Recursive Programming in BASIC

Herbert L. Dershem
Hope College, MI

Recursion can be a valuable tool on microcomputers using BASIC. For this reason, it should be in every programmer's repertoire.

What is recursion?

Anything is recursive if it is defined in terms of itself. In programming, a recursive subroutine calls on itself.

Don't confuse recursion with iteration, since both often can solve the same problem. The distinction is clear: a procedure is iterative if the same process is performed, begun and completed repeatedly; a procedure is recursive if, in the middle of its execution, it calls upon itself. Therefore, a recursive procedure begins another execution of itself before the original execution is finished. Such a procedure has more than one execution in progress at a given time, whereas an iterative procedure never has more than one in progress.

Testing for recursion

Not every language implementation permits recursion. In BASIC, recursion is only possible if it is permissible to call a subroutine from itself and still retain the ability of the original execution to return to the proper point. To test your BASIC for recursion capability use the following program:

```
10   REM TEST FOR THE ABILITY OF THE BASIC TO
     PERFORM RECURSION
20   N=1
30   I=0
40   GOSUB 100
50   IF I<>2*N-1 THEN 170
60   PRINT "RECURSIVE"; N-1; "TIMES."
70   N=N+1
80   GOTO 30
100  I=I+1
110  IF I<N THEN 130
120  RETURN
130  GOSUB 100
140  IF I>2*N-1 THEN 170
150  I=I+1
160  RETURN
170  PRINT "NOT RECURSIVE"; N-1;"TIMES."
180  END
```

Every call of subroutine 100 increments I by one. The Nth call results in the setting of I to N and a return without a recursive call. This return to each of the previous levels increments I by one. On the final return, I should equal 2*N-1, if the process was carried out correctly. Thus, if this program runs for a given value of N, this version of BASIC allows N-1 recursive calls. Many versions of BASIC virtually set no limit on the number of such calls possible.

If the BASIC is not recursive to the appropriate level for some value of N, it usually responds with some type of diagnostic message rather than arriving at statement 170.

Writing recursive subroutines

If you have found that your computer's BASIC does allow recursive subroutines, you are now faced with the problem of writing them. Then follow this general outline of recursive subroutine in BASIC:

1. If the first call, initialize the stack pointer.
2. If termination condition, compute result, decrement stack pointer, return.
3. Do computation.
4. Save necessary values in stack.
5. Increment stack pointer.
6. Recursively call this subroutine.
7. Restore saved values from the stack.
8. Do any remaining computation.
9. Decrement stack pointer.
10. Return.

When you recursively call a subroutine from itself, the variables in the called execution destroy the variables of the same name in the calling execution. To preserve the original values of these variables, save in a dimensioned variable (called a stack) those variables you need to recall later. Suppose your subroutine has three variables – X, Y and Z – that it wishes to save for recall, when it returns from a recursive call. The format of the stacks for these variables, which are introduced by three subscripted variables of the same names, uses the following representation:

```
X(1) X from execution 1
Y(1) Y from execution 1
Z(1) Z from execution 1
X(2) X from execution 2
Y(2) Y from execution 2
Z(2) Z from execution 2
etc.
```

To keep track of the position in the stack where the current execution saves its values, use a pointer. Every recursive call increments this pointer by one. Likewise, each return decrements the pointer by one.

The general recursive procedure just outlined shows only one recursive call in the subroutine. In general, steps 3 to 8 may be repeated several times before returning to steps 9 and 10.

An example: calculating N!

Let's take a recursive subroutine and follow these steps. Let's use the classic example of factorial recursion. Unfortunately, it's also a problem because recursion is not the best way to obtain a solution; however, it is the most familiar and simplest of all examples, so why break tradition?

The common definition of N factorial (N!) is iterative and given by: N! = N*(N-1)*(N-2)* . . . *2*1 for N = 1, 2, . . . (where 0! = 1). But there is also this recursive definition of N factorial: N! = N*(N-1)! for N = 1, 2, . . . (where 0! = 1). Notice that the factorial is defined in terms of itself, but with one escape clause which occurs at 0!. The recursive form of a subroutine to compute N! is exemplified by this sample calling program:

```
960     REM THIS SUBROUTINE COMPUTES N FAC-
        TORIAL RECURSIVELY AND
970     REM STORES THE RESULT IN F. THE FIRST
        CALL IS TO 990.
980     REM SUBSEQUENT CALLS ARE TO 1000.
990     S = 1
1000    IF N<>0 THEN 1040
1010    F = 1
1020    S = S-1
1030    RETURN
1040    N(S) = N
1050    S = S+1
1060    N = N-1
1070    GOSUB 1000
1080    N = N(S)
1090    F = N*F
1100    S = S-1
1110    RETURN

10      REM SAMPLE CALLING PROGRAM FOR RE-
        CURSIVE FACTORIALS
20      DIM N (100)
30      INPUT N
40      GOSUB 990
50      PRINT F
60      GOTO 30
```

In this program, statement 990 corresponds to step 1 in the general algorithm given earlier. Statements 1000-1030 correspond to step 2, in which the termination condition is N = 0. No computations in this program correspond to step 3 of the general algorithm. The remaining steps correspond to statements as follows:

Step	Statement(s)
4	1040
5	1050
6	1060-1070
7	1080
8	1090
9	1100
10	1110

Of course, the iterative version of the factorial subroutine is much simpler and executes much faster. This is an iterative version of a sample calling program:

```
970     REM ITERATIVE SUBROUTINE TO COM-
        PUTE N FACTORIAL
980     REM AND STORE IT IN F.
1000    F = 1
1010    IF N<= 1 THEN 1050
1020    FOR I = 2 TO N
1030       F = F*I
1040    NEXT I
1050    RETURN

10      REM CALLING PROGRAM
        TO COMPUTE FACTORIALS
        ITERATIVELY
20      INPUT N
30      GOSUB 1000
40      PRINT F
50      GOTO 20
```

Computing Fibonacci Series

Another example of recursion is calculating the Fibonacci sequence of numbers. The Nth number in the Fibonacci sequence, F(N), is defined in terms of its two predecessors.

F(0) = 0
F(1) = 1
F(N) = F(N-1) + F(N-2) for N + 2, 3,. . .

The BASIC version of this algorithm and its calling program are:

```
970     REM SUBROUTINE 990 CALCULATES THE
        NTH FIBONACCI NUMBER
980     REM RECURSIVELY
        AND RETURNS IT IN F.
990     S = 1
1000    IF N = 0 THEN 1020
1010    IF N <> 1 THEN 1050
1020    F = N
1030    S = S-1
1040    RETURN
1050    N(S) = N
1060    S = S+1
1070    N = N-1
1080    GOSUB 1000
1090    N = N(S)
1100    F(S) =F
1110    S = S+1
1120    N = N-2
1130    GOSUB 1000
1140    N = N(S)
1150    F = F(S) + F
1160    S = S-1
1170    RETURN

10      REM CALLING PROGRAM
        TO COMPUTE FIBONACCI NOS.
        RECURSIVELY
20      DIM N(50), F(50)
30      INPUT N
40      GOSUB 990
50      PRINT F
60      GOTO 30
```

Again, just as in the case of the factorial, iteration gives a more efficient solution to this problem.

```
970     REM SUBROUTINE 1000 CALCULATES THE
        NTH FIBONACCI NUMBER
980     REM ITERATIVELY AND RETURNS IT IN F.
1000    F = 1
1010    P = 0
1020    FOR I = 1 TO N-1
1030        Q = F
1040        F = F + P
1050        P = Q
1060    NEXT I
1070    RETURN
```

```
10    REM CALLING PROGRAM TO ITERATIVELY
      COMPUTE FIBONACCI NOS.
20    INPUT N
30    GOSUB 1000
40    PRINT F
50    GOTO 20
```

Although iteration gives better answers in both of the first two examples of recursion, recursion is the desired technique for many problems because it greatly simplifies the solution algorithm and its implementation in a BASIC program.

Tower of Hanoi

Let's consider two such problems. The first is the Tower of Hanoi, a well-known problem nicely treated in a recursive manner. This problem consists of three pegs, called pegs 1, 2 and 3, and D disks, all of different radius, and are to be stacked on the pegs. Initially, the disks are stacked on peg 1 in order of decreasing size, with the largest disk on the bottom. In this problem, you must move the disks from peg 1 to peg 2 under the restriction that you can only move them one at a time from one peg to another, and that you may never stack a larger disk on top of a smaller one. The recursive solution, which generalizes the problem of moving D disks from a peg called E to peg F, moves the top D-1 disks on peg E to a third peg, then moves the one remaining disk on peg E to peg F, and then moves all of the disks on the third peg to peg F. In this way, the problem of moving D disks is reduced to making two moves of D-1 disks. Therefore, the recursive algorithm for moving the top D disks from peg E to peg F is:

1. If D =1, move top disk from E to F and return.
2. Let G be the number of the peg which is not E or F.
3. Recursively call this procedure to move the top D-1 disk from E to G.
4. Move the disk on E to F.
5. Recursively call this procedure to move the top D-1 disk from G to F.
6. Return.

In the implementation, store the number of disks on peg I in T(I), for I = 1, 2, 3. When a recursive call is made, the values that need saving are E, F and D. Then, the BASIC version of this algorithm is:

```
960     REM TOWER OF HANOI SUBROUTINE TO
        MOVE THE TOP D(S) DISKS
970     REM FROM PEG E(S) TO PEG F(S). T(I) CON-
        TAINS THE NUMBER
980     REM OF DISKS ON PEG 1. INITIAL CALL IS
        TO 990.
990     S = 1
1000    IF D(S) <> 1 THEN 1050
1010    T(E(S)) = T(E(S)) - 1
1020    T(F(S)) = T(F(S)) + 1
1030    PRINT "MOVE"; E(S); "TO"; F(S)
1040    GOTO 1190
1050    G = 6 - (E(S) + F(S))
1060    S = S + 1
1070    D(S) = D(S-1) - 1
1080    F(S) = G
1090    E(S) = E(S-1)
1100    GOSUB 1000
1110    T(E(S)) = T(E(S)) - 1
1120    T(F(S)) = T(F(S)) + 1
1130    PRINT "MOVE"; E(S); "TO"; F(S)
1140    S = S + 1
1150    D(S) = D(S-1) - 1
1160    E(S) = 6 - (E(S-1) + F(S-1))
1170    F(S) = F(S-1)
1180    GOSUB 1000
1190    S = S-1
1200    RETURN
```

```
10    REM CALLING PROGRAM TO SOLVE TOWER OF
      HANOI PUZZLE.
20    REM T(1) CONTAINS THE NUMBER OF DISKS
      ON TOWER I.
30    REM D(1) IS THE TOTAL NUMBER OF DISKS.
40    REM E(1) AND F(1) ARE THE SOURCE AND DES-
      TINATION TOWERS.
50    DIM E(20), F(20), T(3)
60    INPUT T(1)
70    T(2) = 0
80    T(3) = 0
90    E(1) = 1
100   F(1) = 2
110   D(1) = T(1)
120   GOSUB 990
130   GOTO 60
```

Quicksort algorithm

Let's now consider a final useful application of recursion, the quicksort algorithm. You can easily program these efficient and widely-used sorting algorithms recursively.

Suppose that you have stored values in A(L), . . . , A(H) and wish to place them in ascending order in the same storage locations. The basic quicksort algorithm chooses some arbitrary value from this list, say X = A(K), and then rearranges the values so that all values smaller than X are located before it in the list and all values larger than X are located after it. Then X will be located at its correct sorted position in the list of, say, A(I). The same algorithm is then recursively applied with L and I-1 in place of L and H, and then again applied, with I+1 and H. When calling the algorithm with L = H, it simply returns.

The only part of the algorithm that needs some additional attention is the process of rearranging the list so that X is in its proper position and all other values lie on the proper side of X. By keeping two pointers, I and J, you rearrange the list. I starts by pointing at the first position in the list, L. J points to H. Then as pointer I moves down the list, it encounters a value no smaller than X. This value of A(I) should, therefore, lie below X in the list. Next, pointer J moves up the list until it encounters a number no larger than X. That number is exchanged with A(I); both are then in the proper part of the list relative to the eventual position of X. Repeat the process until I and J cross. At that point, the rearrangement is complete, as the following example of the process shows:

I->12	12	12	12	12	12	12
75	I->75	3	3	3	3	3
60	60	I->60	I->60	44	44	44
13	13	13	13	I->13	13	J->13
X->46	X->46	X->46	X->46	JX->46	JX->46	X->46
44	44	44	J->44	60	60	I->60
94	94	94	94	94	94	94
49	49	49	49	49	49	49
95	95	J->95	95	95	95	95
J->3	J->3	75	75	75	75	75

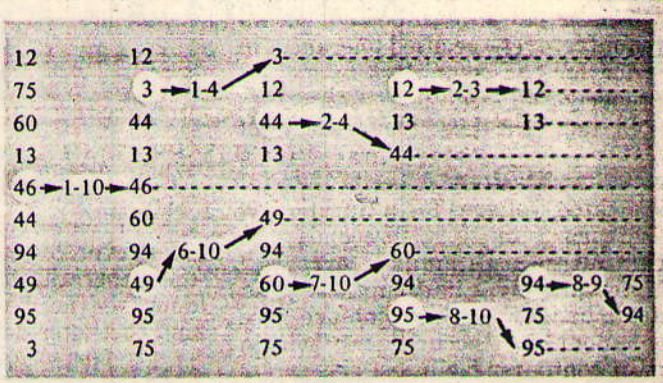
The following program implements this process:

```
950     REM QUICKSORT-SUBROUTINE TO REARRANGE A(L(S)) THRU A(H(S))
960     REM SO THAT ALL VALUES <= X LIE BEFORE X AND ALL VALUES
970     REM > = X LIE AFTER X WHERE X = A (INT ((L(S) + H(S))/2)).
980     REM INITIAL CALL TO 990 SORTS A(1) THRU A(N).
990     S = 1
994     L(1) = 1
997     H(1) = N
1000    IF L(S) >= H(S) THEN 1270
1010    M = INT((L(S) + H(S))/2)
1020    X = A(M)
1030    I = L(S)
1040    J = H(S)
1050    IF A(I) > = X THEN 1080
1060    I = I + 1
1070    GOTO 1050
1080    IF A(J) <= X THEN 1110
1090    J = J-1
1100    GOTO 1080
1110    IF I > J THEN 1170
1120    T = A(I)
1130    A(I) = A(J)
1140    A(J) = T
1150    I = I + 1
1160    J = J - 1
1170    IF I <= J THEN 1050
1180    I(S) = I
1190    S = S + 1
1200    L(S) = L(S-1)
1210    H(S) = J
1220    GOSUB 1000
1230    S = S + 1
1240    L(S) = I(S-1)
1250    H(S) = H(S-1)
1260    GOSUB 1000
1270    S = S - 1
1280    RETURN
10      REM CALLING PROGRAM FOR QUICKSORT.
20      REM A(1) THROUGH A(N) CONTAIN NUMBERS TO BE SORTED.
30      DIM A(100), L(20), H(20), I(20)
40      INPUT N
50      FOR I = 1 TO N
60      A(I) = RND(0)
70      NEXT I
80      GOSUB 990
90      FOR I = 1 TO N
100     PRINT A(I);
110     NEXT I
120     GOTO 40
```

Statement 1000 in this program is the test for termination. Statements 1010-1170 perform the partitioning of the list into those values smaller than X and those larger than X. Two recursive calls follow the partitioning. Only three values, I, L and H, are saved during the recursive call.

Quicksort example

An illustration of an execution of the quicksort algorithm appears as follows. The far left column contains the original list of ten numbers. The circled numbers are the values used for X, and the number-pairs in rectangles are the values of L and H used to partition the list. Note that the procedure begins with L = 1, H = 10, X = 46. Partitioning in the fifth position then places X. Next the process is called for X = 3, L = 1, H = 4, and X = 49, L = 6, H = 10. The process continues in this way until all values are correctly sorted.



Tree searching

These few examples indicate the ease and convenience of doing recursion in BASIC. Another larger area of application for recursion involves tree searching. This application is especially useful for searching game trees or for other applications that need a strategy to be determined through a choice between alternatives.

ABOUT THE AUTHOR

Herbert Dershem is Chairman of the Department of Computer Science at Hope College, Holland, Michigan. He has a Ph.D. in Computer Science from Purdue University.

Rate this article: circle 6L, 6M or 6H on Reader Inquiry Card.

If your address label is printed in red, peel off label, affix to form on page 11, fill out form and return to us to continue your free subscription.