3. What is a call tree?
4. What is the trace of a recursive function call?
5. How can the factorial of n, denoted $n!$, be used to compute the number of possible combinations of n items taken k at a time?
6. What is a natural way to consider decomposing a linked list into substructures helpful for devising a recursive solution to a problem?

3.2 Exercises

1. Write a recursive function that computes x^n, called Power(x,n), where x is a floating point number and n is a nonnegative integer. [*Hint*: Power(x,n) can be defined by the following two equations: Power(x,0) = 1.0 and for $n \geq 1$, Power(x,n) = x * Power(x, n–1)].
2. Write an improved recursive version of Power(x,n) that works by breaking n down into halves (where half of n = n / 2), squaring Power(x,n / 2), and multiplying by x again if n was odd. For example, $x^{11} = (x^5)*(x^5)*x$, whereas $x^{10} = (x^5)*(x^5)$. Find a suitable base case to stop the recursion.
3. Write a recursive function, Mult(m,n), to multiply two positive integers, m and n, using only repeated addition.
4. According to Euclid's algorithm for finding the greatest common divisor, gcd(m,n), of two positive integers m and n, one can take successive pairs of remainders, and when one of the remainders is zero, the other number in the pair is the gcd. Letting (m,n) be the first remainder pair, we can write $m = q * n + r$, such that $0 \leq r < n$. Here, q is the quotient of m upon division by n ($q = m / n$), and r is the remainder of m after division by n ($r = m \% n$). Any divisor (including the gcd) that divides m and n must also divide r, since $r = m - q*n$. Consequently, the gcd of m and n must be the same as gcd(n,r). In Euclid's algorithm, one starts with the pair (m,n) and computes the pair (n,r). Then, if $r = 0$, the gcd(m,n) = gcd(n,r) = n. But if $r \neq 0$, the pair (n,r) is replaced by the next successive remainder pair, which is guaranteed to have the same gcd. Write a recursive version of Euclid's algorithm to compute gcd(m,n).
5. Prove that your solution to the previous exercise correctly computes the gcd. Divide your proof into two parts: (a) *termination*: a proof that your algorithm must terminate, and (b) *correctness*: a proof that after termination, the result is the gcd.
6. Write a "going-up" recursive version of Product(m,n), which gives the product of the integers in the range m:n.
7. Write a recursive string reversal program by refining Program Strategy 3.17, in which the word List is replaced by the word String. In order to do this, you will need to develop auxiliary functions to: (a) get the Head of a string, (b) get the Tail of a string, (c) concatenate two strings, and (d) determine whether a string is empty. How does the efficiency of this method compare with the efficiency of the recursive string reversal Program 3.20?
8. Write another recursive string reversal program, ReverseString(S,R), which uses two string parameters. It repeatedly removes the first character from S and puts it on the front of R until S is empty. It is originally called with the string, S, to be

reversed and an empty string, R. The result is given as the modified value of R after returning from the call.

9. Write a recursive function to find the length of a linked list, where the length of a linked-list, L, is defined to be the number of nodes in list L.
10. Write a recursive function Min(A) to find the smallest integer in an integer array A[n]. [*Hint:* Define an auxiliary function Min2(A,k,j) that finds the smallest integer in A[k:j], and let Min(A) = Min2(A, 0, n – 1).]
11. Ackermann's function, $A(m,n)$, is a two argument function defined as follows:

$$\begin{aligned} A(0,n) &= n+1 && \text{for } n \geq 0, \\ A(m,0) &= A(m-1,1) && \text{for } m > 0, \\ A(m,n) &= A(m-1, A(m,n-1)) && \text{for } m,n > 0. \end{aligned}$$

Write a recursive function that gives the value of Ackermann's function.

12. For what range of integer parameters, (m,n), does the output of your implementation of Ackermann's function, $A(m,n)$, *not* exceed the value of the maximum integer in your C system?
13. Describe in words what the following function P(int n) does:

```
    void PDigit(int d)      /* auxiliary procedure used below in P to write the */
    {                                  /* character corresponding to the digit d */
        printf("%c", (char) ( (int)'0' + d ) );
    }
5
    void P(int n)                      /* assume n is a non-negative integer */
    {
        if (n < 10) {
            PDigit(n);
10      } else {
            P(n / 10);
            PDigit(n % 10);
        }
    }
```

14. What does the following function do?

```
    void R(int n)                      /* where n is a non-negative integer */
    {
        /* Output the rightmost digit of n in a one character field */
            printf("%1d", (n%10) );
5
        if ((n / 10) != 0)  R(n / 10);
    }
```

15. Let x be a positive real. To calculate the square root of x by Newton's Method, so that the square of the solution differs from x to within an accuracy of *epsilon*, we start with an initial approximation $a = x/2$. If $|a*a - x| \leq epsilon$,

we stop with the result a. Otherwise we replace a with the next approximation, defined by $(a + x/a)/2$. Then, we test this next approximation to see if it is close enough and we stop if it is. In general, we keep on computing and testing successive approximations until we find one close enough to stop. Write a recursive function, Sqrt(x), that computes the square root of x by Newton's Method.

16. The binomial coefficients,

$$\binom{n}{k}$$

can be specified recursively. Letting

$$C(n,k) = \binom{n}{k},$$

we can use the following relationships:

$$C(n,0) = 1 \text{ and } C(n,n) = 1 \qquad \text{for } n \geq 0.$$
$$C(n,k) = C(n-1, k) + C(n-1, k-1) \qquad \text{for } n > k > 0$$

Develop a recursive program to compute $C(n,k)$.

17. If your solution to the previous exercise does not already do so, modify it so that it avoids wasteful recomputation of the results previously computed by other recursive calls, so that it computes the result efficiently. Does your solution do the best job possible of ensuring that intermediate partial results do not violate the maximum limits of the numerical variables it uses? How can you compute $C(n,k)$ in a way that is efficient and that works for the largest range of integer inputs n and k?

3.3 Common Pitfall—Infinite Regresses

Learning Objectives

1. To learn about infinite regresses.
2. To understand the symptoms caused by the occurrence of infinite regresses.

We now pause briefly to explore a common pitfall with recursive programs—*infinite regresses*. An infinite regress in a recursive program is somewhat analogous to an endless loop in an iterative program. It occurs when a recursive program calls itself endlessly and never encounters a base case that forces it to stop its execution.

infinite regresses

There are two reasons that a recursive program can call itself endlessly: (1) there is no base case to stop the recursion, or (2) a base case never gets called. The second possibility tends to happen more frequently, and sometimes it happens because the set of values used in a recursive call is bigger than the set of values that the recursive program was designed to handle. It may therefore not be the fault of the program designer, or even the fault of the user who made the call with a value lying outside the set for which the design works. Rather, the fault could lie in faulty or incomplete documentation about the set of values for which the given recursive program is designed to work.

To take a specific example of this, let's reconsider the program for Factorial(n) given earlier as Program 3.12. Now, let's ask two questions: (1) What happens if we call this function with n == 0, by making the call Factorial(0)?, and (2) What happens if we make the function call Factorial(– 1)?

prepare to crash

Warning: Don't fire up your computer, type in Program 3.12, and try out the calls Factorial(0) and Factorial(– 1) without first knowing how to recover from program crashes. Do you know how to perform the crash recovery procedure?

Why this warning? Let's try our method for writing out a call trace, starting with the call, Factorial(0).

```
Factorial(0) = 0 * Factorial(– 1)
             = 0 * (– 1) * Factorial(– 2)
             = 0 * (– 1) * (– 2) * Factorial(– 3)
             = 0 * (– 1) * (– 2) * (– 3) * Factorial(– 4)
             = 0 * (– 1) * (– 2) * (– 3) * (– 4) * Factorial(– 5)
             = and so on, in an infinite regress.
```

a base case that never gets called

This happens because, when n == 0 or n is any negative integer, the condition, n == 1, in the if-condition on line 3 of Program 3.12 evaluates to *false*, after which the else-part is evaluated, causing an evaluation of the expression n*Factorial(n – 1). This results in a recursive call of Factorial(n – 1) with a new argument one less than the previous argument, for which the base case will not be encountered. So yet another recursive call on Factorial(n – 2) will occur—and so on, endlessly.

running out of resources

Well, to be truthful, the process can go on almost endlessly, except for the fact that the computer may run out of resources sooner or later. One thing that may happen is that the computer can run out of space in the region of memory where it stores separate structures containing information about each function call that has been initiated but which has not yet been completed. Each time a function is called in C, some space for a *call frame* for the call is allocated in a call-frame region of memory where call frames are stored for all previous calls that have been made but have not yet been completed. Since each such frame uses up a finite amount of space, an unending sequence of recursive calls results in an unending sequence of frame allocations. This may cause the program to request additional space for call frames, without limit.

Eventually, this call-frame region may try to expand into memory devoted to some other purpose (the *wild case*, resulting in who knows what kind of mayhem), or a