Permutation Generation Methods*

ROBERT SEDGEWICK

Program in Computer Science and Division of Applied Mathematics
Brown University, Providence, Rhode Island 02912

This paper surveys the numerous methods that have been proposed for permutation enumeration by computer. The various algorithms which have been developed over the years are described in detail, and implemented in a modern ALGOL-like language. All of the algorithms are derived from one simple control structure.

The problems involved with implementing the best of the algorithms on real computers are treated in detail. Assembly-language programs are derived and analyzed fully.

The paper is intended not only as a survey of permutation generation methods, but also as a tutorial on how to compare a number of different algorithms for the same task

Key Words and Phrases: permutations, combinatorial algorithms, code optimization, analysis of algorithms, lexicographic ordering, random permutations, recursion, cyclic rotation.

CR Categories: 3.15, 4.6, 5 25, 5.30.

INTRODUCTION

Over thirty algorithms have been published during the past twenty years for generating by computer all $N!$ permutations of N elements. This problem is a nontrivial example of the use of computers in combinatorial mathematics, and it is interesting to study because a number of different approaches can be compared. Surveys of the field have been published previously in 1960 by D. H. Lehmer [26] and in 1970-71 by R. J. Ord-Smith [29, 30]. A new look at the problem is appropriate at this time because several new algorithms have been proposed in the intervening years.

Permutation generation has a long and distinguished history. It was actually one of the first nontrivial nonnumeric problems to be attacked by computer. In 1956, C. Tompkins wrote a paper [44] describing a number of practical areas where permutation generation was being used to solve problems. Most of the problems that he described are now handled with more sophisticated techniques, but the paper stimulated interest in permutation generation by computer *per se*. The problem is simply stated, but not easily solved, and is often used as an example in programming and correctness. (See, for example, [6]).

The study of the various methods that have been proposed for permutation generation is still very instructive today because together they illustrate nicely the relationship between counting, recursion, and iteration. These are fundamental concepts in computer science, and it is useful to have a rather simple example which illustrates so well the relationships between them. We shall see that algorithms which seem to differ markedly have essentially the same structure when expressed in a modern language and subjected to simple program transformations. Many readers may find it surprising to discover that "top-down" (recursive) and "bottom-up"

* This work was supported by the National Science Foundation Grant No. MCS75-23738

CONTENTS

(iterative) design approaches can lead to the same program.

Permutation generation methods not only illustrate programming issues in high-level (procedural) languages; they also illustrate implementation issues in low-level (assembly) languages. In this paper, we shall try to find the fastest possible way to generate permutations by computer. To do so, we will need to consider some program "optimization" methods (to get good implementations) and some mathematical analyses (to determine which implementation is best). It turns out that on most computers we can generate each permutation at only slightly more than the cost of two *store* instructions.

In dealing with such a problem, we must be aware of the inherent limitations. Without computers, few individuals had the patience to record all 5040 permutations of 7 elements, let alone all 40320 permutations of 8 elements, or all 362880 permutations of 9 elements. Computers help, but not as much as one might think. Table 1 shows the values of $N!$ for $N \leq 17$ along with the time that would be taken by a permutation generation program that produces a new permutation each microsecond. For $N > 25$, the time required is far greater than the age of the earth!

For many practical applications, the sheer magnitude of $N!$ has led to the development of "combinatorial search" procedures which are far more efficient than permutation enumeration. Techniques such as mathematical programming and backtracking are used regularly to solve optimization problems in industrial situations, and have led to the resolution of several hard problems in combinatorial mathematics (notably the four-color problem). Full treatment of these methods would be beyond the scope of this paper – they are mentioned here to emphasize that, in practice, there are usually alternatives to the "brute-force" method of generating permutations. We will see one example of how permutation generation can sometimes be greatly improved with a backtracking technique.

In the few applications that remain where permutation generation is really required, it usually doesn't matter much which generation method is used, since the cost of processing the permutations far

TABLE 1. APPROXIMATE TIME NEEDED TO GENERATE ALL PERMUTATIONS OF N (1 μsec per permutation)

N	$N!$	Time
1	1	
2	2	
3	6	
4	24	
5	120	
6	720	
7	5040	
8	40320	
9	362880	
10	3628800	3 seconds
11	39916800	40 seconds
12	479001600	8 minutes
13	6227020800	2 hours
14	87178291200	1 day
15	1307674368000	2 weeks
16	20922789888000	8 months
17	355689428096000	10 years

exceeds the cost of generating them. For example, to evaluate the performance of an operating system, we might want to try all different permutations of a fixed set of tasks for processing, but most of our time would be spent simulating the processing, not generating the permutations. The same is usually true in the study of combinatorial properties of permutations, or in the analysis of sorting methods. In such applications, it can sometimes be worthwhile to generate "random" permutations to get results for a typical case. We shall examine a few methods for doing so in this paper.

In short, the fastest possible permutation method is of limited importance in practice. There is nearly always a better way to proceed, and if there is not, the problem becomes really hopeless when N is increased only a little.

Nevertheless, permutation generation provides a very instructive exercise in the implementation and analysis of algorithms. The problem has received a great deal of attention in the literature, and the techniques that we learn in the process of carefully comparing these interesting algorithms can later be applied to the perhaps more mundane problems that we face from day to day.

We shall begin with simple algorithms that generate permutations of an array by successively exchanging elements; these algorithms all have a common control structure described in Section 1. We then will study a few older algorithms, including some based on elementary operations other than exchanges, in the framework of this same control structure (Section 2). Finally, we shall treat the issues involved in the implementation, analysis, and "optimization" of the best of the algorithms (Section 3).

1. METHODS BASED ON EXCHANGES

A natural way to permute an array of elements on a computer is to exchange two of its elements. The fastest permutation algorithms operate in this way: All $N!$ permutations of N elements are produced by a sequence of $N!-1$ exchanges. We shall use the notation

```
P[1]:=:P[2]
```

to mean "exchange the contents of array elements P[1] and P[2]". This instruction gives both arrangements of the elements P[1], P[2] (i.e., the arrangement before the exchange and the one after). For $N = 3$, several different sequences of five exchanges can be used to generate all six permutations, for example

```
P[1] =:P[2]
P[2]:=:P[3]
P[1] =:P[2]
P[2]:=:P[3]
P[1]:=:P[2].
```

If the initial contents of P[1] P[2] P[3] are A B C, then these five exchanges will produce the permutations B A C, B C A, C B A, C A B, and A C B.

It will be convenient to work with a more compact representation describing these exchange sequences. We can think of the elements as passing through "permutation networks" which produce all the permutations. The networks are comprised of "exchange modules" such as that shown in Diagram 1 which is itself the

DIAGRAM 1

permutation network for $N = 2$. The network of Diagram 2 implements the exchange sequence given above for $N = 3$. The elements pass from right to left, and a new permutation is available after each exchange. Of course, we must be sure that the internal permutations generated are distinct. For $N = 3$ there are $3^5 = 243$ possible networks with five exchange modules, but only the twelve shown in Fig. 1 are "legal" (produce sequences of distinct permutations). We shall most often represent networks as in Fig. 1, namely drawn vertically, with elements passing from top to bottom, and with the permutation se-

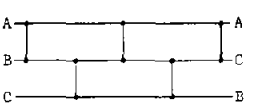

DIAGRAM 2.

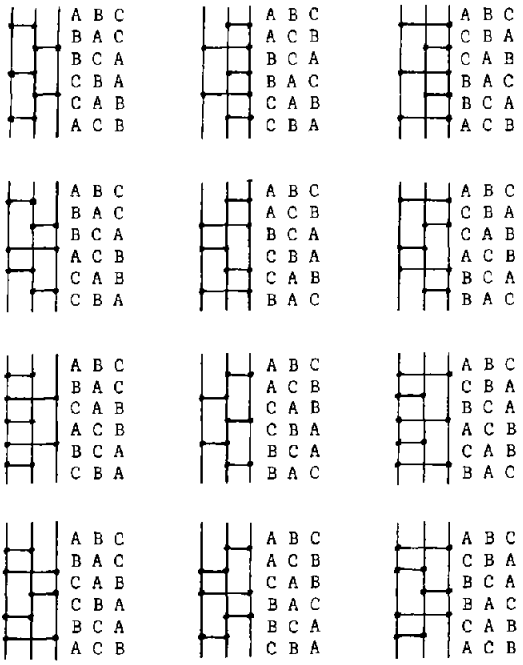

FIGURE 1. Legal permutation networks for three elements.

quences that are generated explicitly written out on the right.

It is easy to see that for larger N there will be large numbers of legal networks. The methods that we shall now examine will show how to systematically construct networks for arbitrary N. Of course, we are most interested in networks with a sufficiently simple structure that their exchange sequences can be conveniently implemented on a computer.

Recursive Methods

We begin by studying a class of permutation generation methods that are very simple when expressed as recursive programs. To generate all permutations of P[1],···,P[N], we repeat N times the step: "first generate all permutations of P[1],···,P[N-1], then exchange P[N] with one of the elements P[1],···,P[N-1]". As this is repeated, a new value is put into P[N] each time. The various methods differ in their approaches to filling P[N] with the N original elements.

The first and seventh networks in Fig. 1 operate according to this discipline. Recursively, we can build up networks for four elements from one of these. For example, using four copies of the first network in Fig. 1, we can build a network for $N = 4$, as shown in Diagram 3. This network fills P[4] with the values D, C, B, A in decreasing alphabetic order (and we could clearly build many similar networks which fill P[4] with the values in other orders).

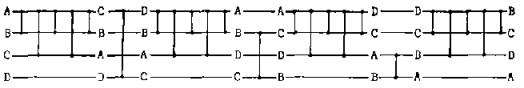

DIAGRAM 3.

The corresponding network for five elements, shown in Diagram 4, is more complicated. (The empty boxes denote the network of Diagram 3 for four elements). To get the desired decreasing sequence in P[5], we must exchange it successively with P[3], P[1], P[3], P[1] in-between generating all permutations of P[1],···,P[4].

In general, we can generate all permutations of N elements with the following recursive procedure:

Algorithm 1.

```
procedure permutations(N);
    begin c:=1;
        loop:
            if N>2 then permutations(N-1)
            endif;
        while c<N:
            P[B[N,c]]:=:P[N];
            c:=c+1
        repeat
    end;
```

This program uses the looping control construct **loop** ··· **while** ··· **repeat** which is described by D. E. Knuth [23]. Statements between **loop** and **repeat** are iterated: when the **while** condition fails, the loop is exited. If the **while** were placed immediately following the **loop**, then the statement would be like a normal ALGOL **while**. In fact, Algorithm 1 might be implemented with a simpler construct like **for**

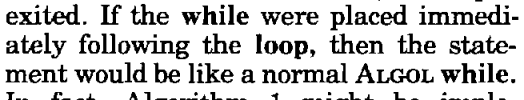

DIAGRAM 4.

c:=1 **until** N **do** ··· were it not for the need to test the control counter c within the loop. The array B[N,c] is an index table which tells where the desired value of P[N] is after P[1],···,P[N-1] have been run through all permutations for the cth time.

We still need to specify how to compute B[N,c]. For each value of N we could specify any one of $(N-1)!$ sequences in which to fill P[N], so there are a total of $(N-1)!(N-2)!(N-3)!\cdots 3!2!1!$ different tables B[N,c] which will cause Algorithm 1 to properly generate all $N!$ permutations of P[1],···,P[N].

One possibility is to precompute B[N,c] by hand (since we know that N is small), continuing as in the example above. If we adopt the rule that P[N] should be filled with elements in decreasing order of their original index, then the network in Diagram 4 tells us that B[5,c] should be 1,3,1,3 for c = 1,2,3,4. For N = 6 we proceed in the same way: if we start with A B C D E F, then the first N = 5 subnetwork leaves the elements in the order C D E B A F, so that B[6,1] must be 3 to get the E into P[6], leaving C D F B A E. The second N = 5 subnetwork then leaves F B A D C E, so that B[6,2] must be 4 to get the D into P[6], etc. Table 2 is the full table for N ≤ 12 generated this way; we could generate permutations with Algorithm 1 by storing these $N(N-1)$ indices.

There is no reason to insist that P[N] should be filled with elements in decreasing order. We could proceed as above to build a table which fills P[N] in any order we choose. One reason for doing so would be to try to avoid having to store the table: there are at least two known versions of this method in which the indices can be easily computed and it is not necessary to precompute the index table.

TABLE 2. INDEX TABLE $B[N, c]$ FOR ALGORITHM 1

N												
	2	1										
	3	1	1									
	4	1	2	3								
	5	3	1	3	1							
	6	3	4	3	2	3						
	7	5	3	1	5	3	1					
	8	5	2	7	2	1	2	3				
	9	7	1	5	5	3	3	7	1			
	10	7	8	1	6	5	4	9	2	3		
	11	9	7	5	3	1	9	7	5	3	1	
	12	9	6	3	10	9	4	3	8	9	2	3

The first of these methods was one of the earliest permutation generation algorithms to be published, by M. B. Wells in 1960 [47]. As modified by J. Boothroyd in 1965 [1, 2], Wells' algorithm amounts to using

$$B[N,c] = \begin{cases} N-c & \text{if } N \text{ is even and } c > 2 \\ N-1 & \text{otherwise,} \end{cases}$$

or, in Algorithm 1, replacing P[B[N,c]]:=:P[N] by

if (N even) **and** (c>2)
then P[N]:=:P[$N-c$]
else P[N]:=:P[$N-1$] **endif**

It is rather remarkable that such a simple method should work properly. Wells gives a complete formal proof in his paper, but many readers may be content to check the method for all practical values of N by constructing the networks as shown in the example above. The complete networks for N = 2,3,4 are shown in Fig. 2.

In a short paper that has gone virtually unnoticed, B.R. Heap [16] pointed out several of the ideas above and described a method even simpler than Wells'. (It is not clear whether Heap was influenced by Wells or Boothroyd, since he gives no references.) Heap's method is to use

$$B(N,c) = \begin{cases} 1 & \text{if } N \text{ is odd} \\ c & \text{if } N \text{ is even,} \end{cases}$$

or, in Algorithm 1, to replace P[B[N,c]]:=:P[N] by

if N *odd* **then** P[N]:=:P[1] **else** P[N]:=:P[c] **endif**

Heap gave no formal proof that his method works, but a proof similar to Wells' will show that the method works for all N. (The reader may find it instructive to verify that the method works for practical values of N (as Heap did) by proceeding as we did when constructing the index table above.) Figure 3 shows that the networks for N = 2,3,4 are the same as for Algorithm 1 with the precomputed index table, but that the network for N = 5, shown in Diagram 5, differs. (The empty boxes denote the network for N = 4 from Fig. 3.)

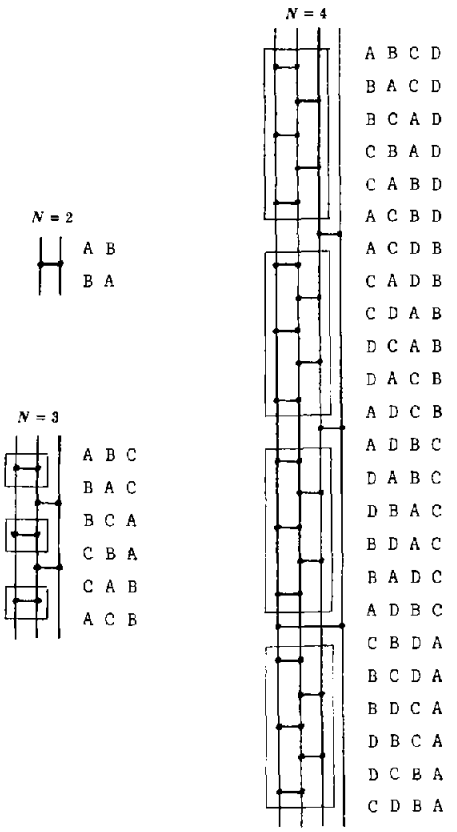

FIGURE 2. Wells' algorithm for $N = 2,3,4$.

Neither Wells nor Heap gave recursive formulations of their methods, although Boothroyd [1] later gave a recursive implementation of his version of Wells' method. Although Wells and Heap undoubtedly arrived at their nonrecursive programs directly, it is instructive here to derive a nonrecursive version of Algorithm 1 by systematically removing the recursion.

The standard method for implementing a recursive procedure is to maintain a stack with the parameters and local variables for each invocation of the procedure. The simple structure of Algorithm 1 makes it more convenient to maintain an array $c[1],\cdots,c[N]$, where $c[i]$ is the value of c for the invocation *permutations*(i). Then by decrementing i on a *call* and incrementing i on *return*, we ensure that $c[i]$ always refers to the proper value of c. Since there is only one recursive call, transfer of control is implemented by jumping to the beginning on *call* and jumping to the place following the call on *return*. The following program results directly when we remove the recursion and the loops from Algorithm 1:

```
        ι:=N;
begin:  c[ι]:=1;
loop:   if ι>2 then ι:=ι-1; go to begin endif;
return: if c[ι]≥ι then go to exit endif;
        P[B[c[ι]]] =:P[ι];
        c[ι]:=c[ι]+1;
        go to loop;
exit:   if ι<N then ι:=ι+1; go to return endif;
```

This program can be simplified by combining the instructions at *begin* and *loop* into a single loop, and by replacing the single **go to** *exit* with the code at *exit*:

```
        ι:=N+1;
loop:   loop while ι>2: ι.=ι-1; c[ι]:=1 repeat;
return. if c[ι]≥ι
        then if ι<N then ι:=ι+1;
                         go to return endif;
        else P[B[c[ι]]] =:P[ι];
             c[ι]:=c[ι]+1;
             go to loop;
        endif;
```

The program can be transformed further if we observe that, after $c[N],\cdots,c[2]$ are all set to 1 (this is the first thing that the program does), we can do the assignment *c[i]:=1* before *ι:=i+1* rather than after *i:=i-1* without affecting the rest of the program. But this means that the loop does nothing but set i to 2 and it can be eliminated (except for the initialization), as in this version:

```
        ι:=N;
        loop: c[ι]:=1 while ι>2. ι:=ι-1 repeat;
return· if c[ι]≥ι
        then if ι<N then c[ι]:=1; ι:=ι+1;
                         go to return endif;
        else P[B[c[ι]]]:=.P[ι];
             c[ι]:=c[ι]+1;
             ι:=2;
             go to return;
        endif;
```

Finally, since the two **go to**'s in this program refer to the same label, they can be replaced with a single **loop**$\cdots$**repeat**. The formulation

```
ι:=N; loop: c[ι]:=1 while ι>2: ι:=ι-1 repeat;
loop:
      if c[ι]<ι then P[B[c[ι]]]:=:P[ι];
                     c[ι]:=c[ι]+1, ι:=2;
                else c[ι]:=1; ι:=ι+1;
      endif;
while ι≤N repeat;
```

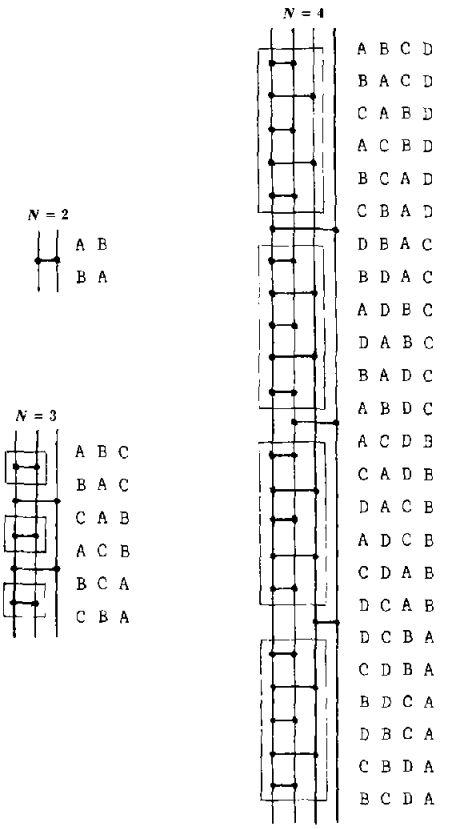

FIGURE 3. Heap's algorithm for $N = 2,3,4$.

is attractive because it is symmetric: each time through the loop, either *c[i]* is initialized and *i* incremented, or *c[i]* is incremented and *i* initialized. (*Note*: in a sense, we have removed too many **go to**'s, since now the program makes a redundant test $i \leq N$ after setting *i*:=2 in the **then** clause. This can be avoided in assembly language, as shown in Section 3, or it could be handled with an "event variable" as described in [24].) We shall examine the structure of this program in detail later.

The programs above merely generate all permutations of P[1],···,P[*N*]; in order to do anything useful, we need to process each permutation in some way. The processing might involve anything from simple counting to a complex simulation. Normally, this is done by turning the permutation generation program into a procedure which returns a new permutation each time it is called. A main program is then written to call this procedure *N*! times, and process each permutation. (In this form, the permutation generator can be kept as a library subprogram.) A more efficient way to proceed is to recognize that the permutation generation procedure is really the "main program" and that each permutation should be processed as it is generated. To indicate this clearly in our programs, we shall assume a macro called *process* which is to be invoked each time a new permutation is ready. In the nonrecursive version of Algorithm 1 above, if we put a call to *process* at the beginning and another call to *process* after the exchange statement, then *process* will be executed *N*! times, once for each permutation. From now on, we will explicitly include such calls to *process* in all of our programs.

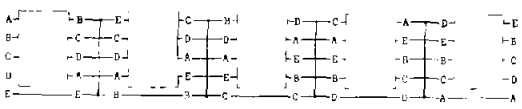

DIAGRAM 5.

The same transformations that we applied to Algorithm 1 yield this nonrecursive version of Heap's method for generating and processing all permutations of P[1],···,P[*N*]:

Algorithm 2 (Heap)

```
i:=N; loop: c[i]:=1 while i>2: i:=i-1 repeat;
process;
loop
    if c[i]<i
    then if i odd then k:=1 else k:=c[i] endif;
        P[i]:=:P[k];
        c[i]:=c[i]+1; i:=2;
        process,
    else c[i]:=1; i:=i+1
    endif;
while i≤N repeat;
```

This can be a most efficient algorithm when implemented properly. In Section 3 we examine further improvements to this algorithm and its implementation.

Adjacent Exchanges

Perhaps the most prominent permutation enumeration algorithm was formulated in 1962 by S. M. Johnson [20] and H. F. Trotter [45], apparently independently. They discovered that it was possible to generate all *N*! permutations of *N* elements with *N*!−1 exchanges of *adjacent* elements.

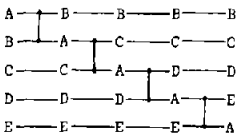

DIAGRAM 6

The method is based on the natural idea that for every permutation of $N-1$ elements we can generate N permutations of N elements by inserting the new element into all possible positions. For example, for five elements, the first four exchange modules in the permutation network are as shown in Diagram 6. The next exchange is P[1]:=:P[2], which produces a new permutation of the elements originally in P[2], P[3], P[4], P[5] (and which are now in P[1], P[2], P[3], P[4]). Following this exchange, we bring A back in the other direction, as illustrated in Diagram 7. Now we exchange P[3]:=:P[4] to produce the next permutation of the last four elements, and continue in this manner until all 4! permutations of the elements originally in P[2], P[3], P[4], P[5] have been generated. The network makes five new permutations of the five elements for each of these (by putting the element originally in P[1] in all possible positions), so that it generates a total of 5! permutations.

Generalizing the description in the last paragraph, we can inductively build the network for N elements by taking the network for $N-1$ elements and inserting chains of $N-1$ exchange modules (to sweep the first element back and forth) in each space between exchange modules. The main complication is that the subnetwork for $N-1$ elements has to shift back and forth between the first $N-1$ lines and the last $N-1$ lines in between sweeps. Figure 4 shows the networks for N = 2,3,4. The modules in boxes identify the subnetwork: if, in the network for N, we connect the output lines of one box to the input lines of the next, we get the network for $N-1$.

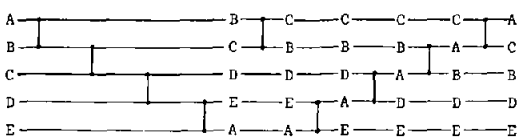

DIAGRAM 7.

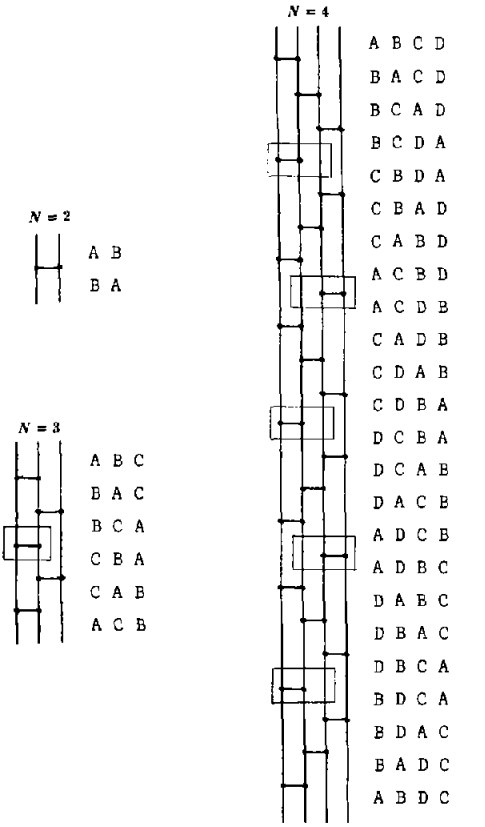

FIGURE 4. Johnson-Trotter algorithm for N = 2, 3, 4.

Continuing the example above, we get the full network for N = 5 shown in Figure 5. By connecting the boxes in this network, we get the network for N = 4.

To develop a program to exchange according to these networks, we could work down from a recursive formulation as in the preceding section, but instead we shall take a bottom-up approach. To begin, imagine that each exchange module is labelled with the number of the network in which it first appears. Thus, for N = 2 the module would be numbered 2; for N = 3 the five modules would be labelled 3 3 2 3 3; for N = 4 the 23 modules are numbered

4 4 4 3 4 4 4 3 4 4 4 2 4 4 4 3 4 4 4 3 4 4 4;

for N = 5 we insert 5 5 5 5 between the numbers above, etc. To write a program to generate this sequence, we keep a set of incrementing counters $c[i]$, $2 \le i \le N$, which are all initially 1 and which satisfy

FIGURE 5 Johnson-Trotter algorithm for $N = 5$.

$1 \leq c[i] \leq i$. We find the highest index i whose counter is not exhausted (not yet equal to i), output it, increment its counter, and reset the counters of the larger indices:

```
i:=1; loop while i≤N: i:=i+1; c[i]:=1 repeat;
c[1]:=0;
loop:
      i:=N;
      loop while c[i]=i: c[i]:=1; i:=i-1 repeat;
while i>1:
            comment exchange module is on level i;
      c[i]:=c[i]+1
repeat;
```

When i becomes 1, the process is completed—the statement $c[1] = 0$ terminates the inner loop in this case. (Again, there are simple alternatives to this with "event variables" [24], or in assembly language.)

Now, suppose that we have a Boolean variable $d[N]$ which is **true** if the original P[1] is travelling from P[1] down to P[N] and **false** if it is travelling from P[N] up to P[1]. Then, when $i = N$ we can replace the **comment** in the above program by

```
if d[N] then k:=c[N] else k:=N-c[N] endif;
P[k]:=:P[k+1];
```

This will take care of all of the exchanges on level N. Similarly, we can proceed by introducing a Boolean $d[N-1]$ for level $N-1$, etc., but we must cope with the fact that the elements originally in P[2], $\cdots$, P[N] switch between those locations and P[1], $\cdots$, P[$N-1$]. This is handled by including an offset x which is incremented by 1 each time a $d[i]$ switches from **false** to **true**. This leads to:

Algorithm 3 (Johnson-Trotter)

```
i:=1;
loop while i<N: i:=i+1; c[i]:=1;
                              d[i]:= true; repeat;
c[1]:=0;
process;
loop:
      i:=N; x:=0;
      loop while c[i]=i:
            if not d[i] then x:=x+1 endif;
            d[i]:= not d[i]; c[i]:=1; i.=i-1;
      repeat;
while i>1:
      if d[i] then k:=c[i]+x
            else k:=i-c[i]+x endif;
      P[k]:=:P[k+1];
      process;
      c[i]:=c[i]+1;
repeat;
```

Although Johnson did not present the algorithm in a programming language, he did give a very precise formulation from which the above program can be derived. Trotter gave an ALGOL formulation which is similar to Algorithm 3. We shall examine alternative methods of implementing this algorithm later.

An argument which is often advanced in favor of the Johnson-Trotter algorithm is that, since it always exchanges adjacent elements, the *process* procedure might be simpler for some applications. It might be possible to calculate the incremental effect of exchanging two elements rather than reprocessing the entire permutation. (This observation could also apply to Algorithms 1 and 2, but the cases when they exchange nonadjacent elements would have to be handled differently.)

The Johnson-Trotter algorithm is often inefficiently formulated [5, 10, 12] because it can be easily described in terms of the values of elements being permuted, rather than their positions. If P[1], $\cdots$, P[N] are originally the integers $1, \cdots, N$, then we might try to avoid maintaining the offset x by noting that each exchange simply involves the smallest integer whose count is not yet exhausted. Inefficient implementations involve actually searching for this smallest integer [5] or maintaining the inverse permutation in order to find it [10]. Both of these are far less efficient than the simple offset method of maintaining the indices of the elements to be exchanged given by Johnson and Trotter, as in Algorithm 3.

Factorial Counting

A careful reader may have become suspicious about similarities between Algo-

rithms 2 and 3. The similarities become striking when we consider an alternate implementation of the Johnson-Trotter method:

Algorithm 3a (Alternate Johnson-Trotter)

```
ι:=N;
loop: c[ι]:=1; d[ι]:=true; while ι>1: ι.=ι−1 repeat;
process,
loop:
        if c[ι]<N+1−ι
        then if d[ι] then k.=c[ι]+x
                     else k:=N+1−ι−c[ι]+x endif;
        P[k]:=:P[k+1];
        process;
        c[ι]:=c[ι]+1; ι:=1; x:=0;
        else if not d[ι] then x:=x+1 endif;
             c[ι]:=1; ι:=ι+1; d[ι]:= not d[ι];
        endif;
while ι≤N repeat;
```

This program is the result of two simple transformations on Algorithm 3. First, change i to $N+1-\iota$ everywhere and redefine the c and d arrays so that $c[N+1-\iota]$, $d[N+1-i]$ in Algorithm 3 are the same as $c[i]$, $d[i]$ in Algorithm 3a. (Thus a reference to $c[i]$ in Algorithm 3 becomes $c[N+1-i]$ when i is changed to $N+1-i$, which becomes $c[i]$ in Algorithm 3a.) Second, rearrange the control structure around a single outer loop. The condition $c[i] < N+1-i$ in Algorithm 3a is equivalent to the condition $c[i] < i$ in Algorithm 3, and both programs perform the exchange and process the permutation in this case. When the counter is exhausted ($c[i] = N+1-\iota$ in Algorithm 3a; $c[i] = i$ in Algorithm 3), both programs fix the offset, reset the counter, switch the direction, and move up a level.

If we ignore statements involving P, k and d, we find that this version of the Johnson-Trotter algorithm is identical to Heap's method, except that Algorithm 3a compares $c[i]$ with $N+1-i$ and Algorithm 2 compares it with i. (Notice that Algorithm 2 still works properly if in both its occurrences *2* is replaced by *1*.)

To appreciate this similarity more fully, let us consider the problem of writing a program to generate all N-digit decimal numbers: to "count" from 0 to $99\cdots9 = 10^N-1$. The algorithm that we learn in grade school is to increment the right-most digit which is not 9 and change all the nines to its right to zeros. If the digits are stored in reverse order in the array $c[N],c[N-1],\cdots,c[2],c[1]$ (according to the way in which we customarily write numbers) we get the program

```
ι:=N, loop c[ι]:=0 while ι>1 ι:=ι−1 repeat;
loop:
        if c[ι]<9 then c[ι]:=c[ι]+1; ι =1
                  else c[ι]:=0; ι =ι+1
        endif;
while ι≤N repeat;
```

From this program, we see that our permutation generation algorithms are controlled by this simple counting process, but in a mixed-radix number system. Where in ordinary counting the digits satisfy $0 \le c[i] \le 9$, in Algorithm 2 they satisfy $1 \le c[i] \le i$ and in Algorithm 3a they satisfy $1 \le c[i] \le N-i+1$. Figure 6 shows the values of $c[1],\cdots,c[N]$ *when process* is encountered in Algorithms 2 and 3a for $N = 2,3,4$.

Virtually all of the permutation generation algorithms that have been proposed are based on such "factorial counting" schemes. Although they appear in the literature in a variety of disguises, they all have the same control structure as the elementary counting program above. We have called methods like Algorithm 2 *recursive* because they generate all sequences of $c[1],\cdots,c[i-1]$ in-between increments of $c[i]$ for all i; we shall call methods like Algorithm 3 *iterative* because they iterate $c[i]$ through all its values in-between increments of $c[i+1],\cdots,c[N]$.

Loopless Algorithms

An idea that has attracted a good deal of attention recently is that the Johnson-Trotter algorithm might be improved by removing the inner loop from Algorithm 3. This idea was introduced by G. Ehrlich [10, 11], and the implementation was refined by N. Dershowitz [5]. The method is also described in some detail by S. Even [12].

Ehrlich's original implementation was complex, but it is based on a few standard programming techniques. The inner loop

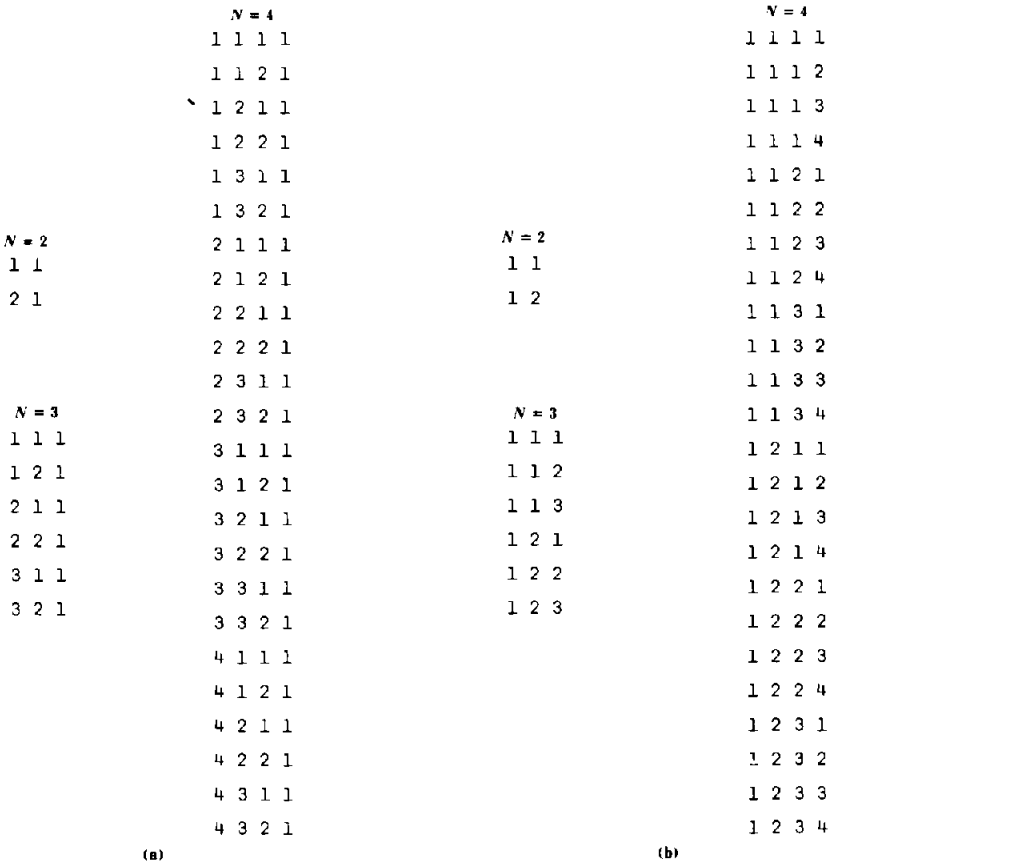

FIGURE 6. Factorial counting: $c[N], \cdots, c[1]$. (a) Using Algorithm 2 (recursive). (b) Using Algorithm 3a (iterative).

in Algorithm 3 has three main purposes: to find the highest index whose counter is not exhausted, to reset the counters at the larger indices, and to compute the offset x. The first purpose can be served by maintaining an array *s[i]* which tells what the next value of i should be when *c[i]* is exhausted: normally $s[i] = i-1$, but when *c[i]* reaches its limit we set *s[i+1]* to *s[i]*. To reset the other counters, we proceed as we did when removing the recursion from Algorithm 1 and reset them just after they are incremented, rather than waiting until they are needed. Finally, rather than computing a "global" offset x, we can maintain an array *x[i]* giving the current offset at level i: when *d[i]* switches from **false** to **true**, we increment *x[s[i]]*. These changes allow us to replace the inner "**loop**$\cdots$**repeat**" in Algorithm 3 by an "**if**$\cdots$**endif**".

Algorithm 3b (Loopless Johnson-Trotter)

```
ι:=0;
loop while ι<N: i:=ι+1; c[ι]:=1; d[ι]:=true;
                s[i]:=ι−1; x[i]:=0 repeat;
process;
loop:
    s[N+1].=N; x[ι]:=0;
    if c[ι]=i then
        if not d[i]
            then x[s[ι]]:=x[s[ι]]+1; endif;
        d[ι]:=not d[ι]; c[i]:=1;
        s[ι+1]:=s[i]; s[i]:=ι−1;
    endif;
    ι:=s[N+1];
while ι>1:
    if d[ι] then k:=c[ι]+x[ι]
        else k:=ι−c[ι]+x[i] endif;
    P[k]:=:P[k+1];
    process;
    c[ι]:=c[ι]+1;
repeat;
```

This algorithm differs from those described in [10, 11, 12], which are based on

the less efficient implementations of the Johnson-Trotter algorithm mentioned above. The loopless formulation is purported to be an improvement because each iteration of the main loop is guaranteed to produce a new permutation in a fixed number of steps.

However, when organized in this form, the unfortunate fact becomes apparent that the loopless Algorithm 3b is slower than the normal Algorithm 3. Loopfree implementation is not an improvement at all! This can be shown with very little analysis because of the similar structure of the algorithms. If, for example, we were to count the total number of times the statement $c[i]:=1$ is executed when each algorithm generates all $N!$ permutations, we would find that the answer would be exactly the same for the two algorithms. The loopless algorithm does not eliminate any such assignments; it just rearranges their order of execution. But this simple fact means that Algorithm 3b must be slower than Algorithm 3, because it not only has to execute all of the same instructions the same number of times, but it also suffers the overhead of maintaining the x and s arrays.

We have become accustomed to the idea that it is undesirable to have programs with loops that could iterate N times, but this is simply not the case with the Johnson-Trotter method. In fact, the loop iterates N times only once out of the $N!$ times that it is executed. Most often ($N-1$ out of every N times) it iterates only once. If N were very large it would be conceivable that the very few occasions that the loop iterates many times might be inconvenient, but since we know that N is small, there seems to be no advantage whatsoever to the loopless algorithm.

Ehrlich [10] found his algorithm to run "twice as fast" as competing algorithms, but this is apparently due entirely to a simple coding technique (described in Section 3) which he applied to his algorithm and not to the others.

Another Iterative Method

In 1976, F. M. Ives [19] published an exchange-based method like the Johnson-Trotter method which *does* represent an improvement. For this method, we build

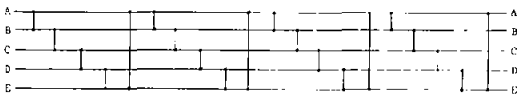

DIAGRAM 8

up the network for N elements from the network for $N-2$ elements. We begin in the same way as in the Johnson-Trotter method. For $N = 5$, the first four exchanges are as shown in Diagram 6. But now the next exchange is P[1]:=:P[5], which not only produces a new permutation of P[1],···,P[4], but also puts P[5] back into its original position. We can perform exactly these five exchanges four more times, until, as shown in Diagram 8, we get back to the original configuration.

At this point, P[1],···,P[4] have been rotated through four permutations, so that we have taken care of the case $N = 4$. If we (inductively) permute three of these elements (Ives suggests the middle three) then the 20 exchanges above will give us 20 new permutations, and so forth. (We shall later see a method which makes exclusive use of this idea that all permutations of N elements can be generated by rotating and then generating all permutations of $N-1$ elements.) Figure 7 shows the networks for $N = 2,3,4$; the full network for $N = 5$ is shown in Fig. 8. As before, if we connect the boxes in the network for N, we get the network for $N-2$. Note that the exchanges immediately preceding the boxes are redundant in that they do not produce new permutations. (The redundant permutations are identified by parentheses in Fig. 7.) However, there are relatively few of these and they are a small price to pay for the lower overhead incurred by this method.

In the example above, we knew that it was time to drop down a level and permute the middle three elements when all of the original elements (but specifically P[1] and P[5]) were back in position. If the elements being permuted are all distinct, we can test for this condition by intially saving the values of P[1],···,P[N] in another array Q[1],···,Q[N]:

Algorithm 4 (Ives)

$\iota:=N$;
loop: $c[\iota]:=\iota$; Q[ι].=P[ι], **while** $\iota<1$. $\iota:=\iota-1$ **repeat**;
process,

```
loop.
    if c[ι]<N+1−ι
    then P[c[ι]]:=:P[c[ι]+1]
        c[ι]:=c[ι]+1; ι:=1;
        process;
     else P[ι].=:P[N+1−ι];
        c[ι]:=ι;
        if P[N+1−ι]=Q[N+1−ι] then ι:=ι+1
                              else ι:=1;
                                   process
        endif,
    endif,
while ι<N+1−ι repeat,
```

This program is very similar to Algorithms 2 and 3a, but it doesn't fall immediately into the "factorial counting" schemes of these programs, because only half of the counters are used. However, we can use counters rather than the test $P[N+1-i] = Q[N+1-i]$, since this test always succeeds after exactly $i-1$ sweeps. We are immediately led to the implementation:

Algorithm 4a (Alternate Ives)

```
ι =N; loop: c[ι]:=1; while ι>1: ι:=ι−1 repeat,
process;
loop
    if c[ι]<N+1−ι
    then if ι odd then P[c[ι]]:=:P[c[ι]+1]
                  else P[ι]:=:P[N+1−ι] endif,
        c[ι]:=c[ι]+1, ι:=1;
        process;
    else c[ι]:=1; ι:=ι+1;
    endif,
while ι≤N repeat;
```

This method does not require that the elements being permuted be distinct, but it is slightly less efficient than Algorithm 4 because more counting has to be done.

Ives' algorithm is more efficient than the Johnson-Trotter method (compare Algorithm 4a with Algorithm 3a) since it does not have to maintain the array d or offset x. The alternate implementation bears a remarkable resemblance to Heap's method (Algorithm 2). Both of these algorithms do little more than factorial counting. We shall compare them in Section 3.

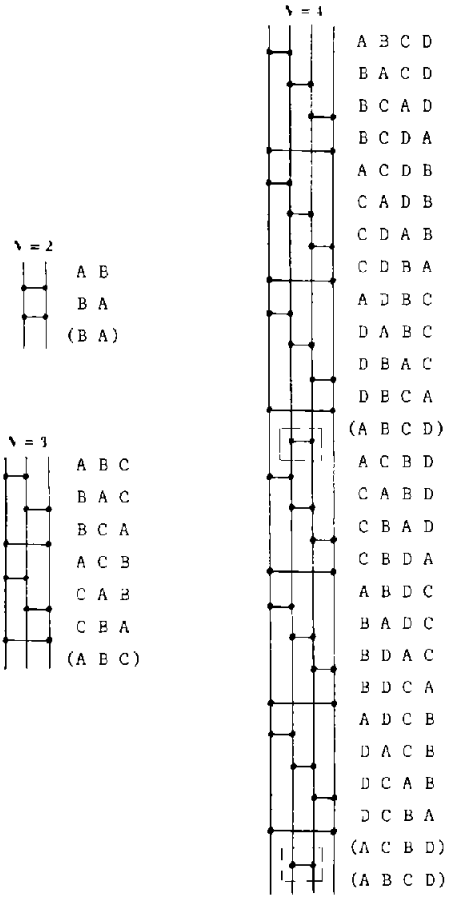

FIGURE 7 Ives' algorithm for $N = 2,3,4$

2. OTHER TYPES OF ALGORITHMS

In this section we consider a variety of algorithms which are *not* based on simple exchanges between elements. These algorithms generally take longer to produce all permutations than the best of the methods already described, but they are worthy of study for several reasons. For example, in some situations it may not be necessary to generate all permutations, but only some "random" ones. Other algorithms may be of practical interest because they are based on elementary operations which could be as efficient as exchanges on some computers. Also, we consider algorithms that generate the permutations in a particular order which is of interest. All of the algorithms can be cast in terms of the basic

FIGURE 8 Ives' algorithm for $N = 5$

"factorial counting" control structure described above.

Nested Cycling

As we saw when we examined Ives' algorithm, N different permutations of P[1],···,P[N] can be obtained by rotating the array. In this section, we examine permutation generation methods which are based solely on this operation. We assume that we have a primitive operation

rotate(i)

which does a cyclic left-rotation of the elements P[1],···,P[i]. In other words, *rotate(i)* is equivalent to

```
t:=P[1]; k:=2;
loop while k≤i: P[k−1]:=P[k] repeat;
P[i]:=t;
```

The network notation for *rotate(5)* is given by Diagram 9. This type of operation can be performed very efficiently on some computers. Of course, it will generally not be more efficient than an exchange, since *rotate(2)* is equivalent to P[1]:=:P[2].

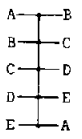

DIAGRAM 9.

The most straightforward way to make use of such an operation is a direct recursive implementation like Algorithm 1:

```
procedure permutations (N);
    begin c:=1;
        loop:
            if N>2 then permutations(N−1)
            endif;
            rotate(N);
        while c<N:
            process;
            c:=c+1
        repeat;
    end;
```

When the recursion is removed from this program in the way that removed the recursion from Algorithm 1, we get an old algorithm which was discovered by C. Tompkins and L. J. Paige in 1956 [44]:

Algorithm 5 (Tompkins-Paige)

```
i:=N; loop: c[i]=1 while i>2: i:=i−1 repeat;
process;
loop:
    rotate(i)
    if c[i]<i then c[i]:=c[i]+1; i:=2;
                   process;
              else c[i]:=1; i:=i+1
    endif;
while i≤N repeat;
```

This is nothing more than a simple counting program with rotation added. The rotation networks and permutation sequences generated by this algorithm are given in Fig. 9. As in Fig. 7, the parenthesized permutations are redundant permutations produced in the course of the computation, which are not passed through the *process* macro. The algorithm is not as inefficient as it may seem, because most of the rotations are short, but it clearly will not compete with the algorithms in Section 1. This method apparently represents the earliest attempt to get a real computer to generate permutations as quickly as possible. An ALGOL implementation was given by Peck and Schrack [33].

An interesting feature of the Tompkins-Paige method, and the reason it generates so many redundant sequences, is that the recursive procedure restores the permutation to the order it had upon entry. Programs that work this way are called backtrack programs [26, 27, 46, 48]. We can easily apply the same idea to exchange methods like Algorithm 1. For example:

```
procedure permutations(N);
    begin c:=1;
        loop:
            P[N]:=:P[c];
            if N>2 then permutations(N−1)
                   else process endif;
            P[c]:=:P[N];
        while c<N:
            c:=c+1
        repeat;
    end;
```

A procedure like this was given by C. T. Fike [13], who also gave a nonrecursive version [37] which is similar to a program developed independently by S. Pleszczyński [35]. These programs are clearly less efficient than the methods of Section 1, which have the same control structure but require many fewer exchanges.

Tompkins was careful to point out that it is often possible easily to achieve great savings with this type of procedure. Often,

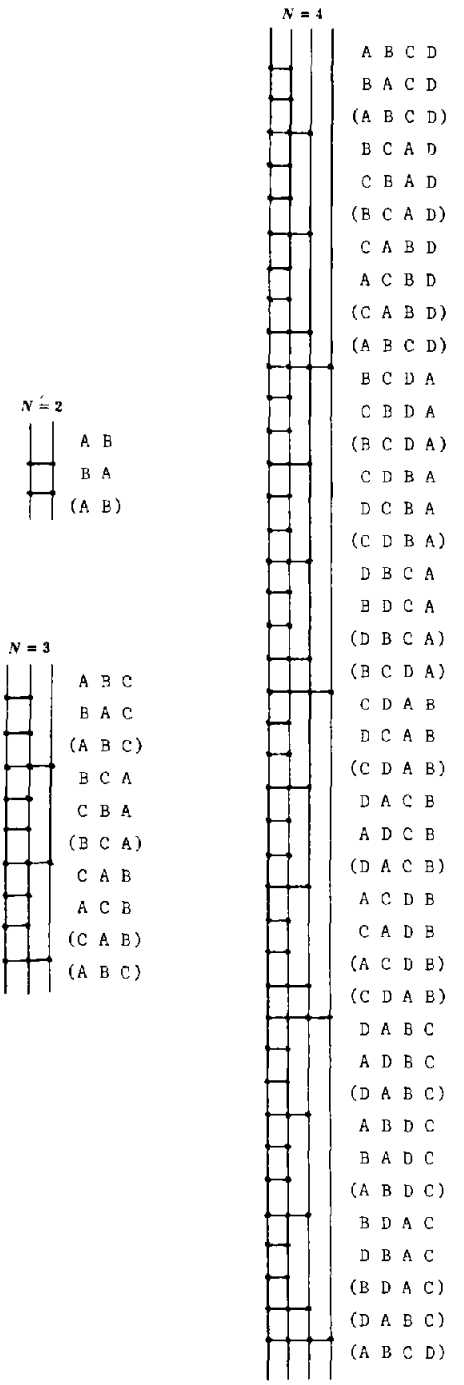

FIGURE 9. Tompkins-Paige algorithm for $N = 2, 3, 4$.

process involves selecting a small set of permutations satisfying some simple criteria. In certain cases, once a permutation is found not to satisfy the criteria, then the permutations generated by permuting its initial elements will not satisfy the criteria either, so the exchanges and the call *permutations(N−1)* can be skipped. Thus large numbers of permutations never need be generated. A good backtracking program will eliminate nearly all of the processing (see [44] for a good example), and such methods are of great importance in many practical applications.

Cycling is a powerful operation, and we should expect to find some other methods which use it to advantage. In fact, Tompkins' original paper [44] gives a general proof from which several methods can be constructed. Remarkably, we can take Algorithm 5 and switch to the counter system upon which the iterative algorithms in Section 2 were based:

```
ι:=N; loop: c[ι]:=1 while ι>1. ι:=ι−1 repeat;
process;
loop:
        rotate(N+1−ι);
        if c[ι]<N+1−ι then c[ι]:=c[ι]+1; ι:=1;
                                process;
                        else  c[ι]:=1; ι:=ι+1
        endif;
while ι≤N repeat,
```

Although fewer redundant permutations are generated, longer rotations are involved, and this method is less efficient than Algorithm 5. However, this method does lend itself to a significant simplification, similar to the one Ives used. The condition $c[i] = N+1-\iota$ merely indicates that the elements in P[1],P[2],$\cdots$, P[$N+1-i$] have undergone a full rotation – that is, P[$N+1-\iota$] is back in its original position in the array. This means that if we initially set Q[i] = P[i] for $1 \le i \le N$, then $c[i] = N+1-i$ is equivalent to P[$N+1-i$] = Q[$N+1-i$]. But we have now removed the only test on $c[i]$, and now it is not necessary to maintain the counter array at all! Making this simplification, and changing ι to $N+1-i$, we have an algorithm proposed by G. Langdon in 1967 [25], shown in Fig. 10.

Algorithm 6 (Langdon)

```
ι:=1; loop: Q[ι]:=P[ι] while ι<N. ι:=ι+1 repeat,
process;
```

```
loop:
    rotate(i);
    if P[i]=Q[i] then i:=N else i:=i-1 endif;
    process;
while i≥1 repeat;
```

This is definitely the most simply expressed of our permutation generation algorithms. If P[1],···,P[N] are initially 1,···,N, then we can eliminate the initialization loop and replace Q[i] by i in the main loop. Unfortunately, this algorithm runs slowly on most computers—only when very fast rotation is available is it the method of choice. We shall examine the implementation of this algorithm in detail in Section 3.

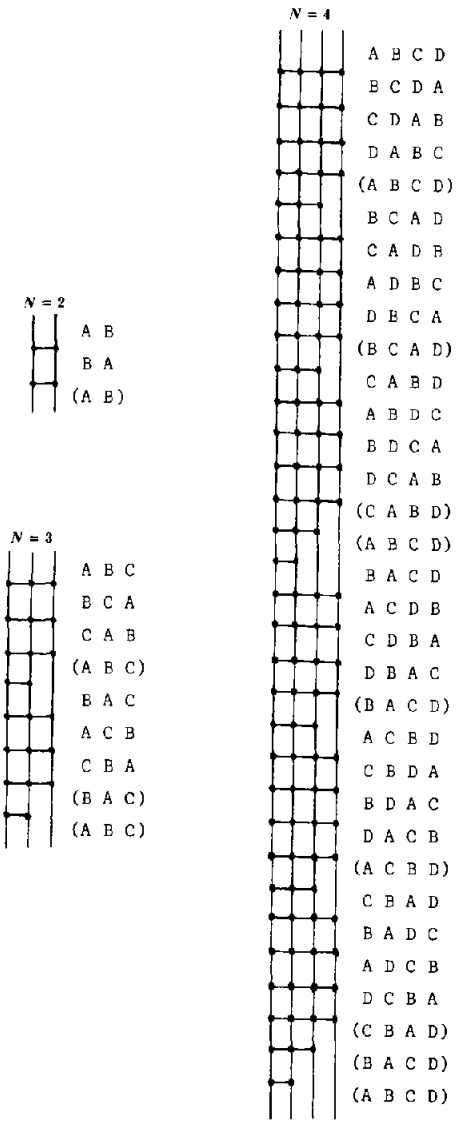

FIGURE 10. Langdon's algorithm for N = 2,3,4.

Lexicographic Algorithms

A particular ordering of the $N!$ permutation of N elements which is of interest is "lexicographic", or alphabetical, ordering. The lexicographic ordering of the 24 permutations of A B C D is shown in Fig. 11a. "Reverse lexicographic" ordering, the result of reading the lexicographic sequence backwards and the permutations from right to left, is also of some interest. Fig. 11b shows the reverse lexicographic ordering of the 24 permutations of A B C D.

The natural definition of these orderings has meant that many algorithms have been proposed for lexicographical permutation generation. Such algorithms are inherently less efficient than the algorithms of Section 1, because they must often use more than one exchange to pass from one permutation to the next in the sequence (e.g., to pass from A C D B to A D B C in Fig. 11a). The main practical reason that has been advanced in favor of lexicographic generation is that, in reverse order, all permutations of P[1],···,P[N−1] are generated before P[N] is moved. As with backtracking, a processing program could, for example, skip (N−1)! permutations in some instances. However, this property is shared by the recursive algorithms of Section 1—in fact, the general structure of Algorithm 1 allows P[N] to be filled in any arbitrary order, which could be even more of an advantage than lexicographic ordering in some instances.

Nevertheless, lexicographic generation is an interesting problem for which we are by now well prepared. We shall begin by assuming that P[1],···P[N] are distinct. Otherwise, the problem is quite different, since it is usual to produce a lexicographic listing with no duplicates (see [3, 9, 39].

We shall find it convenient to make use of another primitive operation, *reverse(i)*. This operation inverts the order of the elements in P[1], ··· P*[i];* thus, *reverse(i)* is equivalent to

A B C D	A B C D
A B D C	B A C D
A C B D	A C B D
A C D B	C A B D
A D B C	B C A D
A D C B	C B A D
B A C D	A B D C
B A D C	B A D C
B C A D	A D B C
B C D A	D A B C
B D A C	B D A C
B D C A	D B A C
C A B D	A C D B
C A D B	C A D B
C B A D	A D C B
C B D A	D A C B
C D A B	C D A B
C D B A	D C A B
D A B C	B C D A
D A C B	C B D A
D B A C	B D C A
D B C A	D B C A
D C A B	C D B A
D C B A	D C B A
(a)	(b)

FIGURE 11. Lexicographic ordering of A B C D. (a) In natural order. (b) In reverse order.

```
ι:=1;
loop while ι<N+1−ι:
    P[ι]:=:P[N+1−ι]; ι:=ι+1 repeat;
```

This operation will not be particularly efficient on most real computers, unless special hardware is available. However, it seems to be inherent in lexicographic generation.

The first algorithm that we shall consider is based on the idea of producing each permutation from its lexicographic predecessor. Hall and Knuth [15] found that the method has been rediscovered many times since being published in 1812 by Fischer and Krause [14]. The ideas involved first appear in the modern literature in a rudimentary form in an algorithm by G. Schrack and M. Shimrat [40]. A full formulation was given by M. Shen in 1962 [41, 42], and Phillips [34] gives an "optimized" implementation. (Dijkstra [6] cites the problem as an example to illustrate a "dramatic improvement in the state of the art" of computer programming since the algorithm is easily expressed in modern languages; the fact that the method is over 160 years old is perhaps a more sobering comment on the state of the art of computer science.)

The direct lexicographic successor of the permutation

B A C F H G D E

is clearly

B A C F H G E D,

but what is the successor of *this* permutation? After some study, we see that H G E D are in their lexicographically highest position, so the next permutation must begin as B A C G ⋯. The answer is the lexicographically lowest permutation that begins in this way, or

B A C G D E F H.

Similarly, the direct successor of

H F E D G C A B

in reverse lexicographic order is

D E G H F C A B,

and its successor is

E D G H F C A B.

The algorithm to generate permutations in reverse lexicographic order can be clearly understood from this example. We first scan from left to right to find the first i such that $P[i] > P[i-1]$. If there is no such i, then the elements are in reverse order and we terminate the algorithm since there is no successor. (For efficiency, it is best to make $P[N+1]$ larger than all the other elements—written $P[N+1] = \infty$—and terminate when $i = N+1$.) Otherwise, we exchange $P[i]$ with the next-lowest element among $P[1], \cdots, P[i-1]$ and then reverse $P[1], \cdots, P[i-1]$. We have

Algorithm 7 (Fischer-Krause)

```
P[N+1]=∞;
process;
loop.
        ι:=2; loop while P[ι]<P[ι−1]: ι:=ι+1 repeat;
while ι<N;
        j:=1; loop while P[j]>P[i]:j:=j + 1 repeat;
        P[ι]:=:P[j];
```

```
        reverse(ι−1);
        process;
    repeat;
```

Like the Tompkins-Paige algorithm, this algorithm is not as inefficient as it seems, since it is most often scanning and reversing short strings.

This seems to be the first example we have seen of an algorithm which does not rely on "factorial counting" for its control structure. However, the control structure here is overly complex; indeed, factorial counters are precisely what is needed to eliminate the inner loops in Algorithm 7.

A more efficient algorithm can be derived, as we have done several times before, from a recursive formulation. A simple recursive procedure to generate the permutations of P[1],···,P[N] in reverse lexicographic order can be quickly devised:

```
procedure lexperms(N);
    begin c:=1;
        loop:
            if N>2
              then lexperms(N−1) endif;
        while c<N:
            P[N]:=:P[c];
            reverse(N−1);
            c:=c+1;
        repeat;
    end;
```

Removing the recursion precisely as we did for Algorithm 1, we get

Algorithm 8 (Ord-Smith)

```
ι:=N; loop c[ι].=1; while ι>2: ι:=ι−1 repeat;
process;
loop:
    if c[ι]<ι then P[ι]:=:P[c[ι]], reverse(ι−1),
                   c[ι]:=c[ι]+1; ι:=2;
                   process;
              else c[ι]:=1; ι:=ι+1
    endif;
while ι≤N repeat;
```

This algorithm was first presented by R. J. Ord-Smith in 1967 [32]. We would not expect a priori to have a lexicographic algorithm so similar to the normal algorithms, but the recursive formulation makes it obvious.

Ord-Smith also developed [31] a "pseudo-lexicographic" algorithm which consists of replacing P[ι]:=:P[*c[i]*]; *reverse(i−1)*; by *reverse(i)* in Algorithm 8. There seem to be no advantages to this method over methods like Algorithm 1. Howell [17, 18] gives a lexicographic method based on treating P[1],···,P[N] as a base-N number, counting in base N, and rejecting numbers whose digits are not distinct. This method is clearly very slow.

Random Permutations

If N is so large that we could never hope to generate all permutations of N elements, it is of interest to study methods for generating "random" permutations of N elements. This is normally done by establishing some one-to-one correspondence between a permutation and a random number between 0 and $N!-1$. (A full treatment of pseudorandom number generation by computer may be found in [22].)

First, we notice that each number between 0 and $N!-1$ can be represented in a mixed radix system to correspond to an array $c[N],c[N-1],\cdots,c[2]$ with $0 \le c[i] \le i-1$ for $2 \le i \le N$. For example, 1000 corresponds to 1 2 1 2 2 0 since $1000 = 6! + 2\cdot5! + 4! + 2\cdot3! + 2\cdot2!$. For $0 \le n < N!$, we have $n = c[2]\cdot1! + c[3]\cdot2! + \cdots + c[N]\cdot(N-1)!$. This correspondence is easily established through standard radix conversion algorithms [22, 27, 47]. Alternatively, we could fill the array by putting a "random" number between 0 and $i-1$ in *c[i]* for $2 \le i \le N$.

Such arrays $c[N],c[N-1],\cdots,c[2]$ can clearly be generated by the factorial counting procedure discussed in Section 1, so that there is an implicit correspondence between such arrays and permutations of $1\ 2\ \cdots\ N$. The algorithms that we shall examine now are based on more explicit correspondences.

The first correspondence has been attributed to M. Hall, Jr., in 1956 [15], although it may be even older. In this correspondence, *c[i]* is defined to be the number of elements to the left of i which are smaller than it. Given an array, the following example shows how to construct the corresponding permutation. To find the permutation of $1\ 2\ \cdots\ 7$ corresponding to

1 2 1 2 2 0

we begin by writing down the first element, 1. Since $c[2] = 0$, 2 must precede 1, or

2 1.

Similarly, $c[3] = 2$ means that 3 must be preceded by both 2 and 1:

2 1 3

Proceeding in this manner, we build up the entire permutation as follows:

2 1 4 3
2 5 1 4 3
2 5 6 1 4 3
2 7 5 6 1 4 3

In general, if we assume that $c[1] = 0$, we can construct the permutation P[1],···, P[N] with the program

```
ι:=1;
loop.
    j:=ι;
    loop while j>c[ι]+1: P[j].=P[j−1]; repeat;
    P[c[ι]+1]:=ι;
    ι:=ι+1;
while ι≤N repeat;
```

This program is not particularly efficient, but the correspondence is of theoretic interest. In a permutation P[1],···,P[N] of 1,···,N, a pair (i,j) such that $i < j$ and P[i] > P[j] is called an inversion. The counter array in Hall's correspondence counts the number of inversions in the corresponding permutation and is called an inversion table. Inversion tables are helpful combinatorial devices in the study of several sorting algorithms (see [23]). Another example of the relationship between inversion tables and permutation generation can be found in [7], where Dijkstra reinvents the Johnson-Trotter method using inversion tables.

D. H. Lehmer [26, 27] describes another correspondence that was defined by D. N. Lehmer as long ago as 1906. To find the permutation corresponding to 1 2 1 2 2 0, we first increment each element by 1 to get

2 3 2 3 3 1.

Now, we write down 1 as before, and for $i = 2,\cdots,N$, we increment all numbers which are $\geq c[i]$ by 1 and write $c[i]$ to the left. In this way we generate

1
1 2
3 1 2
3 4 1 2
2 4 5 1 3
3 2 5 6 1 4
2 4 3 6 7 1 5

so that the permutation 2 4 3 6 7 1 5 corresponds to 1000. In fact, 2 4 3 6 7 1 5 is the 1000th permutation of 1,···,7 in lexicographic order: there are 6! permutations before it which begin with 1, then 2·5! which begin 2 1 or 2 3, then 4! which begin 2 4 1, then 2·3! which begin 2 4 3 1 or 2 4 3 5, and then 2·2! which begin 2 4 3 6 1 or 2 4 3 6 5.

A much simpler method than the above two was apparently first published by R. Durstenfeld [8]. (See also [22], p. 125). We notice that P[i] has $i-1$ elements preceding it, so we can use the c array as follows:

```
ι:=N;
loop while ι≥2: P[ι]:=:P[c[ι]+1]; ι:=ι−1 repeat;
```

If we take P[1],···,P[N] to be initially 1,···,N, then the array 1 2 1 2 2 0 corresponds to the permutation 5 1 4 6 7 3 2. This method involves only one scan through the array and is clearly more efficient than the above two methods.

We could easily construct a program to generate all permutations of 1 2 ··· N by embedding one of the methods described above in a factorial counting control structure as defined in Section 1. Such a program would clearly be much slower than the exchange methods described above, because it must build the entire array P[1],···,P[N] where they do only a simple exchange. Coveyou and Sullivan [4] give an algorithm that works this way. Another method is given by Robinson [37].

3. IMPLEMENTATION AND ANALYSIS

The analysis involved in comparing a number of computer algorithms to perform the same task can be very complex. It is often necessary not only to look carefully at how the algorithms will be implemented on real computers, but also to carry out some complex mathematical analysis. Fortunately, these factors pres-

ent less difficulty than usual in the case of permutation generation. First, since all of the algorithms have the same control structure, comparisons between many of them are immediate, and we need only examine a few in detail. Second, the analysis involved in determining the total running time of the algorithms on real computers (by counting the total number of times each instruction is executed) is not difficult, because of the simple counting algorithms upon which the programs are based.

If we imagine that we have an important application where all $N!$ permutations must be generated as fast as possible, it is easy to see that the programs must be carefully implemented. For example, if we are generating, say, every permutation of 12 elements, then every extraneous instruction in the inner loop of the program will make it run at least 8 minutes longer on most computers (see Table 1).

Evidently, from the discussion in Section 1, Heap's method (Algorithm 2) is the fastest of the recursive exchange algorithms examined, and Ives' method (Algorithm 4) is the fastest of the iterative exchange algorithms. All of the algorithms in Section 2 are clearly slower than these two, except possibly for Langdon's method (Algorithm 6) which may be competitive on machines offering a fast rotation capability. In order to draw conclusions comparing these three algorithms, we shall consider in detail how they can be implemented in assembly language on real computers, and we shall analyze exactly how long they can be expected to run.

As we have done with the high-level language, we shall use a mythical assembly language from which programs on real computers can be easily implemented. (Readers unfamiliar with assembly language should consult [21].) We shall use load (LD), store (ST), add (ADD), subtract (SUB), and compare (CMP) instructions which have the general form

LABEL OPCODE REGISTER, OPERAND (optional)

The first operand will always be a symbolic register name, and the second operand may be a value, a symbolic register name, or an indexed memory reference. For example, ADD I,1 means "increment Register I by 1"; ADD I,J means "add the contents of Register J to Register I"; and ADD I,C(J) means "add to Register I the contents of the memory location whose address is found by adding the contents of Register J to C". In addition, we shall use control transfer instructions of the form

OPCODE LABEL

namely JMP (unconditional transfer); JL, JLE, JE, JGE, JG (conditional transfer according as whether the first operand in the last CMP instruction was $<$, $\leq$, $=$, $\geq$, $>$ than the second); and CALL (subroutine call). Other conditional jump instructions are of the form

OPCODE REGISTER, LABEL

namely JN, JZ, JP (transfer if the specified register is negative, zero, positive). Most machines have capabilities similar to these, and readers should have no difficulty translating the programs given here to particular assembly languages.

Much of our effort will be directed towards what is commonly called code optimization: developing assembly language implementations which are as efficient as possible. This is, of course, a misnomer: while we can usually *improve* programs, we can rarely "optimize" them. A disadvantage of optimization is that it tends to greatly complicate a program. Although significant savings may be involved, it is dangerous to apply optimization techniques at too early a stage in the development of a program. In particular, we shall not consider optimizing until we have a good assembly language implementation which we have fully analyzed, so that we can tell where the improvements will do the most good. Knuth [24] presents a fuller discussion of these issues.

Many misleading conclusions have been drawn and reported in the literature based on empirical performance statistics comparing particular implementations of particular algorithms. Empirical testing can be valuable in some situations, but, as we have seen, the structures of permutation generation algorithms are so similar that the empirical tests which have been per-

formed have really been comparisons of compilers, programmers, and computers, not of algorithms. We shall see that the differences between the best algorithms are very subtle, and they will become most apparent as we analyze the assembly language programs. Fortunately, the assembly language implementations aren't much longer than the high-level descriptions. (This turns out to be the case with many algorithms.)

A Recursive Method (Heap)

We shall begin by looking at the implementation of Heap's method. A direct "hand compilation" of Algorithm 2 leads immediately to Program 1. The right-hand side of the program listing simply repeats the text of Algorithm 2; each statement is attached to its assembly-language equivalent.

This direct translation of Algorithm 2 is more efficient than most automatic translators would produce; it can be further improved in at least three ways. First, as we have already noted in Section 1, the test **while** $i \leq N$ need not be performed after we have set i to 2 (if we assume $N > 1$), so we can replace JMP WHILE in Program 1 by JMP LOOP. But this unconditional jump can be removed from the inner loop by moving the three instructions at LOOP down in its place (this is called rotating the loop—see [24]). Second, the test for whether i is even or odd can be made more efficient by maintaining a separate Register X which is defined to be 1 if i is even and -1 if i is odd. (This improvement applies to most computers, since few have a *jump if even* instruction.) Third, the variable k can be eliminated, and some time saved, if C(I) is updated before the ex-

PROGRAM 1. DIRECT IMPLEMENTATION OF HEAP'S METHOD

```
        LD      Z,1
        LD      I,N             i =N,
INIT    ST      Z,C(I)          loop  c[i] =1,
        CMP     I,2
        JLE     CALL            while i>2
        SUB     I,1                   i =i-1,
        JMP     INIT            repeat,
CALL    CALL    PROCESS         process,
LOOP    LD      J,C(I)          loop
        CMP     J,I
        JE      ELSE                  if c[i]<i
THEN    LD      T,I                   then
        AND     T,1
        JZ      T,EVEN                    if i odd
        LD      K,1                         then k =1
        JMP     EXCH                        else k =c[i]
EVEN    LD      K,J                       endif,
EXCH    LD      T,P(I)
        LD      T1,P(K)
        ST      T1,P(I)
        ST      T,P(K)                    P[i] = P[k],
        ADD     J,1
        ST      J,C(I)                    c[i] =c[i]+1,
        LD      I,2                       i =2,
        CALL    PROCESS                   process,
        JMP     WHILE
ELSE    ST      Z,C(I)                else c[i]:=1,
        ADD     I,1                       i:=i+1 endif,
WHILE   CMP     I,N             while i≤N
        JLE     LOOP            repeat,
```

change is made, freeing Register J. These improvements lead to Program 2. (The program uses the instruction LDN X,X which simply complements Register X.) Notice that even if we were to precompute the index table, as in Algorithm 1, we probably would not have a program as efficient as Program 2. On computers with a memory-to-memory *move* instruction, we might gain further efficiency by implementing the exchange in three instructions rather than in four.

Each instruction in Program 2 is labelled with the number of times it is executed when the program is run to completion. These labels are arrived at through a flow analysis which is not difficult for this program. For example, CALL PROCESS (and the two instructions preceding it) must be executed exactly $N!$ times, since the program generates all $N!$ permutations. The instruction at CALL can be reached via JLE CALL (which happens exactly once) or by falling through from the preceding instruction (which therefore must happen $N!-1$ times). Some of the instruction frequencies are more complicated (we shall analyze the quantities A_N and B_N in detail below), but all of the instructions can be labelled in this manner (see [31]). From these frequencies, we can calculate the total running time of the program, if we know the time taken by the individual instructions. We shall assume that instructions which reference data in memory take two time units, while *jump* instructions and other instructions which do not reference data in memory take one time unit. Under this model, the total running time of Program 2 is

$$19N! + A_N + 10B_N + 6N - 20$$

time units. These coefficients are typical, and a similar exact expression can easily be derived for any particular implementation on any particular real machine.

PROGRAM 2. IMPROVED IMPLEMENTATION OF HEAP'S METHOD

	LD	Z,1	1
	LD	I,N	1
INIT	ST	Z,C(I)	$N-1$
	CMP	I,2	$N-1$
	JLE	CALL	$N-1$
	SUB	I,1	$N-1$
	JMP	INIT	$N-1$
THEN	ADD	J,1	$N!-1$
	ST	J,C(I)	$N!-1$
	JP	X,EXCH	$N!-1$
	LD	J,2	A_N
EXCH	LD	T,P(I)	$N!-1$
	LD	T1,P−1(J)	$N!-1$
	ST	T1,P(I)	$N!-1$
	ST	T,P−1(J)	$N!-1$
CALL	LD	I,2	$N!$
	LD	X,1	$N!$
	CALL	PROCESS	$N!$
LOOP	LD	J,C(I)	$N!+B_N-1$
	CMP	J,I	$N!+B_N-1$
	JL	THEN	$N!+B_N-1$
ELSE	ST	Z,C(I)	B_N
	LDN	X,X	B_N
	ADD	I,1	B_N
	CMP	I,N	B_N
	JLE	LOOP	B_N

The improvements by which we derived Program 2 from Program 1 are applicable to most computers, but they are intended only as examples of the types of simple transformations which can lead to substantial improvements when programs are implemented. Each of the improvements results in one less instruction which is executed $N!$ times, so its effect is significant. Such coding tricks should be applied only when an algorithm is well understood, and then only to the parts of the program which are executed most frequently. For example, the initialization loop of Program 2 could be rotated for a savings of $N-2$ time units, but we have not bothered with this because the savings is so insignificant compared with the other improvements. On the other hand, further improvements within the inner loop will be available on many computers. For example, most computers have a richer set of loop control instructions than we have used: on many machines the last three instructions in Program 2 can be implemented with a single command. In addition, we shall examine another, more advanced improvement below.

To properly determine the effectiveness of these improvements, we shall first complete the analysis of Program 2. In order to do so, we need to analyze the quantities A_N

and B_N. In the algorithm, A_N is the number of times the test *i odd* succeeds, and B_N is the number of times the test $c[i]=1$ succeeds. By considering the recursive structure of the algorithm, we quickly find that the recurrence relations

$$A_N = NA_{N-1} + \begin{cases} 0 & N \text{ even} \\ N-1 & N \text{ odd} \end{cases}$$

and

$$B_N = NB_{N-1} + 1$$

hold for $N > 1$, with $A_1 = B_1 = 0$. These recurrences are not difficult to solve. For example, dividing both sides of the equation for B_N by $N!$ we get

$$\frac{B_N}{N!} = \frac{B_{N-1}}{(N-1)!} + \frac{1}{N!} = \frac{B_{N-2}}{(N-2)!} + \frac{1}{(N-1)!} + \frac{1}{N!}$$
$$= \cdots = \frac{1}{2!} + \frac{1}{3!} + \cdots + \frac{1}{N!}$$

or

$$B_N = N! \sum_{2 \le k \le N} \frac{1}{k!}.$$

This can be more simply expressed in terms of the base of the natural logarithms, e, which has the series expansion $\sum_{k \ge 0} 1/k!$: it is easily verified that

$$B_N = \lfloor N!(e-2) \rfloor$$

That is, B_N is the integer part of the real number $N!(e-2)$ (or $B_N = N!(e-2) + \epsilon$ with $0 \le \epsilon < 1$). The recurrences for A_N can be solved in a similar manner to yield the result

$$A_N = N! \sum_{2 \le k \le N} \frac{(-1)^k}{k!} = \lfloor N!/e \rfloor .$$

Substituting these into the expression above, we find that the total running time of Program 2 is

$$(19 + (1/e) + 10(e-2))N! + 6N + O(1),$$

or about $26.55N!$ time units. Table 3 shows the values of the various quantities in this analysis.

We now have a carefully implemented program whose performance we understand, and it is appropriate to consider how the program can be further "optimized." A standard technique is to identify situations that occur frequently and handle them separately in as efficient a manner as possible. For example, every other exchange performed by Program 2 is simply P[1]:=:P[2]. Rather than have the program go all the way through the main loop to discover this, incrementing and then testing $c[2]$, etc., we can gain efficiency by simply replacing $i:=2$ by $i:=3$;*process*; P[1]:=:P[2] in Algorithm 2. (For the purpose of this discussion assume that there is a statement $i:=2$ following the initialization loop in Algorithm 2.) In general, for any $n > 1$, we can replace $i:=2$ by $i:=n+1$; *process all permutations of* P[1], ···, P[n]. This idea was first applied to the permutation enumeration problem by Boothroyd [2]. For small n, we can quite compactly write in-line code to generate all permutations of P[1], ···, P[n]. For example, taking $n = 3$ we may simply replace

```
CALL    LD      1,2
        LD      X,1
        CALL    PROCESS
```

in Program 2 by the code in Program 3,

TABLE 3. ANALYSIS OF PROGRAM 2 ($T_N = 19! + A_N + 10B_N + 6N - 20$)

N	$N!$	A_N	B_N	T_N	$26.55N!$
1	1	0	0		
2	2	0	1	40	56+
3	6	2	4	154	159+
4	24	8	17	638	637+
5	120	44	86	3194	3186
6	720	264	517	19130	19116
7	5040	1854	3620	133836	133812
8	40320	14832	28961	1070550	1070496
9	362880	133496	260650	9634750	9634464
10	3628800	1334960	2606501	96347210	96344640
11	39916800	14684570	28671512	1059818936	1059791040
12	479001600	176214840	344058145	12717826742	12717492480

which efficiently permutes P[1], P[2], P[3]. (While only the code that differs from Program 2 is given here, "Program 3" refers to the entire improved program.)

The analysis of Program 3 differs only slightly from that of Program 2. This is fortunate, for it is often difficult to determine the exact effect of such major improvements. First, each of the new instructions is clearly executed $N!/6$ times, and each occurrence of $N!$ in Program 2's frequencies becomes $N!/6$ for Program 3; thus, the total running time is

$$(50/6)N! + A'_N + B'_N + 6N - 20.$$

Next, the analysis for A_N and B_N given above still holds, except that the initial conditions are different. We find that

$$A'_N = N! \sum_{4 \le k \le N} \frac{(-1)^k}{k!} = \left\lfloor N!\left(\frac{1}{e} - \frac{1}{3}\right)\right\rfloor$$

$$B'_N = N! \sum_{4 \le k \le N} \frac{1}{k!} = \left\lfloor N!\left(e - \frac{8}{3}\right)\right\rfloor$$

and the total running time of Program 3 is then about $8.88N!$.

By taking larger values of n we can get further improvements, but at the cost of $n+3n!$ lines of code. This is an example of a space-time tradeoff where the time saved is substantial when n is small, but the space consumed becomes substantial when n is large. For $n = 4$, the total running time goes down to about $5.88N!$ and it is probably not worthwhile to go further, since the best that we could hope for would be $5N!$ (the cost of two stores and a call).

On most computers, if Program 2 is "optimized" in the manner of Program 3 with $n = 4$, Heap's method will run faster than any other known method.

PROGRAM 3. OPTIMIZED INNER LOOP FOR PROGRAM 2

```
        :
CALL    LD      I,4
        LD      X,1
        CALL    PROCESS
        LD      T1,P(1)
        LD      T2,P(2)
        LD      T3,P(3)
        ST      T1,P(2)
        ST      T2,P(1)
        CALL    PROCESS
        ST      T3,P(1)
        ST      T2,P(3)
        CALL    PROCESS
        ST      T1,P(1)
        ST      T3,P(2)
        CALL    PROCESS
        ST      T2,P(1)
        ST      T1,P(3)
        CALL    PROCESS
        ST      T3,P(1)
        ST      T2,P(2)
        CALL    PROCESS
        .
```

An Iterative Method (Ives)

The structures of Algorithm 2 and Algorithm 4 are very similar, so that a direct "hand compilation" of Ives' method looks very much like Program 1. By rotating the loop and maintaining the value $N+1-i$ in a separate register we get Program 4, an improved implementation of Ives' method which corresponds to the improved implementation of Heap's method in Program 2.

The total running time of this program is

$$18N! + 21D_N + 10N - 25,$$

where D_N is the number of times $i:=i+1$ is executed in Algorithm 4. Another quantity, C_N, the number of times the test $P[N+1-i] = Q[N+1-i]$ fails, happens to cancel out when the total running time is computed. These quantities can be analyzed in much the same way that A_N and B_N were analyzed for Program 2: they satisfy the recurrences

$$C_N = C_{N-2} + (N-1)! - (N-2)!$$

$$D_N = D_{N-2} + (N-2)!$$

so that

$$C_N = (N-1)! - (N-2)! + (N-3)! - (N-4)! + \cdots$$

$$D_N = (N-2)! + (N-4)! + (N-6)! + \cdots$$

and the total running time of Program 2 is

$$18N! + 21(N-2) + O((N-4)!),$$

or about

$$\left(18 + \frac{21}{N(N-1)}\right)N!$$

Thus Program 4 is faster than Program 2: the improved implementation of Ives' method uses less overhead per permuta-

PROGRAM 4 IMPROVED IMPLEMENTATION OF IVES' METHOD

	LD	I,N	1
INIT	ST	I,C(I)	$N-1$
	LD	V,P(I)	$N-1$
	ST	V,Q(I)	$N-1$
	CMP	I,1	$N-1$
	JLE	CALL	$N-1$
	SUB	I,1	$N-1$
	JMP	INIT	$N-1$
THEN	LD	T,P(J)	$N!-C_N-1$
	LD	T1,P+1(J)	$N!-C_N-1$
	ST	T1,P(J)	$N!-C_N-1$
	ST	T,P+1(J)	$N!-C_N-1$
	ADD	J,1	$N!-C_N-1$
	ST	J,C(I)	$N!-C_N-1$
CALL	LD	I,1	$N!-C_N$
	LD	H,N	$N!-C_N$
	CALL	PROCESS	$N!$
LOOP	LD	J,C(I)	$N!+D_N-1$
	CMP	J,H	$N!+D_N-1$
	JL	THEN	$N!+D_N-1$
ELSE	LD	T,P(I)	C_N+D_N
	LD	T1,P(H)	C_N+D_N
	ST	T1,P(I)	C_N+D_N
	ST	I,C(I)	C_N+D_N
	CMP	T,Q(H)	C_N+D_N
	JNE	CALL	C_N+D_N
	ADD	I,1	D_N
	SUB	H,1	D_N
	CMP	I,H	D_N
	JL	LOOP	D_N

tion than the improved implementation of Heap's method, mainly because it does less counter manipulation. Other iterative methods, like the Johnson-Trotter algorithm (or the version of Ives' method, Algorithm 4a, which does not require the elements to be distinct), are only slightly faster than Heap's method.

However, the iterative methods cannot be optimized quite as completely as we were able to improve Heap's method. In Algorithm 4 and Program 4, the most frequent operation is P[*c[N]*]:=:P[*c[N]*+1]; *c[N]*:=*c[N]*+1; all but 1/*N* of the exchanges are of this type. Therefore, we should program this operation separately. (This idea was used by Ehrlich [10, 11].) Program 4 can be improved by inserting the code given in Program 5 directly after

```
CALL        PROCESS
```

(As before, we shall write down only the new code, but make reference to the entire optimized program as "Program 5".) In this program, Pointer J is kept negative so that we can test it against zero, which can be done efficiently on many computers. Alternatively, we could sweep in the other direction, and have J range from $N-1$ to 0. Neither of these tricks may be necessary on computers with advanced loop control instructions.

To find the total running time of Program 5, it turns out that we need only replace $N!$ by $(N-2)!$ everywhere in the frequencies in Program 4, and then add the frequencies of the new instructions. The result is

$$9N! + 2(N-1)! + 18(N-2)! + O((N-4)!),$$

not quite as fast as the "optimized" version of Heap's algorithm (Program 3). For a fixed value of N, we could improve the program further by completely unrolling the inner loop of Program 5. The second through eighth instructions of Program 5 could be replaced by

```
LD      T,P+1
ST      T,P
ST      V,P+1
CALL    PROCESS
LD      T,P+2
ST      T,P+1
ST      V,P+2
CALL    PROCESS
LD      T,P+3
ST      T,P+2
ST      V,P+3
CALL    PROCESS
```

(This could be done, for example, by a macro generator). This reduces the total running time to

$$7N! + (N-1)! + 18(N-2)! + O((N-4)!)$$

which is not as fast as the comparable highly optimized version of Heap's method (with $n = 4$).

It is interesting to note that the optimization technique which is appropriate for the recursive programs (handling small cases separately) is much more effective than the optimization technique which is

PROGRAM 5 OPTIMIZED INNER LOOP FOR PROGRAM 4

```
        :
EXLP    CALL    PROCESS     (N-1)!
        LD      J,1-N       (N-1)!
INLP    LD      T,P+N+1(J)  N!-(N-1)!
        ST      T,P+N(J)    N!-(N-1)!
        ST      V,P+N+1(J)  N!-(N-1)!
        CALL    PROCESS     N!-(N-1)!
        ADD     J,1         N!-(N-1)!
        JN      J,INLP      N!-(N-1)!
        LD      T,P+1       (N-1)!
        ST      T,P+N       (N-1)!
        ST      V,P+1       (N-1)!
        CMP     T,Q+N       (N-1)!
        JNE     EXLP        (N-1)!
        :
```

appropriate for the iterative programs (loop unrolling).

A Cyclic Method (Langdon)

It is interesting to study Langdon's cyclic method (Algorithm 6) in more detail, because it can be implemented with only a few instructions on many computers. In addition, it can be made to run very fast on computers with hardware rotation capabilities.

To implement Algorithm 6, we shall use a new instruction

```
MOVE TO, FROM(I)
```

which, if Register I contains the number i, moves i words starting at Location FROM to Location TO. That is, the above instruction is equivalent to

```
        LD  J,0
LOOP    T,FROM(J)
        T,TO(J)
        ADD J,1
        CMPJ,I
        JL  LOOP
```

We shall assume that memory references are overlapped, so that the instruction takes $2i$ time units. Many computers have "block transfer" instructions similar to this, although the details of implementation vary widely.

For simplicity, let us further suppose that P[1],$\cdots$,P[N] are initially the integers $0,1,\cdots,N-1$, so that we don't have to bother with the Q array of Algorithm 6.

With these assumptions, Langdon's method is particularly simple to implement, as shown in Program 6. Only eight assembly language instructions will suffice on many computers to generate all permutations of $\{0,1,\cdots,N-1\}$.

As we have already noted, however, the MOVEs tend to be long, and the method is not particularly efficient if N is not small. Proceeding as we have before, we see that E_N and F_N satisfy

$$E_N = \sum_{1\le k\le N-1} k!$$

$$F_N = \sum_{1\le k\le N} kk! = (N+1)! - 1$$

(Here F_N is not the frequency of execution of the MOVE instruction, but the total number of words moved by it.) The total running time of Program 6 turns out to be

$$N!\left(2N + 10 + \frac{9}{N}\right) + (O(N-2)!)$$

It is faster than Program 2 for $N < 8$ and faster than Program 4 for $N < 4$, but it is much slower for larger N.

By almost any measure, Program 6 is the simplest of the programs and algorithms that we have seen so far. Furthermore, on most computer systems it will run faster than any of the algorithms implemented in a high-level language. The algorithm fueled a controversy of sorts (see other references in [25]) when it was first introduced, based on just this issue.

Furthermore, if hardware rotation is available, Program 6 may be the method of choice. Since $(N-1)/N$ of the rotations are of length N, the program may be optimized in the manner of Program 5 around a four-instruction inner loop (call, rotate, compare, conditional jump). On some ma-

PROGRAM 6 IMPLEMENTATION OF LANGDON'S METHOD

```
THEN    LD      I,N-1       N!
        CALL    PROCESS     N!
LOOP    LD      T,P+1       N!+E_N
        MOVE    P,P+1(I)    F_N
        ST      T,P+1(I)    N!+E_N
        CMP     T,I         N!+E_N
        JNE     THEN        N!+E_N
        SUB     I,1         E_N
        JNZ     LOOP        E_N
```

chines, the *rotate* might be performed in, say, two time units (for example, if parallelism were available, or if P were maintained in registers), which would lead to a total time of $5N! + O((N-1)!)$. We have only sketched details here because the issues are so machine-dependent: the obvious point is that exotic hardware features can have drastic effects upon the choice of algorithm.

CONCLUSION

The remarkable similarity of the many permutation enumeration algorithms which have been published has made it possible for us to draw some very definite conclusions regarding their performance. In Section 1, we saw that the method given by Heap is slightly simpler (and therefore slightly more efficient) than the methods of Wells and Boothroyd, and that the method given by Ives is simpler and more efficient than the methods of Johnson and Trotter (and Ehrlich). In Section 2, we found that the cyclic and lexicographic algorithms will not compete with these, except possibly for Langdon's method, which avoids some of the overhead in the control structure inherent in the methods. By carefully implementing these algorithms in Section 3 and applying standard code optimization techniques, we found that Heap's method will run fastest on most computers, since it can be coded so that most permutations are generated with only two *store* instructions.

However, as discussed in the Introduction, our accomplishments must be kept in perspective. An assembly-language implementation such as Program 3 may run 50 to 100 times faster than the best previously published algorithms (in high-level languages) on most computer systems, but this means merely that we can now generate all permutations of 12 elements in one hour of computer time, where before we could not get to $N = 11$. On the other hand, if we happen to be interested only in all permutations of 10 elements, we can now get them in only 15 seconds, rather than 15 minutes.

The problem of comparing different algorithms for the same task arises again and again in computer science, because new algorithms (and new methods of expressing algorithms) are constantly being developed. Normally, the kind of detailed analysis and careful implementation done in this paper is reserved for the most important algorithms. But permutation generation nicely illustrates the important issues. An appropriate choice between algorithms for other problems can be made by studying their structure, implementation, analysis, and "optimization" as we have done for permutation generation.

ACKNOWLEDGMENTS

Thanks are due to P Flanagan, who implemented and checked many of the algorithms and programs on a real computer. Also, I must thank the many authors listed below for providing me with such a wealth of material, and I must apologize to those whose work I may have misunderstood or misrepresented. Finally, the editor, P. J. Denning, must be thanked for his many comments and suggestions for improving the readability of the manuscript.

REFERENCES

[1] BOOTHROYD, J "PERM (Algorithm 6)," *Computer Bulletin* 9, 3 (Dec. 1965), 104

[2] BOOTHROYD, J. "Permutation of the elements of a vector (Algorithm 29)"; and "Fast permutation of the elements of a vector (Algorithm 30)," *Computer J.* 10 (1967), 310–312.

[3] BRATLEY, P "Permutations with repetitions (Algorithm 306)," *Comm ACM* 10, 7 (July 1967), 450

[4] COVEYOU, R R.; AND SULLIVAN, J. G. "Permutation (Algorithm 71)," *Comm. ACM* 4, 11 (Nov 1961), 497.

[5] DERSHOWITZ, N. "A simplified loop-free algorithm for generating permutations," *BIT* 15 (1975), 158–164.

[6] DIJKSTRA, E W *A discipline of programming*, Prentice-Hall, Englewood Cliffs, N J , 1976.

[7] DIJKSTRA, E. W "On a gauntlet thrown by David Gries," *Acta Informatica* 6, 4 (1976), 357.

[8] DURSTENFELD, R. "Random permutation (Algorithm 235)," *Comm. ACM* 7, 7 (July 1964), 420.

[9] EAVES, B. C "Permute (Algorithm 130)," *Comm. ACM* 5, 11 (Nov. 1962), 551 (See also: remarks by R. J. Ord-Smith in *Comm ACM* 10, 7 (July, 1967), 452–3)

[10] EHRLICH, G. "Loopless algorithms for generating permutations, combinations and other combinatorial configurations," *J ACM* 20, 3 (July 1973), 500–513.

[11] EHRLICH, G "Four combinatorial algorithms (Algorithm 466)," *Comm ACM* 16, 11 (Nov 1973), 690–691.

[12] EVEN, S. *Algorithmic combinatorics*, Macmillan, Inc., N.Y., 1973.

[13] FIKE, C. T. "A permutation generation method," *Computer J.* 18, 1 (Feb. 1975), 21–22.

[14] FISCHER, L. L.; AND KRAUSE, K. C., *Lehrbuch der Combinationslehre und der Arithmetik*, Dresden, 1812

[15] HALL, M.; AND KNUTH, D. E. "Combinatorial analysis and computers," *American Math. Monthly* 72, 2 (Feb. 1965, Part II), 21–28.

[16] HEAP, B. R. "Permutations by interchanges," *Computer J.* 6 (1963), 293–4.

[17] HOWELL, J. R. "Generation of permutations by addition," *Math. Comp* 16 (1962), 243–44

[18] HOWELL, J. R. "Permutation generator (Algorithm 87)," *Comm. ACM* 5, 4 (April 1962), 209. (See also: remarks by R. J. Ord-Smith in *Comm ACM* 10, 7 (July 1967), 452–3.)

[19] IVES, F. M. "Permutation enumeration: four new permutation algorithms," *Comm. ACM* 19, 2 (Feb. 1976), 68–72.

[20] JOHNSON, S. M. "Generation of permutations by adjacent transposition," *Math. Comp.* 17 (1963), 282–285.

[21] KNUTH, D. E. "Fundamental algorithms," in *The art of computer programming 1*, Addison-Wesley, Co., Inc., Reading, Mass., 1968.

[22] KNUTH, D. E. "Seminumerical algorithms," in *The art of computer programming 2*, Addison-Wesley, Co., Inc., Reading, Mass., 1969.

[23] KNUTH, D. E. "Sorting and searching," in *The art of computer programming 3*, Addison-Wesley, Co., Inc., Reading, Mass., 1972

[24] KNUTH, D. E. "Structured programming with *go to* statements," *Computing Surveys* 6, 4 (Dec. 1974), 261–301.

[25] LANGDON, G. G., Jr., "An algorithm for generating permutations," *Comm. ACM* 10, 5 (May 1967), 298–9. (See also. letters by R. J. Ord-Smith in *Comm. ACM* 10, 11 (Nov. 1967), 684; by B. E. Rodden in *Comm. ACM* 11, 3 (March 1968), 150; CR Rev. 13,891, *Computing Reviews* 9, 3 (March 1968), and letter by Langdon in *Comm. ACM* 11, 6 (June 1968), 392.)

[26] LEHMER, D. H. "Teaching combinatorial tricks to a computer," in *Proc. of Symposium Appl. Math., Combinatorial Analysis*, Vol. 10, American Mathematical Society, Providence, R.I., 1960, 179–193.

[27] LEHMER, D. H. "The machine tools of combinatorics," in *Applied combinatorial mathematics* (E. F. Beckenbach, [Ed.]), John Wiley, & Sons, Inc., N.Y., 1964

[28] LEHMER, D. H. "Permutation by adjacent interchanges," *American Math. Monthly* 72, 2 (Feb. 1965, Part II), 36–46.

[29] ORD-SMITH, R. J. "Generation of permutation sequences Part 1," *Computer J.* 13, 3 (March 1970), 152–155.

[30] ORD-SMITH, R. J. "Generation of permutation sequences: Part 2," *Computer J.* 14, 2 (May 1971), 136–139.

[31] ORD-SMITH, R. J. "Generation of permutations in pseudo-lexicographic order (Algorithm 308)," *Comm ACM* 10, 7 (July 1967), 452 (See also: remarks in *Comm ACM* 12, 11 (Nov 1969), 638.)

[32] ORD-SMITH, R. J. "Generation of permutations in lexicographic order (Algorithm 323)," *Comm. ACM* 11, 2 (Feb. 1968), 117 (See also: certification by I. M. Leitch in *Comm ACM* 12, 9 (Sept. 1969), 512.)

[33] PECK, J. E. L.; AND SCHRACK, G. F. "Permute (Algorithm 86)," *Comm. ACM* 5, 4 (April 1962), 208.

[34] PHILLIPS, J. P. N. "Permutation of the elements of a vector in lexicographic order (Algorithm 28)," *Computer J.* 10 (1967), 310–311

[35] PLESZCZYŃSKI, S. "On the generation of permutations," *Information Processing Letters* 3, 6 (July 1975), 180–183.

[36] RIORDAN, J. *An introduction to combinatorial analysis*, John Wiley & Sons, Inc., N.Y., 1958.

[37] ROHL, J. S., "Programming improvements to Fike's algorithm for generating permutations," *Computer J.* 19, 2 (May 1976), 156

[38] ROBINSON, C. L. "Permutation (Algorithm 317)," *Comm. ACM* 10, 11 (Nov. 1967), 729.

[39] SAG, T. W. "Permutations of a set with repetitions (Algorithm 242)," *Comm ACM* 7, 10 (Oct 1964), 585.

[40] SCHRAK, G. F.; AND SHIMRAT, M. "Permutation in lexicographic order (Algorithm 102)," *Comm. ACM* 5, 6 (June 1962), 346. (See also: remarks by R. J. Ord-Smith in *Comm ACM* 10, 7 (July 1967), 452–3.)

[41] SHEN, M.-K. "On the generation of permutations and combinations," *BIT* 2 (1962), 228–231

[42] SHEN, M.-K. "Generation of permutations in lexicographic order (Algorithm 202)," *Comm ACM* 6, 9 (Sept. 1963), 517. (See also: remarks by R. J. Ord-Smith in *Comm. ACM* 10, 7 (July 1967), 452–3.)

[43] SLOANE, N. J. A. *A handbook of integer sequences*, Academic Press, Inc., N.Y., 1973

[44] TOMPKINS, C. "Machine attacks on problems whose variables are permutations," in *Proc Symposium in Appl. Math., Numerical Analysis*, Vol. 6, McGraw-Hill, Inc., N.Y., 1956, 195–211

[45] TROTTER, H. F. "Perm (Algorithm 115)," *Comm. ACM* 5, 8 (August 1962), 434–435.

[46] WALKER, R. J. "An enumerative technique for a class of combinatorial problems," in *Proc Symposium in Appl. Math., Combinatorial Analysis*, Vol. 10, American Mathematical Society, Providence, R.I., 1960, 91

[47] WELLS, M. B. "Generation of permutations by transposition," *Math Comp* 15 (1961), 192–195.

[48] WELLS, M. B. *Elements of combinatorial computing*, Pergamon Press, Elmsford, N.Y., 1971.

[49] WHITEHEAD, E. G. Combinatorial algorithms. (Notes by C. Frankfeldt and A. E. Kaplan) Courant Institute, New York Univ., 1973, 11–17.