Summary

1. Recursion is a technique that solves a problem by solving a smaller problem of the same type.
2. When constructing a recursive solution, keep the following four questions in mind:
   a. How can you define the problem in terms of a smaller problem of the same type?
   b. How does each recursive call diminish the size of the problem?
   c. What instance of the problem can serve as the base case?
   d. As the problem size diminishes, will you reach this base case?
3. When constructing a recursive solution, you should assume that a recursive call's result is correct if its precondition has been met.
4. You can use the box trace to trace the actions of a recursive function. These boxes resemble activation records, which many compilers use to implement recursion. Although the box trace is useful, it cannot replace an intuitive understanding of recursion.
5. Recursion allows you to solve problems—such as the Towers of Hanoi—whose iterative solutions are difficult to conceptualize. Even the most complex problems often have straightforward recursive solutions. Such solutions can be easier to understand, describe, and implement than iterative solutions.
6. Some recursive solutions are much less efficient than a corresponding iterative solution due to their inherently inefficient algorithms and the overhead of function calls. In such cases, the iterative solution can be preferable. You can use the recursive solution, however, to derive the iterative solution.
7. If you can easily, clearly, and efficiently solve a problem by using iteration, you should do so.

Exercises

1. The following recursive function `getNumberEqual` searches the array `x` of `n` integers for occurrences of the integer `desiredValue`. It returns the number of integers in `x` that are equal to `desiredValue`. For example, if `x` contains the ten integers 1, 2, 4, 4, 5, 6, 7, 8, 9, and 12, then `getNumberEqual(x, 10, 4)` returns the value 2, because 4 occurs twice in `x`.

```
int getNumberEqual(const int x[], int n, int desiredValue)
{
   int count = 0;

   if (n <= 0)
      return 0;
   else
   {
      if (x[n - 1] == desiredValue)
         count = 1;

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

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

2. Perform a box trace of the following calls to recursive functions that appear in this chapter. Clearly indicate each subsequent recursive call.
   a. `rabbit(5)`
   b. `countDown(5)` (You wrote `countDown` in Checkpoint Question 3.)

3. Write a recursive function that will compute the sum of the first n integers in an array of at least n integers. *Hint:* Begin with the nth integer.

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

5. **a.** Revise the function `writeBackward`, discussed in Section 2.3.1, so that its base case is a string of length 1.

   b. Write a C++ function that implements the pseudocode function `writeBackward2`, as given in Section 2.3.1.

6. Describe the problem with the following recursive function:

```
void printNum(int n)
{
   cout << n << endl;
   printNum(n - 1);
}  // end printNum
```

7. Given an integer $n > 0$, write a recursive C++ function that writes the integers $1, 2, \ldots, n$.

8. Given an integer $n > 0$, write a recursive C++ function that returns the sum of the squares of 1 through n.

9. Write a recursive C++ function that writes the digits of a positive decimal integer in reverse order.

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

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

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

11. What output does the following program produce?

```
int getValue(int a, int b, int n);

int main()
{
   cout << getValue(1, 7, 7) << endl;
   return 0;
}  // end main

int getValue(int a, int b, int n)
{
   int returnValue = 0;

   cout << "Enter: a = " << a << " b = " << b << endl;
   int c = (a + b)/2;
   if (c * c <= n)
      returnValue = c;
   else
      returnValue = getValue(a, c-1, n);

   cout << "Leave: a = " << a << " b = " << b << endl;
   return returnValue;
}  // end getValue
```

12. What output does the following program produce?

```
int search(int first, int last, int n);
int mystery(int n);

int main()
{
   cout << mystery(30) << endl;
   return 0;
}  // end main

int search(int first, int last, int n)
{
   int returnValue = 0;
   cout << "Enter: first = " << first << " last = "
        << last << endl;

   int mid = (first + last)/2;
   if ( (mid * mid <= n) && (n < (mid+1) * (mid+1)) )
      returnValue = mid;
   else if (mid * mid > n)
      returnValue = search(first, mid-1, n);
   else
      returnValue = search(mid+1, last, n);

   cout << "Leave: first = " << first << " last = "
        << last << endl;
   return returnValue;
}  // end search

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

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

```
void displayOctal(int n)
{
   if (n > 0)
   {
      if (n / 8 > 0)
         displayOctal(n / 8);
      cout << n % 8;
   }  // end if
}  // end displayOctal
```

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

14. Consider the following program:

```
int f(int n);

int main()
{
   cout << "The value of f(8) is " << f(8) << endl;
   return 0;
}  // end main

/** @pre n >= 0. */
int f(int n)
{
   cout << "Function entered with n = " << n << endl;
   switch (n)
```

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

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

15. Consider the following function:

```
void recurse(int x, int y)
{
   if (y > 0)
   {
      x++;
      y--;
      cout << x << " " << y << endl;
      recurse(x, y);
      cout << x << " " << y << endl;
   }  // end if
}  // end recurse
```

Execute the function with $x = 5$ and $y = 3$. How is the output affected if `x` is a reference argument instead of a value argument?

16. Perform a box trace of the recursive function `binarySearch`, which appears in Section 2.4.2, with the array 1, 5, 9, 12, 15, 21, 29, 31 for each of the following search values:

a. 5
b. 13
c. 16

17. Imagine that you have 101 Dalmatians; no two Dalmatians have the same number of spots. Suppose that you create an array of 101 integers: The first integer is the number of spots on the first Dalmatian, the second integer is the number of spots on the second Dalmatian, and so on. Your friend wants to know whether you have a Dalmatian with 99 spots. Thus, you need to determine whether the array contains the integer 99.

a. If you plan to use a binary search to look for the 99, what, if anything, would you do to the array before searching it?
b. What is the index of the integer in the array that a binary search would examine first?
c. If all of your Dalmatians have more than 99 spots, exactly how many comparisons will a binary search require to determine that 99 is not in the array?

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

a. Write an iterative function `power1` to compute x^n for $n \geq 0$.
b. Write a recursive function `power2` to compute x^n by using the following recursive formulation:

$$x^0 = 1$$
$$x^n = x \times x^{n-1} \text{ if } n > 0$$

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

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

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

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

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

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

Note how this output corresponds to Figure 2-19.

20. Consider the following recurrence relation:

$$f(1) = 1; f(2) = 1; f(3) = 1; f(4) = 3; f(5) = 5;$$
$$f(n) = f(n-1) + 3 \times f(n-5) \quad \text{for all } n > 5.$$

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

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

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

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

22. Prove that the function `iterativeRabbit`, which appears in Section 2.7, is correct by using invariants. (See Appendix F for a discussion of invariants.)

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

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

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

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

This theorem leads to the following recursive definition:

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

The following function implements this recursive algorithm:

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

a. Prove the theorem.
b. What happens if $b > a$?
c. How is the problem getting smaller? (That is, do you always approach a base case?) Why is the base case appropriate?

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

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

25. Consider the following recursive definition:

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

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

Programming Problems

1. Implement a recursive function that computes a^n, where a is a real number and n is a nonnegative integer.

2. Implement the algorithm `maxArray`, discussed in Section 2.4.3, as a C++ function. What other recursive definitions of `maxArray` can you describe?

3. Implement the `binarySearch` algorithm presented in this chapter for an array of strings.

4. Implement the algorithm `kSmall`, discussed in Section 2.4.4, as a C++ function. Use the first value of the array as the pivot.