recurse forever (or at least until the computer runs out of memory). The recursive solution also has one or more general cases that include recursive calls to the function. The recursive calls must involve a "smaller caller." One (or more) of the actual parameter values must change in each recursive call to redefine the problem to be smaller than it was on the previous call. Thus each recursive call leads the solution of the problem toward the base case(s).

A typical implementation of recursion involves the use of a stack. Each call to a function generates an activation record to contain its return address, parameters, and local variables. The activation records are accessed in a last in, first out manner. Thus a stack is the choice of data structure.

Recursion can be supported by systems and languages that use dynamic storage allocation. The function parameters and local variables are not bound to addresses until an activation record is created at run time. Thus multiple copies of the intermediate values of recursive calls to the function can be supported, as new activation records are created for them.

With static storage allocation, in contrast, a single location is reserved at compile time for each parameter and local variable of a function. No place is provided to store any intermediate values calculated by repeated nested calls to the same function. Therefore, systems and languages with only static storage allocation cannot support recursion.

When recursion is not possible or appropriate, a recursive algorithm can be implemented nonrecursively by using a looping structure and, in some cases, by pushing and popping relevant values onto a stack. This programmer-controlled stack explicitly replaces the system's run-time stack. While such nonrecursive solutions are often more efficient in terms of time and space, they usually involve a tradeoff in terms of the elegance of the solution.

Exercises

1. Explain what is meant by the following:
   a. base case
   b. general (or recursive) case
   c. run-time stack
   d. binding time
   e. tail recursion
2. True or false? If false, correct the statement. *Recursive functions:*
   a. often have fewer local variables than the equivalent nonrecursive routines.
   b. generally use *while* or *for* statements as their main control structure.
   c. are possible only in languages with static storage allocation.
   d. should be used whenever execution speed is critical.
   e. are always shorter and clearer than the equivalent nonrecursive routines.
   f. must always contain a path that does not contain a recursive call.

3. Use the Three-Question Method to verify the `ValueInList` function described in this chapter.
4. Describe the Three-Question Method of verifying recursive routines in relation to an inductive proof.
5. Which data structure would you most likely see in a nonrecursive implementation of a recursive algorithm?
6. Using the recursive function `RevPrint` as a model, write the recursive function `PrintList`, which traverses the elements in the list in forward order. Does one of these routines constitute a better use of recursion? If so, which one?

Use the following function in answering Exercises 7 and 8:

```
int Puzzle(int base, int limit)
{
  if (base > limit)
    return -1;
  else
    if (base == limit)
      return 1;
    else
      return base*Puzzle(base+1, limit);
}
```

7. Identify the following:
   a. the base case(s) of the function `Puzzle`
   b. the general case(s) of the function `Puzzle`
8. Show what would be written by the following calls to the recursive function `Puzzle`:
   a. `cout << Puzzle(14, 10);`
   b. `cout << Puzzle(4, 7);`
   c. `cout << Puzzle(0, 0);`
9. Given the following function:

```
int Func(int num)
{
  if (num == 0)
    return 0;
  else
    return num + Fun(num + 1);
}
```

a. Is there a constraint on the values that can be passed as a parameter for this function to pass the smaller-caller test?

b. Is `Func(7)` a good call? If so, what is returned from the function?

c. Is `Func(0)` a good call? If so, what is returned from the function?

d. Is `Func(-5)` a good call? If so, what is returned from the function?

10. Put comments on the following routines to identify the base and general cases and explain what each routine does.

a.
```
int Power(int base, int exponent)
{
  if (exponent == 0)
    return 1;
  else
    return base * Power(base, exponent-1);
}
```

b.
```
int Factorial(int number)
{
  if (num > 0)
    return num * Factorial(num - 1);
  else
    if (num == 0)
      return 1;
}
```

c.
```
void Sort(int values[], int fromIndex, int toIndex)
{
  int maxIndex;

  if (fromIndex != toIndex)
  {
    maxIndex = MaxPosition(values, fromIndex, toIndex);
    Swap(values[maxIndex], values[toIndex]);
    Sort(values, fromIndex, toIndex - 1);
  }
}
```

11. a. Fill in the blanks to complete the following recursive function:

```
int Sum(int info[], int fromIndex, int toIndex)
// Computes the sum of the items between fromIndex and toIndex.
{
  if (fromIndex __________ toIndex)
    return ________________________;
  else
    return ________________________;
}
```

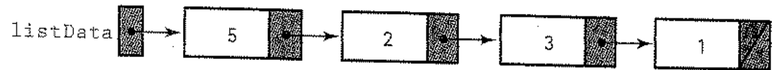
b. Which is the base case and which is the general case?

c. Show how you would call this function to sum all the elements in an array called `numbers`, which contains elements indexed from 0 to `MAX_ITEMS - 1`.

d. What run-time problem might you experience with this function as it is now coded?

A/W 12. You must assign the grades for a programming class. The class is studying recursion, and students have been given this simple assignment: Write a recursive function `SumSquares` that takes a pointer to a linked list of integer elements and returns the sum of the squares of the elements.

Example:



`SumSquares(listPtr)` yields (5 * 5) + (2 * 2) + (3 * 3) + (1 * 1) = 39

Assume that the list is not empty.

You have received quite a variety of solutions. Grade the functions that follow, marking errors where you see them.

a.
```
int SumSquares(NodeType* list)
{
  return 0;
  if (list != NULL)
    return (list->info*list->info) + SumSquares(list->next));
}
```

b.
```
int SumSquares(NodeType* list)
{
  int sum = 0;
  while (list != NULL)
  {
    sum = list->info + sum;
    list = list->next;
  }
  return sum;
}
```

c.
```
int SumSquares(NodeType* list)
{
  if (list == NULL)
    return 0;
  else
    return list->info*list->info + SumSquares(list->next);
}
```

d.
```
int SumSquares(NodeType* list)
{
  if (list->next == NULL)
    return list->info*list->info;
  else
    return list->info*list->info + SumSquares(list->next);
}
```

e.
```
int SumSquares(NodeType* list)
{
  if (list == NULL)
    return 0;
  else
    return (SumSquares(list->next) * SumSquares(list->next));
}
```

13. The Fibonacci sequence is the series of integers

```
0, 1, 1, 2, 3, 5, 8, 21, 34, 55, 89 ...
```

See the pattern? Each element in the series is the sum of the preceding two items. There is a recursive formula for calculating the nth number of the sequence (the 0th number if Fib(0) = 0):

$$\text{Fib}(N) = \begin{cases} N, & \text{if } N = 0 \text{ or } 1 \\ \text{Fib}(N-2) + \text{Fib}(N-1), & \text{if } N > 1 \end{cases}$$

a. Write a recursive version of the function `Fibonacci`.

b. Write a nonrecursive version of the function `Fibonacci`.

c. Write a driver to test the recursive and iterative versions of the function `Fibonacci`.

d. Compare the recursive and iterative versions for efficiency. (Use words, not Big-O notation.)

e. Can you think of a way to make the recursive version more efficient?

14. The following defines a function that calculates an approximation of the square root of a number, starting with an approximate answer (`approx`), within the specified tolerance (`tol`).

$$\text{SqrRoot(number, approx, tol)} = \begin{cases} \text{approx}, & \text{if } |\,\text{approx}^2 - \text{number}\,| <= \text{tol} \\ \text{SqrRoot(number, (approx}^2 + \text{number)/(2*approx), tol)}, & \text{if } |\,\text{approx}^2 - \text{number}\,| > \text{tol} \end{cases}$$

a. What limitations must be made on the values of the parameters if this method is to work correctly?

b. Write a recursive version of the function `SqrRoot`.

c. Write a nonrecursive version of the function `SqrRoot`.

d. Write a driver to test the recursive and iterative versions of the function `SqrRoot`.

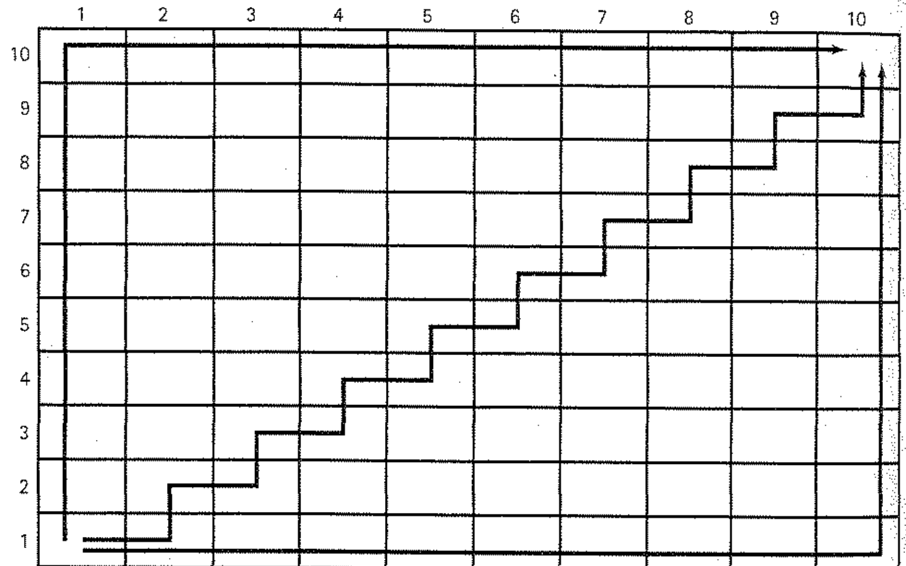
15. A sequential search member function of `SortedType` has the following prototype:

```
void SortedType::Search(int value, bool& found);
```

a. Write the function definition as a recursive search, assuming a linked list implementation.

b. Write the function definition as a recursive search, assuming an array-based implementation.

16. We want to count the number of possible paths to move from row 1, column 1 to row N, column N in a two-dimensional grid. Steps are restricted to going up or to the right, but not diagonally. The illustration that follows shows three of many paths, if $N = 10$:



a. The following function, `NumPaths`, is supposed to count the number of paths, but it has some problems. Debug the function.

```
int NumPaths(int row, int col, int n)
{
  if (row == n)
```

```
        return 1;
    else
        if (col == n)
            return NumPaths + 1;
        else
            return NumPaths(row + 1, col) * NumPaths(row, col + 1);
}
```

b. After you have corrected the function, trace the execution of `NumPaths` with `n = 4` by hand. Why is this algorithm inefficient?

c. You can improve the efficiency of this operation by keeping intermediate values of `NumPaths` in a two-dimensional array of integer values. This approach keeps the function from having to recalculate values that it has already figured out. Design and code a version of `NumPaths` that uses this approach.

d. Show an invocation of the version of `NumPaths` you developed in part (c), including any array initialization necessary.

e. How do the two versions of `NumPaths` compare in terms of time efficiency? Space efficiency?

17. Given the following function:[2]

```
int Ulam(int num)
{
  if (num < 2)
    return 1;
  else
    if (num % 2 == 0)
      return Ulam(num / 2);
    else
      return Ulam (3 * num + 1);
}
```

a. What problems come up in verifying this function?

b. How many recursive calls are made by the following initial calls:

```
cout   << Ulam(7)  << endl;
cout   << Ulam(8)  << endl;
cout   << Ulam(15)  << endl;
```

18. Explain the relationship between dynamic storage allocation and recursion.

[2]One of our reviewers pointed out that the proof of termination of this algorithm is a celebrated open question in mathematics. See *Programming Pearls* by Jon Bentley for a discussion and further references.

19. What do we mean by binding time, and what does it have to do with recursion?
20. Given the following values in `list`:

```
list
  .length  10
  .info    2   6   9   14  23  65  92  96  99  100
          [0] [1] [2] [3] [4] [5] [6] [7] [8] [9]
```

Show the contents of the run-time stack during the execution of this call to `BinarySearch`:

```
BinarySearch(info, 99, 0, 9);
```

21. The parameter to the following two recursive routines is a pointer to a singly linked list of numbers, whose elements are unique (no duplicates) and unsorted. Each node in the list contains two members, `info` (a number) and `next` (a pointer to the next node).
    a. Write a recursive value-returning function, `MinLoc`, that receives a pointer to a list of unsorted numbers and returns a pointer to the node that contains the minimum value in the list.
    b. Write a recursive void function, `Sort`, that receives a pointer to an unsorted list of numbers and reorders the values in the list from smallest to largest. This function may call the recursive `MinLoc` function that you wrote in part (a). (*Hint:* It is easier to swap the values in the `info` part of the nodes than to reorder the nodes in the list.)
22. True or false? If false, correct the statement. *A recursive solution should be used when:*
    a. computing time is critical.
    b. the nonrecursive solution would be longer and more difficult to write.
    c. computing space is critical.
    d. your instructor says to use recursion.
23. Design a maze in which there are starting positions that return "Trapped" when the starting position is Open.