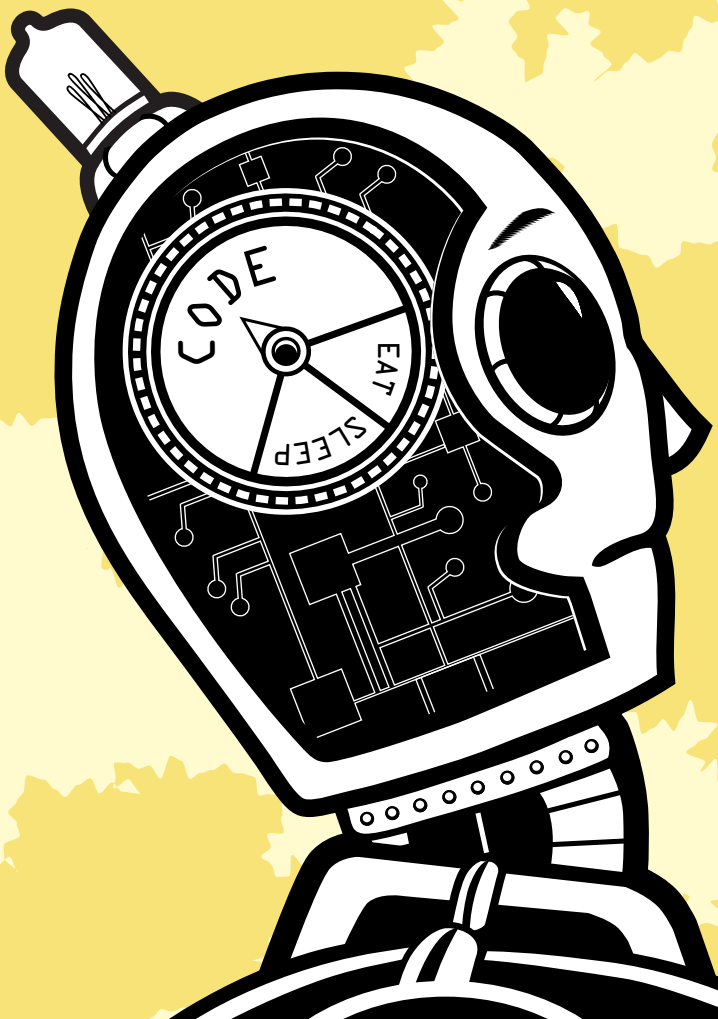
THINK LIKE A PROGRAMMER

AN INTRODUCTION TO
CREATIVE PROBLEM SOLVING

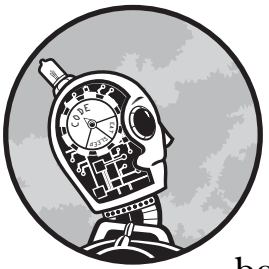
V. ANTON SPRAUL



no starch press

6

SOLVING PROBLEMS WITH RECURSION



This chapter is about *recursion*, which is when a function directly or indirectly calls itself. Recursive programming looks as if it should be simple. Indeed, a good recursive solution often has a simple, almost elegant appearance. However, very often the route to that solution is anything but simple. This is because recursion requires us to think differently than we do with other types of programming. When we process data using loops, we're thinking about processing in a sequential manner, but when we process data using recursion, our normal sequential thinking process won't help. Many good, fledgling programmers struggle with recursion because they can't see a way to apply the problem-solving skills they've learned to recursive problems. In this chapter, we'll discuss how to attack recursive problems systematically. The answer is using what we will call the *Big Recursive Idea*, henceforth referred to as the BRI. It's an idea that's so straightforward it will seem like a trick, but it works.

Review of Recursion Fundamentals

There is not much to know about the *syntax* of recursion; the difficulty arises when you try to use recursion to solve problems. Recursion occurs any time a function calls itself, so the syntax of recursion is just the syntax of a function call. The most common form is *direct recursion*, when a call to a function occurs in the body of that same function. For example:

```
int factorial(int n) {
  ❶if (n == 1) return 1;
   else return n * ❷factorial(n - 1);
}
```

This function, which is a common but highly inefficient demonstration of recursion, computes the factorial of *n*. For example, if *n* is 5, then the factorial is the product of all the numbers from 5 to 1, or 120. Note that in some cases no recursion occurs. In this function, if the parameter is 1, we simply return a value directly without any recursion ❶, which is known as a *base case*. Otherwise, we make the recursive call ❷.

The other form of recursion is *indirect recursion*—for example, if function A calls function B, which later calls function A. Indirect recursion is rarely used as a problem-solving technique, so we won't cover it here.

Head and Tail Recursion

Before we discuss the BRI, we need to understand the difference between head recursion and tail recursion. In *head recursion*, the recursive call, when it happens, comes before other processing in the function (think of it happening at the top, or head, of the function). In *tail recursion*, it's the opposite—the processing occurs before the recursive call. Choosing between the two recursive styles may seem arbitrary, but the choice can make all the difference. To illustrate this difference, let's look at two problems.

PROBLEM: HOW MANY PARROTS?

Passengers on the Tropical Paradise Railway (TPR) look forward to seeing dozens of colorful parrots from the train windows. Because of this, the railway takes a keen interest in the health of the local parrot population and decides to take a tally of the number of parrots in view of each train platform along the main line. Each platform is staffed by a TPR employee (see Figure 6-1), who is certainly capable of counting parrots. Unfortunately, the job is complicated by the primitive telephone system. Each platform can call only its immediate neighbors. How do we get the parrot total at the main line terminal?

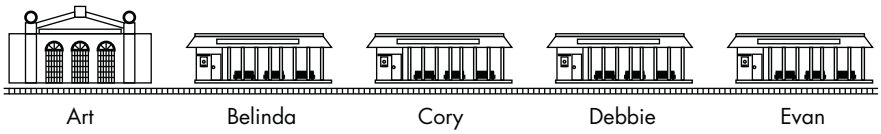


Figure 6-1: The employees at the five stations can communicate only with their immediate neighbors.

Let's suppose that there are 7 parrots by Art at the main terminal, 5 parrots by Belinda, 3 parrots by Cory, 10 parrots by Debbie, and 2 parrots by Evan at the last station. The total number of parrots is thus 27. The question is, how are the employees going to work together to communicate this total to Art? Any solution to this problem is going to require a chain of communications all the way from the main terminal to the end of the line and back. The staff member at each platform will be requested to count parrots and will then report his or her observations. Even so, there are two distinct approaches to this communications chain, and those approaches correspond to the head recursion and tail recursion techniques in programming.

Approach 1

In this approach, we keep a running total of the parrots as we progress through the outbound communications. Each employee, when making the request of the next employee down the line, passes along the number of parrots seen so far. When we get to the end of the line, Evan will be the first to discover the parrot total, which he will pass up to Debbie, who will pass it to Cory, and so on (as shown in Figure 6-2).

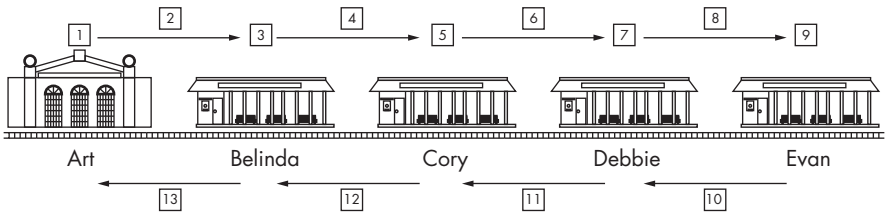


Figure 6-2: Numbering of steps taken in Approach 1 for the parrot-counting problem

1. ART begins by counting the parrots around his platform. He counts 7 parrots.
2. ART to BELINDA: "There are 7 parrots here at the main terminal."
3. BELINDA counts 5 parrots around her platform for a running total of 12.
4. BELINDA to CORY: "There are 12 parrots around the first two stations."
5. CORY counts 3 parrots.

6. CORY to DEBBIE: "There are 15 parrots around the first three stations."
7. DEBBIE counts 10 parrots.
8. DEBBIE to EVAN: "There are 25 parrots around the first four stations."
9. EVAN counts 2 parrots and discovers that the total number of parrots is 27.
10. EVAN to DEBBIE: "The total number of parrots is 27."
11. DEBBIE to CORY: "The total number of parrots is 27."
12. CORY to BELINDA: "The total number of parrots is 27."
13. BELINDA to ART: "The total number of parrots is 27."

This approach is analogous to tail recursion. In tail recursion, the recursive call happens after the processing—the recursive call is the last step in the function. In the communications chain above, note that the "work" of the employees—the parrot counting and summation—happens before they signal the next employee down the line. All of the work happens on the outbound communications chain, not the inbound chain. Here are the steps each employee follows:

1. Count the parrots visible from the station platform.
2. Add this count to the total given by the previous station.
3. Call the next station to pass along the running sum of parrot counts.
4. Wait for the next station to call with the total parrot count, and then pass this total up to the previous station.

Approach 2

In this approach, we sum the parrot counts from the other end. Each employee, when contacting the next station down the line, requests the total number of parrots from that station onward. The employee then adds the number of parrots at his or her own station and passes this new total up the line (as shown in Figure 6-3).

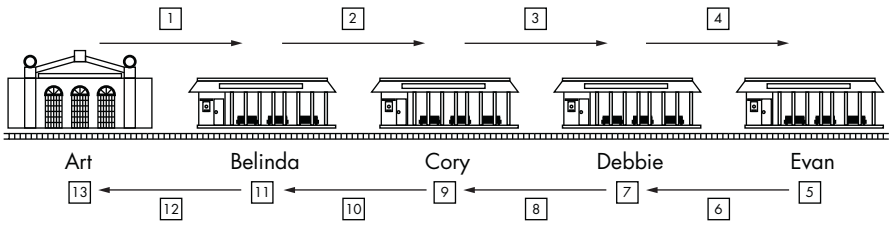


Figure 6-3: Numbering of steps taken in Approach 2 for the parrot-counting problem

1. ART to BELINDA: "What's the total number of parrots from your station to the end of the line?"
2. BELINDA to CORY: "What's the total number of parrots from your station to the end of the line?"

3. CORY to DEBBIE: "What's the total number of parrots from your station to the end of the line?"
4. DEBBIE to EVAN: "What's the total number of parrots from your station to the end of the line?"
5. EVAN is the end of the line. He counts 2 parrots.
6. EVAN to DEBBIE: "The total number of parrots here at the end is 2."
7. DEBBIE counts 10 parrots at her station, so the total from her station to the end is 12.
8. DEBBIE to CORY: "The total number of parrots from here to the end is 12."
9. CORY counts 3 parrots.
10. CORY to BELINDA: "The total number of parrots from here to the end is 15."
11. BELINDA counts 5 parrots.
12. BELINDA to ART: "The total number of parrots from here to the end is 20."
13. ART counts 7 parrots at the main terminal, making a total of 27.

This approach is analogous to head recursion. In head recursion, the recursive call happens before the other processing. Here, the call to the next station happens first, before counting the parrots or the summation. The "work" is postponed until after the stations down the line have reported their totals. Here are the steps each employee follows:

1. Call the next station.
2. Count the parrots visible from the station platform.
3. Add this count to the total given by the next station.
4. Pass the resulting sum up to the previous station.

You may have noticed two practical effects of the different approaches. In the first approach, eventually all of the station employees will learn the overall parrot total. In the second approach, only Art, at the main terminal, learns the full total—but note that Art is the only employee who needs the full total.

The other practical effect will become more important for our analysis when we transition the discussion to actual programming code. In the first approach, each employee passes along the "running total" to the next station down the line when making the request. In the second approach, the employee simply makes the request for information from the next station, without passing any data down the line. This effect is typical of the head recursion approach. Because the recursive call happens first, before any other processing, there is no new information to give the recursive call. In general, the head recursion approach allows the minimum set of data to be passed to the recursive call. Now let's look at another problem.

PROBLEM: WHO'S OUR BEST CUSTOMER?

The manager of DelegateCorp needs to determine which of eight customers produces the most revenue for his company. Two factors complicate this otherwise simple task. First, determining the total revenue for a customer requires going through that customer's whole file and tallying numbers on dozens of orders and receipts. Second, the employees of DelegateCorp, as the name suggests, love to delegate, and each employee passes work along to someone at a lower level whenever possible. To keep the situation from getting out of hand, the manager enforces a rule: When you delegate, you must do some portion of the work yourself, and you have to give the delegated employee less work than you were given.

Tables 6-1 and 6-2 identify the employees and customers of DelegateCorp.

Table 6-1: DelegateCorp Employee Titles and Rank

Title	Rank
Manager	1
Vice manager	2
Associate manager	3
Assistant manager	4
Junior manager	5
Intern	6

Table 6-2: DelegateCorp Customers

Customer Number	Revenue
#0001	$172,000
#0002	$68,000
#0003	$193,000
#0004	$13,000
#0005	$256,000
#0006	$99,000

Following the company rule on delegating work, here's what will happen to the six customer files. The manager will take one file and determine how much revenue that customer has generated for the company. The manager will delegate the other five files to the vice manager. The vice manager will process one file and pass the other four to the associate manager. This process continues until we reach the sixth employee, the intern, who is handed one file and must simply process it, with no further delegation possible.

Figure 6-4 describes the lines of communication and the division of labor. As with the previous example, though, there are two distinct approaches to the communications chain.

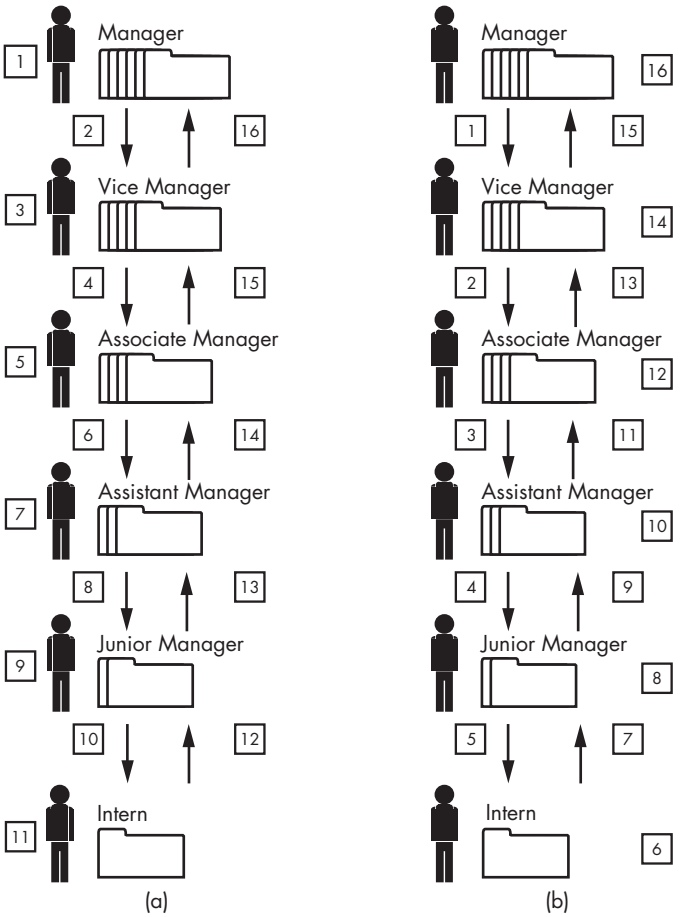


Figure 6-4: The numbering of steps in Approach 1 (a) and Approach 2 (b) for finding the highest-revenue customer

Approach 1

In this approach, when delegating the remaining files, the employee also passes along the highest amount of revenue seen so far. This means that the employee must tally the revenue in one file and compare this to the previous highest amount seen before delegating the remaining files to another employee. Here's an example of how this would proceed in practice.

1. MANAGER tallies the revenue for customer #0001, which is $172,000.
2. MANAGER to VICE MANAGER: "The highest revenue we have seen so far is $172,000, customer #0001. Take these five files and determine the overall highest revenue."
3. VICE MANAGER tallies the revenue for customer #0002, which is $68,000. The highest revenue seen so far is still $172,000, customer #0001.

4. VICE MANAGER to ASSOCIATE MANAGER: "The highest revenue we have seen so far is $172,000, customer #0001. Take these four files and determine the overall highest revenue."
5. ASSOCIATE MANAGER tallies the revenue for customer #0003, which is $193,000. The highest revenue seen so far is now $193,000, customer #0003.
6. ASSOCIATE MANAGER to ASSISTANT MANAGER: "The highest revenue we have seen so far is $193,000, customer #0003. Take these three files and determine the overall highest revenue."
7. ASSISTANT MANAGER tallies the revenue for customer #0004, which is $13,000. The highest revenue seen so far is still $193,000, customer #0003.
8. ASSISTANT MANAGER to JUNIOR MANAGER: "The highest revenue we have seen so far is $193,000, customer #0003. Take these two files and determine the overall highest revenue."
9. JUNIOR MANAGER tallies the revenue for customer #0005, which is $256,000. The highest revenue seen so far is now $256,000, customer #0005.
10. JUNIOR MANAGER to INTERN: "The highest revenue we have seen so far is $256,000, customer #0005. Take this remaining file and determine the overall highest revenue."
11. INTERN tallies the revenue for customer #0006, which is $99,000. The highest revenue seen so far is still $256,000, customer #0005.
12. INTERN to JUNIOR MANAGER: "The highest revenue of all customers is $256,000, customer #0005."
13. JUNIOR MANAGER to ASSISTANT MANAGER: "The highest revenue of all customers is $256,000, customer #0005."
14. ASSISTANT MANAGER to ASSOCIATE MANAGER: "The highest revenue of all customers is $256,000, customer #0005."
15. ASSOCIATE MANAGER to VICE MANAGER: "The highest revenue of all customers is $256,000, customer #0005."
16. VICE MANAGER to MANAGER: "The highest revenue of all customers is $256,000, customer #0005."

This approach, shown in Figure 6-4 (a), uses the tail recursion technique. Each employee processes one customer file and compares the computed revenue for that customer against the highest revenue seen so far. Then the employee passes the result of that comparison to the subordinate employee. The recursion—the passing off of work—happens after the other processing. Each employee's process runs like this:

1. Tally the revenue in one customer file.
2. Compare this total with the highest revenue seen by superiors in other customer files.
3. Pass the remaining customer files to a subordinate employee, along with the highest revenue amount seen so far.
4. When the subordinate employee returns the highest revenue of all the customer files, pass this back to the superior.

Approach 2

In this approach, each employee begins by setting aside one file and then passing the others to the subordinate. In this case, the subordinate isn't asked to determine the highest revenue of all the files, just of the files the subordinate has been given. As with the first sample problem, this simplifies the requests. Using the same data as the first approach, the conversation would be as follows:

1. MANAGER to VICE MANAGER: "Take these five customer files, and tell me the highest revenue."
2. VICE MANAGER to ASSOCIATE MANAGER: "Take these four customer files, and tell me the highest revenue."
3. ASSOCIATE MANAGER to ASSISTANT MANAGER: "Take these three customer files, and tell me the highest revenue."
4. ASSISTANT MANAGER to JUNIOR MANAGER: "Take these two customer files, and tell me the highest revenue."
5. JUNIOR MANAGER to INTERN: "Take this one customer file, and tell me the highest revenue."
6. INTERN tallies the revenue for customer #0006, which is $99,000. This is the only file the INTERN has seen, so that's the highest revenue.
7. INTERN to JUNIOR MANAGER: "The highest revenue in my files is $99,000, customer #0006."
8. JUNIOR MANAGER tallies the revenue for customer #0005, which is $256,000. The highest revenue this employee knows about is $256,000, customer #0005.
9. JUNIOR MANAGER to ASSISTANT MANAGER: "The highest revenue in my files is $256,000, customer #0005."
10. ASSISTANT MANAGER tallies the revenue for customer #0004, which is $13,000. The highest revenue this employee knows about is $256,000, customer #0005.
11. ASSISTANT MANAGER to ASSOCIATE MANAGER: "The highest revenue in my files is $256,000, customer #0005."
12. ASSOCIATE MANAGER tallies the revenue for customer #0003, which is $193,000. The highest revenue this employee knows about is $256,000, customer #0005.
13. ASSOCIATE MANAGER to VICE MANAGER: "The highest revenue in my files is $256,000, customer #0005."
14. VICE MANAGER tallies the revenue for customer #0002, which is $68,000. The highest revenue this employee knows about is $256,000, customer #0005.
15. VICE MANAGER to MANAGER: "The highest revenue in my files is $256,000, customer #0005."
16. MANAGER tallies the revenue for customer #0001, which is $172,000. The highest revenue this employee knows about is $256,000, customer #0005.

This approach, shown in Figure 6-4 (b), uses the head recursion technique. Each employee still has to tally the revenue in one customer file, but that action is postponed until after the subordinate employee determines the highest revenue among the remaining files. The process each employee takes is as follows:

1. Pass all customer files except one to a subordinate employee.
2. Get the highest revenue of those files back from the subordinate employee.
3. Tally the revenue in the one customer file.
4. Pass the larger of those two revenues to the superior.

As in the "counting parrots" problem, the head recursion technique allows each employee to pass the minimum amount of information to the subordinate.

The Big Recursive Idea

We now arrive at the Big Recursive Idea. In fact, if you've read through the steps of the sample problems, you have already seen the BRI in action.

How so? Both of the sample problems follow the form of a recursive solution. Each person in the communications chain performs the same steps on a smaller and smaller subset of the original data. It's important to note, however, that *the problems involve no recursion at all.*

In the first problem, each railway employee makes a request of the next station down the line, and in fulfilling that request, the next employee follows the same steps as the previous employee. But nothing in the wording of the request requires an employee to follow those particular steps. When Art called Belinda using Approach 2, for example, he asked her to count the total number of parrots from her station to the end of the line. He did not dictate a method for discovering this total. If he thought about it, he might have realized that Belinda would have to follow the same steps that he himself was following, but he doesn't have to consider this. To complete his task, all Art required was for Belinda to provide the correct answer to the question he asked.

Likewise, in the second problem, each employee in the management chain hands off as much work as possible to a subordinate. The assistant manager, for example, may know the junior manager well and expect the junior manager to hand all of the files but one to the intern. However, the assistant manager has no reason to care whether the junior manager processes all of the remaining files or passes some of them off to a subordinate. The assistant manager cares only that the junior manager returns the right answer. Because the assistant manager is not going to repeat the work of the junior manager, the assistant manager simply assumes that the result returned by the junior manager is correct and uses that data to solve the overall task that the assistant manager received from the associate manager.

In both problems, when employees make requests of other employees, they are concerned with *what* but not *how*. A question is handed off; an answer is received. This, then, is the Big Recursive Idea: If you follow certain conventions in your coding, *you can pretend that no recursion is taking place*. You can even use a cheap trick (shown below) to move from an iterative implementation to a recursive implementation, without explicitly considering how the recursion is actually solving the problem. Over time, you will develop an intuitive understanding of how recursive solutions work, but before that intuition develops, you can craft recursive implementations and be confident in your code.

Let's put the concept into practice through a code example.

PROBLEM: COMPUTING THE SUM OF AN ARRAY OF INTEGERS

Write a recursive function that is given an array of integers and the size of the array as parameters. The function returns the sum of the integers in the array.

Your first thought may have been that this problem would be trivial to solve iteratively. Indeed, let's start with an iterative solution to this problem:

```
int iterativeArraySum(int integers[], int size) {
    int sum = 0;
    for (int i = 0; i < size; i++) {
        sum += integers[i];
    }
    return sum;
}
```

You saw code very similar to this in Chapter 3, so the function should be simple to understand. The next step is to write code that is halfway between the iterative solution and the final desired recursive solution. We will keep the iterative function and add a second function we will refer to as a *dispatcher*. The dispatcher will hand off most of the work to the previously written iterative function and use this information to solve the overall problem. To write a dispatcher, we have to follow two rules:

1. The dispatcher must completely handle the most trivial case, without calling the iterative function.
2. The dispatcher, when calling the iterative function, must pass a smaller version of the problem.

In applying the first rule to this problem, we must decide what the most trivial case is. If `size` is 0, then the function has conceptually been passed a "null" array, with a sum of 0. One could also make the argument that the most trivial case should be when `size` is 1. In that case, there would be only one number in the logical array, and we could return that number as the

sum. Either of these interpretations will work, but making the first choice allows the function to handle a special case. Note that the original iterative function will not fail when `size` is zero, so it would be preferable to maintain that flexibility.

To apply the second rule to this problem, we must figure out a way to pass a smaller version of the problem from the dispatcher to the iterative function. There is no easy way to pass a smaller array, but we can easily pass a smaller value for `size`. If the dispatcher is given the value of 10 for `size`, the function is being asked to compute the sum of 10 values in the array. If the dispatcher passes 9 as the value of `size` to the iterative function, it is requesting the sum of the first 9 values in the array. The dispatcher can then add the value of the one remaining value in the array (the 10th) to compute the sum of all 10 values. Note that reducing size by 1 when calling the iterative function maximizes the work of the iterative function and thereby minimizes the work of the dispatcher. This is always the desired approach—like the managers of DelegateCorp, the dispatcher function avoids as much work as possible.

Putting these ideas together, here's a dispatcher function for this problem:

```
int arraySumDelegate(int integers[], int size) {
  ❶if (size == 0) return 0;
  ❷int lastNumber = integers[size - 1];
    int allButLastSum = ❸iterativeArraySum(integers, size - 1);
  ❹return lastNumber + allButLastSum;
}
```

The first statement enforces the first rule of dispatchers: It checks for a trivial case and handles it completely, in this case, by returning 0 ❶. Otherwise, control passes to the remaining code, which enforces the second rule. The last number in the array is stored in a local variable called `lastNumber` ❷, and then the sum of all the other values in the array is computed via a call to the iterative function ❸. This result is stored in another local variable, `allButLastSum`, and finally the function returns the sum of the two local variables ❹.

If we have correctly created a dispatcher function, we have already effectively created a recursive solution. This is the Big Recursive Idea in action. To convert this iterative solution to a recursive solution requires but one further, simple step: have the delegate function call itself where it was previously calling the iterative function. We can then remove the iterative function altogether.

```
int ❶arraySumRecursive(int integers[], int size) {
    if (size == 0) return 0;
    int lastNumber = integers[size - 1];
    int allButLastSum = ❷arraySumRecursive(integers, size - 1);
    return lastNumber + allButLastSum;
}
```

Only two changes have been made to the previous code. The name of the function has been changed to better describe its new form ❶, and the function now calls itself where it previously called the iterative function ❷. The logic of the two functions, `arraySumDelegate` and `arraySumRecursive`, is identical. Each function checks for a trivial case in which the sum is already known—in this case, an array of size 0 that has a sum of 0. Otherwise, each function computes the sum of values in the array by making a function call to compute the sum of all of the values, save the last one. Finally, each function adds that last value to the returned sum for a grand total. The only difference is that the first version of the function calls another function, while the recursive version calls itself. The BRI tells us that if we follow the rules outlined above for writing the dispatcher, we can ignore that distinction.

You do not need to literally follow all of the steps shown above to follow the BRI. In particular, you usually would not implement an iterative solution to the problem before implementing a recursive solution. Writing an iterative function as a stepping-stone is extra work that will eventually be thrown away. Besides, recursion is best applied to situations in which an iterative solution is difficult, as explained later. However, you can follow the outline of the BRI without actually writing the iterative solution. The key is thinking of a recursive call as a call to another function, without regards to the internals of that function. In this way, you remove the complexities of recursive logic from the recursive solution.

Common Mistakes

As shown above, with the right approach, recursive solutions can often be very easy to write. But it can be just as easy to come up with an incorrect recursive implementation or a recursive solution that "works" but is ungainly. Most problems with recursive implementations stem from two basic faults: overthinking the problem or beginning implementation without a clear plan.

Overthinking recursive problems is common for new programmers because limited experience and lack of confidence with recursion lead them to think that the problem is more difficult than it really is. Code produced by overthinking can be recognized by its too-careful appearance. For example, a recursive function might have several special cases where it needs only one.

Beginning implementation too soon can lead to overcomplicated "Rube Goldberg" code, where unforeseen interactions lead to fixes that are bolted onto the original code.

Let's look at some specific mistakes and how to avoid them.

Too Many Parameters

As described previously, the head recursion technique can reduce the data passed to the recursive call, while the tail recursion technique can result in passing additional data to recursive calls. Programmers often get stuck in the tail recursion mode because they overthink and start implementation too soon.

Consider our problem of recursively computing the sum of an array of integers. Writing an iterative solution to this problem, the programmer knows a "running total" variable will be needed (in the iterative solution provided, I called this `sum`) and the array will be summed starting from the first element. Considering the recursive solution, the programmer naturally imagines an implementation that most directly mirrors the iterative solution, with a running total variable and the first recursive call handling the first element in the array. This approach, however, requires the recursive function to pass the running total and the location where the next recursive call should begin processing. Such a solution would look like this:

```
int arraySumRecursiveExtraParams(int integers[], int size, ❶int sum, ❷int currentIndex) {
    if (currentIndex == size) return sum;
    sum += integers[currentIndex];
    return arraySumRecursiveExtraParameters(integers, size, sum, currentIndex + 1);
}
```

This code is as short as the other recursive version but considerably more semantically complex because of the additional parameters, `sum` ❶ and `currentIndex` ❷. From the client code's point of view, the extra parameters are meaningless and will always have to be zeroes in the call, as shown in this example:

```
int a[10] = {20, 3, 5, 22, 7, 9, 14, 17, 4, 9};
int total = arraySumRecursiveExtraParameters(a, 10, 0, 0);
```

This problem can be avoided with the use of a *wrapper function*, as described in the next section, but because we can't eliminate those parameters altogether, that's not the best solution. The iterative function for this problem and the original recursive function answer the question, what is the sum of this array with this many elements? In contrast, this second recursive function is being asked, what is the sum of this array if it has this many elements, we are starting with this particular element, and this is the sum of all the prior elements?

The "too many parameters" problem is avoided by choosing your function parameters before thinking about recursion. In other words, force yourself to use the same parameter list you would if the solution were iterative. If you use the full BRI process and actually write the iterative function first, you will avoid this problem automatically. If you skip using the whole process formally, though, you can still use the idea conceptually if you write out the parameter list based on what you would expect for an iterative function.

Global Variables

Avoiding too many parameters sometimes leads programmers into making a different mistake: using global variables to pass data from one recursive call to the other. The use of global variables is generally a poor programming practice, although it is sometimes permissible for performance reasons. Global

variables should always be avoided in recursive functions when possible. Let's look at a specific problem to see how programmers talk themselves into this mistake. Suppose we were asked to write a recursive function that counted the number of zeros appearing in an array of integers. This is a simple problem to solve using iteration:

```
int zeroCountIterative(int numbers[], int size) {
   int sum = 0;
 ❶int count = 0;
   for (int i = 0; i < size; i++) {
      if (numbers[i] == 0) count ++;
   }
   return count;
}
```

The logic of this code is straightforward. We're just running through the array from the first location to the last, counting up the zeroes as we go and using a local variable, `count` ❶, as a tracker. If we have a function like this in our minds when we write our recursive function, though, we may assume that we need a tracker variable in that version as well. We can't simply declare count as a local variable in the recursive version because then it would be a new variable in each recursive call. So we might be tempted to declare it as a global variable:

```
int count;
int zeroCountRecursive(int numbers[], int size) {
   if (size == 0) return count;
   if (numbers[size - 1] == 0) count++;
   zeroCountRecursive(numbers, size - 1);
}
```

This code works, but the global variable is entirely unnecessary and causes all the problems global variables typically cause, such as poor readability and more difficult code maintenance. Some programmers might attempt to mitigate the problem by making the variable local, but static:

```
int zeroCountStatic(int numbers[], int size) {
 ❶static int count ❷= 0;
   if (size == 0) return count;
   if (numbers[size - 1] == 0) count++;
   zeroCountStatic(numbers, size - 1);
}
```

In C++, a local variable declared as *static* retains its value from one function call to the next; thus, the local static variable `count` ❶ would act the same as the global variable in the previous version. So what's the problem? The initialization of the variable to zero ❷ happens only the first time the function is called. This is necessary for the `static` declaration to be of any use, but it means that the function will return a correct answer only the first time it is called. If this function were called twice—first with an array that had three

zeros, then with an array that had five zeros—the function would return an answer of eight for the second array because `count` would be starting where it had left off.

The solution to avoiding the global variable in this case is to use the BRI. We can assume that a recursive call with a smaller value for `size` will return the correct result and compute the correct value for the overall array from that. This will lead to a head-recursive solution:

```
int zeroCountRecursive(int numbers[], int size) {
   if (size == 0) return 0;
 ❶int count = zeroCountRecursive(numbers, size - 1);
 ❷if (numbers[size - 1] == 0) count++;
 ❸return count;
}
```

In this function, we still have a local variable, `count` ❶, but here no attempt is made to maintain its value from one call to the next. Instead, it stores the return value from our recursive call; we optionally increment the variable ❷ before returning it ❸.

Applying Recursion to Dynamic Data Structures

Recursion is often applied to dynamic structures such as linked lists, trees, and graphs. The more complicated the structure, the more the coding can benefit from a recursive solution. Processing complicated structures is often a lot like finding one's way through a maze, and recursion allows us to backtrack to previous steps in our processing.

Recursion and Linked Lists

Let's start, though, with the most basic of dynamic structures, a linked list. For discussions in this section, let's assume we have the simplest of node structures for our linked list, just a single `int` for data. Here are our type declarations:

```
struct listNnode {
    int data;
    listNode * next;
};
typedef listNode * listPtr;
```

Applying the BRI to a singly linked list follows the same general outline regardless of the specific task. Recursion requires us to divide the problem, to be able to pass a reduced version of the original problem to the recursive call. There is only one practical way to divide a singly linked list: the first node in the list and the rest of the list.

In Figure 6-5, we see a sample list divided into unequal parts: the first node and all of the other nodes. Conceptually, we can view the "rest of" the original list as its own list, starting with the second node in the original list. It is this view that allows the recursion to work smoothly.

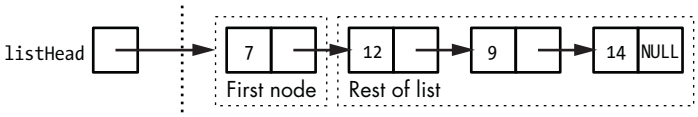


Figure 6-5: A list divided into a first node and "the rest of the list"

Again, though, we are not required to picture all the steps of the recursion to make the recursion work. From the point of view of someone writing a recursive function to process a linked list, it can be conceptualized as the first node, which we have to deal with, and the rest of the list, which we don't and therefore aren't concerned about. This attitude is shown in Figure 6-6.

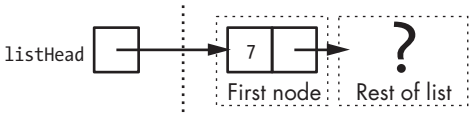


Figure 6-6: The list as a programmer using recursion should picture it: a first node and the rest of the list as a nebulous shape to be passed off to the recursive call

With the division of labor fixed, we can say that recursive processing of singly linked lists will proceed according to the following general plan. Given a linked list L and a question Q:

1. If L is minimal, we directly assign a default value. Otherwise . . .
2. Use a recursive call to produce an answer to Q for the "rest of" list L (the list starting with the second node of L).
3. Inspect the value in the first node of L.
4. Use the results of the previous two steps to answer Q for the whole of L.

As you can see, this is just a straightforward application of the BRI given the practical restrictions on breaking up a linked list. Now let's apply this blueprint to a specific problem.

PROBLEM: COUNTING NEGATIVE NUMBERS IN A SINGLY LINKED LIST

Write a recursive function that is given a singly linked list where the data type is integer. The function returns the count of negative numbers in the list.

The question, Q, we want to answer is, how many negative numbers are in the list? Therefore, our plan can be stated as:

1. If the list has no nodes, the count is 0 by default. Otherwise . . .
2. Use a recursive call to count how many negative numbers are in the "rest of" the list.
3. See whether the value in the first node of the list is negative.

4. Use the results of the previous two steps to determine how many negative numbers are in the whole list.

Here's a function implementation that follows directly from this plan:

```
int countNegative(listPtr head) {
    if (head == NULL) return 0;
    int listCount = countNegative(head->next);
    if (head->data < 0) listCount++;
    return listCount;
}
```

Note how this code follows the same principles as previous examples. It will count the negative numbers "backward," from the end of the list to the front. Also note that the code employs the head recursion technique; we process the "rest of" the list before we process the first node. As before, this allows us to avoid passing extra data in the recursive call or using global variables.

Also notice how linked-list rule 1, "if list L is minimal," is interpreted in the specific implementation of this problem as "if the list has no nodes." That's because it is meaningful to say that a list with no nodes has zero negative values. In some cases, though, there is no meaningful answer for our question Q for a list with no nodes, and the minimal case is a list with one node. Suppose our question was, what's the largest number in this list? That question cannot be answered for a list with no values. If you don't see why, pretend you are an elementary school teacher, and your class happens to be all girls. If your school's principal asked you how many boys in your classroom were members of the boy's choir, you could simply answer zero because you have no boys. If your principal asked you to name the tallest boy in your class, you could not give a meaningful answer to that question—you would have to have at least one boy to have a tallest boy. In the same way, if the question about a data set requires at least one value to be meaningfully answered, the minimal data set is one item. You may still want to return *something* for the "size zero" case, however, if only for flexibility in the use of the function and to guard against a crash.

Recursion and Binary Trees

All of the examples we have explored so far make no more than one recursive call. More complicated structures, however, may require multiple recursive calls. For a taste of how that works, let's consider the structure known as a *binary tree*, in which each node contains "left" and "right" links to other nodes. Here are the types we'll use:

```
struct treeNode {
    int data;
    treeNode * left;
    treeNode * right;
};
typedef treeNode * treePtr;
```

Because each node in the tree points to two other nodes, recursive tree-processing functions require two recursive calls. We conceptualized linked lists as having two parts: a first node and the rest of the list. For applying recursion, we will conceptualize trees as having three parts: the node at the top, known as the *root node*; all of the nodes reached from the left link of the root, known as the *left subtree*; and all of the nodes reached from the right link of the root, known as the *right subtree*. This conceptualization is shown in Figure 6-7. As with the linked list and as developers of a recursive solution, we just focus on the existence of the left and right subtrees, without considering their contents. This is shown in Figure 6-8.

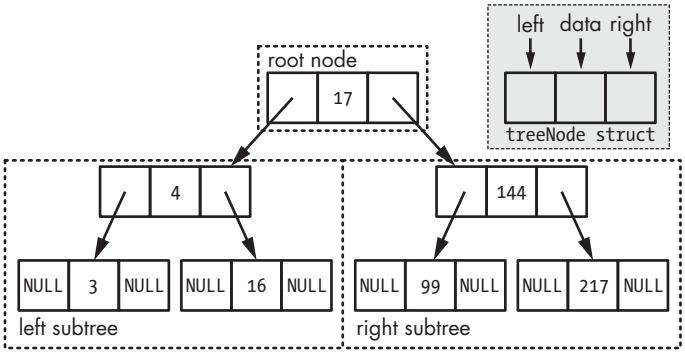


Figure 6-7: A binary tree divided into a root node and left and right subtree

As always, when recursively solving problems involving binary trees, we want to employ the BRI. We will make recursive function calls and assume they return correct results without worrying about how the recursive process solves the overall problem. As with linked lists, we will work with the natural divisions of a binary tree. This produces the following general plan. To answer a question Q for tree T:

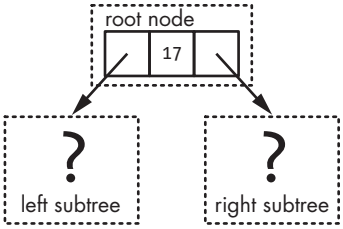


Figure 6-8: A binary tree as a programmer using recursion should picture it: a root node with left and right subtrees of unknown and unconsidered structure

1. If tree T is of minimal size, directly assign a default value. Otherwise . . .
2. Make a recursive call to answer Q for the left subtree of T.
3. Make a recursive call to answer Q for the right subtree of T.
4. Inspect the value in the root node of T.
5. Use the results of the previous three steps to answer Q for all of T.

Now let's apply the general plan to a specific problem.

PROBLEM: FIND THE LARGEST VALUE IN A BINARY TREE

Write a function that, when given a binary tree where each node holds an integer, returns the largest integer in the tree.

Applying the general plan to this specific problem results in the following steps:

1. If the root of the tree has no children, return the value in the root. Otherwise . . .
2. Make a recursive call to find the largest value in the left subtree.
3. Make a recursive call to find the largest value in the right subtree.
4. Inspect the value in the root node.
5. Return the largest of the values in the previous three steps.

With those steps in mind, we can directly write the code for the solution:

```
int maxValue(treePtr root) {
  ❶if (root == NULL) return 0;
  ❷if (root->right == NULL && root->left == NULL)
        return root->data;
  ❸int leftMax = maxValue(root->left);
  ❹int rightMax = maxValue(root->right);
  ❺int maxNum = root->data;
    if (leftMax > maxNum) maxNum = leftMax;
    if (rightMax > maxNum) maxNum = rightMax;
    return maxNum;
}
```

Notice how the minimal tree for this problem is a single node ❷ (although the empty-tree case is covered for safety ❶). This is because the question we are asking can only be meaningfully answered with at least one data value. Consider the practical problem if we tried to make the empty tree the base case. What value could we return? If we return zero, we implicitly require some positive values in the tree; if all of the values in the tree are negative, zero will be erroneously returned as the largest value in the tree. We might solve this problem by returning the lowest (most negative) possible integer, but then we would have to be careful adapting the code for other numeric types. By making a single node the base case, we avoid this decision altogether.

The rest of the code is straightforward. We use recursion to find the maximum values in the left ❸ and right subtrees ❹. Then we find the largest of the three values (value at root, largest in left subtree, largest in right subtree) using a variant of the "King of the Hill" algorithm we've been using throughout this book ❺.

Wrapper Functions

In the previous examples in this chapter, we have discussed only the recursive function itself. In some cases, however, the recursive function needs to be "set up" by a second function. Most commonly, this occurs when we write recursive functions inside of class structures. This can cause a mismatch between the parameters required for the recursive function and the parameters needed for a public method of the class. Because classes typically enforce information hiding, the class client code may not have access to the data or types the recursive function requires. This problem and its solution are shown in the next example.

PROBLEM: FIND THE NUMBER OF LEAVES IN A BINARY TREE

For a class that implements a binary tree, add a publicly accessible method that returns the number of leaves (nodes without children) in the tree. The counting of leaves should be performed using recursion.

Let's sketch the outline of what this class might look like before we try to implement a solution to this problem. For simplicity, we will include only the relevant parts of the class, ignoring the constructors, the destructor, and even the methods that would allow us to build the tree in order to focus on our recursive method.

```
class binaryTree {
    public:
      ❶int countLeaves();
    private:
        struct binaryTreeNode {
            int data;
            binaryTreeNode * left;
            binaryTreeNode * right;
        };
        typedef treeNode * treePtr;
        treePtr _root;
};
```

Note that our leaf-counting function takes no parameters ❶. From an interface point of view, this is exactly correct. Consider a sample call for a previously constructed `binaryTree` object `bt`:

```
int numLeaves = bt.countLeaves();
```

After all, if we are asking the tree how many leaves it has, what information could we possibly provide to the object that it would not already know about itself? As correct as this is for the interface, it's all wrong for the recursive implementation. If there is no parameter, what changes from one recursive

call to the next? Nothing can change in that case, except through global variables, which, as stated earlier, are to be avoided. If nothing changes, there's no way for the recursion to progress or terminate.

The way around this problem is to write the recursive function first, conceptualizing it as a function outside of a class. In other words, we'll write this function to count the leaves in a binary tree in the same style we wrote the function to find the largest value in a binary tree. The one parameter we need to pass is a pointer to our node structure.

This gives us another opportunity to employ the BRI. What is the question Q in this case? It is, how many leaves are in the tree? Applying the general plan for recursively processing binary trees to this specific problem results in the following:

1. If the root of the tree has no children, then the tree has one node total. That node is a leaf by definition, so return 1. Otherwise . . .
2. Make a recursive call to count the leaves in the left subtree.
3. Make a recursive call to count the leaves in the right subtree.
4. In this case, there is no need to inspect the root node because if we get to this step, there is no way the root is a leaf. So . . .
5. Return the sum of steps 2 and 3.

Translating this plan into code results in this:

```
struct binaryTreeNode {
    int data;
    treeNode * left;
    treeNode * right;
};
typedef binaryTreeNode * treePtr;
int countLeaves(treePtr rootPtr) {
    if (rootPtr == NULL) return 0;
    if (rootPtr->right == NULL && rootPtr->left == NULL)
        return 1;
    int leftCount = countLeaves(rootPtr->left);
    int rightCount = countLeaves(rootPtr->right);
    return leftCount + rightCount;
}
```

As you can see, the code is a direct translation of the plan. The question is, how do we get from this independent function to something we can use in the class? This is where the unwary programmer could easily get into trouble, thinking that we need to use a global variable or make the root pointer public. But we don't need to do that; we can keep everything inside the class. The trick is to use a *wrapper function*. First, we put the independent function, with the `treePtr` parameter, in the private section of our class. Then, we write a public function, the wrapper function, which will "wrap" the private function.

Because the public function has access to the private data member root, it can pass this along to the recursive function and then return the results back to the client like this:

```
class binaryTree {
    public:
        int publicCountLeaves();
    private:
        struct binaryTreeNode {
            int data;
            binaryTreeNode * left;
            binaryTreeNode * right;
        };
        typedef binaryTreeNode * treePtr;
        treePtr _root;
        int privateCountLeaves(treePtr rootPtr);
};
❶ int binaryTree::privateCountLeaves(treePtr rootPtr) {
    if (rootPtr == NULL) return 0;
    if (rootPtr->right == NULL && rootPtr->left == NULL)
        return 1;
    int leftCount = privateCountLeaves(rootPtr->left);
    int rightCount = privateCountLeaves(rootPtr->right);
    return leftCount + rightCount;
}
❷ int binaryTree::publicCountLeaves() {
   ❸return privateCountLeaves(_root);
}
```

Although C++ would allow both functions to have the same name, for clarity I've used different names to distinguish between the public and private "count leaves" functions. The code in `privateCountLeaves` ❶ is exactly the same as our previous, independent function `countLeaves`. The wrapper function `publicCountLeaves` ❷ is simple. It calls `privateCountLeaves`, passing the private data member root, and returns the result ❸. In essence, it "primes the pump" of the recursive process. Wrapper functions are very helpful when writing recursive functions inside classes, but they can be used anytime a mismatch exists between the parameter list required by a function and the desired parameter list of a caller.

When to Choose Recursion

New programmers often wonder why anyone has to deal with recursion. They may have already learned that any program can be constructed using basic control structures, such as selection (`if` statements) and iteration (`for` and `while` loops). If recursion is more difficult to employ than basic control structures and unnecessary, perhaps recursion should just be ignored.

There are several rebuttals to this. First, programming recursively helps programmers think recursively, and recursive thinking is employed throughout the world of computer science in such areas as compiler design. Second,

some languages simply require recursion because they lack some basic control structures. Pure versions of the Lisp language, for example, require recursion in almost every nontrivial function.

The question remains, though: If a programmer has studied recursion enough to "get it" and is using a full-featured language such as C++, Java, or Python, should recursion ever be employed? Does recursion have practical use in such languages, or is it just a mental exercise?

Arguments Against Recursion

To explore this question, let's enumerate the bad features of recursion.

Conceptual complexity
: For most problems, it's more difficult for the average programmer to solve a problem using recursion. Even once you understand the Big Recursive Idea, it's still going to be easier in most cases to write code using loops.

Performance
: Function calls incur significant overhead. Recursion involves lots of function calls and, therefore, can be slow.

Space requirements
: Recursion doesn't simply employ many function calls; it also nests them. That is, you can end up with a long chain of function calls waiting for other calls to complete. Each function call that has begun but has yet to end takes additional space on the system stack.

At a glance, this list of features constitutes a strong indictment against recursion as difficult, slow, and wasteful of space. However, these arguments do not hold universally. The most basic rule, then, for deciding between recursion and iteration is, *choose recursion when these arguments do not apply.*

Consider our function that counts the number of leaves in a binary tree. How would you solve this problem without recursion? It's possible, but you would need an explicit mechanism for maintaining the "breadcrumb trail" of nodes for which the left children had already been visited but not the right children. These nodes would need to be revisited at some point so we could travel down the right side. You might store these nodes in a dynamic structure, such as a stack. For comparison, here's an implementation of the function that uses the stack class from the C++ standard template library:

```
int binaryTree::stackBasedCountLeaves() {
    if (_root == NULL) return 0;
    int leafCount = 0;
  ❶stack<❷binaryTreeNode *> nodes;
  ❸nodes.push(_root);
    while (❹!nodes.empty()) {
        treePtr currentNode = ❺nodes.top();
      ❻nodes.pop();
        if (currentNode->left == NULL && currentNode->right == NULL)
            leafCount++;
```

```
        else {
            if (currentNode->right != NULL) ❸nodes.push(currentNode->right);
            if (currentNode->left != NULL) ❸nodes.push(currentNode->left);
        }
    }
    return leafCount;
}
```

This code follows the same pattern as the original, but if you've never used the stack class before, a few comments are in order. The stack class works like the system stack we discussed in Chapter 3; you can add and remove items only at the top. Note that we could perform our leaf count operation using any data structure that doesn't have a fixed size. We could have used a vector, for example, but the use of the stack most directly mirrors the original code. When we declare the stack ❶, we specify the type of items we will store there. In this case, we would store pointers to our `binaryTreeNode` structure ❷. We make use of four stack class methods in this code. The `push` method ❸ places an item (a node pointer, in this case) on the top of the stack. The `empty` method ❹ tells us whether there are any items left on the stack. The `top` method ❺ gives us a copy of the item on top of the stack, and the `pop` method ❻ removes the top item from the stack.

The code solves the problem by placing a pointer to the first node on the stack and then repeatedly removing a pointer to a node from the stack, checking whether it's a leaf, incrementing our counter if it is, and placing pointers to child nodes, if they exist, on the stack. So the stack keeps track of the nodes we have discovered, but have yet to process, in the same way that the chain of recursive calls in the recursive version keeps track of nodes we must revisit. In comparing this iterative version to the recursive version, we see that none of the standard objections to recursion applies with much vigor in this case. First, this code is longer and more complicated than the recursive version, so there is no argument against the recursive version on the basis of conceptual complexity. Second, look how many function calls `stackBasedCountLeaves` makes—for each visit to an interior node (i.e., not a leaf), this function makes four function calls: one each to `empty` and `top`, and two to `push`. The recursive version makes only the two recursive calls for each interior node. (Note that it is possible for us to avoid the function calls to the stack object by incorporating the logic of the stack within the function. This, however, would increase the complexity of the function even further.) Third, while this iterative version doesn't use additional system stack space, it makes explicit use of a private stack. In fairness, this is less space than the system stack overhead of the recursive calls, but it's still an expenditure of system memory in proportion to the maximum depth of the binary tree we are traversing.

Because the objections against recursion are mitigated or minimized in this case, recursion is a good choice for the problem. Put more generally, if a problem is simple to solve iteratively, then iteration should be your first choice. Recursion should be used when iteration would be complicated. Often this involves the necessity of the "breadcrumb trail" mechanism shown here.

Traversals of branching structures, such as trees and graphs, are inherently recursive. Processing linear structures, such as arrays and linked lists, usually does not require recursion, but there are exceptions. You will never go wrong making a first stab at a problem using iteration and seeing how far you get. As a last set of examples, consider the following linked-list problems.

PROBLEM: DISPLAY A LINKED LIST IN ORDER

Write a function that is passed the head pointer of a singly linked list where the data type of each node is an integer and that displays those integers, one per line, in the order they appear in the list.

PROBLEM: DISPLAY A LINKED LIST IN REVERSE ORDER

Write a function that is passed the head pointer of a singly linked list where the data type of each node is an integer and that displays those integers, one per line, in the reverse order they appear in the list.

Because these problems are mirror images of each other, it's natural to assume that their implementations would likewise be mirror images. That is indeed the case for recursive implementations. Using the `listNode` and `listPtr` type given previously, here are recursive functions to solve both of these problems:

```
void displayListForwardsRecursion(listPtr head) {
    if (head != NULL) {
        ❶cout << head->data << "\n";
        ❷displayListForwardsRecursion(head->next);
    }
}
void displayListBackwardsRecursion(listPtr head) {
    if (head != NULL) {
        ❸displayListBackwardsRecursion(head->next);
        ❹cout << head->data << "\n";
    }
}
```

As you can see, the code in these functions is identical except for the order of the two statements inside the `if` statement. That makes all the difference. In the first case, we display the value in the first node ❶ before making the recursive call to display the rest of the list ❷. In the second case, we make the call to display the rest of the list ❸ before we display the value in the first node ❹. This results in an overall backward display.

Because both of these functions are equally succinct, one might assume that recursion is properly used to solve both of these problems, but that's not the case. To see that, let's look at iterative implementations of both of these functions.

```
void displayListForwardsIterative(listPtr head) {
   ❶for (listPtr current = head; current != NULL; current = current->next)
        cout << current->data << "\n";
}
void displayListBackwardsIterative(listPtr head) {
   ❷stack<listPtr> nodes;
   ❸for (listPtr current = head; current != NULL; current = current->next)
        nodes.push(current);
   ❹while (!nodes.empty()) {
      ❺nodePtr current = nodes.top();
      ❻nodes.pop();
      ❼cout << current->data << "\n";
    }
}
```

The function to display the list in order is nothing more than a straightforward traversal loop ❶, such as those we saw back in Chapter 4. The function to display the list in reverse order, though, is more complicated. It suffers from the same requirement for a "breadcrumb trail" as our binary tree problems. Displaying the nodes in a linked list in reverse order requires returning to prior nodes by definition. In a singly linked list, there's no way to do that using the list itself, so a second structure is required. In this case, we need another stack. After declaring the stack ❷, we push all of the nodes in our linked list onto the stack using a `for` loop ❸. Because this is a stack, where each item is added on top of previous items, the first item in the linked list will be on the bottom of the stack, and the last item in the linked list will be on the top. We enter a while loop that continues until the stack is empty ❹, repeatedly grabbing a pointer to the top node on the stack ❺, removing that node pointer from the stack ❻, and then displaying the data in the referenced node ❼. Because the data on the top is the last data in the linked list, this has the effect of displaying the data in the linked list in reverse order.

As with the iterative binary tree function shown earlier, it would be possible to write this function without using a stack (by building a second list within the function that is a reverse of the original). There is no way, however, to make the second function as simple as the first or to avoid effectively traversing two structures instead of one. Comparing the recursive and iterative implementations, it's easy to see that the iterative "forward" function is so simple that there is no practical advantage in employing recursion, and there are several practical disadvantages. In contrast, the recursive "backward" function is simpler than the iterative version and should be expected to perform approximately as well as the iterative version. Therefore, the "backward" function is a reasonable use of recursion, while the "forward" function, though a good recursive programming exercise, is not a good practical use of recursion.

Exercises

As always, trying out the ideas presented in the chapter is imperative!

6-1. Write a function to compute the sum of just the positive numbers in an array of integers. First, solve the problem using iteration. Then, using the technique shown in this chapter, convert your iterative function to a recursive function.

6-2. Consider an array representing a binary string, where every element's data value is 0 or 1. Write a `bool` function to determine whether the binary string has odd parity (an odd number of 1 bits). Hint: Remember that the recursive function is going to return true (odd) or false (even), not the count of 1 bits. Solve the problem first using iteration, then recursion.

6-3. Write a function that is passed an array of integers and a "target" number and that returns the number of occurrences of the target in the array. Solve the problem first using iteration, then recursion.

6-4. Design your own: Find a problem processing a one-dimension array that you have already solved or that is trivial for you at your current skill level, and solve the problem (or solve it again) using recursion.

6-5. Solve exercise 6-1 again, using a linked list instead of an array.

6-6. Solve exercise 6-2 again, using a linked list instead of an array.

6-7. Solve exercise 6-3 again, using a linked list instead of an array.

6-8. Design your own: Try to discover a linked-list processing problem that is difficult to solve using iteration but can be solved directly using recursion.

6-9. Some words in programming have more than one common meaning. In Chapter 4, we learned about the heap, from which we get memory allocated with `new`. The term *heap* also describes a binary tree in which each node value is higher than any in the left or right subtree. Write a recursive function to determine whether a binary tree is a heap.

6-10. A *binary search tree* is a binary tree in which each node value is greater than any value in that node's left subtree but less than any value in the node's right subtree. Write a recursive function to determine whether a binary tree is a binary search tree.

6-11. Write a recursive function that is passed a binary search tree's root pointer and a new value to be inserted and that creates a new node with the new value, placing it in the correct location to maintain the binary search tree structure. Hint: Consider making the root pointer parameter a reference parameter.

6-12. Design your own: Consider basic statistical questions you can ask of a set of numerical values, such as average, median, mode, and so forth. Attempt to write recursive functions to compute those statistics for a binary tree of integers. Some are easier to write than others. Why?