RECURSION AS A PROGRAMMING TOOL

David Gries

TR 75-234

April 1975

Department of Computer Science
Cornell University
Ithaca, New York 14853

RECURSION AS A PROGRAMMING TOOL

David Gries

I. Introduction

Recursion in programming is just about as old as programming itself. However, most programmers and introductory texts on programming seem to regard recursion as a novelty; the usual examples of factorial, traversing trees and compiling are shown but no attempt is made to show how to use recursion in everyday "practical" programming. Frequently, looking at recursive formulations of a problem can lead to better ideas or just to convincing demonstrations of correctness, even though the end product may be iterative rather than recursive.

I present here two examples of the use of recursive thinking. The first example is to find all permutations of a string of n elements, using only transposition of two adjacent elements to get from me permutation to the next. Recursion was the only tool I could think of at the time, to produce a program in which I had confidence.

The second example, to find the mode of an array is due to Mike Griffiths of Grenoble. His idea of looking at the problem actually led to a simpler algorithm.

II. The Problem of Finding all Permutations

Consider the problem of producing all permutations of n elements in such a way that each succeeding permutation arises by exchanging two adjacent elements in the preceding one. For example, for n = 3 we have 123, 132, 312, 321, 231, 213. The algorithm for this, first discovered by the Polish mathematician Hugo Steinhaus (see Martin Gardner's Mathematical Games Section in Scientific American, November 1974), has a nice, simple recursive formulation. Nonrecursive versions have been produced (see e.g. [3]), but they are usually hard to understand. Recently, C. Reinsch of Munich showed me his version. He had run it on the computer for many values of n but was uneasy about it because, while he intuitively felt it was correct, he didn't know how to <u>prove</u> it correct. He asked me to prove it correct for him.

After studying his program for awhile, I came to the conclusion that I couldn't prove <u>his</u> program correct, I would have to construct my own program, producing the program and its correctness proof hand-in-hand. Some more thought led me to believe that I had to start with a simple recursive formulation of the problem and transform it by a series of <u>small</u>, <u>correct</u> steps into a nonrecursive program. This series of transformations is shown below. Of course, this is an idealized version of my work; certainly I wasn't able to produce every step exactly as I needed it, without mistakes, and my work was not so formally done.

The main idea of this permutation algorithm can be explained by an example. Suppose A = 12345 and we want to find all permutations of A. They are produced in the following order:

```
12345      shift 5 to the left
12354
12534
15234
51234      now find next permutation of 1234
51243      and shift 5 to the right
15243
12543
12453
12435      now find next permutation of 1234
14235
14253      etc.
```

Thus, A[n]=5 is propagated to the left until it reaches the end. Then the next permutation of 1234 is found, after which the 5 is propagated to the right. Then the next permutation of 1234 is found and the whole process repeats.

We adopt some notations which will simplify the presentation. Suppose x and y are strings (sequences) of elements. Then

(1) $|x|$ is the length of (number of elements in) x

(2) x o y is string concatenation; x o y is the string consisting of the elements of x followed by the elements of y. Where it is not confusing, we omit the explicit concatenation operation 0, writing x y.

(3) Let $X = \{x_1, x_2, \ldots, x_n\}$ be a set of strings. Then $X \circ y = \{x_1 \circ y, x_2 \circ y, \ldots, x_n \circ y\}$. $y \circ X$ is similarly defined, as is X o Y for sets of strings X and Y. Clearly o is associative.

(4) x[i] is the i<u>th</u> element of x.
x[i:j] is the string x[i]x[i+1]...x[j], if $i \leq j$
x[i:j] = ϵ, the empty string, if i > j.

(5) If $X = \{x_1, x_2, \ldots, x_n\}$, then
$X[i:j] = \{x_1[i:j], x_2[i:j], \ldots, x_n[i:j]\}$

(6) <u>swap</u>(x,i), where $1 \leq i < |x|$, is the string x, but with elements x[i] and x[i+1] interchanged.

Given a string A with $|A| \geq 2$, we can now define p(A), the set of all permutations of A. Let N = |A| be the length of A, and let S be the string consisting of the first N-1 characters of A:

$$S = A[1:N-1]$$

Then

$$(7)\quad p(A) \equiv \begin{cases} N = 2 \Rightarrow \{A[1]A[2], A[2]A[1]\} \\ N > 2 \Rightarrow \bigcup_{\ell=0}^{N-1} \bigcup_{x \in p(S)} x[1:N-\ell-1] \circ A[N] \circ x[N-\ell:N-1] \end{cases}$$

For example, for N = 3 and A = 123, p(S) = {12,21}, and

$$p(A) = \begin{Bmatrix} 123, \\ 213 \end{Bmatrix} \cup \begin{Bmatrix} 132, \\ 231 \end{Bmatrix} \cup \begin{Bmatrix} 312, \\ 321 \end{Bmatrix}$$

4. <u>Program development</u>

p(S) has |S|! members. If these are numbered in some arbitrary order, 0, 1 ,..., |S|!-1, than we can let P(S,q) = the <u>qth</u> element of p(S) and rewrite (7) as

$$p(A) = \begin{cases} N = 2 \Rightarrow \{A, \underline{\text{swap}}(A,1)\} \\ N > 2 \Rightarrow \bigcup_{\ell=0}^{N-1} \bigcup_{q=0}^{(N-1)!-1} P(S,q)[1:N-\ell-1] \circ A[N] \circ P(S,q)[N-\ell:N-1] \end{cases}$$

We now do some simple formula manipulation to begin getting the formula for p(A) to look somewhat like the final program. Looking back at the 12345 case, we first propogated 5 all the way to the left, then all the way to the right, and then to the left, and so on. Formula 8 sort of says this; q being even indicates a left propogation; q odd a right propogation.

$$\text{(8)}\quad p(A) = \begin{cases} N = 2 \Rightarrow \left\{A,\ \underline{\text{swap}}(A,1)\right\} \\[2ex] N > 2 \Rightarrow \displaystyle\bigcup_{q=0,2,4,\ldots}^{(N-1)!-2} \left\{ \bigcup_{\ell=0}^{N-1} P(S,q)[1{:}N-\ell-1] \circ A[N] \circ P(S,q)[N-\ell{:}N-1] \right. \\[2ex] \qquad\qquad \displaystyle\bigcup_{q=1,3,5,\ldots}^{(N-1)!-1} \left. \left\{ \bigcup_{\ell=0}^{N-1} P(S,q)[1{:}\ell] \circ A[N] \circ P(S,q)[\ell+1{:}N-1] \right\} \right. \end{cases}$$

Using equation 8 we now define a particular ordering of the elements of p(A). If $|A| = 2$, define $P(A,0) = A$, $P(A,1) = A[2]A[1]$. For $|A| > 2$, given i, $0 \le i < |A|!$, i can be uniquely defined by

$$i = q \cdot |A| + \ell, \text{ where } 0 \le q < (|A|-1)!,\ 0 \le \ell < |A|$$

Then we let P(A,i) be the member of p(A) denoted by the unique pair (q,ℓ) in equation (8) above.

We will see that the algorithm for producing all permutations using only interchanges will produce these permutations in exactly this order.

It is also easy to show that for $i = |A|!$, $P(A,i)$ is naturally defined as $P(A,|A|!) = P(A,0)$.

Given a string A_0 with $N = |A_0| > 2$ and $S_0 = A_0[1:N-1]$, equation 8 gives us a recursive definition of the $\underline{ith}$ permutation of A_0, in terms of A_0, S_0 and i, $0 \leq i < N!$ However, the algorithm we are attempting to create generates the next permutation at each step not from A_0, but from the current permutation. Accordingly, we now work to define

$$A_{i+1} = PN(A_i,i) = \text{the } i+1^{st} \text{ permutation of an initial } A_0,$$
$$\text{where } A_i \text{ is the } i^{th} \text{ permutation, and } |A_0| > 1.$$

Given $i = q \cdot N + \ell$, $0 \leq \ell \leq N-1$, $0 \leq q < (N-1)!$ we see we can use equation 8 to distinguish four cases.

Case 1: q even, $0 \leq \ell < N-1$. Then $i+1 = q \cdot N+(\ell+1)$.

$$\begin{aligned} PN(A_i,i) &= P(S_0,q)[1:N-\ell-2] \circ A_0[N] \circ P(S_0,q)[N-\ell-1:N-1] \\ &= \underline{\text{swap}}\big(P(S_0,q)[1:N-\ell-1] \circ A_0[N] \circ P(S_0,q)[N-\ell:N-1],\ N-\ell-1\big) \\ &= \underline{\text{swap}}(P(A_0,i),\ N-\ell-1) \\ &= \underline{\text{swap}}(A_i,\ N-(\ell+1)) \end{aligned}$$

Case 2: q odd, $0 \leq \ell < N-1$. Then $i+1 = q \cdot N+(\ell+1)$.

$$\begin{aligned} PN(A_i,i) &= P(S_0,q)[1:\ell+1] \circ A_0[N] \circ P(S_0,q)[(\ell+1)+1:N-1] \\ &= \underline{\text{swap}}\big(P(S_0,q)[1:\ell] \circ A_0[N] \circ P(S_0,q)[\ell+1:N-1],\ \ell+1\big) \\ &= \underline{\text{swap}}(P(A_0,i),\ell+1) \\ &= \underline{\text{swap}}(A_i,\ell+1) \end{aligned}$$

<u>Case 3</u>: q odd, ℓ = N-1. Then i+1 = (q+1)$\cdot$N, with q+1 even.

$$\begin{aligned}
A_i &= P(S_0,q) \circ A_0[N] \\
PN(A_i,i) &= P(S_0,q+1) \circ A_0[N] \\
&= PN(P(S_0,q),q) \circ A_0[N] \\
&= PN(A_i[1:N-1],q) \circ A_i[N]
\end{aligned}$$

<u>Case 4</u>: q even, ℓ = N-1. Then i+1 = (q+1)$\cdot$N, with q+1 odd.

$$\begin{aligned}
A_i &= A_0[N] \circ P(S_0,q) \\
PN(A_i,i) &= A_0[N] \circ P(S_0,q+1) \\
&= A_i[1] \circ PN(P(S_0,q),q) \\
&= A_i[1] \circ PN(A_i[2:N],q)
\end{aligned}$$

Putting this all together, in definition (9), we see we have a neat recursive definition of the next permutation, in which the only real operation being performed is swapping two adjacent elements of the string. Furthermore, we see that such a swap occurs exactly once for each permutation.

(9) PN(A,i) = PN(A, q$\cdot$N+ℓ), where $0 \le \ell \le N-1$, $0 \le q < (N-1)!$,

N = 2	=> <u>swap</u>(A,1)
N > 2, q even, $0 \le \ell < N-1$	=> <u>swap</u>(A,N-(ℓ+1))
N > 2, q odd, $0 \le \ell < N-1$	=> <u>swap</u>(A,ℓ+1)
N > 2, q even, ℓ = N-1	=> A[1] o PN(A[2:N],q)
N > 2, q odd, ℓ = N-1	=> PN(A[1:N-1],q) o A[N]

Noting that if $N = 2$, then $(q,\ell) = (0,0)$ or $(0,1)$, we rewrite (9) as follows:

$$
\begin{array}{lll}
 & PN(A,i) = PN(A,q\cdot N+\ell), \text{ where } N \geq 2,\ 0 \leq \ell \leq N-1,\ 0 \leq q < (N-1)! & \\
 & q \text{ even},\ 0 \leq \ell < N-1 & \Rightarrow \underline{swap}(A,N-(\ell+1)) \\
 & q \text{ odd},\ 0 \leq \ell < N-1 & \Rightarrow \underline{swap}(A,\ell+1) \\
(10) & q \text{ even},\ \ell = N-1,\ N=2 & \Rightarrow \underline{swap}(A,1) \\
 & q \text{ even},\ \ell = N-1,\ N>2 & \Rightarrow A[1] \circ PN(A[2:N],q+1) \\
 & q \text{ odd},\ \ell = N-1,\ N>2 & \Rightarrow PN(A[1:N-1],q+1) \circ A[N]
\end{array}
$$

We want to end up with the following kind of PL/I procedure:

```
      PERM: PROCEDURE(A,M);
      /*A(1:M) is an array, M > 1.  Print out all permutations of A*/
(11)    DO  I = 1,M!
            Print array A;
            Change A to the next permutation;
            END;  END PERM;
```

In transforming (10) into this kind of procedure, we will have to do the following:

1. Instead of using a string of elements A we use an array. This requires us to keep track of the lower bound of the array being used during the computation.
2. We have to keep track of the value of I in such a way as to make it easy to compute the corresponding q and ℓ where $I = q\cdot N+\ell$.
3. We will come across a more efficient way of determining when the $N!^{th}$ permutation has been reached, so that we won't have to explicitly count and test I.

4. We want to change the recursion to iteration.

Problem 1 is rather simple, so let us first tackle problem 2. Suppose A with $N = |A| > 1$ contains the i^{th} permutation with $0 \leq i < N!$:

$$i = q_N \cdot N + \ell_N, \quad 0 \leq q_N < N!, \quad 0 \leq \ell_N \leq N-1$$

Looking at equation (8), we see that the string A with the original element A[N] deleted contains its q_N^{th} permutation, and we have

$$q_N = q_{N-1} \cdot (N-1) + \ell_{N-1}, \quad 0 \leq q_N < (N-1)!, \quad 0 \leq \ell_N \leq N-2$$

In fact, we can see recursively that i defines a sequence

$$(12) \quad \begin{aligned} i = q_{N+1} &= q_N \cdot N + \ell_N, & 0 \leq q_N < N!, \quad 0 \leq \ell_N \leq N-1 \\ q_N &= q_{N-1} \cdot (N-1) + \ell_{N-1}, & 0 \leq q_{N-1} < (N-1)!, \quad 0 \leq \ell_{N-1} \leq N-2 \\ q_{N-1} &= q_{N-2} \; (N-2) + \ell_{N-2} & 0 \leq q_{N-2} < (N-2)!, \quad 0 \leq \ell_{N-2} \leq N-3 \\ &\vdots & \\ q_2 &= q_1 (1) + \ell_1, & 0 \leq q_1 < 1!, \quad 0 \leq \ell_1 \leq 1 \end{aligned}$$

and if A contains the $q_{N+1}^{th} = i^{th}$ permutation, then for $2 \leq k \leq N$, the string consisting of A with the appropriate elements deleted contains its q_k^{th} permutation.

We can therefore keep track of the value i_N by instead keeping track of the pairs of (q_k, ℓ_k), $k = 2,\ldots,N$. Furthermore, from definition 10, we see that we need not know the value of q_k, but just whether it is odd or even. Hence our program will use an array QL[2:N] defined as follows: Suppose A contains the i^{th} permutation, then

$$\text{(13)}\quad QL[k] = \begin{cases} \ell_k + 1 & \text{if } q_k \text{ is odd, for } 2 \le k \le N \\ -(k-\ell_k+1)) & \text{if } q_k \text{ is even, for } 2 \le k \le N \end{cases}$$

We now redefine PN (definition 10), (1) as a PL/I-like procedure, (2) in terms of QL, and (3) using array notation instead of thinking of A as a string. The five cases of definition (10) are given in the same order here, but this time they are in terms of array QL and not q and ℓ. Be careful of LB; the string is in A[LB+1:LB+N] and <u>not</u> in A[LB:LB+N-1].

```
PN: PROC(A,LB,N,QL);
  /*A[LB+1:LB+N], N ≥ 2, contains the i_N^th permutation of an   */
  /*original array segment A[LB+1:LB+N], where i  is defined     */
  /*by (12), QL by (13), and 0 ≤ i  < N!.  Change A[LB+1:LB+N]*/
  /*and QL to the next permutation                               */

  case
     1 ≤ -QL[N] ≤ N-1: A:= swap(A,LB-QL(N)); QL(N):= QL[N]+1

     1 ≤ QL[N]  ≤ N-1: A:= swap(A,LB+QL[N]); QL[N]:= QL[N]+1
(14)
     QL[N] = 0, N = 2: QL[N]:= 1; swap(A,LB+1)
     QL[N] = 0, N > 2: QL[N]:= 1; call PN(A,LB+1,N-1,QL)

     QL[N] = N:        QL:= -(N-1); call PN(A,LB,N-1,QL)

  esac
  END PN;
```

To change this into an iterative rather than recursive version is now quite simple, due to the simple nature of the recursion. We also note that the last permutation, takes place when i_N = N!-1, which means that q is even and ℓ_N = N-1, ℓ_{N-1} = N-2,..., ℓ_2=1. Thus we know when the last permutation is expected: When q_2 is even, ℓ_2 = N-1, and N=2. We use this as our stopping condition. Our final PL/I procedure, transformed from (14), is given below.

In considering the transformation from (14) to the program, remember the following. The 5 mutually exclusive tests in (14) have had to be written in sequential form. The two cases $1 \leq$ -QL[N] $\leq$ N-1 and $1 \leq$ QL[N] $\leq$ N-1 have been handled in one, since the process performed in each is similar enough (swapping two elements and adding one to QL[N]).

```
/* PRINT ALL PERMUTATIONS OF A(1:M), WHERE M > 1 */
PERM: PROCEDURE(A, M);
   DECLARE (A(*), M) FIXED DECIMAL;

   DECLARE (QL(2:M),           /* SEE (13) OF PAPER FOR DEFINITION */
            N, LB,
            ABSQLN,            /* ABS(QL(N)) */
            T, I1)             /* USED LOCALLY IN SWAPPING ARRAY VALUES
                  FIXED DECIMAL;

   /* INITIALIZE ARRAY QL TO DENOTE THE 0TH PERMUTATION */
      DO N = 2 TO M;  QL(N) = -(N-1); END;

   DO WHILE(TRUE);
      PRINT_ARRAY_A;

      /* CHANGE A, QL TO NEXT PERMUTATION, STOP PROCEDURE IF LAST ONE
        N = M; LB = 0;
        CALLPN: ABSQLN = ABS(QL(N));
          IF ABSQLN = N THEN
             DO: QL(N) = -(N-1); N=N-1; GOTO CALLPN: END:
          IF ABSQLN = 0 THEN
             DO; IF N = 2
                 THEN DO; /* SWAP(A, LB+1) */
                              T=A(LB+1); A(LB+1)=A(LB+2); A(LB+2)=
                          RETURN: END;
                 ELSE DO: QL(N)=1; LB=LB+1; N=N-1; GOTO CALLPN;EN
               END;
          /* SWAP(A, LB + |QL(N)| )  */
               I1=LB+ABSQLN;  T=A(I1); A(I1)=A(I1+1); A(I1+1)=T;
          QL(N) = QL(N)+1;
      END; END PERM;
```

4. Finding the Mode of an Array A[1:N].

Now let us take a look at a much simpler problem, that of finding the mode (most frequently occurring value) of a sorted array A[1:N]. Thus for the array containing the values 3,3,4,4,4, 5,8,9,10,10, the mode is 4. I had presented the following program segment to do this. The purpose was not so much to describe this algorithm, but to give a simple example of the use of looking at invariant relations of loops:

```
/*Store in M the mode of A[1:N], N ≥ 1, A is already sorted*/
  /*Invariant relation P:  1 ≤ I ≤ N,           */
  /*           M = mode(A[1:I]),                */
  /*          CM = frequency of M in A[1:I]     */
  /*          CI = frequency of A(I) in A[1:I]*/
  I := 1; M := A[1]; CM := 1; CI := 1;
  while I < N do
      begin I := I+1;
            if A[I] ¬= A[I-1] then CI := 1
            else begin CI := CI+1;
                       if CI > CM
                         then begin CM := CI; M := A[I] end
                 end
      end
```

I knew that the loop could be somewhat simplified by using a slightly different invariant relation, and had used this just as an example. But I wasn't prepared for Griffiths' new idea.

After working for awhile, he came up with a recursive definition of the mode of A[1:N] and its frequency in terms of the mode and frequency of the mode of A[1:N-1]. He then manipulated it to arrive at the following definition:

```
Mode(a,n) where a[1:n] is a sorted array
     returns a pair (m,f) where m is the mode and f
     the number of times m occurs in a[1:n]

Mode(a,n]:
     if n = 1 then (a[1],1)
       else  Let (p,q) = mode(a,n-1)
             if a(n) = a(n-q)
               then (a[n],q+1)
               else (p,q)
```

Transformed to get rid of the recursion we get the following where the mode ends up in P and its frequency in Q:

```
P:= A[1]; Q := 1;
for I := 2 TO N do
   begin if A[I] = A[I-Q]
         then begin P:= A[I]; Q:= Q+1 end
   end
```

This uses one less variable than mine, while the loop body is much simpler. The main idea is to note that the mode(A[1:I]) may be different from that of A[1:I-1] whose frequency is Q, only if

A[I] = A[I-1] = A[I-2] = ... = A[I-Q], in which case A[I] = A[I-Q]!

5. Discussion

Again, the use of recursive definitions in this manner is certainly not new, but few programmers use it. These examples, Knuth's example of its use in topological sorting [2], and others should begin to convince programmers that using recursion at least in the design stages and then "simulating" the recursion (as we have done here) is a powerful programming technique.

Bauer [1] feels that programmers introduce variables too early in the programming process, and that variables are just as dangerous as gotos, because they lead to remote connections of data just the way gotos lead to remote connections of statements.

It is certainly true that most programmers do introduce variables without really defining them properly, and a large portion of the errors introduced in a program arise because of this. If, however, we work at the beginning with more "mathematical" definitions, quite often recursive, we will often find the task easier. Mathematical expressions <u>are</u> easier to manipulate correctly than program segments. And, as done in these examples, variables get introduced later in the development process and with more precision and certainity. This does not mean that "debugging" is not necessary. Transformations such as have been shown here are just as prone to errors as proofs in mathematical papers. My final permutation program ran correctly on the second attempt. The <u>only</u> error arose in the transition from equation 8 to equation 9; I inadvertantly switched the two cases q even, ℓ = N-1 and q odd, ℓ = N-1. Lucky for us, we do have the computer to help us catch these errors; mathematicians do not.

The question of documentation arises; what explanation should accompany such a program which arises from using recursion? Actually the permutation algorithm given here is essentially the same as that given in Algorithm 115 by Trotter [3]. However his is completely unintelligible because no accompanying hints - recursive formulation or definition of the variables, etc. - are given. His is completely "unstructured", or at least the structure is well hidden. My final procedure is also unintelligible, unless accompanied by at the least equations 7,8,9,10,12, the definition (13) of QL, and procedure 14. In effect, we must show where the algorithm came from and show all the transformations. While not a formal proof, this is enough to convince any programmer familiar with recursion and its implementation, of the correctness of the procedure. (A proof is just evidence sufficient to produce belief in the truth of something.)

Should we wish to document the program without the recursive definition and the transformations, then we must work harder to make it look structured, to get rid of the gotos, to use conventional loop structures and to write the invariant relations of the loops. Griffith's iterative version of the mode problem is quite easy to describe. On the other hand, the permutation program developed here must be rewritten and redescribed in another fashion if the recursive details and transformations are not to be part of the documentation.

References

[1] Bauer, F.L., "Variables considered harmful", Internal report of a Seminar über Methodik des Programmierens, Abteilung Mathematik, Technische Universitat Munchen, 1974.

[2] Knuth, D.E. and J.L. Szwarcfiter, "A structured program to generate all topological sorting arrangments", Inf. Proc. Letters 2 (1974), 153-157.

[3] Trotter, H.F., "Algorithm 115, PERM", CACM 5 (Aug 62), 434-435.