CHAPTER

17 Recursion

17.1 Introduction

Suppose we want to find a particular student's exam in a stack of exams that are already sorted into alphabetical order. Our procedure for doing so could be as follows: Pick a random point in the stack, and check for a match. If we find a match, great! We are done. If we don't find the exam (which is the more likely case, initially), we now know the exam we're looking for is either in the upper stack or in the lower stack based on whether the name occurs alphabetically before or after the name at the random point. Here's the key: We can now repeat the same procedure on a stack of exams (lower or upper) that is necessarily smaller than our original stack. For example, say we are looking for Mira's exam. We find at our selected random point Salina's exam. We clearly didn't find Mira's exam, but we know that it must in the portion of the stack that precedes Salina's exam. We repeat our search on that smaller substack. Fairly quickly, we will locate Mira's exam, if it exists in the set.

The technique we've described is *recursive*. We are solving the problem (finding an exam in a stack of exams) by stating that we'll solve it on successively smaller versions of the problem (find the exam on this smaller stack). We examined recursion first in Chapter 8, in the context of subroutines in LC-3 assembly language. Now that we've raised the level of programming abstraction, it's worthwhile for us to revisit recursion in the context of the C programming language.

Recursion is similar to iteration in that both describe a repeating flow of computation. The power of recursion lies in its ability to elegantly express the computation flow for certain programming tasks. There are some programming problems for which the recursive solution is far simpler than the corresponding iterative solution. Nearly always, recursion comes at the cost of additional execution overhead compared to iteration. So recursion must be applied carefully with a thorough understanding of the underlying costs.

In this chapter, we introduce the concept of recursion via several different examples. As we did in Chapter 8, we'll examine how recursive functions are implemented on the LC-3, this time with the run-time stack facilitating the recursion. The elegance of the run-time stack mechanism is that recursive functions require no special handling—they execute in the same manner as any other function. The main purpose of this chapter is to provide an initial but deep exposure to recursion so that you can start implementing recursive algorithms in your code.

17.2 What Is Recursion?

Let's revisit the basic idea of recursion through a simple toy example. A function that calls itself is a recursive function. The function `RunningSum` in Figure 17.1 is an example. This function calculates the sum of all the integers between the input parameter `n` and 1, inclusive. For example, `RunningSum(4)` calculates 4+3+2+1. However, it does the calculation recursively. Notice that the running sum of 4 is really 4 plus the running sum of 3. Likewise, the running sum of 3 is 3 plus the running sum of 2. This *recursive* definition is the basis for a recursive algorithm. In other words,

$$\text{RunningSum}(n) = n + \text{RunningSum}(n-1)$$

In mathematics, we use *recurrence equations* to express such functions. The preceding equation is a recurrence equation for `RunningSum`. In order to complete the evaluation of this equation, we must also supply an initial case. So in addition to the preceding formula, we need to state

$$\text{RunningSum}(1) = 1$$

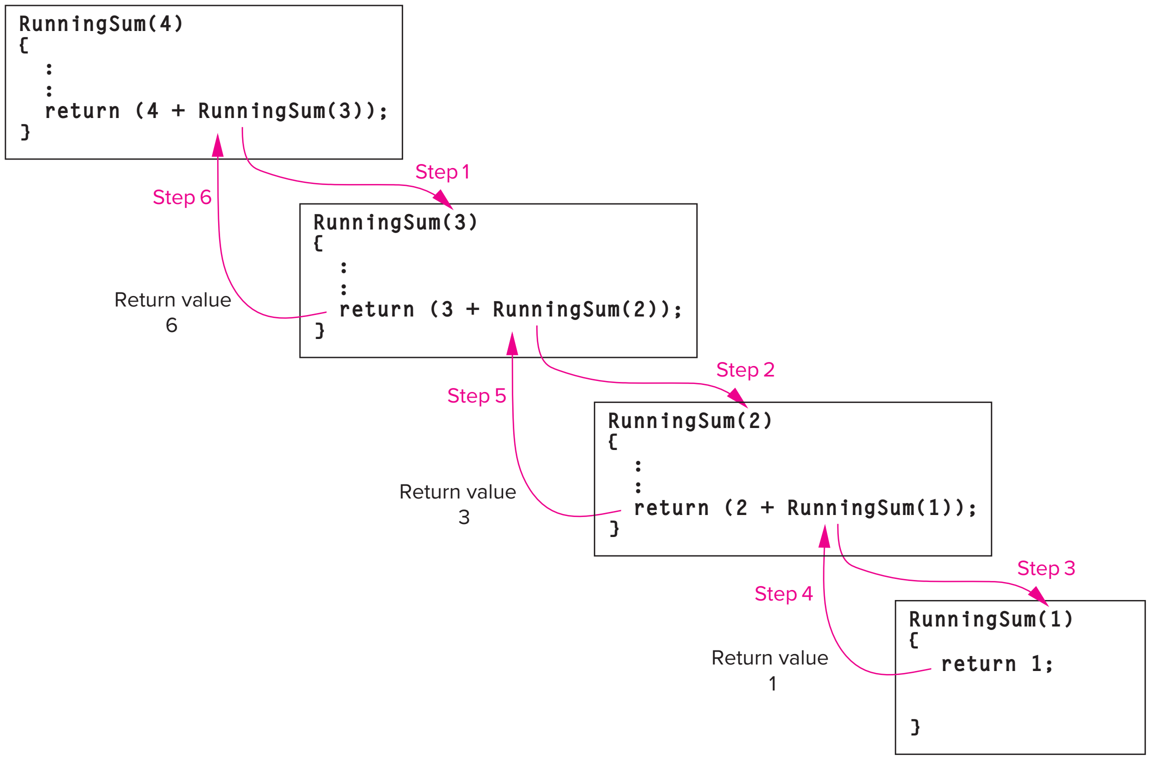
before we can completely evaluate the recurrence. Now we can fully evaluate RunningSum(4):

$$\begin{aligned}\text{RunningSum}(4) &= 4 + \text{RunningSum}(3)\\ &= 4 + 3 + \text{RunningSum}(2)\\ &= 4 + 3 + 2 + \text{RunningSum}(1)\\ &= 4 + 3 + 2 + 1\end{aligned}$$

The C version of `RunningSum` works in the same manner as the recurrence equation. During execution of the function call `RunningSum(4)`, `RunningSum` makes a function call to itself, with an argument of 3 (i.e., `RunningSum(3)`). However, before `RunningSum(3)` ends, it makes a call to `RunningSum(2)`. And before

```
int RunningSum(int n)
{
   if (n == 1)
      return 1;
   else
      return (n + RunningSum(n-1));
}
```

Figure 17.1 An example of a simple recursive function.



```
RunningSum(4)
{
  :
  :
  return (4 + RunningSum(3));
}
```

```
RunningSum(3)
{
  :
  :
  return (3 + RunningSum(2));
}
```

```
RunningSum(2)
{
  :
  :
  return (2 + RunningSum(1));
}
```

```
RunningSum(1)
{
  return 1;

}
```

Figure 17.2 The flow of control when `RunningSum(4)` is executed.

`RunningSum(2)` ends, it makes a call to `RunningSum(1)`. `RunningSum(1)`, however, makes no additional recursive calls and returns the value 1 to `RunningSum(2)`, which enables `RunningSum(2)` to end and return the value 2 + 1 back to `RunningSum(3)`. This enables `RunningSum(3)` to end and pass a value of 3 + 2 + 1 to `RunningSum(4)`. Figure 17.2 pictorially shows how the execution of `RunningSum(4)` proceeds.

17.3 Recursion vs. Iteration

Clearly, we could have (and should have) written `RunningSum` using a `for` loop, and the code would have been more straightforward than its recursive counterpart. We provided a recursive version here in order to demonstrate a recursive call in the context of an easy-to-understand example.

There is a parallel between using recursion and using conventional iteration (such as `for` and `while` loops) in programming. All recursive functions can be written using iteration. For certain programming problems, however, the recursive version is simpler and more elegant than the iterative version. Solutions to certain problems are naturally expressed in a recursive manner, such as problems that are expressed with recurrence equations. It is because of such problems that recursion is an indispensable programming technique. Knowing which problems require recursion and which are better solved with iteration is part of the art of computer programming, and it is a skill one develops through coding experience.

Recursion, as useful as it is, comes at a cost. As an experiment, write an iterative version of `RunningSum` and compare the running time for large *n* with the recursive version. To do this you can use library functions to get the time of day (e.g., `gettimeofday`) before the function starts and when it ends. Plot the running time for a variety of values of *n* and you will notice that the recursive version is relatively slower (provided the compiler did not optimize away the recursion, which it can do through a simple transformation that converts certain types of recursive code to iterative code when the recursion is the last operation in the function, i.e., it occurs at the tail of the function). Recursive functions incur function call overhead that iterative solutions do not. Understanding the underlying overheads of recursion is something we cleanly explore with our bottom-up approach, and it will assist you in knowing when, and when not, to apply recursion for a particular programming task.

17.4 Towers of Hanoi

One problem for which the recursive solution is simpler than iteration is the classic puzzle Towers of Hanoi. The puzzle involves a platform with three posts. On one of the posts sit a number of wooden disks, each smaller than the one below it. The objective is to move all the disks from their current post to one of the other posts. However, there are two rules for moving disks: Only one disk can be moved at a time, and a larger disk can never be placed on top of a smaller disk. For example, Figure 17.3 shows a puzzle where five disks are on post 1. To solve this puzzle, these five disks must be moved to one of the other posts obeying the two rules. As the legend associated with the puzzle goes, when the world was created, the priests at the Temple of Brahma were given the task of moving 64 disks from one post to another. When they completed their task, the world would end.

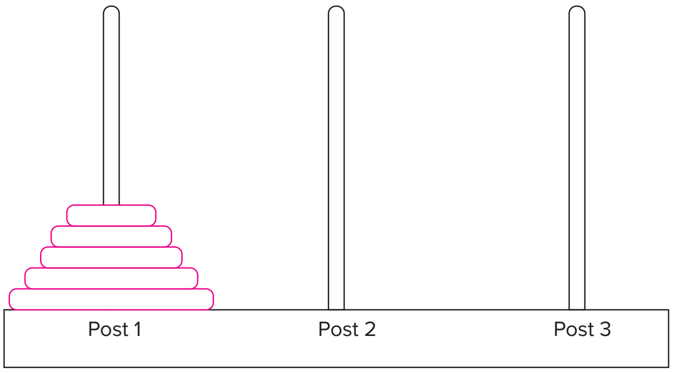


Figure 17.3 **The Towers of Hanoi puzzle with five disks.**

Now how would we go about writing C code to solve this puzzle? If we view the problem from the end first, we can make the following observation: The final sequence of moves **must** involve moving the largest disk from post 1 to the target post, say post 3, and then moving all of the other disks back on top of it. Conceptually, we need to move all $n - 1$ disks off the largest disk and onto the intermediate post, then move the largest disk onto the target post. Finally, we move all $n - 1$ disks from the intermediate post onto the target post. Moving $n - 1$ disks in one move is not a singular, legal move. However, we have stated the problem in such a manner that we can solve by solving two smaller subproblems. We now have a recursive definition of the problem: In order to move n disks to the target post, which we symbolically represent as `Move(n, target)`, we first move $n - 1$ disks to the intermediate post, `Move(n-1, intermediate)`, then move the nth disk to the target, which is a singular move, and finally move $n - 1$ disks from the intermediate to the target, `Move(n-1, target)`. So in order to `Move(n, target)`, two recursive calls are made to solve two smaller subproblems involving $n - 1$ disks.

As with recurrence equations in mathematics, all recursive definitions require a *base case*, which ends the recursion. In the way we have formulated the problem, the base case involves moving the smallest disk (disk 1). Moving disk 1 requires no other disks to be moved since it is always on top and can be moved directly from one post to any another without moving any other disks. Without a base case, a recursive function would never end, similar to an infinite loop in conventional iteration. Taking our recursive definition to C code is fairly straightforward. Figure 17.4 is a recursive C function of this algorithm.

```
// diskNumber is the disk to be moved (disk1 is smallest)
// startPost is the post the disk is currently on
// endPost is the post we want the disk to end on
// midPost is the intermediate post
void MoveDisk(int diskNumber, int startPost, int endPost, int midPost)
{
   if (diskNumber > 1) {
      // Move n-1 disks off the current disk on
      // startPost and put them on the midPost
      MoveDisk(diskNumber-1, startPost, midPost, endPost);

      printf("Move disk %d from post %d to post %d.\n",
              diskNumber, startPost, endPost);

      // Move all n-1 disks from midPost onto endPost
      MoveDisk(diskNumber-1, midPost, endPost, startPost);
   }
   else
      printf("Move disk 1 from post %d to post %d.\n",
              startPost, endPost);
}
```

Figure 17.4 A recursive function to solve the Towers of Hanoi puzzle.

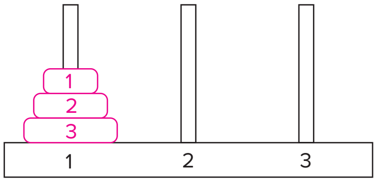


Figure 17.5 The Towers of Hanoi puzzle, initial configuration.

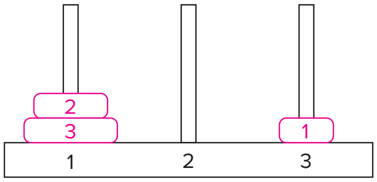


Figure 17.6 The Towers of Hanoi puzzle, after first move.

Let's see what happens when we solve the puzzle with three disks. Following is the initial function call to `MoveDisk`. We start off by saying that we want to move disk 3 (the largest disk) from post 1 to post 3, using post 2 as the intermediate storage post. That is, we want to solve a three-disk Towers of Hanoi puzzle. See Figure 17.5.

```
// diskNumber 3; startPost 1; endPost 3; midPost 2
MoveDisk(3, 1, 3, 2)
```

This call invokes another call to `MoveDisk` to move disks 1 and 2 off disk 3 and onto post 2 using post 3 as intermediate storage. The call is performed at line 10 in the source code.

```
// diskNumber 2; startPost 1; endPost 2; midPost 3
MoveDisk(2, 1, 2, 3)
```

To move disk 2 from post 1 to post 2, we first move disk 1 off disk 2 and onto post 3 (the intermediate post). This triggers another call to `MoveDisk` again from the call on line 10.

```
// diskNumber 1; startPost 1; endPost 3; midPost 2
MoveDisk(1, 1, 3, 2)
```

For this call, the condition of the `if` statement on line 7 will not be true, and disk 1 can be moved directly to the target post. The `printf` statement on lines 19–20 is executed. See Figure 17.6.

```
Move disk 1 from post 1 to post 3.
```

This invocation of `MoveDisk` returns to its caller, which was the call `MoveDisk(2, 1, 2, 3)`. Recall that we were waiting for all disks on top of disk 2 to be moved to post 3. Since that is now complete, we can move disk 2 from post 1 to post 2. The `printf` on lines 19–20 is the next statement to execute, signaling another disk to be moved. See Figure 17.7.

```
Move disk 2 from post 1 to post 2.
```

Next, a call is made to move all disks that were on disk 2 back onto disk 2. This happens at the call on line 16 of the source code for `MoveDisk`.

```
// diskNumber 1; startPost 2; endPost 3; midPost 1
MoveDisk(1, 2, 3, 1)
```

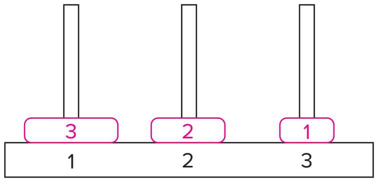


Figure 17.7 The Towers of Hanoi puzzle, after second move.

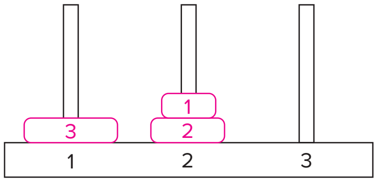


Figure 17.8 The Towers of Hanoi puzzle, after third move.

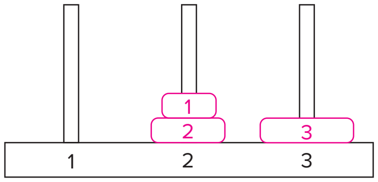


Figure 17.9 The Towers of Hanoi puzzle, after fourth move.

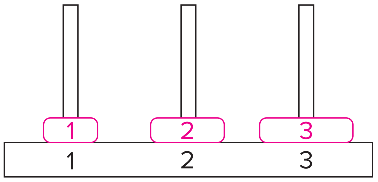


Figure 17.10 The Towers of Hanoi puzzle, after fifth move.

Again, since disk 1 has no disks on top of it, we see the move printed. See Figure 17.8.

```
Move disk number 1 from post 3 to post 2.
```

Control passes back to the call `MoveDisk(2, 1, 2, 3)` which, having completed its task of moving disk 2 (and all disks on top of it) from post 1 to post 2, returns to its caller. Its caller is `MoveDisk(3, 1, 3, 2)`. All disks have been moved off disk 3 and onto post 2. Disk 3 can be moved from post 1 onto post 3. The `printf` is the next statement executed. See Figure 17.9.

```
Move disk 3 from post 1 to post 3.
```

The next subtask remaining is to move disk 2 (and all disks on top of it) from post 2 onto post 3. We can use post 1 for intermediate storage. The following call occurs on line 16 of the source code.

```
// diskNumber 2; startPost 2; endPost 3; midPost 1
MoveDisk(2, 2, 3, 1)
```

In order to do so, we must first move disk 1 from post 2 onto post 1, via the call on line 16.

```
// diskNumber 1; startPost 2; endPost 1; midPost 3
MoveDisk(1, 2, 1, 3)
```

The move requires no submoves. See Figure 17.10.

```
Move disk 1 from post 2 to post 1.
```

Return passes back to the caller `MoveDisk(2, 2, 3, 1)`, and disk 2 is moved onto post 3. See Figure 17.11.

```
Move disk 2 from post 2 to post 3.
```

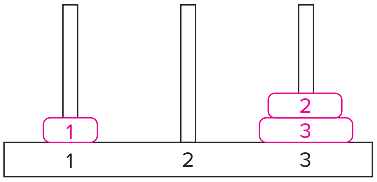


Figure 17.11 The Towers of Hanoi puzzle, after sixth move.

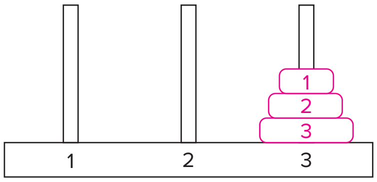


Figure 17.12 The Towers of Hanoi puzzle, completed.

The only thing remaining is to move all disks that were on disk 2 back on top.

```
// diskNumber 1; startPost 1; endPost 3; midPost 2
MoveDisk(1, 1, 3, 2)
```

The move is done immediately. See Figure 17.12.

```
Move disk 1 from post 1 to post 3.
```

and the puzzle is completed!

Let's summarize the action of the recursion by examining the sequence of function calls that were made in solving the three-disk puzzle:

```
MoveDisk(3, 1, 3, 2)   // Initial Call
MoveDisk(2, 1, 2, 3)
MoveDisk(1, 1, 3, 2)
MoveDisk(1, 2, 3, 1)
MoveDisk(2, 2, 3, 1)
MoveDisk(1, 2, 1, 3)
MoveDisk(1, 1, 3, 2)
```

Consider how you would write an iterative version of a solver for this puzzle. You'll no doubt quickly appreciate the simplicity of the recursive version. Returning to the legend of the Towers of Hanoi: The world will end when the monks finish solving a 64-disk version of the puzzle. For a three-disk puzzle, the solution required seven moves. If each move takes one second, how long will it take the monks to solve the 64-disk puzzle? Would the number of moves for an iterative version be any different?

17.5 Fibonacci Numbers

Let's revisit an example from Section 8.3.2. We considered this a bad application of recursion when we introduced it for the LC-3, and it still is in C! But this example is worth reexamining. It's simple enough to express with a short C function, yet complex enough to have interesting stack behavior during execution. Also, we can view a complete translation to LC-3 assembly and take note of how the recursion is handled by the run-time stack.

The following recurrence equations generate a well-known sequence of numbers called the *Fibonacci numbers*, which has some interesting mathematical, geometrical, and natural properties.

$$f(n) = f(n-1) + f(n-2)$$
$$f(1) = 1$$
$$f(0) = 1$$

The *n*th Fibonacci number is the sum of the previous two. The series is 1, 1, 2, 3, 5, 8, 13, … This series was popularized by the Italian mathematician Leonardo of Pisa around the year 1200 (it is thought to have first been described by Indian mathematicians around 200 BC). His father's name was Bonacci, and thus he often called himself Fibonacci as a shortening of *filius Bonacci*, or son of Bonacci. Fibonacci formulated this series as a way of estimating breeding rabbit populations. We have since discovered some fascinating ways in which the series models some other natural phenomena such as the structure of a spiral shell or the pattern of petals on a flower. The ratios of successive numbers in the sequence approximate the Golden Ratio.

We can formulate a recursive function to calculate the *n*th Fibonacci number directly from the recurrence equations. `Fibonacci(n)` is recursively calculated by `Fibonacci(n-1) + Fibonacci(n-2)`. The base case of the recursion is simply the fact that `Fibonacci(1)` and `Fibonacci(0)` both equal 1. Figure 17.13 lists the recursive code to calculate the *n*th Fibonacci number.

```
#include <stdio.h>

int Fibonacci(int n);

int main(void)
{
   int in;
   int number;

   printf("Which Fibonacci number? ");
   scanf("%d", &in);

   number = Fibonacci(in);
   printf("That Fibonacci number is %d\n", number);
}

int Fibonacci(int n)
{
   int sum;

   if (n == 0 || n == 1)
      return 1;
   else {
      sum = (Fibonacci(n-1) + Fibonacci(n-2));
      return sum;
}
```

Figure 17.13 A recursive C function to calculate the *n*th Fibonacci number.

This example is simple enough for us to take a deeper look into how recursion actually is implemented at the lower levels. In particular, we will examine the run-time stack mechanism and see how it naturally handles recursive calls. Whenever a function is called, whether from itself or another function, a new copy of its stack frame is pushed onto the run-time stack. That is, each invocation of the function gets a new, private copy of parameters and local variables. And once each invocation completes, this private copy must be deallocated. The run-time stack enables this in a natural fashion. If the variables of a recursive function were statically allocated in memory, each recursive call to `Fibonacci` would overwrite the values of the previous call.

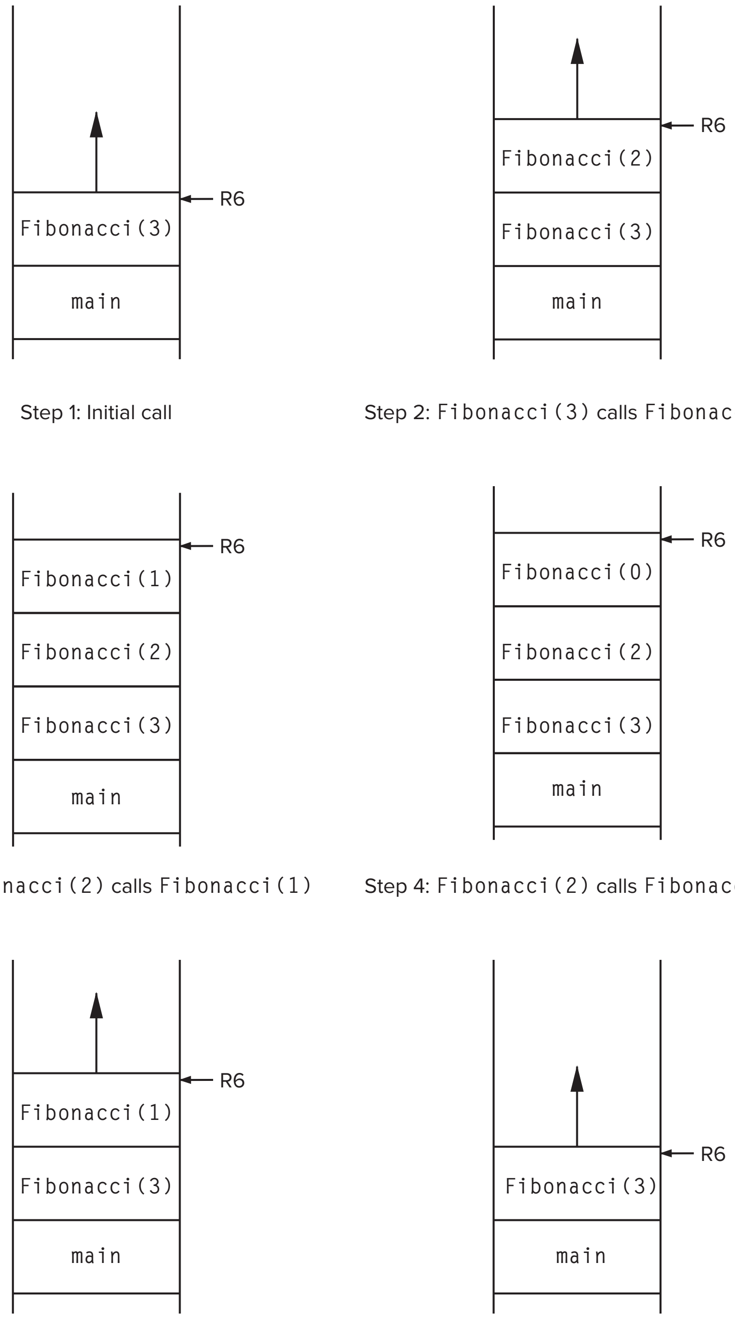
Let's take a look at the run-time stack when we call the function `Fibonacci` with the parameter 3, `Fibonacci(3)`. We start off with the stack frame for `Fibonacci(3)` on top of the run-time stack. Figure 17.14 shows the progression of the stack as the original function call is evaluated. The function call `Fibonacci(3)` will first calculate `Fibonacci(3-1)` as the expression `Fibonacci(n-1) + Fibonacci(n-2)` is evaluated left to right. Therefore, a call is first made to `Fibonacci(2)`, and a stack frame for `Fibonacci(2)` is pushed onto the run-time stack (see Figure 17.14, step 2). For `Fibonacci(2)`, the parameter `n` equals 2 and does not meet the terminal condition; therefore, a call is made to `Fibonacci(1)` (see Figure 17.14, step 3). This call is made in the course of evaluating `Fibonacci(2-1) + Fibonacci(2-2)`. The call `Fibonacci(1)` results in no more recursive calls because the parameter `n` meets the terminal condition. The value 1 is returned to `Fibonacci(2)`, which now can complete the evaluation of `Fibonacci(1) + Fibonacci(0)` by calling `Fibonacci(0)` (see Figure 17.14, step 4). The call `Fibonacci(0)` immediately returns a 1. Now, the call `Fibonacci(2)` can complete and return its subcalculation (its result is 2) to its caller, `Fibonacci(3)`. Having completed the left-hand component of the expression `Fibonacci(2) + Fibonacci(1)`, `Fibonacci(3)` calls `Fibonacci(1)` (see Figure 17.14, step 5), which immediately returns the value 1. Now `Fibonacci(3)` is done—its result is 3 (Figure 17.14, step 6). We could state the recursion of `Fibonacci(3)` algebraically, as follows:

```
Fibonacci(3) = Fibonacci(2) + Fibonacci(1)
             = (Fibonacci(1) + Fibonacci(0)) + Fibonacci(1)
             = 1 + 1 + 1 = 3
```

The sequence of function calls made during the evaluation of `Fibonacci(3)` is as follows:

```
Fibonacci(3)
Fibonacci(2)
Fibonacci(1)
Fibonacci(0)
Fibonacci(1)
```

Walk through the execution of `Fibonacci(4)` and you will notice that the sequence of calls made by `Fibonacci(3)` is a subset of the calls made by `Fibonacci(4)`. No surprise, since `Fibonacci(4) = Fibonacci(3) + Fibonacci(2)`. Likewise, the sequence of calls made by `Fibonacci(4)` is a subset of the calls made by `Fibonacci(5)`. There is an exercise at the end of this



Step 1: Initial call

Step 2: `Fibonacci(3)` calls `Fibonacci(2)`

Step 3: `Fibonacci(2)` calls `Fibonacci(1)`

Step 4: `Fibonacci(2)` calls `Fibonacci(0)`

Step 5: `Fibonacci(3)` calls `Fibonacci(1)`

Step 6: Back to the starting point

Figure 17.14 **Snapshots of the run-time stack for the function call** `Fibonacci(3).`

chapter involving calculating the number of function calls made during the evaluation of `Fibonacci(n)`. As we did in Section 8.3.2, it's worthwhile to compare the running time of the recursive version of `Fibonnaci` in C to an iterative version. The simple recursive version, while it falls directly out of the recurrence equations and is easy to code, is far more costly than its iterative counterpart. It is more expensive not only because of the function call overhead, but also because the recursive solution has a significant number of repeated calculations that the iterative version does not.

Figure 17.15 lists the LC-3 C compiler generated for this Fibonacci program. Notice that no special treatment was required to handle the program's recursive nature. Because of the run-time stack mechanism for activating functions, a recursive function gets treated like every other function. If you examine this code closely, you will notice that the compiler generated a temporary variable in order to translate line 24 of `Fibonacci` properly. Most compilers will generate such temporaries when compiling complex expressions and will allocate storage in the stack frame on top of programmer-declared local variables.

```
Fibonacci:
    ADD R6, R6, #-2    ; push return value/address
    STR R7, R6, #0     ; store return address
    ADD R6, R6, #-1    ; push caller's frame pointer
    STR R5, R6, #0     ;
    ADD R5, R6, #-1    ; set new frame pointer
    ADD R6, R6, #-2    ; allocate space for locals and temps

    LDR R0, R5, #4     ; load the parameter n
    BRZ FIB_BASE       ; check for n == 0
    ADD R0, R0, #-1    ;
    BRZ FIB_BASE       ; n==1

    LDR R0, R5, #4     ; load the parameter n
    ADD R0, R0, #-1    ; calculate n-1
    ADD R6, R6, #-1    ; push n-1
    STR R0, R6, #0     ;
    JSR Fibonacci      ; call to Fibonacci(n-1)

    LDR R0, R6, #0     ; read the return value at top of stack
    ADD R6, R6, #-1    ; pop return value
    STR R0, R5, #-1    ; store it into temporary value
    LDR R0, R5, #4     ; load the parameter n
    ADD R0, R0, #-2    ; calculate n-2
    ADD R6, R6, #-1    ; push n-2
    STR R0, R6, #0     ;
    JSR Fibonacci      ; call to Fibonacci(n-2)

```

Figure 17.15 `Fibonacci` in LC-3 assembly code (Fig. 17.15 continued on next page.)

```
        LDR R0, R6, #0     ; read the return value at top of stack
        ADD R6, R6, #-1    ; pop return value
        LDR R1, R5, #-1    ; read temporary value: Fibonacci(n-1)
        ADD R0, R0, R1     ; Fibonacci(n-1) + Fibonacci(n-2)
        BR FIB_END         ; branch to end of code

  FIB_BASE:
        AND R0, R0, #0     ; clear R0
        ADD R0, R0, #1     ; R0 = 1

  FIB_END:
        STR R0, R5, #3     ; write the return value
        ADD R6, R5, #1     ; pop local variables
        LDR R5, R6, #0     ; restore caller's frame pointer
        ADD R6, R6, #1     ;
        LDR R7, R6, #0     ; pop return address
        ADD R6, R6, #1     ;
        RET
```

Figure 17.15 `Fibonacci` **in LC-3 assembly code (continued Fig. 17.15 from previous page.)**

17.6 Binary Search

We started this chapter by describing a recursive technique for finding a particular exam in a set of exams that are in alphabetical order. The technique is called *binary search*, and it is a rapid way of finding a particular element within a list of elements in sorted order. At this point, given our understanding of recursion and of arrays, we can code a recursive function in C to perform binary search.

Say we want to find a particular integer value in an array of integers that is in ascending order. The function should return the index of the integer, or a -1 if the integer does not exist. To accomplish this, we will use the binary search technique as follows: Given an array and an integer to search for, we will examine the midpoint of the array and determine if the integer is (1) equal to the value at the midpoint, (2) less than the value at the midpoint, or (3) greater than the value at the midpoint. If it is equal, we are done. If it is less than, we perform the search again, but this time only on the first half of the array. If it is greater than, we perform the search only on the second half of the array. Notice that we can express cases (2) and (3) using recursive calls.

What happens if the value we are searching for does not exist within the array? Given this recursive technique of performing searches on smaller and smaller subarrays of the original array, we eventually perform a search on an array that has no elements (e.g., of size 0) if the item we are searching for does not exist. If we encounter this situation, we will return a -1. This will be a base case in the recursion.

Figure 17.16 contains the recursive implementation of the binary search algorithm in C. Notice that in order to determine the size of the array at each step, we pass the starting and ending points of the subarray along with each call to `BinarySearch`. Each call refines the variables `start` and `end` to search smaller

```c
// This function returns the position of 'item' if it exists
// between list[start] and list[end], or -1 if it does not.
int BinarySearch(int item, int list[], int start, int end)
{
   int middle = (end + start) / 2;

   // Did we not find what we are looking for?
   if (end < start)
      return -1;

   // Did we find the item?
   else if (list[middle] == item)
      return middle;

   // Should we search the first half of the array?
   else if (item < list[middle])
      return BinarySearch(item, list, start, middle - 1);

   // Or should we search the second half of the array?
   else
      return BinarySearch(item, list, middle + 1, end);
}
```

Figure 17.16 **A recursive C function to perform binary search.**

and smaller subarrays of the original array `list`. The variable `start` contains the array index of the first data item, and the variable `end` contains the index of the last data item.

Figure 17.17 provides an illustration of this code during execution. The array `list` contains eleven elements as shown. The initial call to `BinarySearch` passes the value we are searching for (`item`) and the array to be searched. (Recall from Chapter 16 that this is the address of the very first element, or base address, of the array.) Along with the array, we provide the *extent* of the array. That is, we provide the starting and ending points of the portion of the array to be searched. In every subsequent recursive call to `BinarySearch`, this extent is made smaller, eventually reaching a point where the subset of the array we are searching has either only one element or no elements at all. These two situations are the base cases of the recursion.

A more straightforward search technique would be to sequentially search through the array. That is, we could examine `list[0]`, then `list[1]`, then `list[2]`, etc., and eventually either find the item or determine that it does not exist. Binary search, however, will require fewer comparisons and can potentially execute faster if the array is large enough. In subsequent computing courses, you will analyze binary search and determine that its running time is proportional to $\log_2 n$, where n is the size of the array. Sequential search, on the other hand, is proportional to n. For sufficiently large values of n, $\log_2 n$ is much smaller than n. For example, if $n = 1{,}000{,}000$, then $\log_2 n$ is 19.93. Note that we aren't saying that recursive binary search is more efficient than iterative binary search, because it

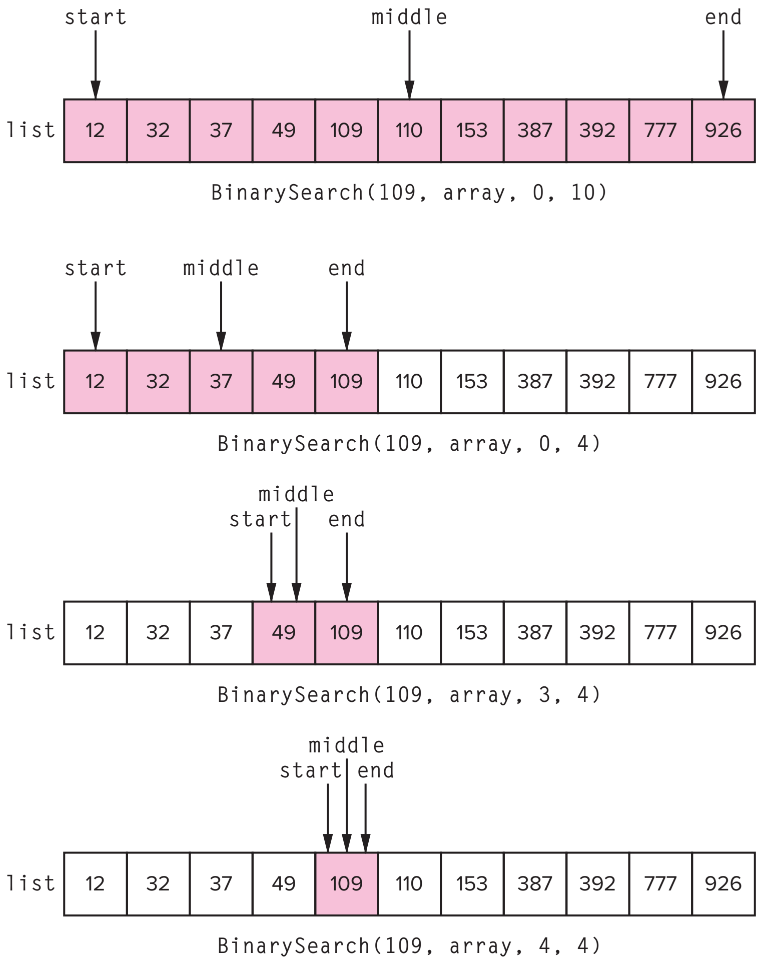

start
middle
end
list
12
32
37
49
109
110
153
387
392
777
926
BinarySearch(109, array, 0, 10)
BinarySearch(109, array, 0, 4)
BinarySearch(109, array, 3, 4)
BinarySearch(109, array, 4, 4)

Figure 17.17 `BinarySearch` **performed on an array of eleven elements. We are searching for the element 109.**

isn't (due to function call overhead). But we are saying that binary search (iterative or recursive) is more efficient than sequential search, and significantly so.

17.7 Escaping a Maze

Recursion is commonly applied in solving games and puzzles. Recursion allows for a systematic search through all possibilities, making it useful for solving problems where it is feasible to examine every possible solution in search of the right one. Solving a Soduku puzzle, or finding a path through a maze, for example, are problems that are amenable to recursive search.

We wrote a simple maze solver in Chapter 8 in LC-3 assembly, and we'll revisit the problem again in the context of C. There are many ways we can approach a maze solver, but the recursive approach is perhaps the simplest. Let's start by considering how to represent a maze in C. As we did in Chapter 8, we'll use a two-dimensional array, with each element in the array representing whether the maze position is blocked (because it contains a wall), is open, or contains an exit. The maze will be a character array, where each element of the array contains the character *X* for a blocked space, a space ' ' character for an open space, and the character *E* for an exit. Our maze solver will take this 2D array as input, along with coordinates of the initial starting point, and will search through the maze to find a path to the exit, if it exists. Figure 17.18 shows an example of a simple 4×4 maze represented in our 2D array format. In this example, `maze[3][0] = 'E'` designates the exit, and `maze[0][2] = 'X'` is an example of a space that is blocked. If we set the initial position to `maze[1][2]`, then there exists a path to the exit.

	X	X	
X			X
		X	
E	X		

Figure 17.18 An example of how to represent a maze with a two-dimensional character array.

Now the question is, how might a maze solver compute the path to the exit? We can use the following idea: From the starting point, if a path to the exit exists, it must go through the space either directly to the left, directly to the right, directly up, or directly down from the initial point. So we set the new position to one of those spaces and recursively solve the maze from there. That is, from the new point, if a solution exists, it must go through space to the left, to the right, directly up, or directly down from the new point.

Eventually, we'll hit a blocked space *X* or the exit *E*. And these designate base cases for the recursion. We'll also need to mark spaces that we've already visited with a *V*. This will enable us to skip over spaces that we've already evaluated, and it will prevent us from going in circles (literally!). This will also be a base case.

Figure 17.19 provides the C code implementation of the `ExitMaze` function. It takes as its parameters a maze represented as a 2D character array of dimensions `MAZE_HEIGHT` by `MAZE_WIDTH`, and two integers that represent the current position as *x* and *y* indices in the maze array. The initial portion of the function checks to see if we've hit any of our terminal conditions: Either `xpos` or `ypos` is out of the maze, or the current position is the exit or corresponds to a space in the maze that has already been visited or is blocked. If not, we mark the current position as visited and recursively check the neighboring positions.

This algorithm performs something called a *depth-first search* through all the possible paths through the maze. We can represent the recursive calls as a graph, where each node in the graph corresponds to an invocation of the function ExitMaze. In the general case, each invocation of `ExitMaze` can make up to four

```c
#define MAZE_HEIGHT 4
#define MAZE_WIDTH 4

int ExitMaze(char maze[MAZE_HEIGHT][MAZE_WIDTH], int xpos, int ypos)
{
   if (xpos < 0 || xpos >= MAZE_HEIGHT || ypos < 0 || ypos >= MAZE_WIDTH)
      return 0;

   if (maze[xpos][ypos] == 'E')  // Found the Exit!
      return 1;

   if (maze[xpos][ypos] != ' ')  // Space is not empty (possibly X or V)
      return 0;

   maze[xpos][ypos]='V';         // Mark this space as visited

   // Go Down
   if (ExitMaze(maze, xpos + 1, ypos)) {
      maze[xpos][ypos]='P';
      return 1;
   }

   // Go Right
   if (ExitMaze(maze, xpos, ypos + 1)) {
      maze[xpos][ypos]='P';
      return 1;
   }

   // Go Up
   if (ExitMaze(maze, xpos - 1, ypos)) {
      maze[xpos][ypos]='P';
      return 1;
   }

   // Go Left
   if (ExitMaze(maze, xpos, ypos - 1)) {
      maze[xpos][ypos]='P';
      return 1;
   }

   // No path to Exit
   return 0;
}
```

Figure 17.19 **A recursive C function to find an escape path in a maze.**

recursive calls (one call for each direction). So each node can end up creating up to four new invocations of ExitMaze. Figure 17.20 provides a graphical depiction of `ExitMaze` in action. The initial call shows the initial maze configuration, along with the starting point (`maze[1][2]`). That initial call generates four new calls,

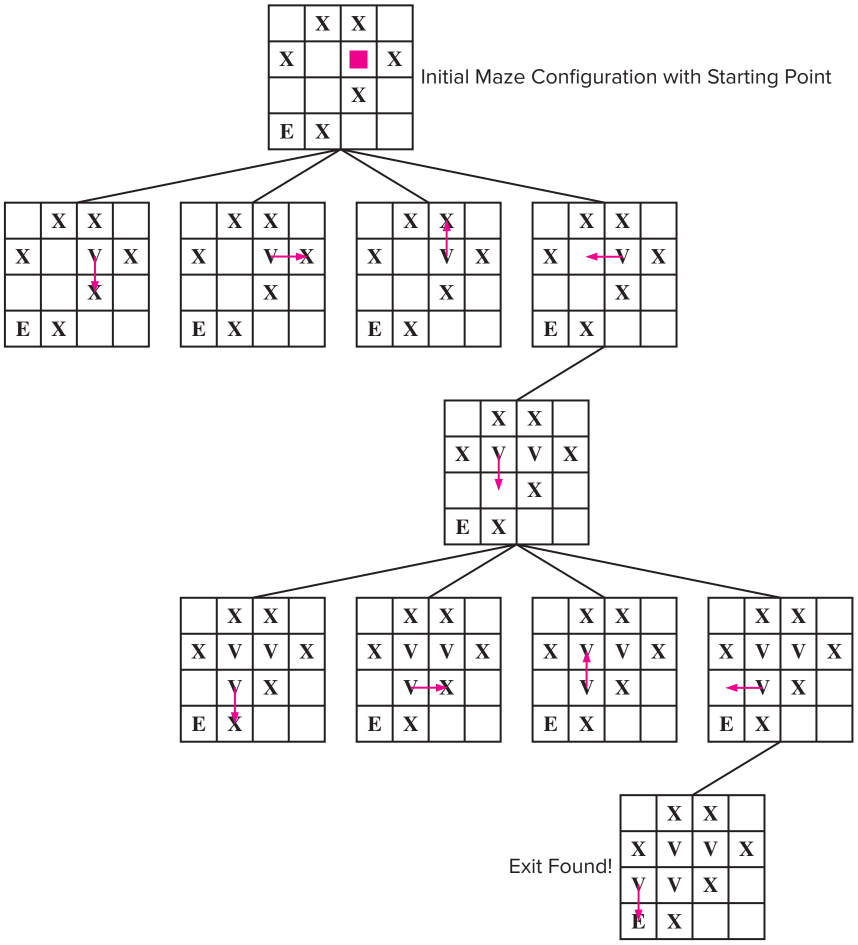


Figure 17.20 The total set of recursive calls made by our maze solver.

each evaluated one after the other. The first call (move down) returns immediately, as does the next call (move right), as does the third call (move up). The fourth call (move left) is a move into an open space, and thus it causes new calls to be generated.

17.8 Summary

We initially examined recursion in the context of LC-3 assembly language in Chapter 8. In this chapter, we examine it again in the context of C. With recursion, it is important for us to develop an understanding of the types of problems that are amenable to a recursive solution and also to understand the overheads involved with recursion.

We can solve a problem recursively by using a function that calls itself on smaller subproblems. With recursion, we state the function, say $f(n)$, in terms of the same function on smaller values of n, say for example, $f(n-1)$. The Fibonacci series, for example, is recursively stated as

$$\text{Fibonacci}(n) = \text{Fibonacci}(n-1) + \text{Fibonacci}(n-2);$$

For the recursion to eventually terminate, recursive calls require a base case. Recursion is a powerful programming tool that, when applied to the right problem, can make the task of programming considerably easier. For example, the Towers of Hanoi puzzle can be solved in a simple manner with recursion. It is much harder to formulate using iteration. In future courses, you will examine ways of organizing data involving pointers (e.g., trees and graphs) where the simplest techniques to manipulate the data structure involve recursive functions. At the lower levels, recursive functions are handled in exactly the same manner as any other function call. The run-time stack mechanism enables this by systematically allocating a stack frame for each function invocation, providing storage for each invocation such that it doesn't interfere with storage for any other invocation.

Exercises

17.1 For these questions, refer to the examples that appear in the chapter.

a. How many calls to `RunningSum` are made for the call `RunningSum(10)`?
b. How about for the call `RunningSum(n)`? Give your answer in terms of n.
c. How many calls to `MoveDisk` are made in the Towers of Hanoi problem if the initial call is `MoveDisk(4, 1, 3, 2)`? This call plays out a four-disk game.
d. How many calls are made for an n-disk game?
e. How many calls to `Fibonacci` (see Figure 17.13) are made for the initial call `Fibonacci(10)`?
f. How many calls are required for the nth Fibonacci number?
g Is a square with a 'P' ever encountered at line 12 of the `ExitMaze` code?

17.2 Is the return address for a recursive function always the same at each function call? Why or why not?

17.3 What is the maximum number of recursive calls made to solve a maze using `ExitMaze`?

17.4 What does the following function produce for `count(20)`?

```
int count(int arg)
{
   if (arg < 1)
      return 0;
   else if (arg % 2)
      return(1 + count(arg - 2));
   else
      return(1 + count(arg - 1));
}
```

17.5 Consider the following C program, and the run-time stack in Figure 17.21:

```
#include <stdio.h>
int Power(int a, int b);
int main(void)
{
   int x, y, z;

   printf("Input two numbers: ");
   scanf("%d %d", &x, &y);
   if ((x > 0) && (y > 0))
      z = Power(x,y);
   else
      z = 0;
   printf("The result is %d.\n", z);
}

int Power(int a, int b)
{
   if (a < b)
      return 0;
   else
      return 1 + Power(a/b, b);
}
```

a. State the complete output if the input is
   (1) 4 9
   (2) 27 5
   (3) −1 3

b. What does the function `Power` compute?

c. Figure 17.21 is a snapshot of the stack after a call to the function `Power`. Two stack frames are shown, with some of the entries filled in. Assume the snapshot was taken just before execution of one of the `return` statements in `Power`. What are the values in the entries marked with a question mark? If an entry contains an address, use an arrow to indicate the location the address refers to.

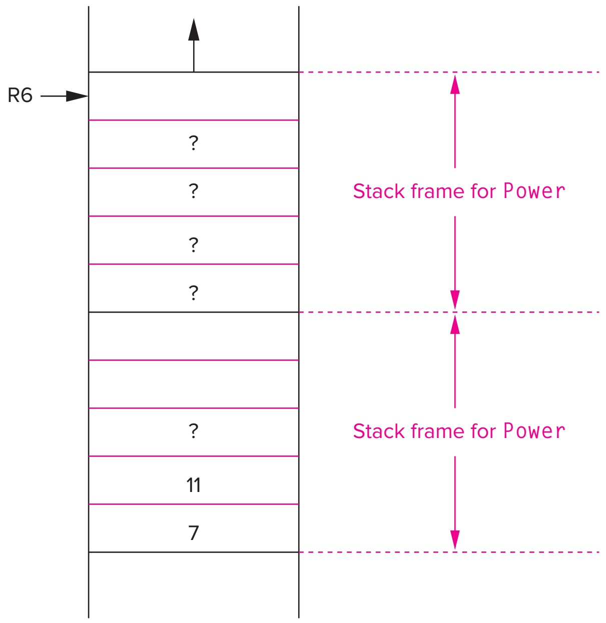


Figure 17.21 **Run-time stack after the function** `Power` **is called.**

17.6 Consider the following C function:

```
int Sigma( int k )
{
   int l;
   l = k - 1;
   if (k==0)
      return 0;
   else
      return (k + Sigma(l));
}
```

a. Convert the recursive function into a nonrecursive function. Assume `Sigma()` will always be called with a nonnegative argument.

b. Exactly 1 KB of contiguous memory is available for the run-time stack, and addresses and integers are 16 bits wide. How many recursive function calls can be made before the program runs out of stack space? Assume no storage is needed for temporary values.

17.7 The following C program is compiled and executed on the LC-3. When the program is executed, the run-time stack starts at memory location xFEFF and grows toward xC000 (the stack can occupy up to 16 KB of memory).

```
int SevenUp(int x)
{
   if (x == 1)
      return 7;
   else
      return (7 + sevenUp(x - 1));
}

int main(void)
{
   int a;

   printf("Input a number \n");
   scanf("%d", &a);
   a = SevenUp(a);
   printf("%d is 7 times the number\n", a);
}
```

a. What is the largest input value for which this program will run correctly? Explain your answer.
b. If the run-time stack can occupy only 4 KB of memory, what is the largest input value for which this program will run correctly?

17.8 Write an iterative version of a function to find the *n*th Fibonacci number. Plot the running time of this iterative version to the running time of the recursive version on a variety of values for *n*. Why is the recursive version significantly slower?

17.9 The binary search routine shown in Figure 17.16 searches through an array that is in ascending order. Rewrite the code so that it works for arrays in descending order.

17.10 Following is a very famous algorithm whose recursive version is significantly easier to express than the iterative one. It was originally proposed by the Greek mathematician Euclid! For the following subproblems, provide the final value returned by the function.

```
int ea(int x, int y)
{
    if (y == 0)
       return x;
    else
       return ea(y, x % y);
}
```

a. `ea(12, 15)`
b. `ea(6, 10)`
c. `ea(110, 24)`

d. What does this function calculate? Consider how you might construct an iterative version to calculate the same thing.

17.11 Write a program without recursive functions equivalent to the following C program:

```
int main(void)
{
   printf("%d", M());
}

void M()
{
   int num, x;
   printf("Type a number: ");
   scanf("%d", &num);
   if (num <= 0)
      return 0;
   else {
      x = M();
      if (num > x)
         return num;
      else
         return x;
   }
}
```

17.12 Consider the following recursive function:

```
int func (int arg)
{
   if (arg % 2 != 0)
      return func(arg - 1);
   if (arg <= 0)
      return 1;
   return func(arg/2) + 1;
}
```

a. Is there a value of `arg` that causes an infinite recursion? If so, what is it?

b. Suppose that the function `func` is part of a program whose `main` function is as follows. How many function calls are made to `func` when the program is executed?

```
int main(void)
{
   printf("The value is %d\n", func(10));
}
```

c. What value is output by the program?

17.13 A magic square is an $n \times n$ grid where each cell contains one of the integers between 1 and n^2. Each cell contains a different integer, and the sum of the cells in a column, in a row, and on each diagonal is equal. Given a partial solution to a magic square (provided as a 2D integer array), find a complete solution if it exists. Use a recursive technique to build a solver.

17.14 What is the output of the following C program?

```
#include <stdio.h>
void Magic(int in);
int Even(int n);

int main(void)
{
   Magic(10);
}

void Magic(int in)
{
   if (in == 0)
      return;
   if (Even(in))
      printf("%i\n", in);
   Magic(in - 1);
   if (!Even(in))
      printf("%i\n", in);
   return;
}

int Even(int n)
{
  if (n % 2) return 1 else 0;
}
```