is more efficient—iteration or recursion. The results are somewhat surprising, and lead to a discussion of the philosophy of *measurement and tuning* as a good software engineering practice.

Recursive algorithms are also used in Chapter 6, which explores the analysis of algorithms. There we investigate the use of recurrence relations to analyze algorithms. Chapter 7 illustrates how C can use run-time stacks to implement recursion. Chapter 13 uses recursive algorithms to solve sorting problems in a way that makes them easy to analyze.

Finally, Chapter 14 explores advanced topics in recursion, such as the use of recursion as a descriptive tool to define things and recognize things. Also explored is the relation between recursion and proofs of correctness by structural induction.

3.2 Thinking Recursively

Learning Objectives

1. To learn to think recursively.
2. To learn how strategies for recursion involve both base cases and recursion cases.
3. To learn how to search for different ways of decomposing a problem into subproblems.
4. To understand how to use call trees and traces to reason about how recursive programs work.

a gradual introduction

A good way to get a gradual introduction to the idea of recursion is to examine a sequence of solutions to simple problems. First, we study the simple problem of adding up the squares of some integers. Three different recursive solutions are presented to illustrate different ways of breaking problems into subproblems. We also discuss base cases and show how they are used to terminate the execution of recursive procedures. Then we introduce call trees and traces and show how they can reveal the way recursive programs work.

After showing how the decomposition techniques explored in summing squares of integers can be applied to multiplying integers as well, we study a nonstandard way of computing the factorial function recursively. We also briefly mention the use of factorials in computing permutations and combinations.

We then broaden the range of examples of recursion to show how we can treat nonnumeric data such as linked-lists and strings. We study some recursive solutions for reversing linked-lists and strings in order to illustrate some possibilities. Then it's time to generalize. We look back on our examples and try to extract the essence of what recursion involves.

How to Make Things Add Up Recursively

adding up squares

Our first example is a simple program to add up all the squares of the numbers from m to n. That is, given two positive integers, m and n, where $m \leq n$, we want to find

```
1  int SumSquares(int m, int n)
2  {
3      int i, sum;
4
5      sum = 0;                          /* Recall that the assignment */
6      for (i = m; i <= n; ++i) sum += i*i;   /* sum += i*i has the */
7      return sum;                       /* same effect in C as the */
8  }                                     /* assignment sum = sum + i*i */
```

Program 3.1 Iterative Sum of Squares

SumSquares$(m,n) = m^2 + (m+1)^2 + \ldots + n^2$. For example, SumSquares(5,10) $= 5^2 + 6^2 + 7^2 + 8^2 + 9^2 + 10^2 = 355$.[1]

the iterative way

An ordinary iterative program to compute SumSquares(m,n) is shown in Program 3.1. The strategy for this program is familiar to beginners. The variable sum holds partial sums during the iteration, and, initially, sum is set to 0. On line 6, a for-statement lets its controlled variable, i, range over the successive values in the range m:n (where the range m:n consists of the integers i, such that $m \le i \le n$).

iteration builds up solutions stepwise

For each such integer i, the partial sum in the variable, sum, is increased by the square of i (by adding i*i to it). After the iteration is finished, sum holds the total of all the contributions of the squares i*i, for each i in the range m:n. The value of sum is finally returned as the value of the function on line 7.

In an iterative solution such as this, a typical pattern is to build up the final solution in stages, using a repetitive process that enumerates contributions step-by-step and combines the contributions with a partial solution that, stepwise, gets closer and closer to the overall solution.

recursion combines subproblem solutions

To compute SumSquares(m,n) recursively, a new way of thinking needs to be used. The idea is to find a way of solving the overall problem by breaking it into smaller subproblems, such that some of the smaller subproblems can be solved using the *same* method as that used to solve the overall problem. The solutions to the subproblems are then *combined* to get the solution to the overall problem.

refining the strategy

Program Strategy 3.2 illustrates one such way of thinking about the solution. To refine this program strategy into an actual recursive solution, all we have to do is to replace the comments with appropriate implementations in C, as illustrated in Program 3.3.

recursive calls

Line 4 of Program 3.3 contains the *recursive call*, SumSquares(m+1,n). A recursive call is one in which a function calls itself *inside* itself. In effect, what line 4 says

[1] When implementing practical applications, professional programmers would be unlikely to write iterative or recursive programs to compute SumSquares(m,n), since this sum can be computed more directly and efficiently by evaluating a simple algebraic formula involving m and n, such as SumSquares$(m,n) = (n^3 - m^3)/3 + (n^2 + m^2)/2 + (n - m)/6$. But for the purpose of this chapter, the value of writing iterative and recursive programs to compute SumSquares(m,n) lies not in their efficiency but rather in their value as a device for illustrating the principles of recursion.

```
    int SumSquares(int m, int n)
    {
        (to compute the sum of the squares in the range m:n, where m ≤ n)

5       if (there is more than one number in the range m:n) {
            (the solution is gotten by adding the square of m to)
            (the sum of the squares in the range m+1:n)
        } else {
            (there is only one number in the range m:n, so m == n, and)
10          (the solution is therefore just the square of m)
        }
    }
```

Program Strategy 3.2 Recursive Sum of Squares

is that the solution to the overall problem can be gotten by adding (a) the solution to the smaller subproblem of summing the squares in the range m+1:n, and (b) the solution to the subproblem of finding the square of m. Moreover, the smaller subproblem (a) can be solved by the same method as the overall problem by making the recursive call, SumSquares(m+1,n), in which the function SumSquares calls itself *within* itself.

base cases

It is important to be aware that there is always a potential danger that a recursive function could attempt to go on endlessly splitting problems into subproblems and calling itself recursively to solve these subproblems, without ever stopping. Consequently, each properly designed recursive function should have a *base case* (or several base cases).

Line 6 of Program 3.3 contains the base case that occurs when the range m:n contains just one number, in which case m == n. The solution can then be computed directly by returning the square of m. This stops the recursion, since SumSquares is *not* called recursively on line 6.

the general pattern

Now let's generalize. The overall pattern of a recursive function is that it breaks the overall problem into smaller and smaller subproblems, which are solved by calling the function recursively, until the subproblems get small enough that they become

```
    int SumSquares(int m, int n)                    /* assume m ≤ n */
    {
        if (m < n) {
            return  m*m + SumSquares(m+1,n);        /* the recursion */
5       } else {
            return m*m;                             /* the base case */
        }
    }
```

Program 3.3 Recursive Sum of Squares

base cases and can be solved by giving their solutions directly (without using any more recursive calls). At each stage, solutions to the subproblems are combined to yield a solution to the overall problem. Let's look at two more recursive solutions to SumSquares(m,n) that fit this overall general pattern.

reading what a program does

Program 3.4 gives a new solution that is only slightly different than the first solution given in Program 3.3. The process of reading what a program such as this does involves inferring the goals that the individual parts of the program achieve and describing how the program achieves these goals. Therefore one way of summarizing the results of reading what a program does is to replace the program by the program strategy that describes its goals and methods. (This can be thought of as the reverse of the process of refining a program strategy into a specific realization of the strategy—a process of *antirefinement*, so-to-speak, in which we infer the program strategy from the specific program text.) The result of this process is shown in Program Strategy 3.5. Comparing Program 3.4 with Program 3.3 reveals that the decomposition of the overall problem into subproblems is slightly different. Program 3.3 specifies that to get the sum of the squares in the range m:n, we add m^2 to the sum of the squares in the range m+1:n, whereas, Program 3.4 gets the same sum by adding n^2 to the sum of the squares in the range m:n–1. The former could be called a *going-up recursion* since the successive subproblems called in the recursive calls "go upward," starting with the full range m:n, progressing next to subranges (m+1:n), (m+2:n), ..., and finally stopping at the uppermost subrange containing the base case (n:n). Program 3.4 could be called a *going-down recursion* since the successive subproblems called in the recursive calls "go downward," starting with the full range m:n, progressing next to subranges (m:n–1), (m:n–2), ..., and finally stopping at the bottommost subrange containing the base case (m:m).

going-up recursions

going-down recursions

The final example for computing SumSquares(m,n) uses yet another decomposition principle to break the overall problem into subproblems—namely, splitting the overall problem into two *halves*.

splitting a range in halves

In Program 3.6, if a range of numbers, m:n, contains just one number (i.e., if m == n), then the base case solution m^2 is given on line 6. Otherwise, the range m:n

```
    int SumSquares(int m, int n)                          /* assume m ≤ n */
    {

        if (m < n) {
5           return SumSquares(m, n – 1) + n*n;             /* the recursion */
        } else {
            return n*n;                                    /* the base case */
        }

    }
```

Program 3.4 Going-Down Recursion

```
    int SumSquares(int m, int n)
    {
        (to compute the sum of the squares in the range m:n, where m ≤ n)

5       if (there is more than one number in the range m:n) {
            (the solution is gotten by adding the square of n to)
            (the sum of the squares in the range m:n–1)
        } else {
            (there is only one number in the range m:n, so m == n, and)
10          (the solution is therefore just the square of n)
        }

    }
```

Program Strategy 3.5 Strategy for Going-Down Recursion

```
    int SumSquares(int m, int n)                          /* assume m ≤ n */
    {
        int middle;

5       if (m == n) {
            return m*m;                                   /* the base case */
        } else {
            middle = (m+n) / 2;
            return SumSquares(m, middle) + SumSquares(middle +1, n) ;
10      }
    }
```

Program 3.6 Recursion Combining Two Half-Solutions

contains more than one number and is split into two half-ranges. Assuming that middle is a midpoint for the range m:n, then the left half-range, m:middle, goes from m up to and including the middle. The right half-range, middle+1:n, goes from the number just past the middle up to and including n.

The recursion case on line 9 simply says that the sum of the squares of the entire range of integers m:n can be obtained by adding the sum of the squares of the left half-range, m:middle, to the sum of the squares of the right half-range, middle +1:n.

Let's take a closer look at the method used on line 8 to compute the middle. An example, shown in Fig. 3.7, is splitting the range 5:10 into two half-ranges.

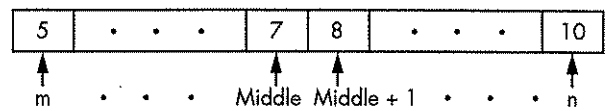

Figure 3.7 Splitting a range m:n into two halves

computing the middle

In this example, the range 5:10 is split into a left half-range, 5:7, and a right half-range, 8:10, using a middle == 7. To compute the middle of the range m:n, we divide m + n by 2, using *integer division* by 2, which keeps the quotient and throws away the remainder. The / operator in C performs integer division on two integer operands.

In the case of the range 5:10, we set middle = (5 + 10) / 2, on line 8 of Program 3.6, which gives middle the value 7, since (5 + 10) = 15, and 15 divided by 2 gives a quotient of 7 and a remainder of 1/2. This remainder is discarded in the integer division, 15 / 2. (If the range m:n contains an odd number of integers, as in the example 10:20, which contains 11 integers, then the middle value 15 divides the range into two half-ranges of unequal size. In this case, the left half-range, 10:15, contains six integers, and the right half-range, 16:20, contains only five integers.)

In summary, Program 3.6 presents a way of decomposing the overall problem into two subproblems different from that in Programs 3.3 and 3.4, because it uses two subproblems whose size is (roughly) half the size of the overall problem, and, whenever the range includes more than one number, it uses two recursive calls to compute the sum of the squares of the numbers in the two half-ranges.

Call Trees and Traces

It is informative to look at the *call tree* of Program 3.6, when the function call SumSquares(5,10) is evaluated. This is illustrated in Fig. 3.8. The evaluation of the calling expression, SumSquares(5,10), causes a function call on Program 3.6 generating two more recursive calls, SumSquares(5,7) and SumSquares(8,10). These latter two calls are shown as the descendants of the topmost call, SumSquares(5,10). In Fig. 3.8, we see that the calls of the form SumSquares(m,m) have no descendants beneath them in the call tree since they do not generate any further recursive calls. This is because calls of the form SumSquares(m,m) result in base cases in Program 3.6, in which m^2 is returned as the direct result.

annotating call trees

Suppose we annotate each calling expression in the call tree of Fig. 3.8 with the results returned by each recursive call. To do this, assume each call resulting in a base

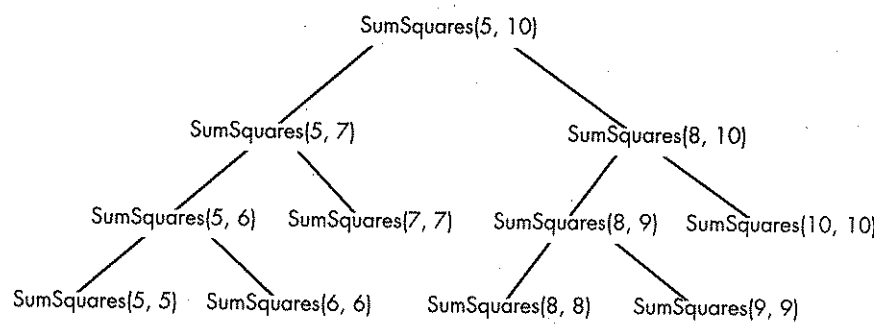

Figure 3.8 Call Tree for SumSquares(5,10) of Program 3.6