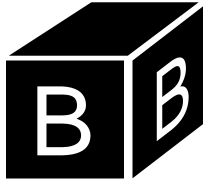


Chapter 19

Recursion

Did you ever look up the definition of some unknown word in the dictionary only to discover that the dictionary defined it in terms of another unknown word? Then, when you looked up the second word, you discovered that it was defined in terms of the first word! The problem with the dictionary is that you did not know the meaning of the first word to begin with. Had the second word been defined in terms of a third word that you knew, you would have been satisfied.

Definition of recursion

A recursive definition of an item is a definition in terms of that same item. In the dictionary example, the recursion is circular or mutual. In mathematics, a recursive definition of a function is a definition that uses the function itself. For example, suppose a function, $f(n)$, is defined as follows:

$$f(n) = nf(n-1)$$

You want to use this definition to determine $f(4)$, so you substitute 4 for n in the definition.

$$f(4) = 4f(3)$$

But now you do not know what $f(3)$ is. So you substitute 3 for n in the definition and get

$$f(3) = 3f(2)$$

Substituting this into the formula for $f(4)$ gives

$$f(4) = 4(3)f(2)$$

But now you do not know what $f(2)$ is. The definition tells you it is 2 times $f(1)$. So the formula for $f(4)$ becomes

$$f(4) = 4(3)(2)f(1)$$

You can see the problem with this definition. With nothing to stop the process, you will continue to compute $f(4)$ endlessly.

$$f(4) = 4(3)(2)(1)(0)(-1)(-2)\ldots$$

It is as if the dictionary gave you an endless string of definitions, each based on another unknown word.

To be complete, the definition must specify the value of $f(n)$ for a specific value of n. Then the preceding process will terminate, and you can compute $f(n)$ for any n. Here is a complete recursive definition of $f(n)$:

$$\begin{cases} f(0) = 1 \\ f(n) = n f(n-1) & \text{for } n > 0 \end{cases}$$

A recursive definition of factorial

This definition says you can stop the previous process at $f(0)$. So $f(4)$ is

$$\begin{aligned} f(4) &= 4f(3) \\ &= 4(3)f(2) \\ &= 4(3)(2)f(1) \\ &= 4(3)(2)(1)f(0) \\ &= 4(3)(2)(1)(1) \\ &= 24 \end{aligned}$$

You should recognize this definition as the factorial function.

A recursive factorial function

A recursive function in Component Pascal is a function that calls itself. There is no special recursion statement with a new recursion syntax to learn. The method of storage allocation on the run-time stack is the same as with nonrecursive functions. The only difference is that a recursive function contains a statement that calls itself.

Figure 19.1 shows a dialog box for the factorial function. It is identical to the dialog box of Figure 14.3, which is linked to a procedure that also calculates the factorial function but using iteration instead of recursion. As far as the user is concerned, there is no difference between the results of the two programs.

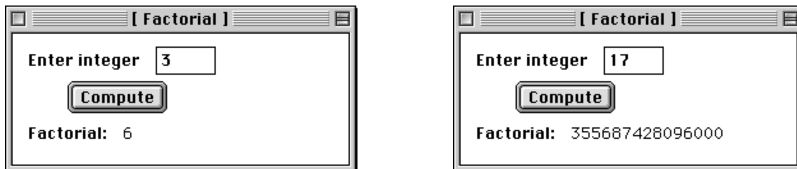


Figure 19.1
The dialog box for the factorial function of Figure 19.2.

The function in Figure 19.2 computes the factorial of a number recursively. It is a direct application of the recursive definition of $f(n)$, which is shown above.

```
MODULE Pbox19A;
    IMPORT Dialog;
    VAR
        d*: RECORD
            num*: INTEGER;
            factorial-: LONGINT
        END;

    PROCEDURE Factorial (n: INTEGER): LONGINT;
    BEGIN
        ASSERT((0 <= n) & (n <= 20), 20);
        IF n = 0 THEN
            RETURN 1
        ELSE
            RETURN n * Factorial(n - 1)  (* ra2 *)
        END
    END Factorial;

    PROCEDURE ComputeFactorial*;
    BEGIN
        IF d.num >= 0 THEN
            d.factorial := Factorial(d.num)  (* ra1 *)
        ELSE
            d.factorial := 0
        END;
        Dialog.Update(d)
    END ComputeFactorial;

BEGIN
    d.num := 0;
    d.factorial := 1
END Pbox19A.
```

Figure 19.2
A program to compute the factorial recursively.

When a function is called, memory allocation on the run-time stack takes place in the following order.

Allocation for function procedures

- Push storage for the returned value.
- Push the parameters.
- Push the return address.
- Push storage for the local variables.

Figure 19.3 is a trace that shows the run-time stack. Figure 19.3(a) shows the values for global variables d.num and d.factorial after the user has entered 3 for d.num in the dialog box. When she clicks the button in the dialog box, the framework calls procedure ComputeFactorial, pushing the return address onto the run-time stack, as shown in Figure 19.3(b). The return address is ra0, which represents the address of some statement in the framework. Figure 19.3(c) shows the stack frame after procedure ComputeFactorial calls Factorial. The return address on the run-time stack is ra1, the address of the statement

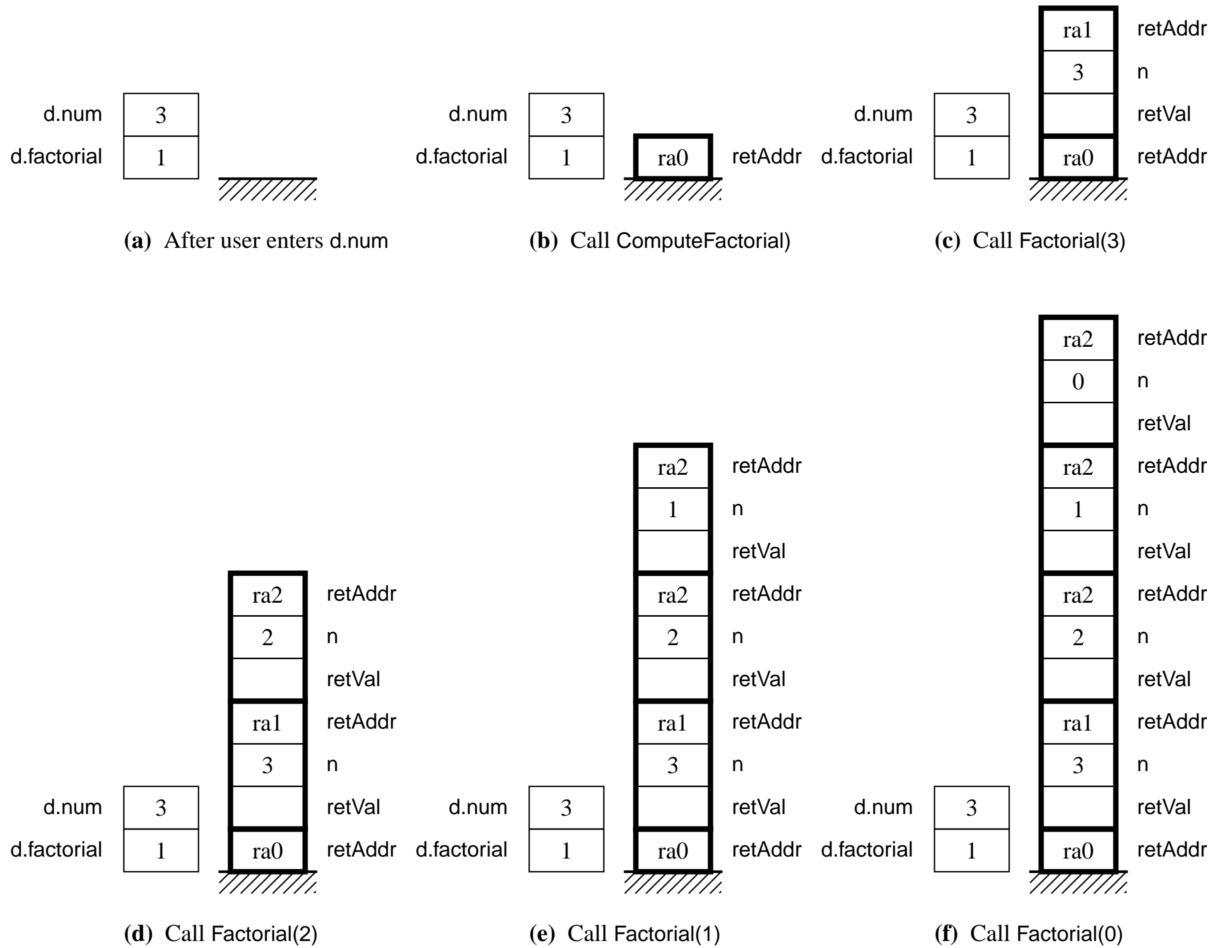


Figure 19.3
The run-time stack for Figure 19.2.

```
d.factorial := Factorial(d.num); (* ra1 *)
```

The first statement in the function tests if n equals 0. Because it does not, the ELSE part of the IF statement executes, which causes the first recursive call.

Figure 19.3(d) shows the stack after the first recursive call. Procedure Factorial calls itself. The statement that makes the call is

```
RETURN n * Factorial(n - 1) (* ra2 *)
```

so the return address is the address of this statement, ra2. The function is suspended and a new instance of the same function begins executing. The actual parameter is n - 1, and the current value of n is 3. Because the parameter is called by value, the formal parameter n for the new stack frame gets the value 2.

Figure 19.3(d) shows a curious situation that is typical of recursive calls. The program in Listing 19.2 shows only one declaration of n in the formal parameter list of Factorial. But Figure 19.3(d) shows two instances of n. The old instance of n has the

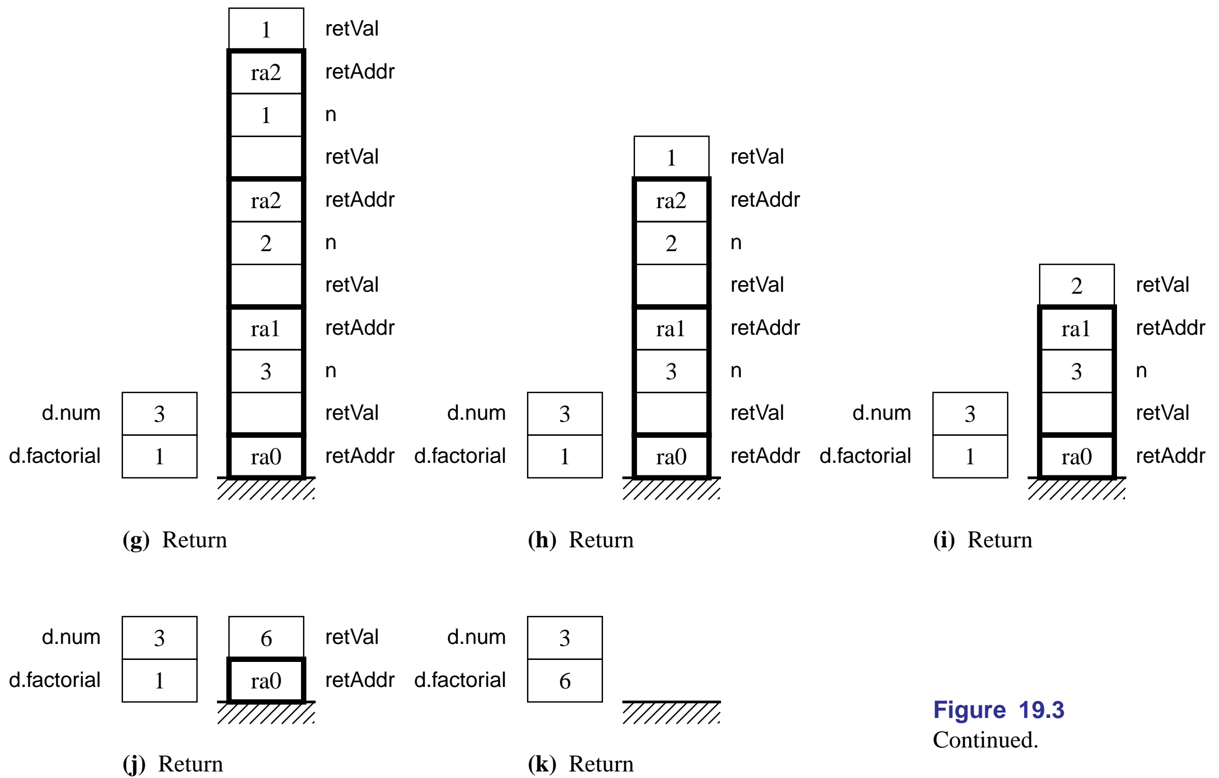


Figure 19.3
Continued.

value 3 from procedure ComputeFactorial. But the new instance of n has the value 2 from the recursive call.

The computer suspends the old execution of the function and begins a new execution of the same function from its beginning. The first statement in the function tests n for 0. But which n? Figure 19.3(d) shows two n's on the run-time stack. The rule is that any reference to a local variable or formal parameter is to the one on the top stack frame. Because the value of n is 2, the ELSE part executes.

But now the function makes another recursive call. It allocates another stack frame as Figure 19.3(e) shows, then another as Figure 19.3(f) shows. Each time, the newly allocated formal parameter gets a value one less than the old value of n because the function call is

```
Factorial(n - 1)
```

Finally, in Figure 19.3(f), n has the value 0. The statement

```
RETURN 1
```

gives 1 to the cell on the run-time stack allocated for the returned value. The RETURN statement also triggers a return to the calling statement.

The same events transpire with a recursive return as with a nonrecursive return.

The bottom cell of the stack frame gets the returned value, and the return address tells which statement to execute next. In Figure 19.3(f), the bottom cell of the top stack frame gets 1, and the return address is the calling statement in the function. The top frame is deallocated as shown in Figure 19.3(g). The calling statement

```
RETURN n * Factorial(n - 1)  (* ra2 *)
```

completes its execution. It multiplies its value of n, which is 1, by the value returned, which is 1, and assigns the result to the cell reserved for the returned value. So, the cell for the returned value gets 1, as Figure 19.3(h) shows. A similar sequence of events occurs on each return. Figure 19.3(i) and (j) show that the value returned from the second call is 2 and from the third call is 6.

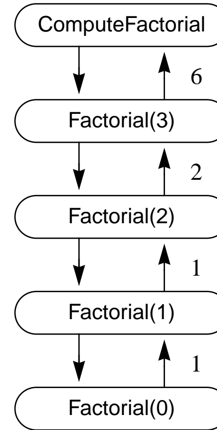


Figure 19.4
The calling sequence for Figure 19.2.

Figure 19.4 shows the calling sequence for Figure 19.2. The down arrows represent function calls, and the up arrows represent returns. The value returned is next to each up arrow. Procedure ComputeFactorial calls Factorial. Then Factorial calls itself three times. In this example, Factorial is called a total of four times.

You can see that the program computes the factorial of 3 the same way you would compute $f(3)$ from its recursive definition. You start by computing $f(3)$ as 3 times $f(2)$. Then you must suspend your computation of $f(3)$ to compute $f(2)$. After you get your result for $f(2)$, you can multiply it by 3 to get $f(3)$. Similarly, the program must suspend its execution of the function to call the same function again. The run-time stack keeps track of the current values of the variables so they can be used when that instance of the function resumes.

Thinking recursively

The microscopic viewpoint of recursion

You can take two different viewpoints when dealing with recursion—microscopic and macroscopic. Figure 19.3 illustrates the microscopic viewpoint and shows precisely what happens inside the computer during execution. It is the viewpoint that considers the details of the run-time stack during a trace of the program. The macroscopic viewpoint does not consider the individual trees. It considers the forest as a whole.

You need to know the microscopic viewpoint to understand how Component Pascal implements recursion. The details of the run-time stack are necessary when you study how recursion is implemented at the machine level. But to write a recursive function you should think macroscopically, not microscopically.

The most difficult aspect of writing a recursive function is the assumption that you can call the procedure that you are in the process of writing. To make that assumption, you must think macroscopically and forget about the run-time stack. The two key elements of designing a recursive function are

- Compute the function for the basis.
- Assuming the function for n – 1, write it for n.

Imagine you are writing function Factorial. You get to this point:

```
PROCEDURE Factorial (n: INTEGER): LONGINT;
BEGIN
   IF n = 0 THEN
      RETURN 1
   ELSE
```

and wonder how to continue. You have computed the function for the basis, n = 0. But now you must assume that you can call function Factorial, even though you have not finished writing Factorial. You must assume that Factorial(n - 1) will return the correct value for the factorial.

The macroscopic viewpoint of recursion

Here is where you must think macroscopically. If you start wondering how Factorial(n - 1) will return the correct value, and if visions of stack frames begin dancing in your head, you are not thinking correctly. In writing Factorial, you must assume you can call Factorial(n - 1), with no questions asked.

Recursive programs are based on a divide and conquer strategy. It is appropriate when you can solve a large problem in terms of a smaller one. Each recursive call makes the problem smaller and smaller until the program reaches the smallest problem of all, the basis, which is simple to solve.

Recursive addition

Here is another example of a recursive problem. Figure 19.5 shows the input/output for a program that scans a list of integers into an array. It computes their sum and displays it on the Log. Because this problem can be solved much more efficiently using the techniques of earlier chapters without an array, this example serves only to illustrate recursion.

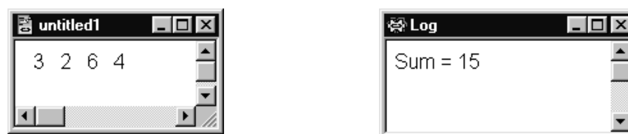


Figure 19.5
The input and output for the module in Figure 19.6.

Suppose v is an array of n integers. You want to find the sum of all n integers in the list recursively. The first step is to formulate the solution of the large problem in terms of a smaller problem. If you knew how to find the sum of the first n-1 integers, you could simply add it to the nth integer in v. You would then have the sum of all n integers.

The next step is to design a function with the appropriate parameters. The function will compute the sum of n integers by calling itself to compute the sum of n-1 integers. So the parameter list must have a parameter that tells how many integers in the array to add. These considerations should lead you to the following function head:

```
PROCEDURE Sum (IN v: ARRAY OF INTEGER; n: INTEGER): INTEGER;
   (* Returns the sum of the first n elements of v *)
```

How do you establish the basis? That is simple. If n is less than 1, there are no numbers left to add, and the function should return 0. Now you can write

```
BEGIN
   IF n < 1 THEN
      RETURN 0
   ELSE
```

Now think macroscopically. You can assume that Sum(v, n - 1) will return the sum of the first n-1 integers. Have faith. All you need to do is add that sum to v[n - 1]. Listing 19.6 shows the function in a finished program. v is called by constant reference because it is an array whose initial values are defined an not changed by Sum.

```
MODULE Pbox19B;
   IMPORT TextControllers, PboxMappers, StdLog;

   PROCEDURE Sum (IN v: ARRAY OF INTEGER; n: INTEGER): INTEGER;
      (* Returns the sum of the first n elements of v *)
   BEGIN
      ASSERT (n >= 0, 20);
      IF n = 0 THEN
         RETURN 0
      ELSE
         RETURN v[n - 1] + Sum(v, n - 1)  (* ra2 *)
      END
   END Sum;

   PROCEDURE ComputeSum*;
      VAR
         cn: TextControllers.Controller;
         sc: PboxMappers.Scanner;
         list: ARRAY 1024 OF INTEGER;
         numItems: INTEGER;
   BEGIN
      cn := TextControllers.Focus();
      IF cn # NIL THEN
         sc.ConnectTo(cn.text);
         sc.ScanIntVector(list, numItems);
         StdLog.String("Sum = ");
         StdLog.Int(Sum(list, numItems));  (* ra1 *)
         StdLog.Ln
      END
   END ComputeSum;

END Pbox19B.
```

Figure 19.6
A recursive function that returns the sum of the first n numbers in an array.

Even though you write the function without considering the microscopic view, you can still trace the run-time stack. Figure 19.7 shows the stack frames for the first two calls to Sum. The stack frame consists of the value returned, the parameters, v

and n, and the return address. Because there are no local variables, no storage for them is allocated on the run-time stack. v is called by constant reference. Hence, a reference to its actual parameter is passed on the run-time stack.

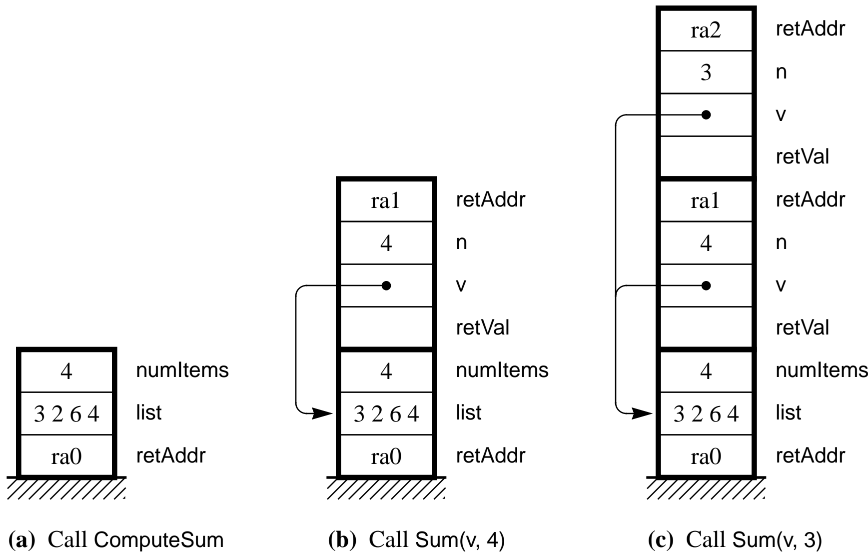


Figure 19.7 The run-time stack for Figure 19.6.

A recursive greatest common divisor function

To reduce a fraction to lowest terms you must determine the greatest common divisor (*gcd*) of the numerator and denominator, then divide them both by the *gcd*.

Example 19.1 The fraction $24/30$ is not in lowest terms. The divisors of 24, other than 1 and 24 itself, are 2, 3, 4, 6, 8, and 12. The divisors of 30 are 2, 3, 5, 6, 10, and 15. The common divisors of 24 and 30 are 2, 3, and 6. Of these common divisors the greatest is 6. Therefore, to reduce $24/30$ to lowest terms you divide 24 by 6 and 30 by 6 to get $4/5$. ∎

An elegant algorithm for computing the *gcd* of two integers is based on the following mathematical property.

$$gcd(m, n) = \begin{cases} m & \text{if } n = 0 \\ gcd(n, m \bmod n) & \text{if } n > 0 \end{cases}$$

It is a recursive property, because the *gcd* function is defined in terms of itself.

Example 19.2 The recursive property of the *gcd* computes the greatest common divisor of 24 and 30 as follows.

$$
\begin{aligned}
gcd(24, 30) &= gcd(30, 24 \bmod 30) \\
&= gcd(30, 24) \\
&= gcd(24, 30 \bmod 24) \\
&= gcd(24, 6) \\
&= gcd(6, 24 \bmod 6) \\
&= gcd(6, 0) \\
&= 6
\end{aligned}
$$

The algorithm is based on the fact that if an integer k divides both m and n, then it divides m mod n. To see why this is true consider the relationship between div and mod.

- m div n is the quotient of $m \div n$.
- m mod n is the remainder of $m \div n$.

Let

$$
\begin{aligned}
q &= m \text{ div } n \\
r &= m \bmod n
\end{aligned}
$$

Then the relationship between div and mod is expressed mathematically as

$$m = q \cdot n + r \qquad 0 \le r < n$$

Example 19.3 For $m = 30$ and $n = 24$,

$$
\begin{aligned}
q &= m \text{ div } n = 30 \text{ div } 24 = 1 \\
r &= m \bmod n = 30 \bmod 24 = 6
\end{aligned}
$$

The mathematical relationship between div and mod states that

$$30 = 1 \cdot 24 + 6 \qquad 0 \le 6 < 24$$

What does it mean for k to divide m? It means that there exists some integer, call it m', such that $m = km'$. So, if k divides both m and n, then

$$
\begin{aligned}
m &= km' \\
n &= kn'
\end{aligned}
$$

The mathematical relationship between div and mod becomes

$$km' = q \cdot kn' + r$$

Solving for r and factoring out k yields

$$
\begin{aligned}
r &= km' - qkn' \\
&= k \cdot (m' - qn')
\end{aligned}
$$

Because r is k times some integer, the last equation says that k divides r, which is m mod n. So, if k divides both m and n, then it divides m mod n. That is why the *gcd* of m and n is the *gcd* of n and m mod n.

The algorithm terminates by making the second parameter smaller until it reaches zero, at which time the first parameter is the *gcd* of the two numbers. Example 19.2 shows that if $m < n$ the first call to $gcd(m, n)$ simply switches m and n. Thus, the call has made the second parameter smaller. That operation is mathematically justified because the *gcd* of two integers does not depend on their order.

On the other hand, if m is not less than n then the call gives n to the first parameter and computes m mod n as the second parameter. Because the mod operation is the remainder when you divide m by n, that remainder is guaranteed to be between 0 and $n - 1$. It is, therefore, smaller than m as well. Because subsequent calls make the second parameter smaller than both m and n, the algorithm is guaranteed to terminate.

The algorithm terminates when n equals 0. But because n gets its value recursively from m mod n, that can only be possible if n divides m with no remainder. But that implies that n is the *gcd* of m and n. Implementation of the *gcd* function is left as a problem for the student at the end of the chapter.

A recursive binomial coefficient function

The next example of a recursive function has a more complex calling sequence. It is a function to compute the coefficient in the expansion of a binomial expression.

Consider the following expansions:

$$
\begin{aligned}
(x+y)^1 &= x + y \\
(x+y)^2 &= x^2 + 2xy + y^2 \\
(x+y)^3 &= x^3 + 3x^2y + 3xy^2 + y^3 \\
(x+y)^4 &= x^4 + 4x^3y + 6x^2y^2 + 4xy^3 + y^4
\end{aligned}
$$

The coefficients of the terms are called binomial coefficients. If you write the coefficients without the terms, they form a triangle of values called Pascal's triangle. Figure 19.8 is Pascal's triangle for the coefficients up to the seventh power.

You can see from Figure 19.8 that each coefficient is the sum of the coefficient immediately above and the coefficient above and to the left. For example, the binomial coefficient in row 5, column 2, which is 10, equals 4 plus 6. Six is above 10, and 4 is above and to the left.

Mathematically, the binomial coefficient $b(n, k)$ for power n and term k is

$$
b(n, k) = b(n-1, k) + b(n-1, k-1)
$$

Power, n	0	1	2	3	4	5	6	7
1	1	1						
2	1	2	1					
3	1	3	3	1				
4	1	4	6	4	1			
5	1	5	10	10	5	1		
6	1	6	15	20	15	6	1	
7	1	7	21	35	35	21	7	1

Term number, k

Figure 19.8
Pascal's triangle.

That is a recursive definition, because it defines the function $b(n, k)$ in terms of itself. You can also see that if k equals 0, or if n equals k, the value of the binomial coefficient is 1. The complete mathematical definition is

A recursive definition of the binomial coefficient

$$\begin{cases} b(n, 0) = 1 \\ b(k, k) = 1 \\ b(n, k) = b(n-1, k) + b(n-1, k-1) \qquad 0 < k < n \end{cases}$$

where the first two equations are the basis for the recursive function.

```
MODULE Pbox19C;
   IMPORT StdLog;

   PROCEDURE BinomCoeff (n, k: INTEGER): INTEGER;
      VAR
         y1, y2: INTEGER;
   BEGIN
      ASSERT((0 <= k) & (k <= n), 20);
      IF (0 = k) OR (k = n) THEN
         RETURN 1
      ELSE
         y1 := BinomCoeff(n - 1, k);  (* ra2 *)
         y2 := BinomCoeff(n - 1, k - 1);  (* ra3 *)
         RETURN y1 + y2
      END
   END BinomCoeff;

   PROCEDURE ComputeBinomCoeff*;
   BEGIN
      StdLog.String("BinomCoeff(3, 1) = ");
      StdLog.Int(BinomCoeff(3, 1));  (* ra1 *)
      StdLog.Ln
   END ComputeBinomCoeff;

END Pbox19C.
```

Figure 19.9
A recursive computation of the binomial coefficient.

The program in Figure 19.9 computes the value of a binomial coefficient recursively. It is based directly on the recursive definition of $b(n, k)$. To keep the following figures simple, the program always computes the same coefficient and does not ask the user for an arbitrary coefficient.

Figure 19.10 shows a trace of the run-time stack. Figures 19.10(b), (c), and (d) show the allocation of the stack frames for the first three calls to procedure BinomCoeff. They represent calls to BinomCoeff(3, 1), BinomCoeff(2, 1), and BinomCoeff(1, 1). The first stack frame has the return address ra1 of the calling program in procedure ComputeBinomCoeff. The next two stack frames have the return address ra2 of the y1 assignment statement.

Figure 19.10(e) shows the return from BinomCoeff(1, 1). y1 gets the value 1 returned by the function in Figure 19.10(f). Then the y2 assignment statement calls the function BinomCoeff(1, 0). Figure 19.10(g) shows the run-time stack just after the function call to BinomCoeff(1, 0). Each stack frame has a different return address.

The calling sequence for this program is different from the previous recursive programs. The other programs keep allocating stack frames until the run-time stack reaches its maximum height. Then they keep deallocating stack frames until the run-

Figure 19.10
The run-time stack for Figure 19.9. BC stands for BinomCoeff.

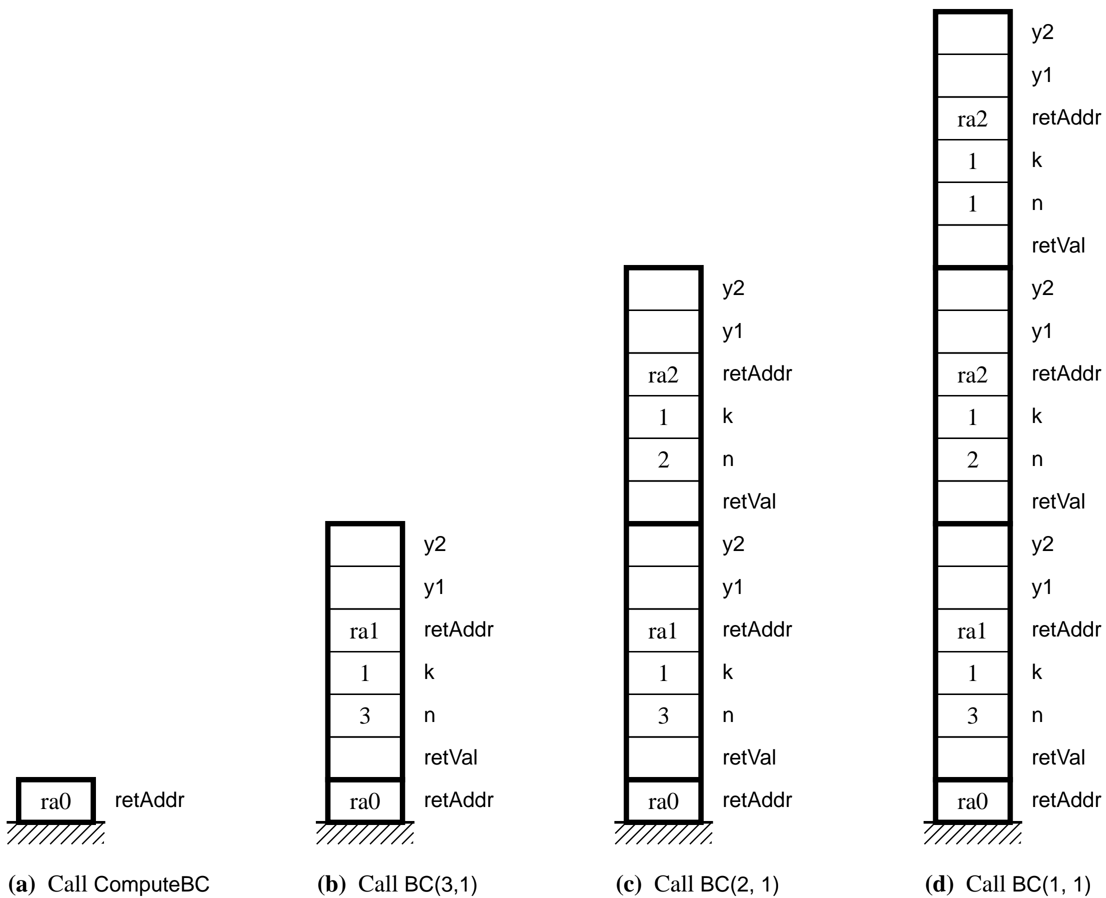
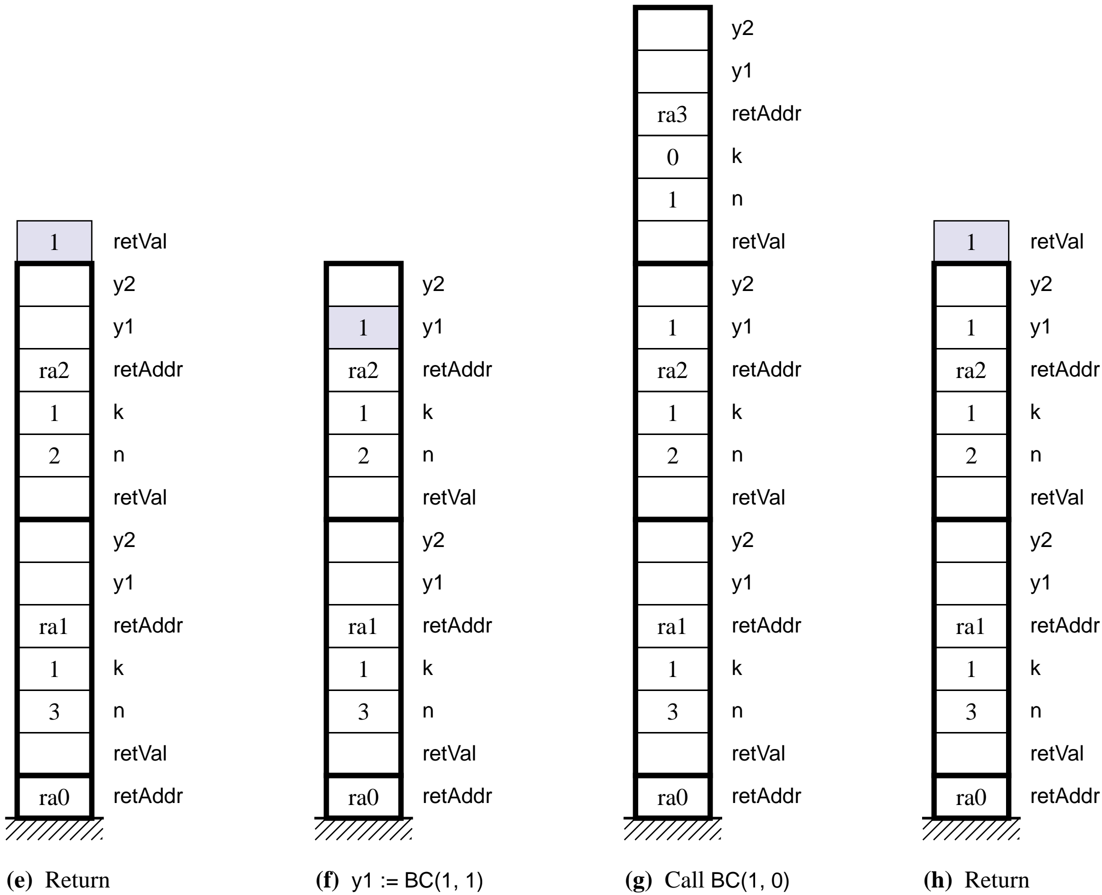


Figure 19.10
Continued.



(e) Return **(f)** y1 := BC(1, 1) **(g)** Call BC(1, 0) **(h)** Return

time stack is empty.

This program allocates stack frames until the run-time stack reaches its maximum height. It does not deallocate stack frames until the run-time stack is empty, however. From Figures 19.10(d) to (e) and (f) it deallocates, but from Figure 19.10(f) to (g) it allocates. From Figures 19.10(g) to (h), (i), (j), and (k) it deallocates, but from Figure 19.10(k) to (l) it allocates. Why?

Because this function has two recursive calls instead of one. If the basis step is true, the function makes no recursive call. But if the basis step is false, the function makes *two* recursive calls, one for y1 and one for y2.

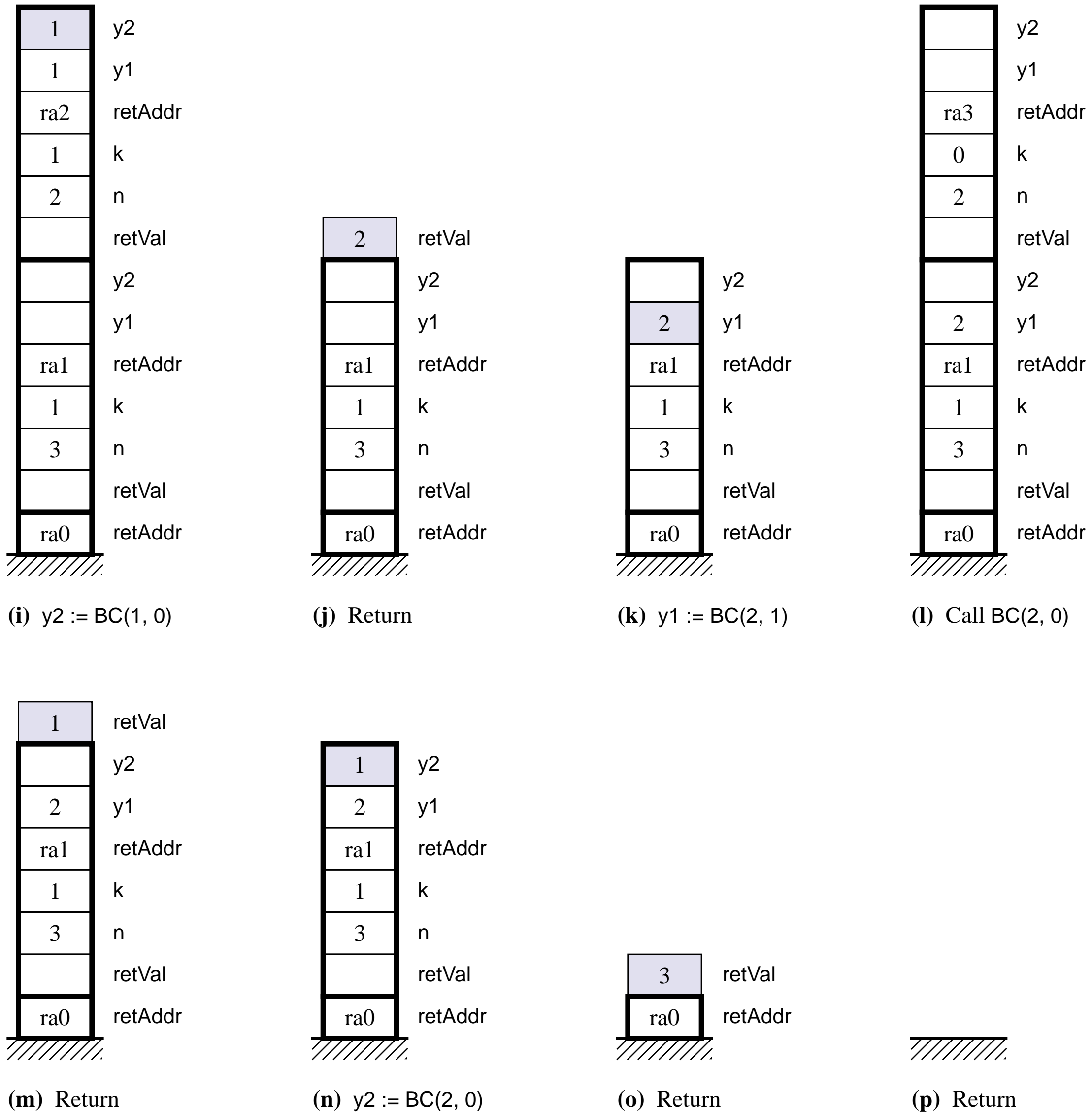


Figure 19.10
Continued.

Figure 19.11 shows the calling sequence for the program. Notice that it is in the shape of an inverted tree with the root of the tree at the top and the leaves at the bottom. Each node of the tree represents a function call. Except for the main program, a node has either two children or no children, corresponding to two recursive calls or no recursive calls.

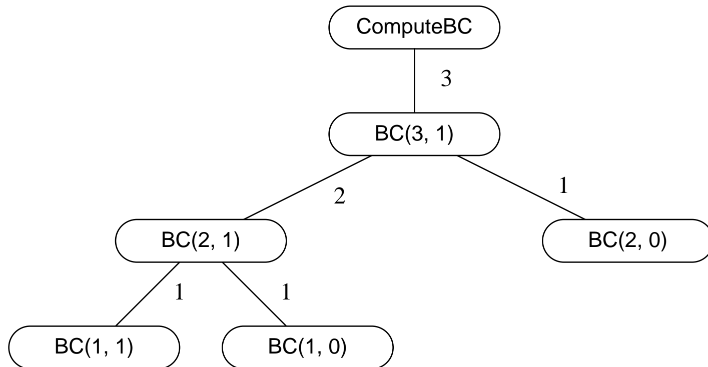


Figure 19.11
The call tree for Figure 19.9. The numbers next to the branches represent the value returned by the function.

Referring to Figure 19.11, the sequence of calls and returns is

ComputeBinomCoeff
    Call BC (3, 1)
        Call BC (2, 1)
            Call BC (1, 1)
        Return to BC (2, 1)
            Call BC (1, 0)
        Return to BC (2, 1)
    Return to BC (3, 1)
        Call BC (2, 0)
    Return to BC (3, 1)
Return to ComputeBinomCoeff

Figure 19.12 shows how to visualize the execution sequence. You can think of the call tree as a land mass in an ocean. A boat begins at the left side of the root and sails along the coastline. It continually moves forward in and out of the bays always keeping the land mass to its left until it arrives back at the right side of the root.

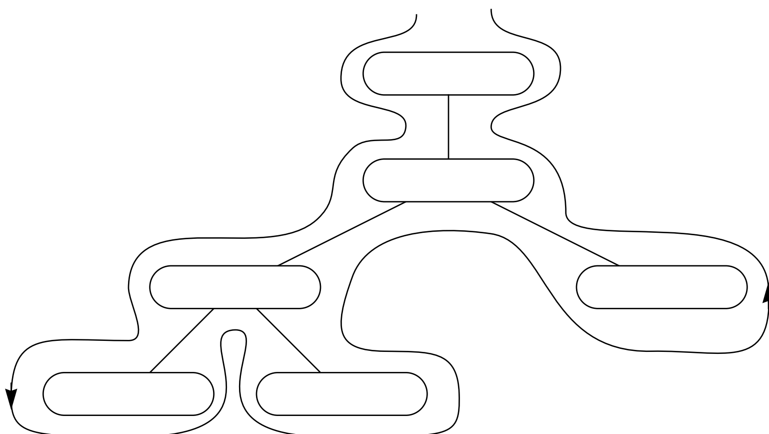


Figure 19.12
A visualization of the calling sequence for the call tree of Figure 19.11.

When analyzing a recursive program from a microscopic point of view, it is easier to construct the call tree before you construct the trace of the run-time stack. Once you have the tree, it is easy to see the behavior of the run-time stack. Every time a transition is made to a lower node in the tree, the program allocates one stack frame. Every time a transition is made to a higher node in the tree, the program deallocates one stack frame.

You can determine the maximum height of the run-time stack from the call tree. Just keep track of the net number of stack frames allocated when you get to the lowest node of the call tree. That will correspond to the maximum height of the run-time stack. In this program, the maximum number of stack frames is four, which occurs twice—once with the call to BinomCoeff(1, 1) and once with the call to BinomCoeff(1, 0).

Drawing the call tree in the order of execution is not the easiest way. The previous execution sequence started

```
ComputeBinomCoeff
   Call BC (3, 1)
      Call BC (2, 1)
         Call BC (1, 1)
```

You should not draw the call tree in that order. It is easier to start with

```
ComputeBinomCoeff
   Call BC (3, 1)
      Call BC (2, 1)
         ...
      Call BC (2, 0)
         ...
   Return to BC (3, 1)
Return to BinomCoeff
```

recognizing from the program listing that BC (3, 1) will call itself twice—BC(2, 1) once, and BC (2, 0) once. Then you can go back to BC (2, 1) and determine its children. In other words, determine all the children of a node before analyzing the deeper calls from any one of the children.

This is a "breadth first" construction of the tree as opposed to the "depth first" construction that follows the execution sequence. The problem with the depth-first construction arises when you return up several levels in a complicated call tree to some higher node. You might forget the state of execution the node is in and not be able to determine its next child node. If you determine all the children of a node at once, you no longer need to remember the state of execution of the node.

Reversing an array

The module in Figure 19.13 has a recursive proper procedure instead of a function. It reverses the elements in an array of characters. Remember that to solve a problem recursively, you need to think of solving a problem with a large size in terms of the

same problem with a smaller size. Suppose you want to reverse the string “Backward” to produce “drawkcaB”. This is a problem involving eight characters. But what if you could assume that you had a procedure that would automatically reverse the middle six characters.

Here is the paradox of designing recursive solutions: You need to write a procedure assuming that it is already written. In this problem, you write a procedure that will reverse eight characters by assuming you can call the *same* procedure to reverse the middle six characters. So, the parameter list must include the indices of the first and last positions of the array. To reverse all eight characters in an array, simply exchange the first and last characters, then call the procedure recursively to reverse the middle six characters.

In general the procedure reverses the characters in the array str between str[first] and str[last]. The calling procedure wants to reverse the characters between ‘B’ and ‘d’. So it calls Reverse with 0 for first and 7 for last. The called procedure switches str[first] with str[last] and calls itself recursively to switch all the characters between str[first + 1] and str[last - 1]. If first is ever greater than or equal to last, no switching is necessary and the procedure does nothing.

```
MODULE Pbox19D;
   IMPORT StdLog;

   PROCEDURE Reverse (VAR str: ARRAY OF CHAR; first, last: INTEGER);
      (* Reverses the characters between str[first] and str[last] *)
      VAR
         temp: CHAR;
   BEGIN
      ASSERT((0 <= first) & (last < LEN(str$)), 20);
      IF first < last THEN
         temp := str[first];
         str[first] := str[last];
         str[last] := temp;
         Reverse(str, first + 1, last - 1)
      END  (* ra2 *)
   END Reverse;

   PROCEDURE ComputeReverse*;
      VAR
         word: ARRAY 16 OF CHAR;
   BEGIN
      word := "Backward";
      Reverse(word, 0, LEN(word$) - 1);
      StdLog.String(word);  (* ra1 *)
      StdLog.Ln
   END ComputeReverse;

END Pbox19D.
```

Figure 19.13
A recursive procedure to reverse the elements of an array.

Figure 19.14 shows the beginning of a trace of the run-time stack. In this program, str must be called by reference because the procedure changes the values of

the array in the actual parameter list. Even though there are multiple copies of str, one in each stack frame, they all refer to word in the calling procedure.

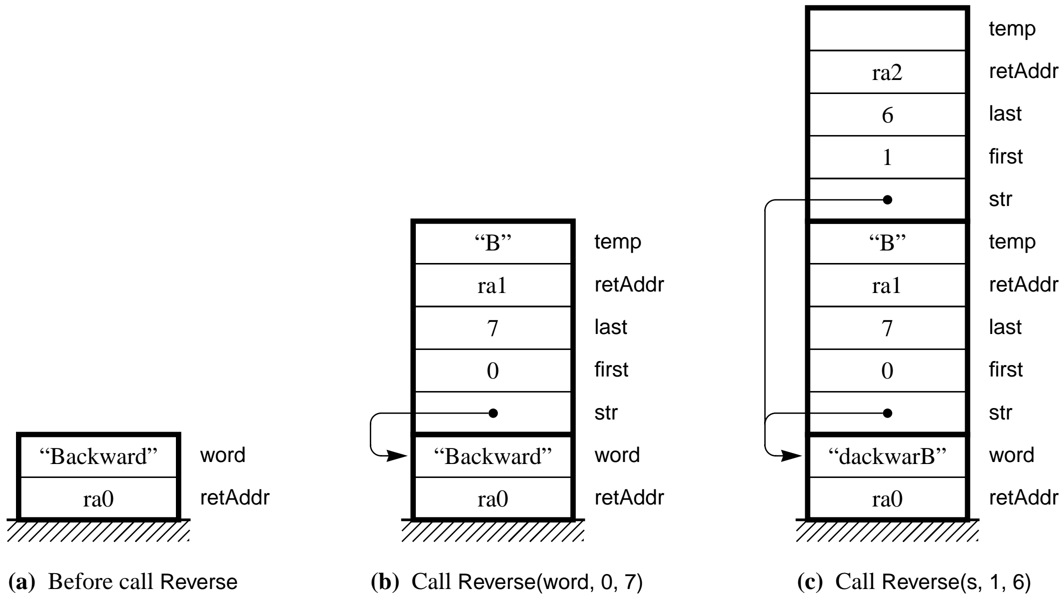


Figure 19.14
The run-time stack for Figure 19.13.

Permutations

The next example of recursion has the most complex calling sequence yet. The problem is to print all the permutations of a list of characters. For example, the characters abcd have 24 permutations as follows:

abcd	bacd	cabd	dabc
abdc	badc	cadb	dacb
acbd	bcad	cbad	dbac
acdb	bcda	cbda	dbca
adbc	bdac	cdab	dcab
adcb	bdca	cdba	dcba

In general, we need a procedure that will print all the permutations of any number of characters.

Remember, the key to a recursive solution is to solve the large problem assuming you already have the solution to a smaller problem. This problem is to print the permutations of the characters in some array of characters, say str, between str[0] and str[3]. As usual the procedure heading will include the array for which we want the permutation and the limits of the indices.

```
PROCEDURE Permute (str: ARRAY OF CHAR; first, last: INTEGER);
   (* Print the permutations of str between str[first] and str[last] *)
```

In this problem we will call the procedure with str having a value of “abcd”, and first and last having values 0 and 3.

What can you assume? You can assume you have a procedure that will print all the permutations between str[1] and str[3]. For example, if you give the procedure the characters “xabc” for str and values 1 and 3 for first and last, you can assume that it will print the following six permutations:

xabc xacb xbac xbca xcab xcba

Look at the pattern of permutations for four characters. It is simply four groups of six permutations. Each group starts with one of the four characters and contains the six permutations of the remaining three characters. You can print the permutations of four characters as follows:

Make str[0] ‘a’
Print permutations from str[1] to str[3] with str[0] at the beginning
Make str[0] ‘b’
Print permutations from str[1] to str[3] with str[0] at the beginning
Make str[0] ‘c’
Print permutations from str[1] to str[3] with str[0] at the beginning
Make str[0] ‘d’
Print permutations from str[1] to str[3] with str[0] at the beginning

This is obviously a job for a loop. To make the first character of str each letter in turn, simply exchange it with each of the other characters.

```
FOR i := 0 TO 3 DO
   Exchange str[0] with str[i]
   Print all permutations from str[1] to str[3]
END
```

For this scheme to work, the procedure that prints the permutations cannot change any of the values in str. The array of characters must be called by value.

Starting with abcd, the loop exchanges ‘a’ with ‘a’ and prints the first group of six permutations starting with abcd. Then it exchanges ‘a’ with ‘b’ and prints the group of six permutations starting with bacd. Then it exchanges ‘b’ with ‘c’ and prints the group of six permutations starting with cabd. Then it exchanges ‘c’ with ‘d’ and prints the group of six permutations starting with dabc.

In general, you want to be able to print the permutations between str[first] and str[last], where first and last are parameters. The loop generalizes to

```
FOR i := first TO last DO
   Exchange str[first] with str[i]
   Print permutations from str[first + 1] to str[last]
END
```

Figure 19.15 shows the completed program.

```
MODULE Pbox19E;
   IMPORT StdLog;

   PROCEDURE Exchange (VAR s: ARRAY OF CHAR; i, j: INTEGER);
      VAR
         temp: CHAR;
   BEGIN
      temp := s[i];
      s[i] := s[j];
      s[j] := temp
   END Exchange;

   PROCEDURE Permute (str: ARRAY OF CHAR; first, last: INTEGER);
      (* Print the permutations of str between str[first] and str[last] *)
      VAR
         i: INTEGER;
   BEGIN
      ASSERT((0 <= first) & (first <= last) & (last < LEN(str$)), 20);
      IF first = last THEN
         StdLog.String(str); StdLog.Ln
      ELSE
         FOR i := first TO last DO
            Exchange(str, first, i);
            Permute(str, first + 1, last)
         END
      END
   END Permute;

   PROCEDURE ComputePermutation*;
   BEGIN
      Permute("abc", 0, 2);
   END ComputePermutation;

END Pbox19E.
```

Figure 19.15
A recursive procedure that prints the permutations of the elements in an array.

Figure 19.16 shows the call tree for the case where procedure Permute is given initial values of “abc” for s, and 0 and 2 for first and last. Procedure ComputePermutation calls the procedure once with first equals 0 and last equals 2. The FOR loop executes three times. Each time it executes it makes a recursive call to Permute. So, the node below the main program has three children.

Each child has first equals 1 and last equals 2. Their FOR loops execute twice, so they each have two children. The bottom nodes have first equals 2 and last equals 2. They have no children. They simply print the value they received in the array. The six leaves on the tree print the six permutations.

Towers of Hanoi

The Towers of Hanoi puzzle is a classic computer science problem that is conveniently solved by the recursive technique. The puzzle consists of three pegs and a set

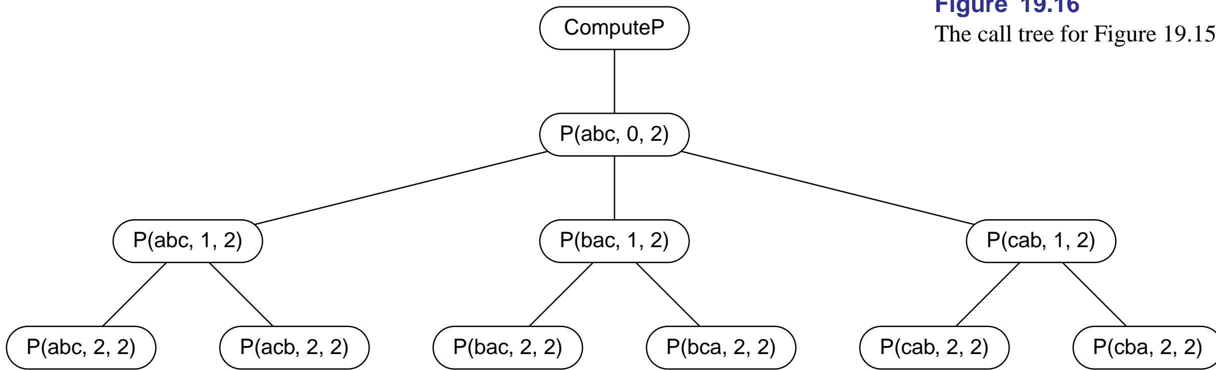


Figure 19.16
The call tree for Figure 19.15.

of disks with different diameters. The pegs are numbered 1, 2, and 3. Each disk has a hole at its center so that it can fit onto one of the pegs. The initial configuration of the puzzle consists of all the disks on one peg in a way that no disk rests directly on another disk with a smaller diameter. Figure 19.17 is the initial configuration for four disks.

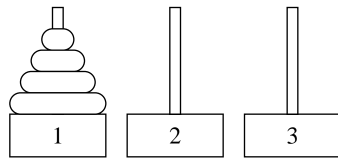


Figure 19.17
The Towers of Hanoi puzzle.

The problem is to move all the disks from the starting peg to another peg under the following conditions:

- You may only move one disk at a time. It must be the top disk from one peg, which is moved to the top of another peg.
- You may not place one disk on another disk having a smaller diameter.

The procedure for solving this problem has three parameters, n, i, and j, where

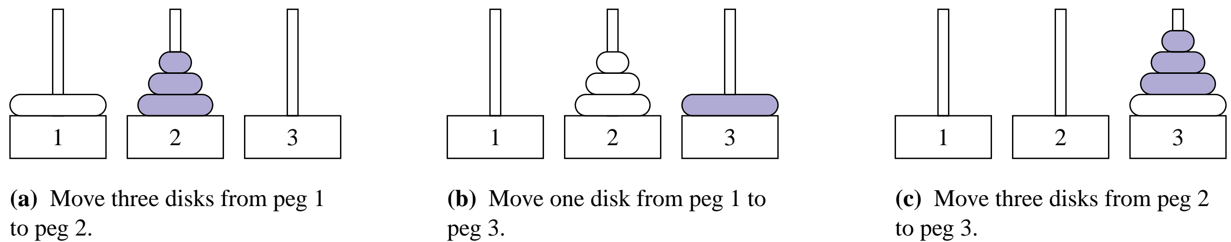
- n is the number of disks to move
- i is the starting peg
- j is the goal peg

i and j are integers that identify the pegs. Given the values of i and j, you can calculate the intermediate peg, which is the one that is neither the starting peg nor the goal peg, as 6 - i - j. For example, if the starting peg is 1 and the goal peg is 3 then the intermediate peg is $6 - 1 - 3 = 2$.

To move the n disks from peg i to peg j, first check to see if n = 1. If it does, then simply move the one disk from peg i to peg j. But if it does not, then decompose the problem into several smaller parts.

- Move n - 1 disks from peg i to the intermediate peg.
- Move one disk from peg i to peg j.
- Move n - 1 disks from the intermediate peg to peg j.

Figure 19.18 shows this decomposition for the problem of moving four disks from peg 1 to peg 3.



(a) Move three disks from peg 1 to peg 2.

(b) Move one disk from peg 1 to peg 3.

(c) Move three disks from peg 2 to peg 3.

Figure 19.18 The solution for moving four disks from peg 1 to peg 3.

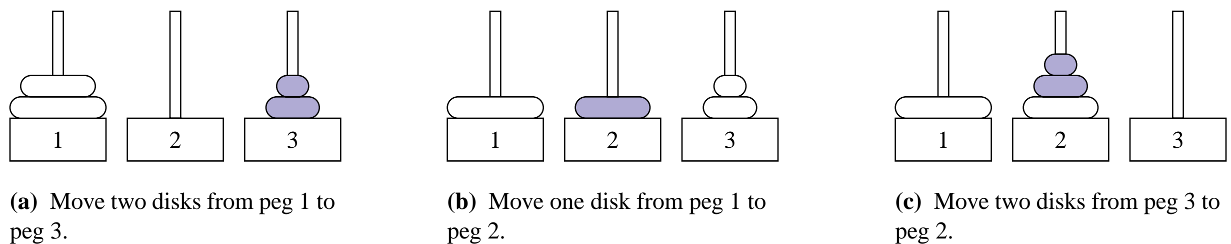
This procedure guarantees that a disk will not be placed on another disk with a smaller diameter, assuming that the original n disks are stacked correctly. Suppose, for example, that four disks are to be moved from peg 1 to peg 3 as in Figure 19.18. The procedure says that you should move the top three disks from peg 1 to peg 2, move the bottom disk from peg 1 to peg 3, and then move the three disks from peg 2 to peg 3.

In moving the top three disks from peg 1 to peg 2, you will leave the bottom disk on peg 1. Remember that it is the disk with the largest diameter, so any disk you place on it in the process of moving the other disks will be smaller.

In order to move the bottom disk from peg 1 to peg 3, peg 3 must be empty. You will not place the bottom disk on a smaller disk in this step either.

When you move the three disks from peg 2 to peg 3, you will place them on the largest disk, now on the bottom of peg 3. So the three disks will be placed on peg 3 correctly.

The procedure is recursive. In the first step, you must move three disks from peg 1 to peg 2. To do that, move two disks from peg 1 to peg 3, then one disk from peg 1 to peg 2, then two disks from peg 3 to peg 2. Figure 19.19 shows this sequence.



(a) Move two disks from peg 1 to peg 3.

(b) Move one disk from peg 1 to peg 2.

(c) Move two disks from peg 3 to peg 2.

Figure 19.19 The solution for moving three disks from peg 1 to peg 2.

Using the previous reasoning, these steps will be carried out correctly. In the process of moving two disks from peg 1 to peg 3, you may place any of these two disks on the bottom two disks of peg 1 without fear of breaking the rules.

Eventually you will reduce the problem to the basis step where you only need to move one disk. But the solution with one disk is easy. Programming the solution to the Towers of Hanoi puzzle is left as a problem at the end of the chapter.

Mutual recursion

Some problems are best solved by procedures that do not call themselves directly but that are recursive nonetheless. Suppose a procedure calls procedure A, and procedure A contains a call to procedure B. If procedure B contains a call to procedure A, then A and B are mutually recursive. Even though procedure A does not call itself directly, it does call itself indirectly through procedure B.

There is nothing different about the implementation of mutual recursion compared to plain recursion. Stack frames are allocated on the run-time stack the same way, with parameters allocated first, followed by the return address, followed by local variables.

There is one slight problem in specifying mutually recursive procedures in a Component Pascal program, however. It arises from the fact that procedures must be declared before they are used.

If procedure A calls procedure B, the declaration of procedure B must appear before the declaration of procedure A in the listing. But, if procedure B calls procedure A, the declaration of procedure A must appear before the declaration of procedure B in the listing. The problem is that if each calls the other, each must appear before the other in the listing, an obvious impossibility.

For this situation, Component Pascal provides the forward declaration, which allows the programmer to write the first procedure heading without the rest of the procedure. In a forward declaration, you include the heading with the formal parameter list, but you insert the ^ character after the reserved word PROCEDURE. After the forward declaration comes the declaration of the second procedure, followed by the declaration of the first procedure with formal parameters that match the formal parameters of the forward declaration.

Figure 19.20 is an outline of the structure of the mutually recursive procedures A and B as just discussed:

```
MODULE Alpha;
CONST, TYPE, VAR of Alpha

PROCEDURE^ A (x: SomeType);

PROCEDURE B (Y: SomeOtherType);
    Procedure body for B, including CONST, TYPE, VAR, etc.

PROCEDURE A (x: SomeType);
    Procedure body for A, including CONST, TYPE, VAR, etc.

BEGIN
    etc.
END Alpha.
```

Figure 19.20 The structure of a program with mutual recursion.

If B has a call to A, the compiler will be able to verify that the number and types of the actual parameters match the formal parameters of A scanned earlier in the forward declaration. If A has a call to B, the call will be in the procedure body of A. The compiler will have scanned the declaration of B because it occurs before the procedure body of A.

Mutual recursion is rare in practice with one notable exception. Some compilers are based on a technique called recursive descent, which uses mutual recursion heavily. You can get an idea of why this is so by considering the structure of Component Pascal statements. It is possible to nest an IF inside of a WHILE, which is nested in turn inside of another IF. A compiler that uses recursive descent has a procedure to translate IF statements and another procedure to translate WHILE statements. When the procedure that is translating the outer IF statement encounters the WHILE statement, it calls the procedure that translates WHILE statements. But when that procedure encounters the nested IF statement, it calls the statement that translates IF statements; hence the mutual recursion. We will leave complete examples of mutual recursion to the problems following this chapter.

The cost of recursion

The selection of examples in this section was based on only one criterion—the ability of the example to illustrate recursion. You can see that recursive solutions require much storage for the run-time stack. It also takes time to allocate and deallocate the stack frames. Recursive solutions are expensive in both space and time.

If you can solve a problem easily without recursion, the nonrecursive solution will usually be better than the recursive solution. Function procedure Factorial in Figure 13.4, the nonrecursive function to calculate the factorial, is certainly better than the recursive factorial function of Figure 19.2. Both procedure Sum in Figure 19.6 and procedure Reverse in Figure 19.13 can easily be programmed iteratively with a loop. The iterative versions are simpler and more efficient than the recursive versions.

The binomial coefficient $b(n, k)$ has a nonrecursive definition that is based on factorials.

$$b(n, k) = \frac{n!}{k!(n-k)!}$$

If you compute the factorials nonrecursively, a program based on this definition may be more efficient than the corresponding recursive program. Here the choice is a little less clear, because the nonrecursive solution requires multiplication and division but the recursive solution requires only addition. Also, the nonrecursive version overflows with smaller values of n and k compared to the recursive version.

Some problems are recursive by nature and can only be solved nonrecursively with great difficulty. The problems of printing the permutations of n letters and solving the Towers of Hanoi puzzle are recursive by nature. You can try to solve them without recursion to see how difficult it would be.

Exercises

1. The function Sum in Figure 19.6 is called for the first time by procedure ComputeSum. From the second time on it is called by itself. **(a)** How many times is it called altogether assuming the input of Figure 19.5? **(b)** Draw a picture of the run-time stack just after the function is called for the third time. You should have four stack frames, including the one for ComputeSum.

2. For the following call statements from ComputeBinomCoeff

(a) StdLog.Int(BinomCoeff(4, 1)); (* ra1 *)
(b) StdLog.Int(BinomCoeff(5, 1)); (* ra1 *)
(c) StdLog.Int(BinomCoeff(3, 2)); (* ra1 *)
(d) StdLog.Int(BinomCoeff(4, 4)); (* ra1 *)
(e) StdLog.Int(BinomCoeff(4, 2)); (* ra1 *)

(1) Draw the call tree as in Figure 19.11. (2) How many times is procedure BinomCoeff called? (3) What is the maximum number of stack frames (including the frame for ComputeBinomCoeff) on the run-time stack during the execution? (4) Write the sequence of calls and returns using the indentation style as on page 414.

3. For Exercise 2, draw the run-time stack as in Figure 19.10 just before the return from the following function calls.

(a) BinomCoeff(2, 1) **(b)** BinomCoeff(3, 1)
(c) BinomCoeff(1, 1) **(d)** BinomCoeff(4, 4)
(e) BinomCoeff(2, 1)

In part (e), BinomCoeff(2, 1) is called twice. Draw the run-time stack just before the return from the second call of the function.

4. Draw the calling sequence for Figure 19.13. How many times is procedure Reverse called? What is the maximum number of stack frames (including the frame for ComputeReverse) allocated on the run-time stack? Draw the run-time stack just after the third call to procedure Reverse.

5. Answer the three questions below and draw the call tree as in Figure 19.16 for procedure Permute of Figure 19.15 for the following call statements from procedure ComputePermutation.

(a) Permute ("wxyz", 0, 3) **(b)** Permute ("wxyz", 1, 3)
(c) Permute ("wxyz", 1, 2) **(d)** Permute ("wxyz", 2, 2)

How many times is procedure Permute called? What is the maximum number of stack frames (including the frame for ComputePermutation) on the run-time stack during the execution? In what order does the program make the calls and returns?

6. For Exercise 5, draw the run-time stack just after the following function calls.

(a) Permute ("xwyz", 2, 3) **(b)** Permute ("wyzx", 3, 3)
(c) Permute ("wyxz", 2, 2) **(d)** Permute ("wxyz", 2, 2)

7. The mystery numbers are defined recursively as

$$\begin{cases} Myst(0) = 2 \\ Myst(1) = 1 \\ Myst(n) = 2 \times Myst(n-1) + 4 \times Myst(n-2) & \text{for } n > 1 \end{cases}$$

(a) Draw the call tree for $Myst(4)$. **(b)** What is the value of $Myst(4)$?

8. For your solution to Problem 13, draw the call tree as in Figure 19.16 for the following Fibonacci numbers.

(a) $Fib(3)$ **(b)** $Fib(4)$ **(c)** $Fib(5)$

For each of these calls, (1) how many times is *Fib* called? (2) What is the maximum number of stack frames (including the frame for the procedure linked to the dialog box button) allocated on the run-time stack?

9. For your solution to Problem 15, **(a)** draw the call tree as in Figure 19.16 for the problem to move four disks from peg 1 to peg 3. **(b)** How many times is your procedure called? **(c)** What is the maximum number of stack frames (including the frame for the procedure linked to the dialog box button) on the run-time stack?

10. For your solution to Problem 20, **(a)** draw the call tree as in Figure 19.16 for the example input given in the problem. **(b)** How many times is procedure Comb called? **(c)** What is the maximum number of stack frames (including the frame for the procedure that calls Comb the first time) on the run-time stack?

11. For your solution to Problem 21, **(a)** draw the call tree as in Figure 19.16 for the example input given in the problem. **(b)** How many times is procedure Select called? **(c)** What is the maximum number of stack frames (including the frame for the procedure that calls Select the first time) on the run-time stack?

12. Examine the Component Pascal module that follows. **(a)** Draw the run-time stack just after procedure What is called for the last time. **(b)** What is the output of the program?

```
MODULE Pbox19Exercise12;
   IMPORT StdLog;

   PROCEDURE What (VAR word: ARRAY OF CHAR; j: INTEGER);
   BEGIN
      IF j > 3 THEN
         word[j - 1] := word[6 - j];
         What(word, j - 1)
      END (* ra2 *)
   END What;
```

```
PROCEDURE Mystery*;
    VAR
        string: ARRAY 8 OF CHAR;
BEGIN
    string := 'abcdef';
    What (string, 6);
    StdLog.String(string) (* ra1 *)
END Mystery;

END Pbox19Exercise12.
```

Problems

13. The Fibonacci sequence is

0 1 1 2 3 5 8 13 21 …

Each Fibonacci number is the sum of the preceding two Fibonacci numbers. The sequence starts with the first two Fibonacci numbers, defined as

$$\begin{cases} Fib(0) = 0 \\ Fib(1) = 1 \\ Fib(n) = Fib(n-1) + Fib(n-2) & \text{for } n > 1 \end{cases}$$

Design a dialog box with one input field for the value of n and one output field for the Fibonacci number. Use a recursive function to compute $Fib(n)$ and output it. Assert a precondition in your function that n cannot be negative, and do nothing if the user enters a negative value. By experimentation, determine the largest value of n that will cause an overflow. Implement a precondition in the function and a test in the calling procedure to prevent the overflow.

14. Design a dialog box with two integer input fields, one field that outputs a message, and one button labeled GCD. Use a recursive function to output the greatest common divisor in the message field. Assert a precondition in your function that at least one parameter must be nonzero. If the user enters two negative numbers display an appropriate message in the message field.

15. Write a program in Component Pascal that prints the solution to the Towers of Hanoi puzzle. It should present the user with a dialog box with three input fields—the number of disks in the puzzle, the peg on which all of the disks are placed initially, and the peg on which the disks are to be moved. Display the solution in a new window. For example, if the user requests the solution for moving three disks from peg 3 to peg 2, your window should display the following solution.

```
Move a disk from peg 3 to peg 2.
Move a disk from peg 3 to peg 1.
Move a disk from peg 2 to peg 1.
Move a disk from peg 3 to peg 2.
Move a disk from peg 1 to peg 3.
Move a disk from peg 1 to peg 2.
Move a disk from peg 3 to peg 2.
```

Assert a precondition in your function that the number of disks must be greater than zero and that the peg numbers must be in 1..3. If the user violates the preconditions write an appropriate error message on the window. Pass the formatter as a parameter as in Figure 12.2.

16. (Silas Smith) Write the recursive function procedure

```
PROCEDURE NumDiskMoves (n: INTEGER): LONGINT
```

that returns the number of moves it takes to transfer n disks in the Towers of Hanoi puzzle from one peg to another one. Output the number of moves after the instructions for moving the disks in Problem 15. Do not use any global variables in NumDiskMoves, and do not change its parameter list as specified above.

17. Write a recursive version of RotateLeft in Figure 15.7. To rotate n items left, rotate the first n - 1 items left recursively, then exchange items n - 2 and n - 1. For example, to rotate the five items

5.0 –2.3 7.0 8.0 0.1

to the left, recursively rotate the first four items to the left,

–2.3 7.0 8.0 5.0 0.1

then exchange items four and five.

–2.3 7.0 8.0 0.1 5.0

Do not use a loop. Test your procedure with input/output as in Figure 15.7. Your program must work with the empty list.

18. Write a function

```
PROCEDURE Maximum (IN v: ARRAY OF INTEGER; last: INTEGER): INTEGER
```

that returns the largest value of the integers in v between v[0] and v[last]. Use recursion without a loop. Test your procedure with a tool dialog box containing one button to load an array and another to compute the maximum. Display the maximum in a read-only field in the dialog box. Assert a precondition in your function that v contains at least one value and verify in the calling procedure that the precondition is not violated. Also assert an appropriate precondition on the upper bound of last.

19. Write a recursive version of boolean function IsPalindrome described in Chapter 15, Problem 29. You will need to modify the parameter list. Do not use a loop.

20. Write a program to print all combinations of n letters taken r at a time. As opposed to permutations, the order of the elements in combinations is irrelevant. For example, the combinations of six letters taken four at a time are the possible sets of four letters from abcdef as follows:

abcd	bcde	cdef
abce	bcdf	
abcf	bcef	
abde	bdef	
abdf		
abef		
acde		
acdf		
acef		
adef		

The solution for selecting four letters from abcdef is to first output ‘a’ followed by the solution for selecting three letters from bcdef. Then output ‘b’ followed by the solution for selecting three letters from cdef. Then output ‘c’ followed by the solution for selecting three letters from def.

The following is the parameter list for a procedure to output the combinations.

```
PROCEDURE Comb (prefix: ARRAY OF CHAR; n, r: INTEGER; suffix: ARRAY OF CHAR);
   (* Prints the prefix string followed by the combination *)
   (* of n characters taken r at a time from the suffix string. *)
   (* Assumes suffix contains n characters. *)
```

To produce the previous list of combinations, the main program called

```
Comb("", 6, 4, "abcdef")
```

The top level recursive calls were

```
Comb("a", 5, 3, "bcdef")
Comb("b", 4, 3, "cdef")
Comb("c", 3, 3, "def")
```

Before each recursive call you will need to concatenate the first character from the suffix onto the last character of the prefix, then strip the first character from the suffix. Test your program with a dialog box for the user to enter a string of up to seven characters and the number of characters from the string to output. Before calling procedure Comb, verify that the number of characters to output is not greater than the length of the string. When the user clicks the compute button, output the combinations to the Log.

21. Write a program to print *n* selections of *m* letters with duplication. As opposed to the elements in combinations of Problem 20, the elements in selections can be duplicated. For example, the two selections of four letters with duplication from abcd are as follows:

aa	ba	ca	da
ab	bb	cb	db
ac	bc	cc	dc
ad	bd	cd	dd

The solution for selecting two letters from abcd is first to output ‘a’ followed by the solution for selecting one letter from abcd. Then output ‘b’ followed by the solution for selecting one letter from abcd. Next output ‘c’ followed by the solution for selecting

one letter from abcd. Finally output 'd' followed by the solution for selecting one letter from abcd.

The following is the parameter list for a procedure to output the selections.

```
PROCEDURE Select (prefix: ARRAY OF CHAR; n: INTEGER; IN suffix: ARRAY OF CHAR);
    (*Prints the prefix string followed by the selection *)
    (* of n characters taken from the suffix string. *)
```

To produce the previous list of selections, the main program called

```
Select("", 2, "abcd")
```

The top level recursive calls were

```
Select("a", 1, "abcd")
Select("b", 1, "abcd")
Select("c", 1, "abcd")
Select("d", 1, "abcd")
```

Before each recursive call you will need to concatenate one character from the suffix onto the last character of the prefix. Test your program with a dialog box for the user to enter a string of up to seven characters and the number of characters from the string to output. Before calling procedure Select, verify that the number of characters to output is not negative. When the user clicks the compute button, output the selections to the Log.

22. The determinant of an $n \times n$ matrix is defined recursively in terms of the determinants of $(n-1) \times (n-1)$ matrices. For example, the 3×3 determinant

$$\begin{vmatrix} 6 & 4 & 7 \\ 0 & 2 & 5 \\ 8 & 9 & 1 \end{vmatrix}$$

is defined recursively in terms of the 2×2 determinants as follows:

$$6\begin{vmatrix} 2 & 5 \\ 9 & 1 \end{vmatrix} - 4\begin{vmatrix} 0 & 5 \\ 8 & 1 \end{vmatrix} + 7\begin{vmatrix} 0 & 2 \\ 8 & 9 \end{vmatrix}$$

In general, the coefficients that multiply the smaller determinants come from the first row of the larger determinant and alternate in sign starting with positive. Each smaller determinant comes from the larger one by eliminating the first row and the column of the coefficient. For example, the second determinant comes from eliminating the first row and the second column of the large determinant, because the coefficient, 4, is in the second column. The determinant of a 1×1 matrix is simply the value of the single element. Write a program that inputs a matrix of integer values from the focus window and outputs the value of its determinant to the Log. The determinant value of the above matrix is –210.

23. At the start of any particular day, a machine is either broken down or in operating condition. If the machine is broken at the start of day n, the probability is p that it will be successfully repaired and in operating condition at the start of day $(n+1)$ and $(1-p)$ that it will still be broken. If the machine is in operating condition at the start of day n, the probability is q that it will have a failure causing it to be broken down at the start of day $(n+1)$ and $(1-q)$ that it will still be in operating condition. At the start of day one, the machine is in operating condition.

The problem is to calculate the probability that the machine is in operating condition on day m. That state can occur in two ways, depending on its state the previous day. Either the machine was broken on the previous day and was repaired with probability p, or it was operating on the previous day and remained operating with probability $(1-q)$. Mathematically,

$$Prob(\text{Operating on day } m) = pProb(\text{Broken on day } (m-1)) + (1-q)Prob(\text{Operating on day } (m-1))$$

Similarly

$$Prob(\text{Broken on day } m) = (1-p)Prob(\text{Broken on day } (m-1)) + qProb(\text{Operating on day } (m-1))$$

Notice that these two relationships are mutually recursive. What is the basis of the recursion?

(a) Declare ProbOperate and ProbBroken, two mutually recursive functions. Use them in a program that inputs the day, m, and probabilities, p and q, and outputs the probability that the machine is operating on day m. **(b)** Draw the call tree for m = 3. **(c)** Use your program to calculate the probability that the machine is operating on days m = 1, 2, 3, 4, 5 if p = 0.4 and q = 0.2. Plot your data. Experiment with different values of p and q and discuss your results.

24. (Gregory Boudreaux) The sum of the first four 1's is

$$\sum_{i=1}^{4} 1 = 1+1+1+1 = 4$$

The sum of the first four integers is

$$\sum_{i=1}^{4} i = 1+2+3+4 = 10$$

The sum of the first four squares is

$$\sum_{i=1}^{4} i^2 = 1+4+9+16 = 30$$

In general, the sum of the first n powers to the j is

$$\sum_{i=1}^{n} i^j = n \sum_{i=1}^{n} i^{j-1} - \sum_{k=2}^{n} \sum_{i=1}^{k-1} i^{j-1}$$

which is a recursive relationship. Defining the function $sum(n, j)$ to be the sum of the first n terms to the power j,

$$sum(n, j) = \sum_{i=1}^{n} i^j$$

the recursive relationship is expressed as

$$\begin{cases} sum(n, 0) = n \\ sum(1, i) = 1 \\ sum(n, j) = n \cdot sum(n, j-1) - \sum_{k=2}^{n} sum(k-1, j-1) \qquad \text{for } n > 1, i > 0 \end{cases}$$

(a) Write a recursive function procedure

```
PROCECURE Sum (n, j: LONGINT): LONGINT
```

that returns the sum of the first n terms to the power j. Test your program with a dialog box containing two input fields for n and j, and one output field for the sum. **(b)** Draw the call tree for the evaluation of Sum(3, 2). Show the value returned for each call on the call tree and verify that the final value returned is the sum of the first three squares.