Recursive techniques in problem solving

by A. JAY GOLDSTEIN

Bell Telephone Laboratories, Incorporated
Murray Hill, New Jersey

INTRODUCTION

Recursive methods have long been a standard tool of compiler writers. However, other kinds of computer users have not fully realized that recursion is a powerful technique for solving and programming problems.* Even when a language with recursive facilities is not available, an attack on a problem from a recursive point of view will often show how an iterative program can be written.

In this paper, three problems are approached using recursion: generation of permutations of N objects, evaluation of polynomials in several variables, and generation of all the spanning trees of a graph. These examples illustrate that, when applicable, recursive methods significantly reduce the difficulties of (1) formulating a solution algorithm for the problem, (2) programming the algorithm, and (3) making the program readable.

For those not familiar with recursive programming, the following anthropomorphic analogy may be instructive. John wishes to use a recursive program P which has within it a call to P. He makes a large number of printed copies of the program P and obtains the services of a large number of subordinates. To use the program P, John chooses a subordinate John 1, gives him a copy of P with a list of the arguments of P and tells him to follow the instructions in P and to report back when he is finished with his task. John 1 starts his task and finds in the process that he must call for program P with a new list of arguments. John 1 then chooses a subordinate, John 2, gives him a copy of P with the new list of arguments, and tells him to follow the instructions in P and to report back when he is finished with his task. John 2 starts his task and may have to choose a subordinate, John 3, who may have to choose a subordinate, John 4, etc. Eventually, a subordinate John N starts his task and finishes it without needing a subordinate. John N reports his results to his supervisor John N-1, who completes his task and reports his results to his supervisor John N-2, etc. Sometimes an intermediate supervisor, after receiving the result from his subordinate, may have to choose another subordinate in order to make a further call to program P. Thus, the chain of supervisors can oscillate in length before John 1 finally completes his task and reports the results to John. John is happy.

The actual way in which a language with recursive facilities is implemented on a computer need not concern us here. However, the analogy gives the essence of how recursion works.

Permutations

The need to generate all the arrangements of N distinct objects often arises in combinatoric problems. Here is one such problem.

In the layout of electrical packages, one often wants to place the packages on a frame so as to minimize the amount of wire needed to make the specified connections between packages. Sometimes the actual cost of wiring is crucial, but more often the desire is to minimize electromagnetic and electrostatic coupling between wires by minimizing the amount of wire. Consider the simple one-dimensional problem where the packages are all identical in shape and must be placed along a line. Let us simplify the problem further by assuming that the packages (N+2 of them) are points and are to be placed at the integer points 0, 1, ..., N+1. The designer insists that two of the packages, the 0-th and (N+1)st which are input-output packages, must be located at 0 and N+1. The interconnections are specified by C(I,J)—the number of connections between packages I and J. The location of package I is denoted by P(I) with P(0)=0 and P(N+1) = N + 1.

A naive approach to the minimization problem would be to compute the amount of wire

\[ \sum_{1 < J} C(I,J) \mid P(I) - P(J) \mid \]
for each of the N! arrangements. This requires N(N−1)/2 subtractions and multiplications for each arrangement.

A more sophisticated approach would be to generate the arrangements in such a way that the incremental change in wire length, from one arrangement to the next, would be easy to compute. One way to accomplish this is to generate the arrangements so that each differs from its predecessor by an interchange of adjacent packages. Then the incremental change in wire length can easily be computed by keeping tables NL(J) and NR(J) of the number of wires from package J to packages on its left and right, respectively. If the next arrangement is obtained by interchanging packages L and R in positions p and p + 1, then the incremental change in wire length is

\[ \text{increment} = \text{NL}(L) - \text{NR}(L) - \text{C}(L,R) + \text{NR}(R) - \text{NL}(R) + \text{C}(L,R) \]

Due to the interchange, the quantities NL(J), NR(J) for J = R,L must be modified by C(L,R).

Using this method of generating arrangements, the amount of computation per arrangement is very small and is independent of N, in contrast to the original method in which the amount of computation is proportional to N².

There is a tacit assumption that the importance of obtaining an absolute minimum is great enough to warrant an exhaustive search through the N! arrangements. If the importance is not that great, then successive approximation techniques are available.

We now ask the crucial question: Can all the arrangements on N distinct objects be generated one after another such that each differs from its predecessor by an interchange of adjacent objects?

A recursive approach to this problem begins with the hypothesis: suppose we know how to do it for N − 1 objects we can generate N permutations ga...f, ag...f, ..., a...gf, a...fg on N objects where each differs from the preceding by an interchange of adjacent objects. Applying the hypothesis, we take the next arrangement on N − 1 objects \( a'...f' \) (differing from a...f by an adjacent interchange) and append g on the right. Then, by repeatedly interchanging g with its left neighbor, generate the next N arrangements. This elegant recursive algorithm is due to S. M. Johnson.¹

Frequently we want a program which upon each call returns the next permutation. INTER, given below, is such a function procedure written in PL/I. Each CALL returns an integer INTER which says:

“to obtain the next permutation, interchange the objects in position INTER and INTER+1”. INTER = 0 indicates that all permutations have been generated. Note that the user starts with one arrangement and hence needs only N! − 1 calls to INTER. The N! call must be made—with a return of INTER = 0—in order to clean up the program.

The previous discussion of the generation process shows that INTER

1. N−1 must know location of the N-th object \( a_N \) with respect to \( a_1, ..., a_{N−1} \).

2. (N)− must know whether \( a_N \) is shifting right or left, and

3. (3,N)− must take special action if \( a_N \) is at the left or right end. I.e. the direction of shift must be reversed and a recursive call to INTER must be made to find out what the next interchange on \( a_1, ..., a_{N−1} \) is.

At the next level of recursion produced by the call (3,N), the procedure must have the information in (1,N−1) and (2,N−1) and take the action in (3,N−1). In general (1,K), (2) and (3,K) are needed.

To store this information INTER uses a one-dimensional array P of length N. The sign of P(K) is plus or minus if the K-th object is moving right or left, respectively. The magnitude of P(K) is the relative position of the K-th object, that is, \( |P(K)| \) is 1 + the number of objects \( a_1, ..., a_{K−1} \) to the left of the \( a_K \).

INTER must handle three cases. Case 1. \( P(K) = K \), i.e., \( a_K \) is at the right end and shifting right. Then the direction of shift must be reversed by replacing \( P(K) \) by \( −K \) and the position INTER(K−1) returned. Recall that this says that the next interchange involves the objects in position INTER(K−1) and INTER(K−1)+1. Case 2. \( P(K) = −1 \), i.e., \( a_K \) is at the left end and is shifting left. Then the direction of shift must be reversed by replacing \( P(K) \) by +1 and the position INTER(K−1)+1 returned—one is added because \( a_K \) is at the left. Case 3. \( P(K) \neq K, −1 \). In this case \( P(K) \) is returned if \( P(K) \) is positive and \( |P(K)|−1 \) is returned if \( P(K) \) is negative, and \( P(K) \) is updated to \( P(K) + 1 \).

INTER: PROCEDURE (K) FIXED RECURSIVE;
DECLARE K FIXED, P(50) FIXED STATIC;
/*ARE WE FINISHED*/
IF K ≤ 1 THEN
DO; P = 1; RETURN (0); END;
/*CASE 1*/
IF P(K) = K THEN
DO; P(K) = −K; RETURN (INTER

¹ From the collection of the Computer History Museum (www.computerhistory.org)
/*CASE 2*/
IF P(K) = -1 THEN
  DO; P(K) = 1; RETURN (INTER(K-1) +1); END;
/*CASE 3*/
(PK) = PK+ 1
IF P(K) > 0 THEN RETURN (P(K) - 1 ELSE RETURN iP)K)j);
END INTER;

Polynomial evaluation

Horner's method for evaluation polynomials:

\[ P = A(K) = A(K-1) \cdot X + \ldots + A(1) \cdot X^{(K-1)} \]

has the merit of requiring only 2(K-1) arithmetic operations. In this section we show how the computational advantage of Horner's method can be extended to polynomials in several variables. First, Horner's method is examined from a recursive point of view. Second, consideration is given to the storage limitations due to the astronomical number of possible monomials in a polynomial in several variables. Clearly, we can store only the nonzero monomials. The effect of this mode of storage on the previous recursive method is then examined for the special case of polynomials in one variable and a new recursive method is given. Third, this recursive method is generalized to the case of many variables.

Looking at polynomial evaluation from a recursive point of view, we note that

\[ P = A(K) + X \cdot (A(K-1) + \ldots + A(1) \cdot X^{(K-2)}) \]

That is, we can evaluate a polynomial with K terms if we can first evaluate a polynomial with K - 1 terms. This is the recursive step, and we can define a recursive program

\[ \text{POLYVAL}(A,X,K) = \begin{cases} 0 & \text{for } K = 0 \\ A(K) + X \cdot \text{POLYVAL}(A,X,K-1) & \text{for } K > 1 \end{cases} \]

POLYVAL is just Horner's method.

Before generalizing this to the several variable case, we should consider a storage problem. A polynomial in several variables can have \( (d_1+1)(d_2+1) \ldots \) monomials where the i-th variable appears with maximal degree \( d_i \). We cannot afford the storage for all these monomials and will store only the nonzero ones. The effect of this on Horner's method will be examined next, since, as is typical with recursive approaches, the special case is easily generalized.

Eliminating the nonzero monomials gives the polynomial

\[ A(K) \cdot X^{E(K)} + \ldots + A(1) \cdot X^{E(1)} \]

with \( E(1) > E(2) > \ldots \). It can be expressed in the recursive form

\[ X^{E(K)}(A(K) + (A(K-1) \cdot X^{E(K-1)}) + \ldots) \]

Thus, we can write the recursive procedure

\[
\text{POLYVAL}(A,E,X,K) = \begin{cases} 0 & \text{if } K = 0 \\ X^{E(K)}(A(K) + (A(K-1) \cdot X^{E(K-1)}) + \ldots) & \text{if } K > 0 \end{cases}
\]

Note that \( E - E(K) \) says: subtract \( E(K) \) from every element of \( E \).

The PL/I procedure below is an implementation of the recursion. In order that the array \( E \) of exponents not be modified and to avoid creating a temporary array, the procedure uses the device of saving \( E(K) \) in ELAST.

\[
\text{POLYVAL:PROCEDURE (A,E,X,K) FOAT RECURSIVE;}
\text{DECLARE(K,E(K),ELAST) FIXED,A(K),X) FLOAT;}
\text{1*ELAST IS THE EXPONENT OF THE PREVIOUS TERM. WE CANNOT USE E(K+1) SINCE IT DOES NOT EXIST ON THE FIRST INVOCATION OF POLYVAL.*}
\text{ELAST = 0; GO TO START;}\]
\text{POLYVAL:PROCEDURE (A,E,X,K) FOAT RECURSIVE;}\]
\text{DECLARE(K,E(K),ELAST) FIXED,A(K),X) FLOAT;}
\text{1*ELAST IS THE EXPONENT OF THE PREVIOUS TERM. WE CANNOT USE E(K+1) SINCE IT DOES NOT EXIST ON THE FIRST INVOCATION OF POLYVAL.*}
\text{ELAST = 0; GO TO START;}\]
\text{POLYVAL:PROCEDURE (A,E,X,K) FOAT RECURSIVE;}\]
\text{DECLARE(K,E(K),ELAST) FIXED,A(K),X) FLOAT;}
\text{1*ELAST IS THE EXPONENT OF THE PREVIOUS TERM. WE CANNOT USE E(K+1) SINCE IT DOES NOT EXIST ON THE FIRST INVOCATION OF POLYVAL.*}
\text{ELAST = 0; GO TO START;}\]

To generalize POLYVAL to several variables, we must change \( A(K) \) to a polynomial in several variables and modify the RETURN statement where \( A(K) \) appears. We assume that the polynomial in \( V A R \) variables is expressed as a sum of polynomials (called terms) each consisting of the product of a power of \( X(V A R) \) and a coefficient polynomial in the variables \( X(1), \ldots, X(V A R-1) \). The coefficient polynomial has the same format.

Now we must specify the data structure for representing the polynomials. A PL/I structure for each
of the terms of the polynomial will work very well. The structure must contain the exponent \( E \); a pointer, \( \text{COEF} \), to the polynomial which is the coefficient; a pointer, \( \text{NEXT} \), to the next term in the polynomial; and a number \( \text{VALUE} \) which is the numerical value of the coefficient when the polynomial is a constant. We assume the exponent \( E \) in the \( \text{NEXT} \) term is larger than the present exponent. With this data format decided upon, the generalization of the previous procedure to the several variable case is not difficult.

**POLYVAL**: PROCEDURE (\( P,X,\text{VAR} \)) FLOAT RECURSIVE;

DECLARE

\[ P \text{ POINTER, } /* A POINTER TO THE FIRST TERM IN THE POLYNOMIAL */ \]

\[ \text{VAR} \text{ FIXED, } /* THE NUMBER OF VARIABLES */ \]

\[ X(\text{VAR}) \text{ FLOAT, } /* THE VALUES OF THE VARIABLES */ \]

\[ \text{ELAST} \text{ FIXED, } /* THE EXPONENT OF THE PREVIOUS TERM */ \]

1 TERM CONTROLLED (\( P \))

2 E FIXED, /* EXPONENT OF \( X(\text{VAR}) \) IN THIS TERM */

2 NEXT POINTER, /* POINTER TO THE NEXT TERM. ITS EXPONENT IS LARGER */

2 COEF POINTER, /* A POINTER TO THE FIRST TERM OF THE COEFFICIENT */

2 VALUE FLOAT; /* VALUE OF COEFFICIENT IF \( \text{VAR} = 0 \) */

ELAST = 0; GO TO START;

PV: ENTRY (\( P, X, \text{VAR}, \text{ELAST} \)) FLOAT;

START: IF \( P = \text{NULL} \) THEN RETURN (0);

IF \( \text{VAR} = 0 \) THEN RETURN (\( \text{VALUE} \));

IF \( E = 0 \) THEN \( Y = 1.0 \); ELSE \( Y = X(\text{VAR})^*(E-\text{ELAST}) \);

RETURN (\( Y*(\text{PV}(\text{COEF}, X, \text{VAR}, -1, 0) + \text{PV}(\text{NEXT}, X, \text{VAR}, E)) \));

END POLYVAL;

The McIlroy method for the generation of all spanning trees

For the analysis of electrical networks by topological methods, it is sometimes necessary to find all the spanning trees of a graph. M. Douglas McIlroy has developed an elegant recursive procedure for generating them [2].

In order to understand the algorithm we will first examine one that is less efficient but more transparent. The procedure is TREE1 (\( G,DE,T \)) where \( G \) is the graph, \( DE \) a set of edges deleted from \( G \), and \( T \) is a tree of \( G \) which does not necessarily span \( G \). For any \( T \), the procedure grows all the spanning trees which contain \( T \) as subtree. Each time a spanning tree is generated, it is handed to the user by calling his procedure JOB(T). JOB returns to the TREE1 upon completion of its task and the next tree is generated. The process is initiated by calling TREE1 with \( DE \) and \( T \) empty. We describe TREE1 in a dialect of PL/I.

**TREE1**: Procedure (\( G,DE,T \)) Recursive;

Step 1. If \( T \) is empty then \( T = \) any node of \( G \);

Step 2. If \( T \) spans \( G \) then (call JOB(T); Return);

Step 3. Choose an edge \( e = (b,c) \) in \( G - DE \) with node \( b \) in \( T \) and node \( c \) not in \( T \); If there are none, then return;

/* We now grow all trees which contain \( T \). These trees are divided into two disjoint kinds, namely, those which do not contain the edge \( e \) and those which do contain the edge \( e \).* /

Step 4. Call TREE1 (\( G,DE+e,T \))

Step 5. Call TREE1 (\( G,DE,T+e \))

Return;

End TREE1;

McIlroy's algorithm TREE is obtained from TREE1 by making a crucial observation about step 3. At this step, an edge \( e \), from node \( b \) in \( T \) to node \( c \) not in \( T \), is chosen. At various other stages in the recursion, all the edges from \( c \) to other nodes in \( T \) are considered, one by one. This set of edges is called an attachment set. McIlroy handles all these edges simultaneously by formalizing the notion of the attachment set of a node and introducing the notion of a family of trees. The attachment set \( \text{ATCH}(I) \) of a node \( I \) is a set of nodes which are adjacent to \( I \) and describes a set of edges at node \( I \). A family of trees is described by the array of attachment sets \( \text{ATCH}(1), \text{ATCH}(2), \text{etc.}, \) one for each node of the graph \( G \). A tree of the family is obtained by selecting for each \( I \), one node \( N_I \) from \( \text{ATCH}(I) \) and constructing the tree whose edges are \( \{(I,N_I)\} \). Each possible set of selections gives a tree.

As an example, consider the graph which is a square plus one diagonal. Its edges are \((1,2) (2,3)\)
(3,4) (4,1) (2,4). Then one family of trees is given by

\\begin{align*}
\text{ATCH}(1) &= \text{empty}, \quad \text{ATCH}(2) = \{1\}, \\
\text{ATCH}(3) &= \{2,4\}, \quad \text{ATCH}(4) = \{1,2\}.
\end{align*}

This describes the family of trees

1. (2,1) (3,2) (4,1)
2. (2,1) (3,2) (4,2)
3. (2,1) (3,4) (4,1)
4. (2,1) (3,4) (4,2)

The technique of generating the trees in families produces a tremendous increase in efficiency.

To facilitate the description of TREE, we use set notation. Let the graph $G$ be described by giving for each node $I$, the set $G(I)$, of nodes adjacent to $I$. Let $\text{NODES}$ be the set of nodes of $G$ and let $\text{TNODES}$ be the set of nodes in the growing tree. Finally, let ATCH(I) be the attachment set at node I as above. The steps of TREE parallel those of TREEI except that the set of deleted edges $DE$ is not used. Instead, the graph $G$ is modified before making recursive calls and is restored before returning.

TREE: Procedure $(G, \text{NODES}, \text{TNODES}, \text{ATCH})$

Recursive;

Step 1. If $\text{TNODES}$ is empty then add any node $I$ to it; ATCH(I) = empty;

Step 2. /*Does the family span $G$?*/ If $\text{TNODES} = \text{NODES}$ then (call JOB $(\text{NODES}, \text{ATCH})$; Return;)

Step 3. /*Construct a new attachment set.*/ Choose node $I$ in $\text{NODES}$-$\text{TNODES}$ which is adjacent to at least one node in $\text{TNODES}$; if there are none then return; $\text{TEMP} = G(I) \cap \text{TNODES}$; /*$\text{TEMP}$ is the attachment set at $I$*/

Step 4. /*Grow the family of trees which does not have edges from $I$ to a node in $\text{TEMP}$.*/ $G(I) = G(I) - \text{TEMP}$; Call TREE $(G, \text{NODES}, \text{TNODES}, \text{ATCH})$;

Step 5. /*Grow the family of trees which does have edges from $I$ to a node in $\text{TEMP}$.*/ $G(I) = G(I) + \text{TEMP}$; $\text{TNODES} = \text{TNODES} \cup \{I\}$; ATCH(I) = TEMP; Call TREE $(G, \text{NODES}, \text{TNODES}, \text{ATCH})$; Return;

End TREE;

REMARKS

A recursive procedure sometimes runs more slowly than a corresponding iterative procedure. This phenomenon occurs in recursive procedures which recomputes the same quantity in different branches of the recursion. The wasteful recomputation can be eliminated by an exchange of storage for time. By allocating a table for intermediary results, with flags to indicate which entries have been computed, the procedure can retrieve previously computed values. Using this technique, the recursive procedure, except for the cost of procedure calls, is just as fast as a corresponding iterative procedure. Moreover, the added storage is usually just that storage need to implement the corresponding iterative procedure.

Some other problems which are ideal for a recursive approach are: adaptive integration, the towers of Hanoi game (also called the end of the world game), and determining if one multivariable polynomial divides another.

REFERENCES

1. S M Johnson
   Generation of Permutations by Adjacent Interchanges
   Math Comp 17 283-285 1963
2. M D McIlroy
   to be published
3. W M McKeeman
   Adaptive Numerical Integration by Simpson Rule
   Algorithm 145 CACM 5 604 1962