
```
Write string s minus its first character backward
Write the first character of string s
```

In other words, you write the first character of *s* only *after* you have written the rest of *s* backward. This approach leads to the following pseudocode solution:

writeBackward2 writes a string backward

```
writeBackward2(in s:string)
```

The initial call is made, and the method begins execution:



Output line: **t**

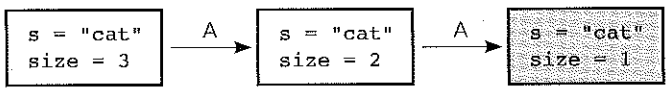
Point A (`writeBackward(s, size-1)`) is reached, and the recursive call is made.

The new invocation begins execution:

Output line: **ta**

Point A is reached, and the recursive call is made.

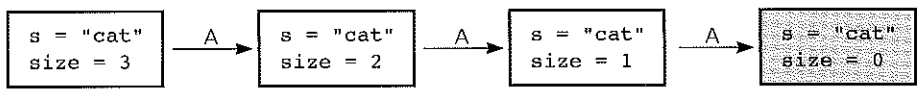
The new invocation begins execution:



Output line: **tac**

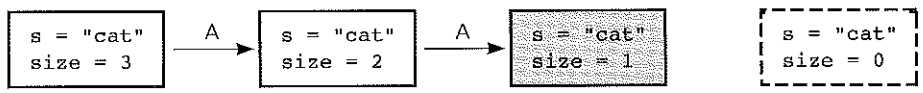
Point A is reached, and the recursive call is made.

The new invocation begins execution:



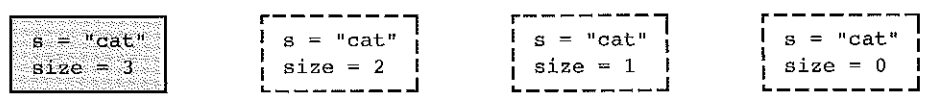
This is the base case, so this invocation completes.

Control returns to the calling box, which continues execution:



This invocation completes. Control returns to the calling box, which continues execution:

This invocation completes. Control returns to the calling box, which continues execution:



This invocation completes. Control returns to the statement following the initial call.

FIGURE 3-7

Box trace of ***writeBackward("cat", 3)***

```
if (the string s is empty) {
   Do nothing -- this is the base case
}
else {
   writeBackward2(s minus its first character)
   Write the first character of s
}  // end if
```

The translation of *writeBackward2* into Java is similar to that of the original *writeBackward* method and is left as an exercise.

It is instructive to carefully trace the actions of the two pseudocode methods *writeBackward* and *writeBackward2*. First, add statements to each method to provide output that is useful to the trace, as follows:

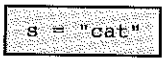
Output statements can help you trace the logic of a recursive method

```
writeBackward(in s:string)

  System.out.println("Enter writeBackward, string: " + s );
  if (the string s is empty) {
    Do nothing -- this is the base case
  }
  else {
    System.out.println("About to write last character of " +
                       "string: " + s);
    Write the last character of s
    writeBackward(s minus its last character)  // Point A
  }  // end if
  System.out.println("Leave writeBackward, string: " + s);

writeBackward2(in s:string)

  System.out.println("Enter writeBackward2, string: " + s);
  if (the string s is empty) {
    Do nothing -- this is the base case
  }
  else {
    writeBackward2(s minus its first character) // Point A
    System.out.println("About to write first character of" +
                       "string: " + s);
    Write the first character of s
  }  // end if
  System.out.println("Leave writeBackward2, string: " + s);
```

Figures 3-8 and 3-9 show the output of the revised pseudocode methods *writeBackward* and *writeBackward2*, when initially given the string "cat".

You need to be comfortable with the differences between these two methods. The recursive calls that the two methods make generate a different

The initial call is made, and the method begins execution:



Output stream:

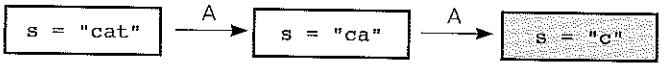
```
Enter writeBackward, string: cat
About to write last character of string: cat
t
```

Point A is reached, and the recursive call is made. The new invocation begins execution:

Output stream:

```
Enter writeBackward, string: cat
About to write last character of string: cat
t
Enter writeBackward, string: ca
About to write last character of string: ca
a
```

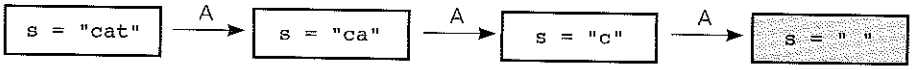
Point A is reached, and the recursive call is made. The new invocation begins execution:



Output stream:

```
Enter writeBackward, string: cat
About to write last character of string: cat
t
Enter writeBackward, string: ca
About to write last character of string: ca
a
Enter writeBackward, string: c
About to write last character of string: c
c
```

Point A is reached, and the recursive call is made. The new invocation begins execution:



This invocation completes execution, and a return is made.

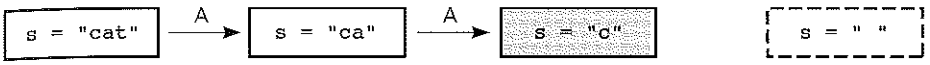
Output stream:

```
Enter writeBackward, string: cat
About to write last character of string: cat
t
```

FIGURE 3-8

Box trace of ***writeBackward("cat",3)*** in pseudocode

```
Enter writeBackward, string: ca
About to write last character of string: ca
a
Enter writeBackward, string: c
About to write last character of string: c
c
Enter writeBackward, string:
Leave writeBackward, string:
```



This invocation completes execution, and a return is made.

Output stream:

```
Enter writeBackward, string: cat
About to write last character of string: cat
t
Enter writeBackward, string: ca
About to write last character of string: ca
a
Enter writeBackward, string: c
About to write last character of string: c
c
Enter writeBackward, string:
Leave writeBackward, string:
Leave writeBackward, string: c
```

This invocation completes execution, and a return is made.

Output stream:

```
Enter writeBackward, string: cat
About to write last character of string: cat
t
Enter writeBackward, string: ca
About to write last character of string: ca
a
Enter writeBackward, string: c
About to write last character of string: c
c
Enter writeBackward, string:
Leave writeBackward, string:
Leave writeBackward, string: c
Leave writeBackward, string: ca
```

This invocation completes execution, and a return is made.

(continues)

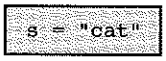
FIGURE 3-8

(continued)

Output stream:

```
Enter writeBackward, string: cat
About to write last character of string: cat
t
Enter writeBackward, string: ca
About to write last character of string: ca
a
Enter writeBackward, string: c
About to write last character of string: c
c
Enter writeBackward, string:
Leave writeBackward, string:
Leave writeBackward, string: c
Leave writeBackward, string: ca
Leave writeBackward, string: cat
```

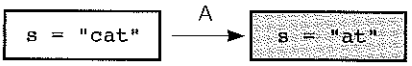
The initial call is made, and the method begins execution:



Output stream:

```
Enter writeBackward2, string: cat
```

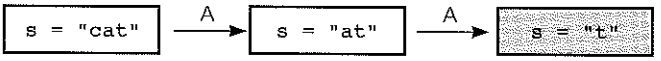
Point A is reached, and the recursive call is made. The new invocation begins execution:



Output stream:

```
Enter writeBackward2, string: cat
Enter writeBackward2, string: at
```

Point A is reached, and the recursive call is made. The new invocation begins execution:



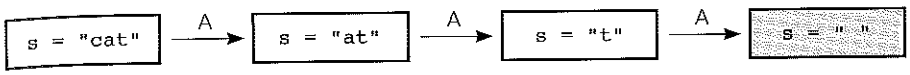
Output stream:

```
Enter writeBackward2, string: cat
Enter writeBackward2, string: at
Enter writeBackward2, string: t
```

FIGURE 3-9

Box trace of *writeBackward2("cat",3)* in pseudocode

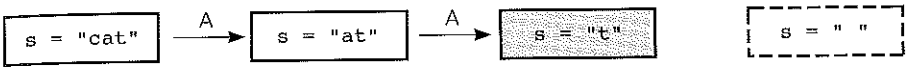
Point A is reached, and the recursive call is made. The new invocation begins execution:



This invocation completes execution, and a return is made.

Output stream:

```
Enter writeBackward2, string: cat
Enter writeBackward2, string: at
Enter writeBackward2, string: t
Enter writeBackward2, string:
Leave writeBackward2, string:
```



This invocation completes execution, and a return is made.

Output stream:

```
Enter writeBackward2, string: cat
Enter writeBackward2, string: at
Enter writeBackward2, string: t
Enter writeBackward2, string:
Leave writeBackward2, string:
About to write first character of string: t
t
Leave writeBackward2, string: t
```

This invocation completes execution, and a return is made.

Output stream:

```
Enter writeBackward2, string: cat
Enter writeBackward2, string: at
Enter writeBackward2, string: t
Enter writeBackward2, string:
Leave writeBackward2, string:
About to write first character of string: t
t
Leave writeBackward2, string: t
About to write first character of string: at
a
Leave writeBackward2, string: at
```

(continues)

FIGURE 3-9

(continued)

This invocation completes execution, and a return is made.

Output stream:

```
Enter writeBackward2, string: cat
Enter writeBackward2, string: at
Enter writeBackward2, string: t
Enter writeBackward2, string:
Leave writeBackward2, string:
About to write first character of string: t
t
Leave writeBackward2, string: t
About to write first character of string: at
a
Leave writeBackward2, string: at
About to write first character of string: cat
c
Leave writeBackward2, string: cat
```

sequence of values for the argument *s*. Despite this fact, both methods correctly write the string argument backward. They compensate for the difference in the sequence of values for *s* by writing different characters in the string at different times relative to the recursive calls. In terms of the box traces in Figures 3-8 and 3-9, *writeBackward* writes a character just before generating a new box (just before a new recursive call), whereas *writeBackward2* writes a character just after crossing off a box (just after returning from a recursive call). When these differences are put together, the result is two methods that employ different strategies to accomplish the same task.

This example also illustrates the value of the box trace, combined with well-placed *System.out.println* statements, in debugging recursive methods. The *System.out.println* statements at the beginning, interior, and end of the recursive methods report the value of the argument *s*. In general, when debugging a recursive method, you should also report both the values of local variables and the point in the method where each recursive call occurred, as in this example:

Well-placed but temporary ***System.out.println*** statements can help you to debug a recursive method

```
abc(...)

  System.out.println("Calling abc from point A.");
  abc(...)  // this is point A

  System.out.println("Calling abc from point B.");
  abc(... )  // this is point B
```

Realize that the `System.out.println` statements do not belong in the final version of the method.

Remove `System.out.println` statements after you have debugged the method

3.2 Counting Things

The next three problems require you to count certain events or combinations of events or things. They are good examples of problems with more than one base case. They also provide good examples of tremendously inefficient recursive solutions. Do not let this inefficiency discourage you. Your goal right now is to understand recursion by examining simple problems. Soon you will see useful and efficient recursive solutions.

Multiplying Rabbits (The Fibonacci Sequence)

Rabbits are very prolific breeders. If rabbits did not die, their population would quickly get out of hand. Suppose we assume the following "facts," which were obtained in a recent survey of randomly selected rabbits:

- Rabbits never die.
- A rabbit reaches sexual maturity exactly two months after birth, that is, at the beginning of its third month of life.
- Rabbits are always born in male-female pairs. At the beginning of every month, each sexually mature male-female pair gives birth to exactly one male-female pair.

Suppose you started with a single newborn male-female pair. How many pairs would there be in month 6, counting the births that took place at the beginning of month 6? Since 6 is a relatively small number, you can figure out the solution easily:

Month 1:	1 pair, the original rabbits.
Month 2:	1 pair still, since it is not yet sexually mature.
Month 3:	2 pairs; the original pair has reached sexual maturity and has given birth to a second pair.
Month 4:	3 pairs; the original pair has given birth again, but the pair born at the beginning of month 3 is not yet sexually mature.
Month 5:	5 pairs; all rabbits alive in month 3 (2 pairs) are now sexually mature. Add their offspring to those pairs alive in month 4 (3 pairs) to yield 5 pairs.
Month 6:	8 pairs; 3 newborn pairs from the pairs alive in month 4 plus 5 pairs alive in month 5.

You can now construct a recursive solution for computing *rabbit*(n), the number of pairs alive in month n. You must determine how you can use *rab-*

bit(n – 1) to compute *rabbit*(n). Observe that *rabbit*(n) is the sum of the number of pairs alive just prior to the start of month n and the number of pairs born at the start of month n. Just prior to the start of month n, there are *rabbit*(n – 1) pairs of rabbits. Not all of these rabbits are sexually mature at the start of month n. Only those who were alive in month n – 2 are ready to reproduce at the start of month n. That is, the number of pairs born at the start of month n is *rabbit*(n – 2). Therefore, you have the recurrence relation

$$rabbit(n) = rabbit(n-1) + rabbit(n-2)$$

Figure 3-10 illustrates this relationship.

This recurrence relation introduces a new point. In some cases, you solve a problem by solving more than one smaller problem of the same type. This change does not add much conceptual difficulty, but you must be very careful when selecting the base case. The temptation is simply to say that *rabbit*(1) should be the base case because its value is 1 according to the problem's statement. But what about *rabbit*(2)? Applying the recursive definition to *rabbit*(2) would yield

$$rabbit(2) = rabbit(1) + rabbit(0)$$

Thus, the recursive definition would need to specify the number of pairs alive in month 0—an undefined quantity.

Two base cases are necessary because there are two smaller problems of the same type

One possible solution is to define *rabbit*(0) to be 0, but this approach seems artificial. A slightly more attractive alternative is to treat *rabbit*(2) itself as a special case with the value of 1. Thus, the recursive definition has two base cases, *rabbit*(2) and *rabbit*(1). The recursive definition becomes

$$rabbit(n) = \begin{cases} 1 & \text{if } n \text{ is 1 or 2} \\ rabbit(n-1) + rabbit(n-2) & \text{if } n > 2 \end{cases}$$

Incidentally, the series of numbers *rabbit*(1), *rabbit*(2), *rabbit*(3), and so on is known as the **Fibonacci sequence,** which models many naturally occurring phenomena.

A Java method to compute *rabbit*(n) is easy to write from the previous definition:

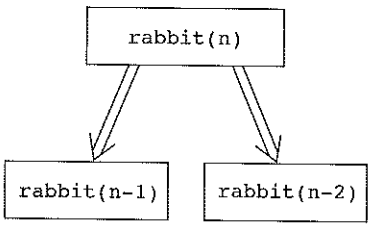


FIGURE 3-10

Recursive solution to the rabbit problem

```
public static int rabbit(int n) {
// ------------------------------------------------
// Computes a term in the Fibonacci sequence.
// Precondition: n is a positive integer.
// Postcondition: Returns the nth Fibonacci number.
// ------------------------------------------------
   if (n <= 2) {
      return 1;
   }
   else  { // n > 2, so n-1 > 0 and n-2 > 0
      return rabbit(n-1) + rabbit(n-2);
   }  // end if
}  // end rabbit
```

- `rabbit` computes the Fibonacci sequence but does so inefficiently

Should you actually use this method? Figure 3-11 illustrates the recursive calls that `rabbit(7)` generates. Think about the number of recursive calls that `rabbit(10)` generates. At best, the method `rabbit` is inefficient. Thus, its use is not feasible for large values of n. This problem is discussed in more detail at the end of this chapter, at which time you will see some techniques for generating a more efficient solution from this same recursive relationship.

Organizing a Parade

You have been asked to organize the Fourth of July parade, which will consist of bands and floats in a single line. Last year, adjacent bands tried to outplay each other. To avoid this problem, the sponsors have asked you never to place one band immediately after another. In how many ways can you organize a parade of length n?

Assume that you have at least n marching bands and n floats from which to choose. When counting the number of ways to organize the parade, assume that the sequences *band-float* and *float-band*, for example, are different entities and count as two ways.

The parade can end with either a float or a band. The number of ways to organize the parade is simply the sum of the number of parades of each type. That is, let

$P(n)$ be the number of ways to organize a parade of length n

$F(n)$ be the number of parades of length n that end with a float

$B(n)$ be the number of parades of length n that end with a band

Then

$$P(n) = F(n) + B(n)$$

First, consider $F(n)$. You will have a parade of length n that ends with a float simply by placing a float at the end of *any* acceptable parade of length $n - 1$. Hence, the number of acceptable parades of length n that end with a float is

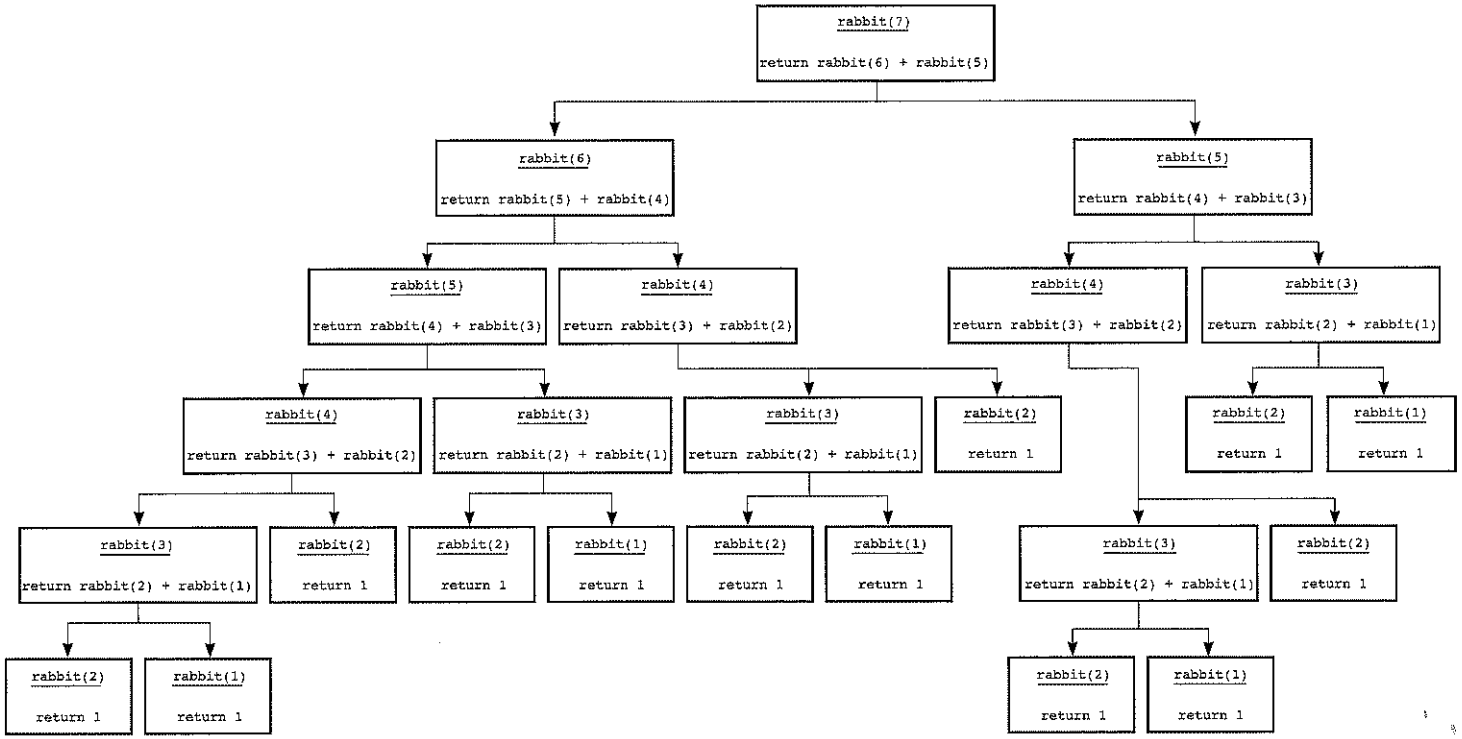


FIGURE 3-11

The recursive calls that *rabbit(7)* generates

precisely equal to the total number of acceptable parades of length $n - 1$; that is,

The number of acceptable parades of length n that end with a float

$$F(n) = P(n-1)$$

Next, consider $B(n)$. The only way a parade can end with a band is if the unit just before the end is a float. (If it is a band, you will have two adjacent bands.) Thus, the only way to organize an acceptable parade of length n that ends with a band is first to organize a parade of length $n - 1$ that ends with a float and then add a band to the end. Therefore, the number of acceptable parades of length n that end with a band is precisely equal to the number of acceptable parades of length $n - 1$ that end with a float:

The number of acceptable parades of length n that end with a band

$$B(n) = F(n-1)$$

You use the earlier fact that $F(n) = P(n-1)$ to obtain

$$B(n) = P(n-2)$$

Thus, you have solved $F(n)$ and $B(n)$ in terms of the smaller problems $P(n-1)$ and $P(n-2)$, respectively. You then use

$$P(n) = F(n) + B(n)$$

to obtain

$$P(n) = P(n-1) + P(n-2)$$

The number of acceptable parades of length n

The form of this recurrence relation is identical to the solution for the multiplying rabbits problem.

As you saw in the rabbit problem, two base cases are necessary because the recurrence relation defines a problem in terms of two smaller problems. As you did for the rabbit problem, you can choose $n = 1$ and $n = 2$ for the base cases. Although both problems use the same n's for their base cases, there is no reason to expect that they use the same values for these base cases. That is, there is no reason to expect that *rabbit*(1) is equal to $P(1)$ and that *rabbit*(2) is equal to $P(2)$.

A little thought reveals that for the parade problem,

Two base cases are necessary because there are two smaller problems of the same type

$P(1) = 2$ (The parades of length 1 are *float* and *band*.)

$P(2) = 3$ (The parades of length 2 are *float-float*, *band-float*, and *float-band*.)

In summary, the solution to this problem is

A recursive solution

$$P(1) = 2$$

$$P(2) = 3$$

$$P(n) = P(n-1) + P(n-2) \quad \text{for } n > 2$$

This example demonstrates the following points about recursion:

- Sometimes you can solve a problem by breaking it up into cases—for example, parades that end with a float and parades that end with a band.
- The values that you use for the base cases are extremely important. Although the recurrence relations for P and *rabbit* are the same, the different values for their base cases ($n = 1$ or 2) cause different values for larger values of n. For example, *rabbit*(20) = 6,765, while $P(20) = 17{,}711$. The larger the value of n, the larger the discrepancy. You should think about why this is so.

Mr. Spock's Dilemma (Choosing k out of n Things)

The five-year mission of the *U.S.S. Enterprise* is to explore new worlds. The five years are almost up, but the *Enterprise* has just entered an unexplored solar system that contains n planets. Unfortunately, time will allow for visits to only k planets. Mr. Spock begins to ponder how many different choices are possible for exploring k planets out of the n planets in the solar system. Because time is short, he does not care about the order in which he visits the same k planets.

Mr. Spock is especially fascinated by one particular planet, Planet X. He begins to think—in terms of Planet X—about how to pick k planets out of the n. "There are two possibilities: Either we visit Planet X, or we do not visit Planet X. If we do visit Planet X, I will have to choose $k - 1$ other planets to visit from the $n - 1$ remaining planets. On the other hand, if we do not visit Planet X, I will have to choose k planets to visit from the remaining $n - 1$ planets."

Mr. Spock is on his way to a recursive method of counting how many groups of k planets he can possibly choose out of n. Let $c(n, k)$ be the number of groups of k planets chosen from n. Then, in terms of Planet X, Mr. Spock deduces that

$c(n, k) =$ (the number of groups of k planets that include Planet X)

\+

(the number of groups of k planets that do not include Planet X)

But Mr. Spock has already reasoned that the number of groups that include Planet X is $c(n - 1, k - 1)$, and the number of groups that do not include Planet X is $c(n - 1, k)$. Mr. Spock has figured out a way to solve his counting problem in terms of two smaller counting problems of the same type:

The number of ways to choose k out of n things is the sum of the number of ways to choose $k - 1$ out of $n - 1$ things and the number of ways to choose k out of $n - 1$ things

$$c(n, k) = c(n - 1, k - 1) + c(n - 1, k)$$

Mr. Spock now has to worry about the base case(s). He also needs to demonstrate that each of the two smaller problems eventually reaches a base case. First, what selection problem does he immediately know the answer to? If the *Enterprise* had time to visit all the planets (that is, if $k = n$), no decision

would be necessary; there is only one group of all the planets. Thus, the first base case is

$$c(k, k) = 1$$

Base case: There is one group of everything

If $k < n$, it is easy to see that the second term in the recursive definition, $c(n - 1, k)$, is "closer" to the base case $c(k, k)$ than is $c(n, k)$. However, the first term, $c(n - 1, k - 1)$, is not closer to $c(k, k)$ than is $c(n, k)$—they are the same "distance" apart. *When you solve a problem by solving two (or more) smaller problems, each of the smaller problems must be closer to a base case than the original problem.*

Mr. Spock realizes that the first term does, in fact, approach another trivial selection problem. This problem is the counterpart of his first base case, $c(k, k)$. Just as there is only one group of all the planets ($k = n$), there is also only one group of zero planets ($k = 0$). When there is no time to visit any of the planets, the *Enterprise* must head home without any exploration. Thus, the second base case is

$$c(n, 0) = 1$$

Base case: There is one group of nothing

This base case does indeed have the property that $c(n - 1, k - 1)$ is closer to it than is $c(n, k)$. (Alternatively, you could define the second base case to be $c(n, 1) = n$.)

Mr. Spock adds one final part to his solution:

$$c(n, k) = 0 \quad \text{if } k > n$$

Although k could not be greater than n in the context of this problem, the addition of this case makes the recursive solution more generally applicable.

To summarize, the following recursive solution solves the problem of choosing k out of n things:

$$c(n, k) = \begin{cases} 1 & \text{if } k = 0 \\ 1 & \text{if } k = n \\ 0 & \text{if } k > n \\ c(n-1, k-1) + c(n-1, k) & \text{if } 0 < k < n \end{cases}$$

The number of groups of k things recursively chosen out of n things

You can easily derive the following method from this recursive definition:

```
public static int c(int n, int k) {
// ------------------------------------------------
// Computes the number of groups of k out of n things.
// Precondition: n and k are nonnegative integers.
// Postcondition: Returns c(n, k).
// ------------------------------------------------
  if ( (k == 0) || (k == n) ) {
     return 1;
```

```
    }
    else if (k > n) {
       return 0;
    }
    else {
       return c(n-1, k-1) + c(n-1, k);
    }  // end if
}  // end c
```

Like the *rabbit* method, this method is inefficient and not practical to use. Figure 3-12 shows the number of recursive calls that the computation of *c(4, 2)* requires.

3.3 Searching an Array

Searching is an important task that occurs frequently. This chapter began with an intuitive approach to a binary search algorithm. This section develops the binary search and examines other searching problems that have recursive solutions. The goal is to develop further your notion of recursion.

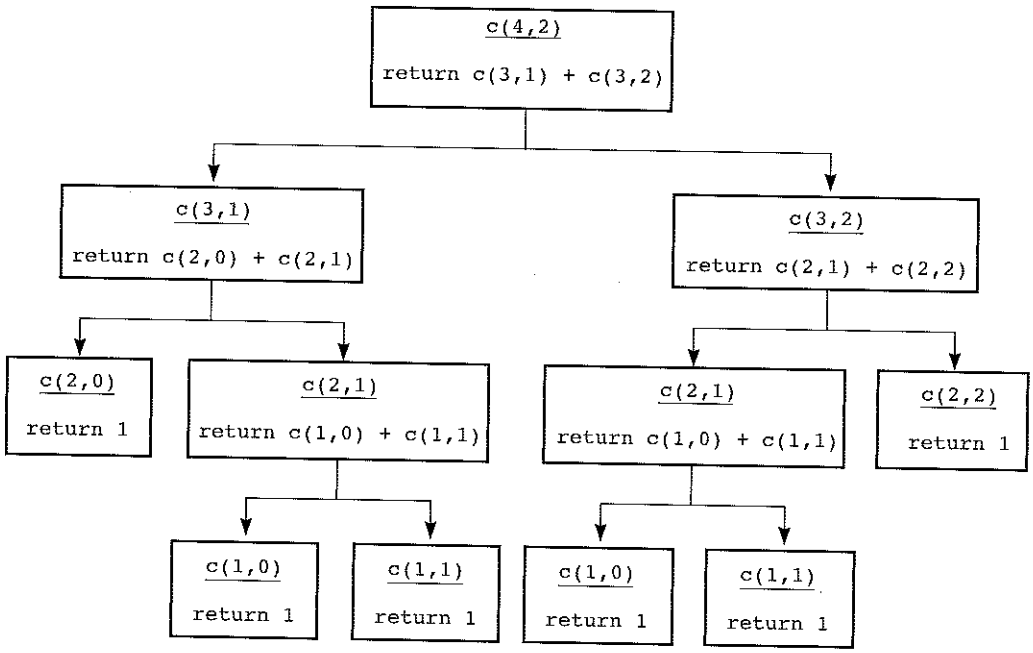


FIGURE 3-12

The recursive calls that *c(4, 2)* generates

Finding the Largest Item in an Array

Suppose that you have an array *anArray* of integers and you want to find the largest one. You could construct an iterative solution without too much difficulty, but instead consider a recursive formulation:

```
if (anArray has only one item) {
   maxArray(anArray) is the item in anArray
}
else if (anArray has more than one item) {
   maxArray(anArray) is the maximum of
      maxArray(left half of anArray) and
      maxArray(right half of anArray)
}  // end if
```

Notice that this strategy fits the divide-and-conquer model that the binary search algorithm used at the beginning of this chapter. That is, the algorithm proceeds by dividing the problem and conquering the subproblems, as Figure 3-13 illustrates. However, there is a difference between this algorithm and the binary search algorithm. While the binary search algorithm conquers only one of its subproblems at each step, *maxArray* conquers both. In addition, after *maxArray* conquers the subproblems, it must reconcile the two solutions—that is, it must find the maximum of the two maximums. Figure 3-14 illustrates the computations that are necessary to find the largest integer in the array that contains 1, 6, 8, and 3 (denoted here by <1, 6, 8, 3>).

maxArray conquers both of its subproblems at each step

You should develop a recursive solution based on this strategy. In so doing, you may stumble on several subtle programming issues. The binary search problem that follows raises virtually all of these issues, but this is a good opportunity for you to get some practice implementing a recursive solution.

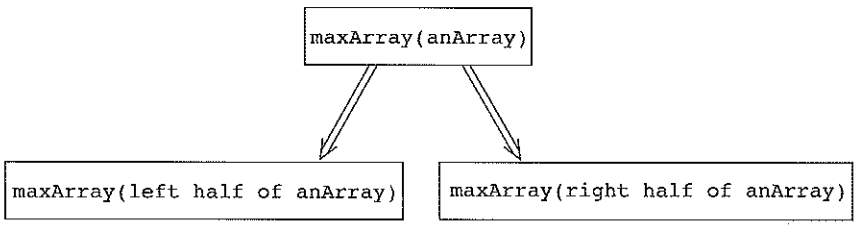


FIGURE 3-13

Recursive solution to the largest-item problem

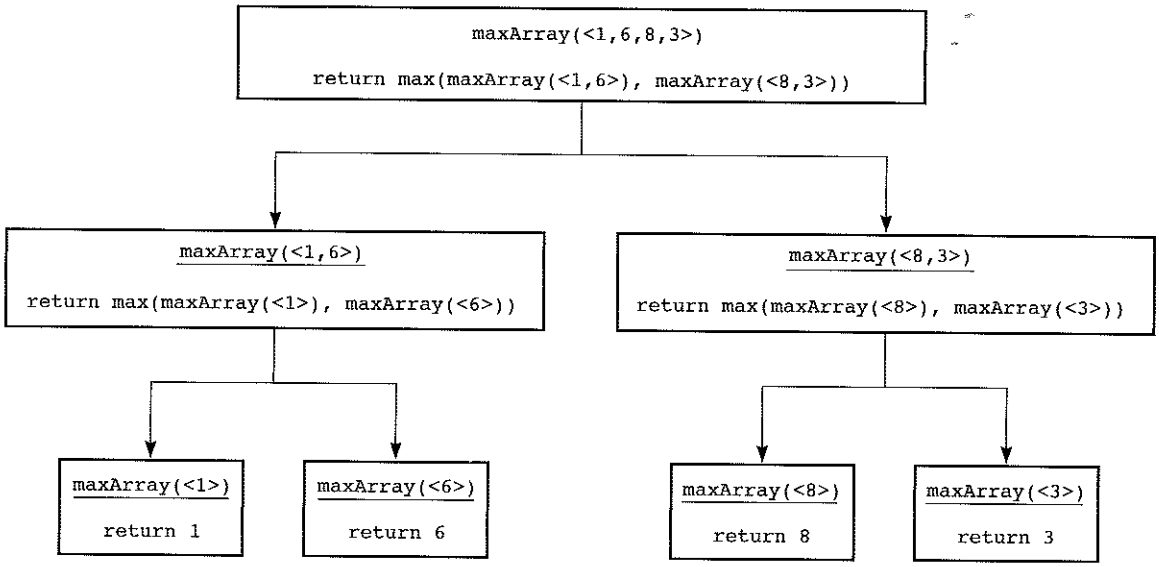


FIGURE 3-14

The recursive calls that *maxArray(<1,6,8,3>)* generates

Binary Search

The beginning of this chapter presented—at a high level—a recursive binary search algorithm for finding a word in a dictionary. We now develop this algorithm fully and illustrate some important programming issues.

Recall the earlier solution to the dictionary problem:

```
search(in theDictionary:Dictionary, in aWord: string)
  if (theDictionary is one page in size) {
    Scan the page for aWord
  }
  else {
    Open theDictionary to a point near the middle
    Determine which half of theDictionary contains
        aWord
    if (aWord is in first half of theDictionary) {
      search(first half of theDictionary, aWord)
    }
    else {
      search(second half of theDictionary, aWord)
    } // end if
  } // end if
```

Now alter the problem slightly by searching an array *anArray* of integers for a given value. The array, like the dictionary, must be sorted, or else a binary search is not applicable. Hence, assume that

An array must be sorted before you can apply a binary search to it

$$anArray[0] \leq anArray[1] \leq anArray[2] \leq \cdots \leq anArray[size-1]$$

where *size* is the size of the array. A high-level binary search for the array problem is

```
binarySearch(in anArray:ArrayType, in value:ItemType)

  if (anArray is of size 1) {
    Determine if anArray's item is equal to value
  }
  else {
    Find the midpoint of anArray
    Determine which half of anArray contains value
    if (value is in the first half of anArray) {
      binarySearch(first half of anArray, value)
    }
    else {
      binarySearch(second half of anArray, value)
    }  // end if
  }  // end if
```

Although the solution is conceptually sound, you must consider several details before you can implement the algorithm:

1. **How will you pass "half of *anArray*" to the recursive calls to *binarySearch*?** You can pass the entire array at each call but have *binarySearch* search only *anArray[first..last]*,[1] that is, the portion *anArray[first]* through *anArray[last]*. Thus, you would also pass the integers *first* and *last* to *binarySearch*:

   ```
   binarySearch(anArray, first, last, value)
   ```

   With this convention, the new midpoint is given by

   ```
   mid = (first + last)/2
   ```

   The array halves are ***anArray[first..mid-1]*** and ***anArray[mid+1..last]***; neither half contains ***anArray[mid]***

   Then *binarySearch(first half of anArray, value)* becomes

   ```
   binarySearch(anArray, first, mid-1, value)
   ```

1. You will see this notation in the rest of the book to represent a portion of an array.

and `binarySearch(second half of anArray, value)` becomes

```
binarySearch(anArray, mid+1, last, value)
```

2. **How do you determine which half of the array contains `value`?** One possible implementation of

```
if (value is in the first half of anArray)
```

is

```
if (value < anArray[mid])
```

However, there is no test for equality between `value` and `anArray[mid]`. This omission can cause the algorithm to miss `value`. After the previous halving algorithm splits `anArray` into halves, `anArray[mid]` is not in either half of the array. (In this case, two halves do not make a whole!) Therefore, you must determine whether `anArray[mid]` is the value you seek *now* because later it will not be in the remaining half of the array. The interaction between the halving criterion and the termination condition (the base case) is subtle and is often a source of error. We need to rethink the base case.

Determine whether `anArray[mid]` is the value you seek

3. **What should the base case(s) be?** As it is written, `binarySearch` terminates only when an array of size 1 occurs; this is the only base case. By changing the halving process so that `anArray[mid]` *remains in one of the halves,* it is possible to implement the binary search correctly so that it has only this single base case. However, it can be clearer to have two distinct base cases as follows:

Two base cases

   a. `first > last`. You will reach this base case when `value` is not in the original array.

   b. `value == anArray[mid]`. You will reach this base case when `value` is in the original array.

   These base cases are a bit different from any you have encountered previously. In a sense, the algorithm determines the answer to the problem from the base case it reaches. Many search problems have this flavor.

3. **How will `binarySearch` indicate the result of the search?** If `binarySearch` successfully locates `value` in the array, it could return the index of the array item that is equal to `value`. Since this index would never be negative, `binarySearch` could return a negative value if it does not find `value` in the array.

The Java method `binarySearch` that follows implements these ideas. The two recursive calls to `binarySearch` are labeled as X and Y for use in a later box trace of this method.

```
public static int binarySearch(int anArray[], int first,
                               int last, int value) {
// Searches the array items anArray[first] through
// anArray[last] for value by using a binary search.
// Precondition: 0 <= first, last <= SIZE-1, where
// SIZE is the maximum size of the array, and
// anArray[first] <= anArray[first+1] <= ... <=
// anArray[last].
// Postcondition: If value is in the array, the method
// returns the index of the array item that equals value;
// otherwise the method returns -1.
  int index;
  if (first > last) {
    index = -1;       // value not in original array
  }
  else {
    // Invariant: If value is in anArray,
    //            anArray[first] <= value <= anArray[last]
    int mid = (first + last)/2;
    if (value == anArray[mid]) {
      index = mid;  // value found at anArray[mid]
    }
    else if (value < anArray[mid]) {
     // point X
      index = binarySearch(anArray, first, mid-1, value);
    }
    else {
     // point Y
      index = binarySearch(anArray, mid+1, last, value);
    }  // end if
  }  // end if

  return index;
}  // end binarySearch
```

Notice that *binarySearch* has the following invariant: If *value* occurs in the array, then *anArray[first]* ≤ *value* ≤ *anArray[last]*.

Figure 3-15 shows box traces of *binarySearch* when it searches the array containing 1, 5, 9, 12, 15, 21, 29, and 31. Notice how the labels *X* and *Y* of the two recursive calls to *binarySearch* appear in the diagram. Exercise 16 at the end of this chapter asks you to perform other box traces with this method.

There is another implementation issue—one that deals specifically with Java—to consider. Recall that an array is an object, and when the method *binarySearch* is called, only the reference to the array is copied to the method, not the entire array contents. This aspect of Java is particularly useful in a recursive method such as *binarySearch*. If the array *anArray* is large,

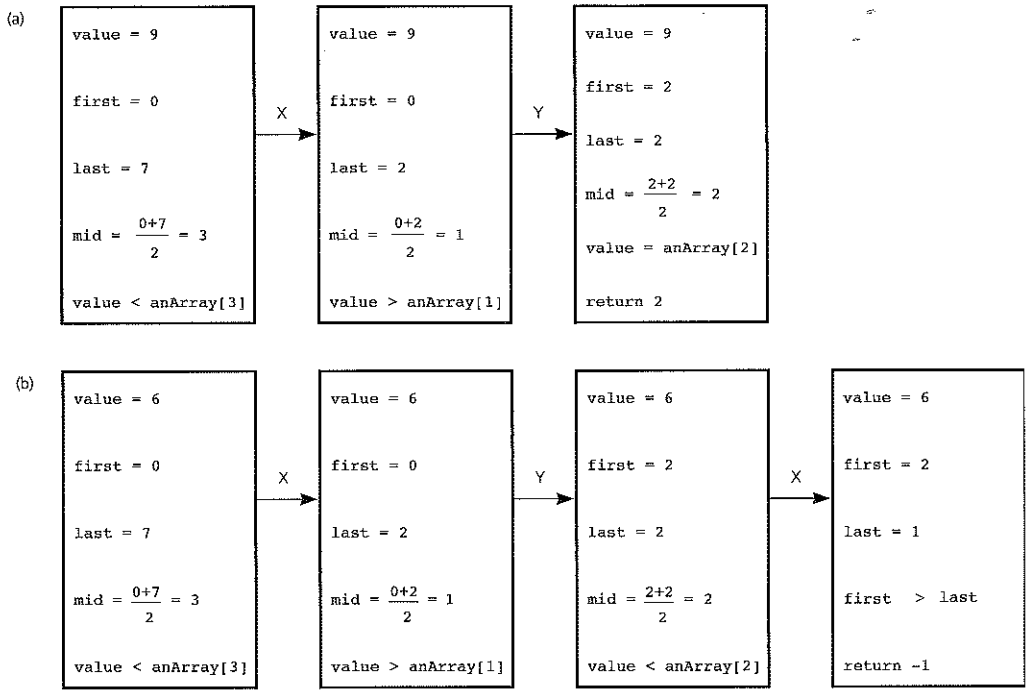


FIGURE 3-15

Box traces of `binarySearch` with `anArray = <1,5,9,12,15,21,29,31>`: (a) a successful search for 9; (b) an unsuccessful search for 6

many recursive calls to `binarySearch` may be necessary. If each call copied `anArray`, much memory and time would be wasted.

A box trace of a recursive method that has an array argument requires a new consideration. Because only the reference to `anArray` is passed and it is not a local variable, the contents of the array are not a part of the method's local environment and should not appear within each box. Therefore, as Figure 3-16 shows, you represent `anArray` outside the boxes, and all references to `anArray` affect this single representation.

Finding the k^{th} Smallest Item in an Array

Our discussion of searching concludes with a more difficult problem. Although you could skip this example now, Chapter 10 uses aspects of it in a sorting algorithm.

The previous two examples presented recursive methods for finding the largest item in an arbitrary array and for finding an arbitrary item in a sorted array. This example describes a recursive solution for finding the k^{th} smallest item in an arbitrary array `anArray`. Would you ever be interested in such an

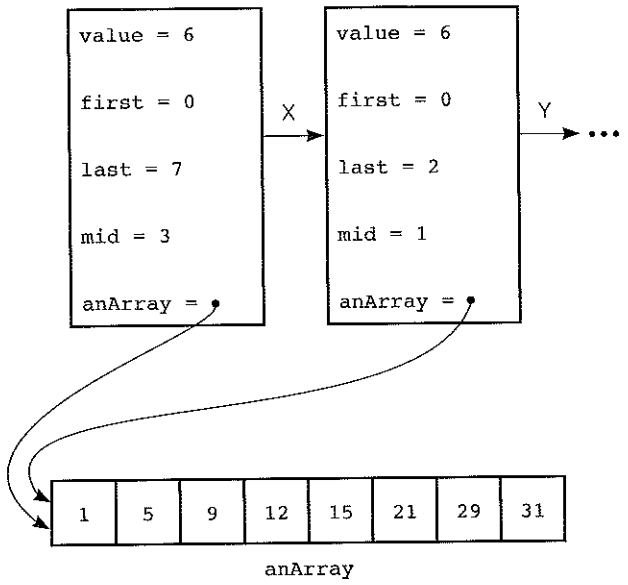


FIGURE 3-16

Box trace with a reference to an array

item? Statisticians often want the median value in a collection of data. The median value in an ordered collection of data occurs in the middle of the collection. In an unordered collection of data, there are about the same number of values smaller than the median value as there are larger values. Thus, if you have 49 items, the 25th smallest item is the median value.

Obviously, you could solve this problem by sorting the array. Then the k^{th} smallest item would be `anArray[k-1]`. Although this approach is a legitimate solution, it does more than the problem requires; a more efficient solution is possible. The solution outlined here finds the k^{th} smallest item without completely sorting the array.

By now, you know that you solve a problem recursively by writing its solution in terms of one or more smaller problems of the same type in such a way that this notion of *smaller* ensures that you will always reach a base case. For all of the earlier recursive solutions, the reduction in problem size between recursive calls is *predictable*. For example, the factorial method always decreases the problem size by 1; the binary search always halves the problem size. In addition, the base cases for all the previous problems except the binary search have a static, predefined size. Thus, by knowing only the size of the original problem, you can determine the number of recursive calls that are necessary before you reach the base case.

For all previous examples, you know the amount of reduction made in the problem size by each recursive call

The solution that you are about to see for finding the k^{th} smallest item departs from these traditions. Although you solve the problem in terms of a smaller problem, just how much smaller this problem is depends on the items in the array and cannot be predicted in advance. Also, the size of the base

You cannot predict in advance the size of either the smaller problems or the base case in the recursive solution to the k^{th} smallest item problem

case depends on the items in the array, as it did for the binary search. (Recall that you reach a base case for a binary search when the middle item is the one sought.)

This "unpredictable" type of solution is caused by the nature of the problem: The relationship between the rankings of the items in any predetermined parts of the array and the ranking of the items in the entire array is not strong enough to determine the k^{th} smallest item. For example, suppose that `anArray` contains the items shown in Figure 3-17. Notice that 6, which is in `anArray[3]`, is the third smallest item in the first half of `anArray` and that 8, which is in `anArray[4]`, is the third smallest item in the second half of `anArray`. Can you conclude from these observations anything about the location of the third smallest item in all of `anArray`? The answer is no; these facts about parts of the array do not allow you to draw any useful conclusions about the entire array. You should experiment with other fixed splitting schemes as well.

The recursive solution proceeds by

1. Selecting a **pivot item** in the array
2. Cleverly arranging, or **partitioning,** the items in the array about this pivot item
3. Recursively applying the strategy to *one* of the partitions

Consider the details of the recursive solution: You want to find the k^{th} smallest item in the array segment `anArray[first..last]`. Let the pivot p be any item of the array segment. (For now, ignore how to choose p.) You can partition the items of `anArray[first..last]` into three regions: S_1, which contains the items less than p; the pivot p itself; and S_2, which contains the items greater than or equal to p. This partition implies that all the items in S_1 are smaller than all the items in S_2. Figure 3-18 illustrates this partition.

Partition *anArray* into three parts: items $< p$, p, and items $\geq p$

All items in `anArray[first..pivotIndex-1]`, in terms of array subscripts, are less than p, and all items in `anArray[pivotIndex+1..last]` are greater than or equal to p. Notice that the sizes of the regions S_1 and S_2 depend on both p and the other items of `anArray[first..last]`.

This partition induces three "smaller problems," such that the solution to one of the problems will solve the original problem:

1. If S_1 contains k or more items, S_1 contains the k smallest items of the array segment `anArray[first..last]`. In this case, the k^{th} smallest item must

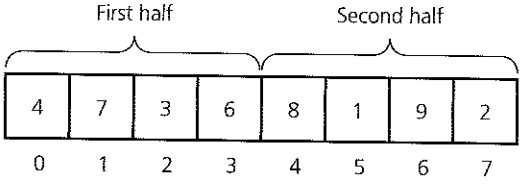


FIGURE 3-17

A sample array

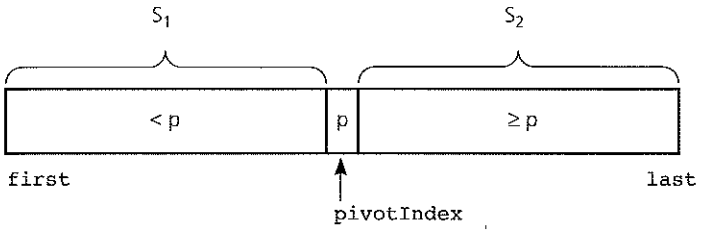


FIGURE 3-18

A partition about a pivot

be in S_1. Since S_1 is the array segment `anArray [first..pivotIndex-1]`, this case occurs if $k <$ `pivotIndex - first` + 1.

2. If S_1 contains $k - 1$ items, the k^{th} smallest item must be the pivot p. This is the base case; it occurs if $k =$ `pivotIndex - first` + 1.
3. If S_1 contains fewer than $k - 1$ items, the k^{th} smallest item in `anArray[first..last]` must be in S_2. Because S_1 contains `pivotIndex - first` items, the k^{th} smallest item in `anArray [first..last]` is the $(k -$ (`pivotIndex - first` + 1))th smallest item in S_2. This case occurs if $k >$ `pivotIndex - first` + 1.

A recursive definition can summarize this discussion. Let

`kSmall(k, anArray, first, last)` = k^{th} smallest item in `anArray[first..last]`

After you select the pivot item p and partition `anArray[first..last]` into S_1 and S_2, you have that

$$\texttt{kSmall(k, anArray, first, last)} = \begin{cases} \texttt{kSmall(k, anArray, first, pivotIndex-1)} & \text{if } k < \texttt{pivotIndex} - \texttt{first} + 1 \\ p & \text{if } k = \texttt{pivotIndex} - \texttt{first} + 1 \\ \texttt{kSmall(k-(pivotIndex-first+1), anArray, pivotIndex+1, last)} & \text{if } k > \texttt{pivotIndex} - \texttt{first} + 1 \end{cases}$$

The k^{th} smallest item in `anArray [first..last]`

There is always a pivot, and since it is not part of either S_1 or S_2, the size of the array segment to be searched decreases by at least 1 at each step. Thus, you will eventually reach the base case: The desired item is a pivot. A high-level pseudocode solution is as follows:

```
kSmall(in k:integer, in anArray:ArrayType, in first:integer,
       in last:integer)
// Returns the k^th smallest value in anArray[first..last].

   Choose a pivot item p from anArray[first..last]
   Partition the items of anArray[first..last] about p
```

```
if (k < pivotIndex - first + 1) {
   return kSmall(k, anArray, first, pivotIndex-1)
}
else if (k == pivotIndex - first + 1) {
   return p
}
else {
   return kSmall(k-(pivotIndex-first+1), anArray,
                 pivotIndex+1, last)
}  // end if
```

This pseudocode is not far from a Java method. The only questions that remain are how to choose the pivot item *p* and how to partition the array about the chosen *p*. The choice of *p* is arbitrary. Any *p* in the array will work, although the sequence of choices will affect how soon you reach the base case. Chapter 10 gives an algorithm for partitioning the items about *p*. There you will see how to turn the method `kSmall` into a sorting algorithm.

3.4 Organizing Data

Given some data organized in one way, you might need to organize the data in another way. Thus, you will actually change some aspect of the data and not, for example, simply search it. The problem in this section is called the Towers of Hanoi. Although this classic problem probably has no direct real-world application, we consider it because its solution so well illustrates the use of recursion.

The Towers of Hanoi

Many, many years ago, in a distant part of the Orient—in the Vietnamese city of Hanoi—the Emperor's wiseperson passed on to join his ancestors. The Emperor needed a replacement wiseperson. Being a rather wise person himself, the Emperor devised a puzzle, declaring that its solver could have the job of wiseperson.

The Emperor's puzzle consisted of *n* disks (he didn't say exactly how many) and three poles: *A* (the source), *B* (the destination), and *C* (the spare). The disks were of different sizes and had holes in the middle so that they could fit on the poles. Because of their great weight, the disks could be placed only on top of disks larger than themselves. Initially, all the disks were on pole *A*, as shown in Figure 3-19a. The puzzle was to move the disks, one by one, from pole *A* to pole *B*. A person could also use pole *C* in the course of the transfer, but again a disk could be placed only on top of a disk larger than itself.

As the position of wiseperson was generally known to be a soft job, there were many applicants. Scholars and peasants alike brought the Emperor their solutions. Many solutions were thousands of steps long, and many contained `goto`'s. "I can't understand these solutions," bellowed the Emperor. "There must be an easy way to solve this puzzle."

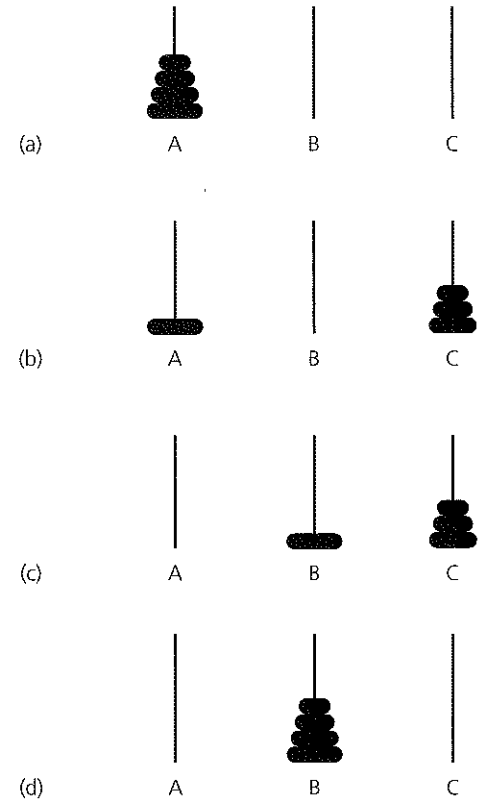


FIGURE 3-19

(a) The initial state; (b) move $n - 1$ disks from A to C; (c) move one disk from A to B; (d) move $n - 1$ disks from C to B

And indeed there was. A great Buddhist monk came out of the mountains to see the Emperor. "My son," he said, "the puzzle is so easy, it almost solves itself." The Emperor's security chief wanted to throw this strange person out, but the Emperor let him continue.

"If you have only one disk (that is, $n = 1$), move it from pole A to pole B." So far, so good, but even the village idiot got that part right. "If you have more than one disk (that is, $n > 1$), simply

1. "Ignore the bottom disk and solve the problem for $n - 1$ disks, with the small modification that pole C is the destination and pole B is the spare." (See Figure 3-19b.)
2. "After you have done this, $n - 1$ disks will be on pole C, and the largest disk will remain on pole A. So solve the problem for $n = 1$ (recall that even the village idiot could do this) by moving the large disk from A to B." (See Figure 3-19c.)

3. "Now all you have to do is move the $n - 1$ disks from pole C to pole B; that is, solve the problem with pole C as the source, pole B as the destination, and pole A as the spare." (See Figure 3-19d.)

There was silence for a few moments, and finally the Emperor said impatiently, "Well, are you going to tell us your solution or not?" The monk simply gave an all-knowing smile and vanished.

The Emperor obviously was not a recursive thinker, but you should realize that the monk's solution is perfectly correct. The key to the solution is the observation that you can solve the Towers problem of n disks by solving three smaller—in the sense of number of disks—Towers problems. Let `towers(count, source, destination, spare)` denote the problem of moving `count` disks from pole `source` to pole `destination`, using pole `spare` as a spare. Notice that this definition makes sense even if there are more than `count` disks on pole `source`; in this case, you concern yourself with only the top `count` disks and ignore the others. Similarly, the poles `destination` and `spare` might have disks on them before you begin; you ignore these, too, except that you may place only smaller disks on top of them.

The problem statement

You can restate the Emperor's problem as follows: Beginning with n disks on pole A and 0 disks on poles B and C, solve `towers(n, A, B, C)`. You can state the monk's solution as follows:

The solution

Step 1. Starting in the initial state—with all the disks on pole A—solve the problem

```
towers(n-1, A, C, B)
```

That is, ignore the bottom (largest) disk and move the top $n - 1$ disks from pole A to pole C, using pole B as a spare. When you are finished, the largest disk will remain on pole A, and all the other disks will be on pole C.

Step 2. Now, with the largest disk on pole A and all others on pole C, solve the problem

```
towers(1, A, B, C)
```

That is, move the largest disk from pole A to pole B. Because this disk is larger than the disks already on the spare pole C, you really could not use the spare. However, fortunately—and obviously—you do not need to use the spare in this base case. When you are done, the largest disk will be on pole B and all other disks will remain on pole C.

Step 3. Finally, with the largest disk on pole B and all the other disks on pole C, solve the problem

```
towers(n-1, C, B, A)
```

That is, move the $n - 1$ disks from pole C to pole B, using A as a spare. Notice that the destination pole B already has the largest disk, which you ignore. When you are done, you will have solved the original problem: All the disks will be on pole B.

The problem *towers(count, source, destination, spare)* has the following pseudocode solution:

```
solveTowers(in count:integer, in source:Pole,
            in destination:Pole, in spare:Pole)

  if (count is 1) {
     Move a disk directly from source to destination
  }
   else {
     solveTowers(count-1, source, spare, destination)
     solveTowers(1, source, destination, spare)
     solveTowers(count-1, spare, destination, source)
   }  // end if
```

This recursive solution follows the same basic pattern as the recursive solutions you saw earlier in this chapter:

The solution to the Towers problem satisfies the four criteria of a recursive solution

1. You solve a Towers problem by solving other Towers problems.
2. These other Towers problems are smaller than the original problem; they have fewer disks to move. In particular, the number of disks decreases by 1 at each recursive call.
3. When a problem has only one disk—the base case—the solution is easy to solve directly.
4. The way that the problems become smaller ensures that you will reach a base case.

Solving the Towers problem requires you to solve many smaller Towers problems recursively. Figure 3-20 illustrates the resulting recursive calls and their order when you solve the problem for three disks.

Now consider a Java implementation of this algorithm. Notice that since most computers do not have arms (at the time of this writing), the method moves a disk by giving directions to a human. Thus, the formal parameters that represent the poles are of type *char*, and the corresponding actual arguments could be *'A'*, *'B'*, and *'C'*. The call *solveTowers(3, 'A', 'B', 'C')* produces this output:

The solution for three disks

```
Move top disk from pole A to pole B
Move top disk from pole A to pole C
Move top disk from pole B to pole C
Move top disk from pole A to pole B
Move top disk from pole C to pole A
Move top disk from pole C to pole B
Move top disk from pole A to pole B
```

The Java method follows:

```
public static void solveTowers(int count, char source,
                               char destination, char spare) {
  if (count == 1) {
    System.out.println("Move top disk from pole " + source +
                       " to pole " + destination);
  }
  else {
    solveTowers(count-1, source, spare, destination); // X
    solveTowers(1, source, destination, spare);       // Y
    solveTowers(count-1, spare, destination, source); // Z
  }  // end if
}  // end solveTowers
```

The three recursive calls in the method are labeled *X*, *Y*, and *Z*. These labels appear in the box trace of `solveTowers(3, 'A', 'B', 'C')` in Figure 3-21. The recursive calls are also numbered to correspond to the numbers used in Figure 3-20. (Figure 3-21 abbreviates `destination` as `dest` to save space.)

3.5 Recursion and Efficiency

Recursion is a powerful problem-solving technique that often produces very clean solutions to even the most complex problems. Recursive solutions can be easier to understand and to describe than iterative solutions. By using recursion, you can often write simple, short implementations of your solution.

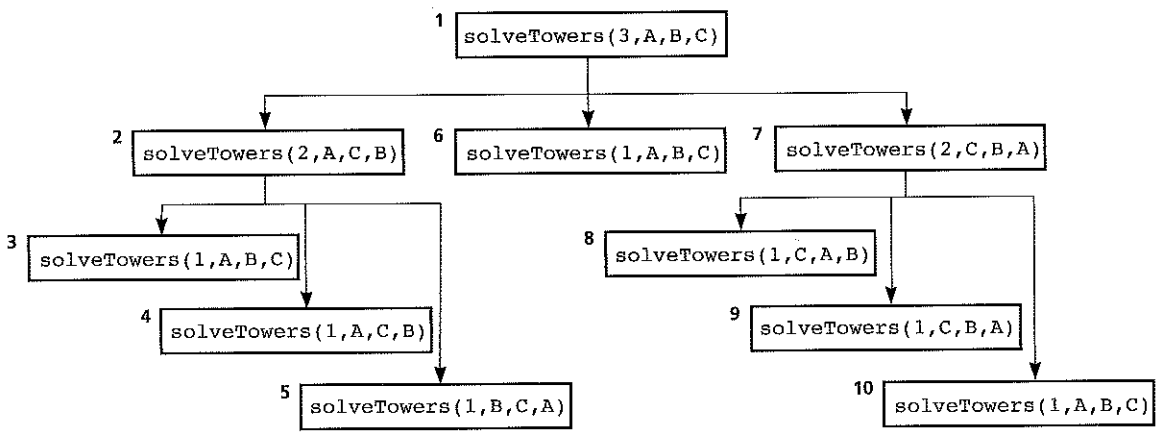
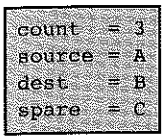


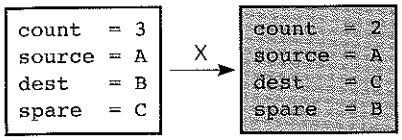
FIGURE 3-20

The order of recursive calls that results from `solveTowers(3, A, B, C)`

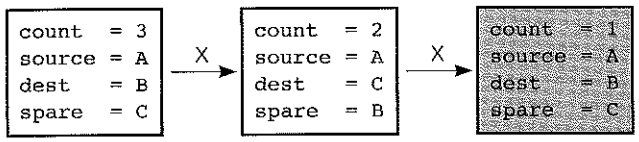
The initial call 1 is made, and `solveTowers` begins execution:



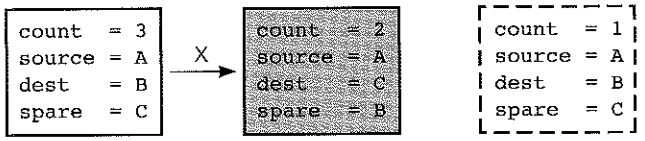
At point X, recursive call 2 is made, and the new invocation of the method begins execution:



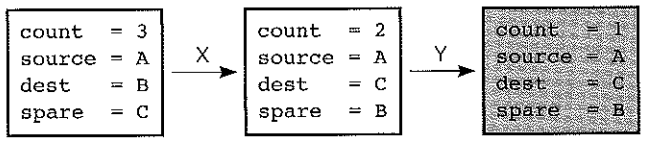
At point X, recursive call 3 is made, and the new invocation of the method begins execution:



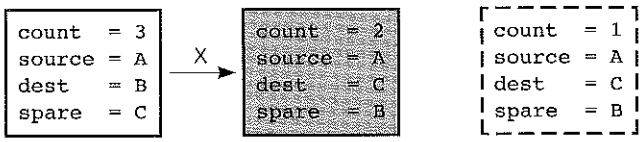
This is the base case, so a disk is moved, the return is made, and the method continues execution.



At point Y, recursive call 4 is made, and the new invocation of the method begins execution:



This is the base case, so a disk is moved, the return is made, and the method continues execution.



At point Z, recursive call 5 is made, and the new invocation of the method begins execution:

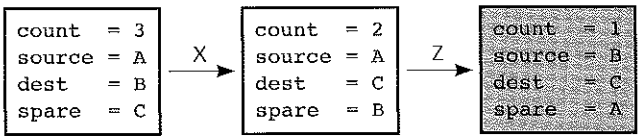
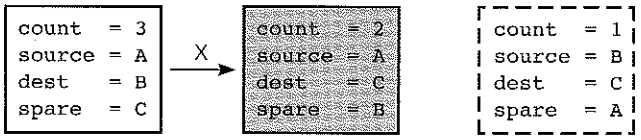


FIGURE 3-21

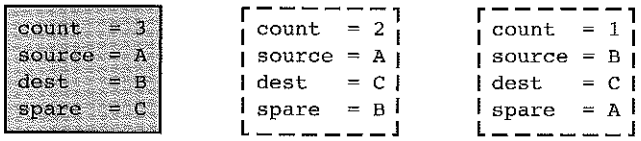
Box trace of *solveTowers(3, 'A', 'B', 'C')* (continues)

(continued)

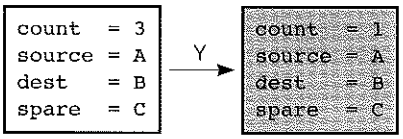
This is the base case, so a disk is moved, the return is made, and the method continues execution.



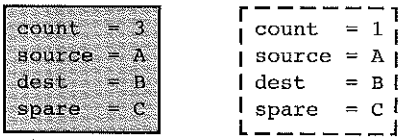
This invocation completes, the return is made, and the method continues execution.



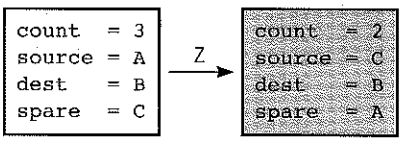
At point Y, recursive call 6 is made, and the new invocation of the method begins execution:



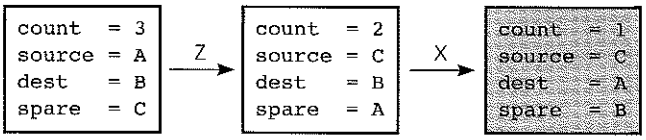
This is the base case, so a disk is moved, the return is made, and the method continues execution.



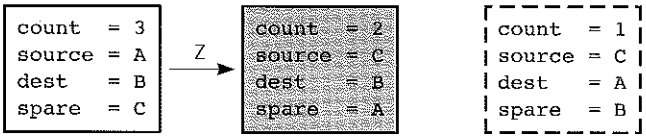
At point Z, recursive call 7 is made, and the new invocation of the method begins execution:



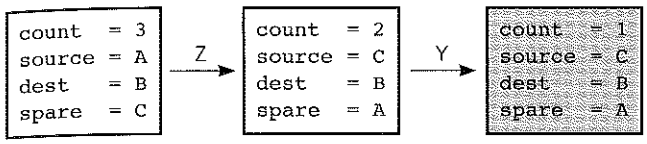
At point X, recursive call 8 is made, and the new invocation of the method begins execution:



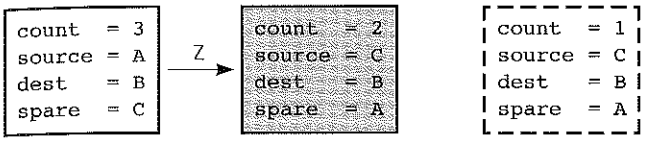
This is the base case, so a disk is moved, the return is made, and the method continues execution.



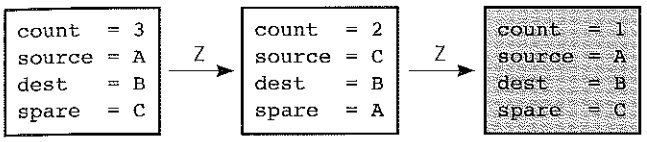
At point Y, recursive call 9 is made, and the new invocation of the method begins execution:



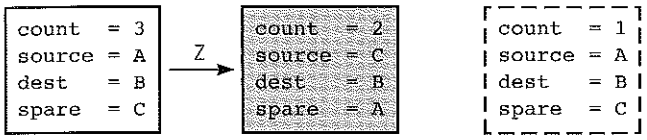
This is the base case, so a disk is moved, the return is made, and the method continues execution.



At point Z, recursive call 10 is made, and the new invocation of the method begins execution:



This is the base case, so a disk is moved, the return is made, and the method continues execution.



This invocation completes, the return is made, and the method continues execution.

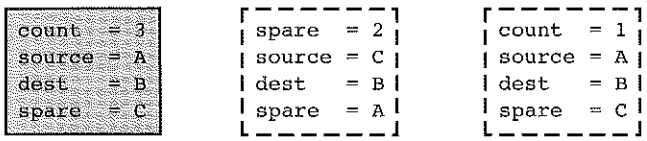


FIGURE 3-21

The overriding concern of this chapter has been to give you a solid understanding of recursion so that you will be able to construct recursive solutions on your own. Most of our examples, therefore, have been simple. Unfortunately, many of the recursive solutions in this chapter are so inefficient that you should not use them. The recursive methods *binarySearch* and *solveTowers* are the notable exceptions, as they are quite efficient.[2]

Two factors contribute to the inefficiency of some recursive solutions:

- The overhead associated with method calls
- The inherent inefficiency of some recursive algorithms

Factors that contribute to the inefficiency of some recursive solutions

2. Chapters 6 and 10 present other practical, efficient applications of recursion.

The first of these factors does not pertain specifically to recursive methods but is true of methods in general. In most implementations of Java and other high-level programming languages, a method call incurs a bookkeeping overhead. As was mentioned earlier, each method call produces an activation record, which is analogous to a box in the box trace. Recursive methods magnify this overhead because a single initial call to the method can generate a large number of recursive calls. For example, the call `fact(n)` generates n recursive calls. On the other hand, the use of recursion, as is true with modularity in general, can greatly clarify complex programs. This clarification frequently more than compensates for the additional overhead. Thus, the use of recursion is often consistent with the multidimensional view of the cost of a computer program, as Chapter 2 describes.

Recursion can clarify complex solutions

However, you should not use recursion just for the sake of using recursion. For example, you probably should not use the recursive factorial method in practice. You easily can write an iterative factorial method given the iterative definition that was stated earlier in this chapter. The iterative method is almost as clear as the recursive one and is more efficient. There is no reason to incur the overhead of recursion when its use does not gain anything. *Recursion is truly valuable when a problem has no simple iterative solutions.*

Do not use a recursive solution if it is inefficient and you have a clear, efficient iterative solution

The second point about recursion and efficiency is that some recursive algorithms are inherently inefficient. This inefficiency is a very different issue than that of overhead. It has nothing to do with how a compiler happens to implement a recursive method but rather is related to the method of solution that the algorithm employs.

As an example, recall the recursive solution for the multiplying rabbits problem that you saw earlier in this chapter:

$$rabbit(n) = \begin{cases} 1 & \text{if } n \text{ is 1 or 2} \\ rabbit(n-1) + rabbit(n-2) & \text{if } n > 2 \end{cases}$$

The recursive version of **`rabbit`** is inherently inefficient

The diagram in Figure 2-11 illustrated the computation of *rabbit(7)*. Earlier, you were asked to think about what the diagram would look like for *rabbit(10)*. If you thought about this question, you may have come to the conclusion that such a diagram would fill up most of this chapter. The diagram for *rabbit(100)* would fill up most of this universe!

The fundamental problem with *rabbit* is that it computes the same values over and over again. For example, in the diagram for *rabbit(7)*, you can see that *rabbit(3)* is computed five times. When n is moderately large, many of the values are recomputed literally trillions of times. This enormous number of computations makes the solution infeasible, even if each computation required only a trivial amount of work (for example, if you could perform 100 million of these computations per second).

However, do not conclude that the recurrence relation is of no use. One way to solve the rabbit problem is to construct an iterative solution based on this same recurrence relation. The iterative solution goes forward instead of backward and computes each value only once. You can use the following iterative method to compute *rabbit(n)* even for very large values of n.

You can use **`rabbit`**'s recurrence relation to construct an efficient iterative solution

```
public static int iterativeRabbit(int n) {
// Iterative solution to the rabbit problem.
  // initialize base cases:
  int previous = 1;   // initially rabbit(1)
  int current = 1;    // initially rabbit(2)
  int next = 1;       // result when n is 1 or 2

  // compute next rabbit values when n >= 3
  for (int i = 3; i <= n; i++) {
    // current is rabbit(i-1), previous is rabbit(i-2)
    next = current + previous;  // rabbit(i)

    previous = current;         // get ready for
    current = next;             // next iteration
  }  // end for

  return next;
}  // end iterativeRabbit
```

Convert from recursion to iteration if it is easier to discover a recursive solution but more efficient to use an iterative solution

Thus, an iterative solution can be more efficient than a recursive solution. In certain cases, however, it may be easier to discover a recursive solution than an iterative solution. Therefore, you may need to convert a recursive solution to an iterative solution. This conversion process is easier if your recursive method calls itself once, instead of several times. Be careful when deciding whether your method calls itself more than once. Although the method *rabbit* calls itself twice, the method *binarySearch* calls itself once, even though you see two calls in the Java code. Those two calls appear within an *if* statement; only one of them will be executed.

A tail-recursive method

Converting a recursive solution to an iterative solution is even easier when the solitary recursive call is the last *action* that the method takes. This situation is called **tail recursion.** For example, the method *writeBackward* exhibits tail recursion because its recursive call is the last action that the method takes. Before you conclude that this is obvious, consider the method *fact*. Although its recursive call appears last in the method definition, *fact*'s last action is the multiplication. Thus, *fact* is not tail recursive.

Recall the definition of *writeBackward*:

```
public static void writeBackward(String s, int size) {
  if (size > 0) {
    // write the last character
    System.out.println(s.substring(size-1, size));
    writeBackward(s, size - 1);      // write rest
  }  // end if
}  // end writeBackward
```

Because this method is tail recursive, its last recursive call simply repeats the method's action with altered arguments. You can perform this repetitive action

Removing tail recursion is often straightforward

by using an iteration that will be straightforward and often more efficient. For example, the following definition of *writeBackward* is iterative:

```
public static void writeBackward(String s, int size) {
// Iterative version.
   while (size > 0) {
      System.out.println(s.substring(size-1, size));
      --size;
   } // end while
} // end writeBackward
```

Because tail-recursive methods are often less efficient than their iterative counterparts and because the conversion of a tail-recursive method to an equivalent iterative method is rather mechanical, some compilers automatically replace tail recursion with iteration. Eliminating other forms of recursion is usually more complex, as you will see in Chapter 7, and is a task that *you* would need to undertake, if necessary.

Some recursive algorithms, such as *rabbit*, are inherently inefficient, while other recursive algorithms, such as the binary search,[3] are extremely efficient. You will learn how to determine the relative efficiency of a recursive algorithm in more advanced courses concerned with the analysis of algorithms. Chapter 10 introduces some of these techniques briefly.

Chapter 6 will continue the discussion of recursion by examining several difficult problems that have straightforward recursive solutions. Other chapters in this book use recursion as a matter of course.

Summary

1. Recursion is a technique that solves a problem by solving a smaller problem of the same type.
2. When constructing a recursive solution, keep the following four questions in mind:
   a. How can you define the problem in terms of a smaller problem of the same type?
   b. How does each recursive call diminish the size of the problem?
   c. What instance of the problem can serve as the base case?
   d. As the problem size diminishes, will you reach this base case?
3. When constructing a recursive solution, you should assume that a recursive call's postcondition is true if its precondition is true.
4. You can use the box trace to trace the actions of a recursive method. These boxes resemble activation records, which many compilers use to implement recursion. (Chapter 6 discusses implementing recursion further.) Although the box trace is useful, it cannot replace an intuitive understanding of recursion.

3. The binary search algorithm also has an iterative formulation.

5. Recursion allows you to solve problems—such as the Towers of Hanoi—whose iterative solutions are difficult to conceptualize. Even the most complex problems often have straightforward recursive solutions. Such solutions can be easier to understand, describe, and implement than iterative solutions.

6. Some recursive solutions are much less efficient than a corresponding iterative solution, due to their inherently inefficient algorithms and the overhead of method calls. In such cases, the iterative solution can be preferable. You can use the recursive solution, however, to derive the iterative solution.

7. If you can easily, clearly, and efficiently solve a problem by using iteration, you should do so.

Cautions

1. A recursive algorithm must have a base case, whose solution you know directly without making any recursive calls. Without a base case, a recursive method will generate an infinite sequence of calls. When a recursive method contains more than one recursive call, you will often need more than one base case.
2. A recursive solution must involve one or more smaller problems that are each closer to a base case than is the original problem. You must be sure that these smaller problems eventually reach the base case. Failure to do so could result in an algorithm that does not terminate.
3. When developing a recursive solution, you must be sure that the solutions to the smaller problems really do give you a solution to the original problem. For example, ***binarySearch*** works because each smaller array is sorted and the value sought is between its first and last items.
4. The box trace, together with well-placed ***System.out.println*** statements, can be a good aid in debugging recursive methods. Such statements should report the point in the program from which each recursive call occurs as well as the values of input arguments and local variables at both entry to and exit from the methods. Be sure to remove these statements from the final version of the method.
5. A recursive solution that recomputes certain values frequently can be quite inefficient. In such cases, iteration may be preferable to recursion.

Self-Test Exercises

1. The following method computes the product of the first $n \geq 1$ real numbers in an array. Show how this method satisfies the properties of a recursive method.

```
public static double product(double anArray[], int n) {
// Precondition: 1 <= n <= max size of anArray.
// Postcondition: Returns the product of the first n
// items in anArray; anArray is unchanged.
  if (n == 1) {
    return anArray[0];
  }
  else {
    return anArray[n-1] * product(anArray, n-1);
```

```
  } // end if
} // end product
```

2. Given an integer $n > 0$, write a recursive method `countDown` that writes the integers n, $n - 1$, . . . , 1. *Hint*: What task can you do and what task can you ask a friend to do for you?
3. Write a recursive method that computes the product of the items in the array `anArray[first..last]`.
4. Of the following recursive methods that you saw in this chapter, identify those that exhibit tail recursion: `fact`, `writeBackward`, `writeBackward2`, `rabbit`, `c` in the Spock problem, `p` in the parade problem, `maxArray`, `binarySearch`, and `kSmall`. Are the methods in Self-Test Exercises 1 through 3 tail recursive?
5. Compute `c(4,2)` in the Spock problem.
6. Trace the execution of the method `solveTowers` to solve the Towers of Hanoi problem for two disks.

Exercises

1. The following recursive method `getNumberEqual` searches the array `x` of `n` integers for occurrences of the integer `val`. It returns the number of integers in `x` that are equal to `val`. For example, if `x` contains the 9 integers 1, 2, 4, 4, 5, 6, 7, 8, and 9, then `getNumberEqual(x, 9, 4)` returns the value 2 because 4 occurs twice in `x`.

```
public static int getNumberEqual(int x[], int n,
                                  int val) {
  int count;
  if (n <= 0) {
    return 0;
  }
 else {
    if (x[n-1] == val) {
      count = 1;
    }
    else {
      count = 0;
     } // end if

  return getNumberEqual(x, n-1, val) + count;
  } // end if
} // end getNumberEqual
```

   Demonstrate that this method is recursive by listing the criteria of a recursive solution and stating how the method meets each criterion.

2. Perform a box trace of the following calls to recursive methods that appear in this chapter. Clearly indicate each subsequent recursive call.

   a. `rabbit(5)`

   b. `countDown(5)` (You wrote `countDown` in Self-Test Exercise 2.)

3. Write a recursive method that will compute the sum of the first n integers in an array of at least n integers. *Hint*: Begin with the n^{th} integer.

4. Given two integers *start* and *end*, where *end* is greater than *start*, write a recursive Java method that returns the sum of the integers from *start* through *end*, inclusive.

5. Revise the method `writeBackward`, discussed in the section "A Recursive `void` Method: Writing a String Backward," so that its base case is a string of length 1.

6. Describe the problem with the following recursive method:

```
public void printNum (int n )
{
      System.out.println(n);
      printNum (n - 1);
}
```

7. Given an integer $n > 0$, write a recursive Java method that writes the integers 1, 2, . . . , n.

8. Given an integer n, write a recursive Java method that returns the sum of 1 through n.

9. Write a recursive Java method that writes the digits of a positive decimal integer in reverse order.

10. a. Write a recursive Java method `writeLine` that writes a character repeatedly to form a line of n characters. For example, `writeLine('*', 5)` produces the line `*****`.

    b. Now write a recursive method `writeBlock` that uses `writeLine` to write m lines of n characters each. For example, `writeBlock('*', 5, 3)` produces the output

```
*****
*****
*****
```

11. What output does the following program produce?

```
public class Exercise11 {
  public static int getValue(int a, int b, int n) {
    int returnValue;
    System.out.println("Enter: a = " + a + " b = " + b);

    int c = (a + b)/2;
    if (c * c <= n) {
       returnValue = c;
    }
    else {
      returnValue = getValue(a, c-1, n);
    }  // end if
    System.out.println("Leave: a = " + a + " b = " + b);
    return returnValue;
  }  // end getValue

  public static void main(String[] args) {
    System.out.println(getValue(1, 7, 7));
  }  // end main
}  // end Exercise11
```

12. What output does the following program produce?

```
public class Exercise12 {

  public static int mystery(int n) {
    return search(1, n, n);
  }  // end mystery

  private static int search(int first, int last, int n) {
    int returnValue;
    System.out.println("Enter: first = " + first +
                       " last = " + last);
    int mid = (first + last)/2;
    if ((mid * mid <= n) && (n < (mid+1) * (mid+1))) {
      returnValue = mid;
    }
    else if (mid * mid > n) {
      returnValue = search(first, mid-1, n);
    }
    else {
      returnValue = search(mid+1, last, n);
    }  // end if
    System.out.println("Leave: first = " + first +
                       " last = " + last);
    return returnValue;
  }  // end search

  public static void main(String[] args) {
    System.out.println(mystery(30));
  }  // end main
}  //end Exercise12
```

13. Consider the following method that converts a positive decimal number to base 8 and displays the result.

```
public static void displayOctal(int n) {
  if (n > 0) {
      if (n/8 > 0) {
         displayOctal(n/8);
      }  // end if
      System.out.println(n%8);
   }  // end if
}  // end displayOctal
```

Describe how the algorithm works. Trace the method with $n = 100$.

14. Consider the following program:

```
public class Exercise14 {

  public static int f(int n) {
  // Precondition: n >= 0.
    System.out.println("Method f entered, n = " + n);
    switch (n) {
```

```
      case 0: case 1: case 2:
        return n + 1;
      default:
        return f(n-2) * f(n-4);
    }  // end switch
  }  // end f

  public static void main(String[] args) {
    System.out.println("f(8) is equal to " + f(8));
  } // end main
} // end Exercise14
```

Show the exact output of the program. What argument values, if any, could you pass to the method `f` to cause the program to run forever?

15. Consider the following method:

```
public static void recurse(int x, int y) {
  if (y > 0) {
    ++x;
    --y;
    System.out.println(x + " " + y);
    recurse(x, y);
    System.out.println(x + " " + y);
  }  // end if
}  // end recurse
```

Execute the method with $x = 5$ and $y = 3$.

16. Perform a box trace of the recursive method *binarySearch*, which appears in the section "Binary Search," with the array 1, 5, 9, 12, 15, 21, 29, 31 for each of the following search values:

    a. 5 b. 13 c. 16

17. Imagine that you have 101 dalmatians; no two dalmatians have the same number of spots. Suppose that you create an array of 101 integers: The first integer is the number of spots on the first dalmatian, the second integer is the number of spots on the second dalmatian, and so on.

    Your friend wants to know whether you have a dalmatian with 99 spots. Thus, you need to determine whether the array contains the integer 99.

    a. If you plan to use a binary search to look for the 99, what, if anything, would you do to the array before searching it?

    b. What is the index of the integer in the array that a binary search would examine first?

    c. If all your dalmatians have more than 99 spots, exactly how many comparisons will a binary search require to determine that 99 is not in the array?

18. This problem considers several ways to compute x^n for some $n \geq 0$.

    a. Write an iterative method *power1* to compute x^n for $n \geq 0$.

    b. Write a recursive method *power2* to compute x^n by using the following recursive formulation:

$x^0 = 1$
$x^n = x * x^{n-1}$ if $n > 0$

c. Write a recursive method `power3` to compute x^n by using the following recursive formulation:

$x^0 = 1$
$x^n = (x^{n/2})^2$ if $n > 0$ and n is even
$x^n = x * (x^{n/2})^2$ if $n > 0$ and n is odd

d. How many multiplications will each of the methods `power1`, `power2`, and `power3` perform when computing 3^{32}? 3^{19}?

e. How many recursive calls will `power2` and `power3` make when computing 3^{32}? 3^{19}?

19. Modify the recursive `rabbit` method so that it is visually easy to follow the flow of execution. Instead of just adding "Enter" and "Leave" messages, indent the trace messages according to how "deep" the current recursive call is. For example, the call `rabbit (4)` should produce the output

```
Enter rabbit:   n = 4
    Enter rabbit:   n = 3
        Enter rabbit:   n = 2
        Leave rabbit:   n = 2   value = 1
        Enter rabbit:   n = 1
        Leave rabbit:   n = 1   value = 1
    Leave rabbit:   n = 3   value = 2
    Enter rabbit:   n = 2
    Leave rabbit:   n = 2   value = 1
Leave rabbit:   n = 4   value = 3
```

Note how this output corresponds to figures such as Figure 3-11.

20. Consider the following recurrence relation:

$f(1) = 1; f(2) = 1; f(3) = 1; f(4) = 3; f(5) = 5;$
$f(n) = f(n-1) + 3 * f(n-5)$ for all $n > 5$.

a. Compute $f(n)$ for the following values of n: 6, 7, 12, 15.

b. If you were careful, rather than computing $f(15)$ from scratch (the way a recursive Java method would compute it), you would have computed $f(6)$, then $f(7)$, then $f(8)$, and so on up to $f(15)$, recording the values as you computed them. This ordering would have saved you the effort of ever computing the same value more than once. (Recall the nonrecursive version of the `rabbit` method discussed at the end of this chapter.)

Note that during the computation, you never need to remember all the previously computed values—only the last five. By taking advantage of these observations, write a Java method that computes $f(n)$ for arbitrary values of n.

21. Write iterative versions of the following recursive methods: `fact`, `writeBackward`, `binarySearch`, `kSmall`.

22. Prove that the method `iterativeRabbit`, which appears in the section "Recursion and Efficiency," is correct by using invariants.

* 23. Consider the problem of finding the **greatest common divisor (gcd)** of two positive integers a and b. The algorithm presented here is a variation of Euclid's algorithm, which is based on the following theorem:[4]

THEOREM. If a and b are positive integers with $a > b$ such that b is not a divisor of a, then $gcd(a, b) = gcd(b, a \bmod b)$.

This relationship between $gcd(a, b)$ and $gcd(b, a \bmod b)$ is the heart of the recursive solution. It specifies how you can solve the problem of computing $gcd(a, b)$ in terms of another problem of the same type. Also, if b does divide a, then $b = gcd(a, b)$, so an appropriate choice for the base case is $(a \bmod b) = 0$.

This theorem leads to the following recursive definition:

$$\gcd(a, b) = \begin{cases} b & \text{if } (a \bmod b) = 0 \\ \gcd(b, a \bmod b) & \text{otherwise} \end{cases}$$

The following method implements this recursive algorithm:

```
public static int gcd(int a, int b) {
   if (a % b == 0) { // base case
      return b;
   }
   else {
      return gcd(b, a % b);
   } // end if
} // end gcd
```

a. Prove the theorem.

b. What happens if $b > a$?

c. How is the problem getting smaller? (That is, do you always approach a base case?) Why is the base case appropriate?

* 24. Let $C(n)$ be the number of different groups of integers that can be chosen from the integers 1 through $n - 1$ so that the integers in each group add up to n (for example, $4 = 1 + 1 + 1 + 1 = 1 + 1 + 2 = 2 + 2 \ldots$). Write recursive definitions for $C(n)$ under the following variations:

a. You count permutations. For example, 1, 2, 1 and 1, 1, 2 are two groups that each add up to 4.

b. You ignore permutations.

25. Consider the following recursive definition:

$$Acker(m, n) = \begin{cases} n + 1 & \text{if } m = 0 \\ Acker(m - 1, 1) & \text{if } n = 0 \\ Acker(m - 1, Acker(m, n - 1)) & \text{otherwise} \end{cases}$$

4. This book uses mod as an abbreviation for the mathematical operation modulo. In Java, the modulo operator is `%`.

This function, called **Ackermann's function,** is of interest because it grows rapidly with respect to the sizes of m and n. What is $Acker(1, 2)$? Implement the function as a method in Java and do a box trace of $Acker(1, 2)$. (*Caution:* Even for modest values of m and n, Ackermann's function requires *many* recursive calls.)

Programming Problems

1. Implement a recursive function that computes a^n where a is a real number and n is a nonnegative integer.
2. Implement `maxArray`, discussed in the section "Finding the Largest Item in an Array," as a Java method. What other recursive definitions of `maxArray` can you describe?
3. Implement the `binarySearch` algorithm presented in this chapter for an array of strings.
4. Implement `kSmall`, discussed in the section "Finding the k^{th} Smallest Item in an Array," as a Java method. Use the first item of the array as the pivot.