Generation of permutation sequences

A. D. Woodall

Department of Computing, North Staffordshire Polytechnic, Blackheath Lane, Stafford

A recursive program for direct lexicographic generation of permutation seqences is described. From this, an approach to developing algorithms for non-lexicographic generation of sequences is presented, and one such algorithm is examined and proved.

(Received April 1976)

Algorithms for the generation of permutation sequences have attracted a good deal of interest in the computing literature. A survey of existing algorithms, and an analysis of their characteristics was given in two papers by Professor Ord-Smith of Bradford (Ord-Smith, 1970 and 1971). In the second of these papers, Ord-Smith refers to an algorithm of the present author which has not so far been published.

A recent paper by Fike (1975), which gave a new algorithm, has prompted preparation of the present paper, describing the algorithm in question. Fike's algorithm had proved, on test, to be one of the fastest known, and since the algorithm presented in the present paper appears to be faster than Fike's, it seems to be worth describing.

The method by which the algorithm was developed is described: It shows that there are many related algorithms which seem worthy of study.

Lexicographic algorithm

The first stage in the development was writing a recursive algorithm for generating permutations in lexicographic order. In essence, to generate all permutations of a set of n marks, held initially in order, the steps are:

1. If there are only two marks, the next permutation is got by swapping them; otherwise:
2. We hold the first mark unchanged while we generate all of the permutations of the remaining $n - 1$ marks (using the algorithm being described).
3. On completing these, the $n - 1$ marks will have been reversed (since we are generating in lexicographic order). Unless we have finished, we reverse them, which leaves them back in order.
4. Now we swap the first of the n marks with the largest of the following $n - 1$ marks which has not yet occupied the first position.
5. We now repeat from 2.

Iteration will be finished when all marks have occupied the first position while the remaining positions have been through a complete sequence of permutations.

An ALGOL 60 procedure for this algorithm is given in **Fig. 1.** The marks are preset in the integer array A. The procedure EXEC is called after each new permutation has been produced—in the test it simply printed the sequences. The procedure is invoked by the statements: EXEC;LEXPERM(n); A is indexed from right to left.

Non-lexicographic algorithm

If we remove the restriction that permutations appear in lexicographic order, we can still use the algorithm described above, but we omit the reversing of marks described in step 3.

We omit the restriction that the swap in 4 is of the largest of the following $n - 1$ marks not yet used—instead we take any of them that has not appeared before.

This leaves us with the problem of choosing the right item for a swap, so that we select a different mark each time: of course it must be a different mark—rather than a mark from a different position.

A simple procedure, which works, is as follows:

1. When n is odd, the swap of the mark in the nth position is always with the last of the $n - 1$ remaining marks.
2. When n is even, the swap is with marks in the positions starting with that next to the nth mark, and working across.

To make the procedure clear, **Fig. 2** gives the full print of the permutations of the integer marks 1 to 4 using this method.

An ALGOL 60 procedure for this algorithm is shown in **Fig. 3.** Testing for 'even' or 'odd' is by means of a global Boolean array which is preset 'true' or 'false' as appropriate. In other ways the procedure is similar to that in Fig. 1.

Proof of algorithm

To show that the algorithm of Fig. 3 does in fact generate permutations, it is necessary to show that the swapping procedures used always pick on a different mark.

In the 'odd' case, where swapping is always with the first position, we must show that the algorithm applied to the outer $n - 1$ marks leaves a different mark in the first position after each call. In the 'even' case, the marks must be shown to rotate appropriately.

```
procedure LEXPERM(n); value n; integer n;
 begin integer w;
 if n = 2 then
  begin w := A[1];
  A[1] := A[2]; A[2] := w;
  EXEC
  end
 else
  begin integer mp, hlen, i;
  for mp := n − 1 step −1 until 1 do
   begin LEXPERM(n − 1);
   hlen := (n − 1) ÷ 2;
   for i := 1 step 1 until hlen do
    begin w := A[i];
    A[i] := A[n − i];
    A[n − i] := w
    end;
   w := A[n]; A[n] := A[mp]; A[mp] := w;
   EXEC
   end;
  LEXPERM(n − 1)
  end
 end;
```

Fig. 1 **The lexicographic algorithm. Uses global array A (for the marks), and a global procedure EXEC**

As a first stage in the proof of the algorithm, we notice that it is simple to check whether or not the procedure works for all values of n up to any given number. The check proceeds as follows:

For each value of n we both check that the algorithm works, and discover how the marks are reordered in a complete cycle of $n!$ permutations.

With $n = 2$, the complete cycle is: C D ; D C .
The algorithm clearly works, and the reorder is a reversal (it could scarcely be anything else).

With $n = 3$, the complete cycle is:

```
B C D
B D C
C D B
C B D
D B C
D C B
```

In this case the algorithm works (the check is simply that each mark has appeared in the left hand column) and the reordering is again a reversal.

With $n = 4$, we no longer need to run through all the permutations. We have:

```
A B C D
. . . .
A D C B  (using the n = 3 result)
D A C B  (using the algorithm)
. . . .
D B C A  (using the n = 3 result)
C B D A  (using the algorithm)
. . . .
C A D B  (using the n = 3 result)
B A D C  (using the algorithm)
. . . .
B C D A  (using the n = 3 result).
```

Again the algorithm is checked (each mark has appeared in the left hand column). This time the reordering is cyclic.

Desk checking becomes tedious, and a program to perform the check was written. However, it is not needed since the pattern of reordering revealed by the check leads to an inductive proof of the algorithm.

We note that, if n is odd, the algorithm reorders the marks initially in positions M_n to M_1 into positions as follows:

$$\begin{aligned} &M_n \text{ goes to position } 1 \\ &M_1 \text{ goes to position } n \end{aligned} \qquad (1)$$

other marks finish in their starting positions.

If n is even the reordering gives:

$$\begin{aligned} &M_n \text{ goes to position } 1 \\ &M_{n-1} \text{ to position } n \\ &M_i \text{ goes to position } i-1 \ (2 < i < n-1) \\ &M_2 \text{ goes to position } n-1 \\ &M_1 \text{ goes to position } n-2 \end{aligned} \qquad (2)$$

The above results have been verified by the desk check above for small values of n. To complete the proof by induction, consider first the application of the check where n is even.

The positions $n - 1$ to 1 (an odd number of positions) are reordered as in (1) above; then we swap the nth position with (successively) the $n - 1$th down to the first, reordering from $n - 1$ to 1 between each, and after the last.

If the fortunes of the mark originally in position n are followed, it will be seen to move initially to position $n - 1$, and thereafter to alternate between positions 1 and $n - 1$, finishing in position 1 (remember that n is even, $n - 1$ odd).

In a similar way, the resting place of each mark is found to accord with the scheme (2) above.

The discussion of the proof with n odd is similar: In each case the steps are straightforward but tedious. It follows, by induction, that the algorithm is universally valid.

1	2	3	4	3	2	4	1
1	2	4	3	3	2	1	4
1	3	4	2	3	4	1	2
1	3	2	4	3	4	2	1
1	4	2	3	3	1	2	4
1	4	3	2	3	1	4	2
4	1	3	2	2	1	4	3
4	1	2	3	2	1	3	4
4	3	2	1	2	4	3	1
4	3	1	2	2	4	1	3
4	2	1	3	2	3	1	4
4	2	3	1	2	3	4	1

Fig. 2 Sequences generated by the non-lexicographic algorithm

```
procedure PERMALL(n); value n; integer n;
 begin integer w;
 if n = 2 then
  begin w := A[1];
  A[1] := A[2]; A[2] := w;
  EXEC
  end
 else
  begin integer mp, swpt;
  for mp := n − 1 step −1 until 1 do
   begin PERMALL(n − 1);
   swpt := if even[n] then mp else 1;
   w := A[n]; A[n] := A[swpt]; A[swpt] := w;
   EXEC
   end;
  PERMALL(n − 1)
  end
 end;
```

Fig. 3 Non-lexicographic algorithm (recursive). Uses global arrays A (for the marks) and even, and global procedure EXEC

Efficient implementation

To give an efficient program for the permutation algorithm, recursion is eliminated. The result is the procedure of **Fig. 4.**

This is a useful, general way of producing effective algorithms: A high level, often recursive, procedure reveals the structure of the algorithm. Eliminating recursion gives a fast, but often less readable procedure. Examples of this methodology are to be found, for example, in Knuth's tree visiting algorithms (Knuth, 1968) and in Fike's paper already cited (Fike, 1975). The present algorithm gives a particularly effective example.

(*a*) To start with, the depth of recursion corresponds to one of the positions where a mark is held. This is kept by the value of a pointer p (in fact $n - p + 1$ corresponds to the depth of recursion).

(*b*) Secondly, we note that when the recursive procedure is entered, the first step is a recursive call, and this proceeds down to the depth where $p = 2$. In the non-recursive version (Fig. 4), we set $p = 2$ initially, jumping straight to the bottom level, rather than descending through intermediate levels.

(*c*) Thirdly, we note that while the natural return from a recursive call is to the level immediately above, after the last call from a procedure of the level below, there is no more work to be done and the procedure exits at once (to the appropriate level above).

To allow for this in the non-recursive procedure we keep an array 'ret', holding the value of p to which control is to be returned. On the last descent of any cycle, we set the return value for the immediately lower value of p to the point where the present level is to return.

At the same time we must set the current level's ret to the one above, since the normal initiation will be avoided by the streamlining of step (*b*) above.

(*d*) Finally to further speed the program, we include the $p = 2$ interchange along with the next one in a loop (every alternate interchange is at $p = 2$). This is why the loop control sets $p = \text{ret}(2)$, rather than $p = 2$ as (*b*) above would suggest. It also means that the reset of ret(2) becomes a special case, which is taken care of by the first statement in the loop—setting ret(2) to 3 again whenever it has changed, but after the change has been effective.

The local variable *mp* of the recursive procedure reappears in Fig. 4 as the array *M*, indexed by *p*. The even/odd information is held by the sign of *M*.

Analysis of work done

The program of Fig. 4 has an extremely simple structure. The loop is traversed exactly once for each two permutations, apart from the last two. That is the loop is traversed $n!/2 - 1$ times.

At the first test in the loop, $p > 3$ will be false on exactly $\frac{1}{3}$ of the traversals.

Apart from this, there are just two paths within the loop, depending on the outcome of the test $mp = 0$. Let S be the number of occasions where the test shows $mp = 0$. At the top level ($p = n$) there is one such occasion. At $p = n - 1$ there is one for each of the n descents from above, and so on. Thus

$$S = 1 + n + n(n-1) + \ldots + n(n-1)\ldots 4$$
$$= n!\left(\frac{1}{n!} + \frac{1}{(n-1)!} + \frac{1}{(n-2)!} + \ldots \frac{1}{3!}\right)$$

if n is large we have $S \simeq n!(e - 2\cdot5)$.

Thus the proportion of loop traversals where $mp = 0$ is approximately $S/(n!/2 - 1) \simeq 2(e - 2\cdot5) \simeq 0\cdot436$.

The proportion where $mp \neq 0$ will thus be approximately 0·564. A simple count of the program will now give the average computational cost per permutation. We remember that each loop generates two permutations, and ignore the two outside the loop, and the other initialisation—which is slight.

We now have (for large n): cost per permutation =

1 exchange of a pair of marks
+4·205 array accesses
+5·769 assignments
+0·936 add/subtract operations
+2·5 'if' tests.

This permits comparison with previous algorithms.

Further improvements

The decision to include the permutations of the end two marks explicitly in the loop is arbitrary. It is equally simple to run through the five changes of the end three marks within the loop, or for that matter the 23 changes of the end four marks.

This kind of improvement of a permutation algorithm is not new; it has been used effectively by Boothroyd (1967). The details are simple—the value of p is set to the appropriate 'ret' value (3 or 4), the values of the marks are captured as variables at the beginning of the loop, and the work per permutation is drastically reduced.

The details of these programs and their analyses are not given: they can be simply constructed.

```
procedure perm(A, n, exec); value n; integer n;
integer array A; procedure exec;
 begin integer p, Mp, swpt, i, w;
 integer array M, KM[3:n], ret[2:n];
 for i := 4 step 2 until n do
  begin M[i] := KM[i] := i − 1;
  M[i − 1] := KM[i − 1] := 2 − i
  end;
 if n ÷ 2 × 2 ≠ n then KM[n] := M[n] := 1 − n;
 for i := 2 step 1 until n do ret[i] := i + 1;
 for p := ret[2] while p ⩽ n do
  begin
  if p > 3 then ret[2] := 3;
  exec;
  w := A[1]; A[1] := A[2]; A[2] := w;
  Mp := M[p];
  if Mp < 0 then
   begin swpt := 1; Mp := Mp + 1 end
  else
   begin swpt := Mp; Mp := Mp − 1 end;
  if Mp = 0 then
   begin
   M[p] := KM[p];
   ret[p − 1] := ret[p];
   ret[p] := p + 1
   end
  else M[p] := Mp;
  exec;
  w := A[p]; A[p] := A[swpt]; A[swpt] := w
  end;
 exec;
 w := A[1]; A[1] := A[2]; A[2] := w;
 exec
 end of perm
```

Fig. 4 The final form of the algorithm as a self-contained ALGOL 60 procedure

Other algorithms

Reverting to the recursive version of the algorithm, the particular procedure used for choosing which mark to swap with the nth mark is not the only possible one.

A similar, but slightly more complicated procedure yields the familiar sequence due to Wells (1961). Here, if n is odd, the first two swaps are with the adjacent mark, thereafter swapping is with positions starting at the end and stepping across. With n odd, the swap is always with the adjacent item.

Once an algorithm has been established, the non-recursive version can be simply obtained. In fact there is an obvious correspondence with Wells' algorithm, the array *M* of Fig. 4 corresponding to the variable radix counters used as a 'signature' by Wells. The improvement involving the array ret could also be devised, simply by considering ways of economising the counter mechanism.

However, the derivation via the recursive algorithm seems more natural and shows clearly how other sequences can be found. In fact, if the places for each of the $n - 1$ swaps with the nth position in a cycle at depth n are held as the nth row of a two-dimensional array, no algorithm need be found. The array can be constructed, for example, so that the swap to the nth position is always of the largest mark not yet to have been there (taking 'largest' to mean leftmost in the original order). This approach, with a study of the array formed, seems likely to yield interesting results.

Conclusion

The traditional approach to the generation of permutation sequences has been to assume that a direct program is too difficult to write. This has usually led to study of a cognate problem—generation of all states of a function whose states can be placed in one to one correspondence with permutations. The function is chosen so that its states are simply generated and the correspondences are amenable to mathematical study.

The present paper shows that the problem can be solved by direct programming.

References

Boothroyd, J. (1967). Algorithm 30, *The Computer Journal*, Vol. 10, p. 310

Fike, C. T. (1975). A Permutation Generation Method. *The Computer Journal*, Vol. 18, pp. 21-22

Knuth, D. E. (1968). *The Art of Computer Programming*, Volume 1: Fundamental Algorithms, Addison-Wesley Publishing Company, Reading, Massachusetts.

Ord-Smith, R. J. (1970). Generation of Permutation Sequences: part 1. *The Computer Journal*, Vol. 13, pp. 152-155

Ord-Smith, R. J. (1971). Generation of Permutation Sequences: part 2. *The Computer Journal*, Vol. 14, pp. 136-139

Wells, Mark B. (1961). Generation of Permutations by Transposition. *Mathematics of Computation*, Vol. 15, p. 192

Book reviews

Modelling and Performance Evaluation of Computer Systems, edited by H. Beilner and E. Gelenbe, 1977; 515 pages. (*North-Holland*, Dfl. 95.00)

This book consists of 31 papers presented at the 1976 International Workshop of Modelling and Performance Evaluation of Computer Systems. Two thirds of the contributions are from West European research laboratories and universities, and the remainder from Eastern Europe and USA.

The largest group of papers aims at an understanding of the processes within a computer, hence mathematical models of virtual storage systems, multiprogramming facilities, file assignment and other characteristics associated with large computers are presented. Some of the authors provide solutions to classes of queueing networks which have made advances to probability theory, whereas others use existing modelling techniques such as queueing theory, simulation, Markov processes, or dynamic programming. The other major group of papers is concerned with scheduling of externally given tasks through the computer. Simple statistical models are described as well as more complicated heuristic models. In addition, two papers concentrate on measurement aspects of information inside a computer, and there is a textbook type contribution on 'Statistics and simulation'.

The papers vary in length from two to 32 pages and differ considerably in quality. Some authors present no more than a theorem or mathematical formula with sparse explanation, whereas others describe their models in some detail together with experimental evidence. The editors give no guidance to the reader who wishes to select certain topics, nor are the papers arranged in any coherent sequence.

This is not a book for the commercial computer installation, but could be of benefit to the mathematically orientated researcher in computer science or probability theory.

R. Holstein (London)

Interactive Data Analysis, by R. R. McNeil, 1977; 186 pages. (*John Wiley*, £7·00, paper only)

The title of this book requires interpretation. 'Data analysis' is a branch of statistics pioneered by J. W. Tukey of Princeton University, whose influence is pervasive throughout the book—it may indeed be regarded as an introduction to Tukey's *Exploratory Data Analysis*, recently published by Addison-Wesley after some years of informal circulation. 'Interactive' is taken to imply that the results of one piece of data analysis can be used immediately as input to the next piece, without a computer necessarily being involved.

On the statistical side, the original flavour of the book may be illustrated by the fact that it contains no mention of significance tests of any kind, and that the mean as a measure of location is only introduced (in a rather complex form) in the final chapter. The methods illustrated are essentially descriptive. They are based on medians and inter-quartile distances as measures of location and spread and their output is usually a simple plot shown on a typewriter. The problems tackled are mostly concerned with expressing the data in a simple way, such as with equal spreads in different groups or straight line relationships between variables, and with spotting outlying values which require special attention. The approach is a refreshing change from that of the standard statistical texts, though (as with the parent book) it is not always clear just what the proposed analysis is *for*; the recommended approach seems to be 'Here's some nice data—let's analyse them' rather than one which starts with the purposes for which the data were gathered in the first place. Symptomatically, the brief section on experimental design is almost worthless.

The book contains algorithms both in APL and FORTRAN for doing the various analyses. These are potentially quite useful but could have been far more so had a little more care been taken. I cannot speak for the APL, but the FORTRAN contains many non-standard features and the algorithms are imagined as embedded in an operating system which is fairly sophisticated in its handling of temporary files, default parameter values and so on. I hope to try them out in practice, but I do not anticipate that this will be a particularly easy task.

M. J. R. Healy (Harrow)

Systems: Analysis, Administration and Architecture, by J. W. Sutherland, 1975; 339 pages. (*van Nostrand Reinhold*, £6·95)

Computer scientists and computer practitioners make frequent use of the word 'systems' but rarely so in the sense in which it has become used in system science circles. Indeed system science has, so far, not had much practical impact on the computer field except perhaps tangentially in the field of 'systems analysis'. However, even there very few systems analysts are in fact aware of the thinking that has gone into the system science area these past 20 years.

It is of course not only in the analysis of a potential computer application, and the design of the hardware and software to implement it, that system science can make a contribution. Examination of the complex problems facing the software industry, the design of the software process, the implementation and maintenance of large programs, all constitute problems that are typical of those for whose solution system science offers an approach. Similarly, as computer system design moves further and further into the distributed computer area, the system approach becomes increasingly important.

Systems thinking is thus a must for the serious computer scientist or practitioner, the system architect, the software designer and the executive with computing responsibilities.

Sutherland's book, *Systems: Analysis, Administration and Architecture*, gives a first class introduction to the subject area. It is relatively simple reading yet comprehensive. Once picked up it is difficult to put down and this reviewer can only recommend it in the strongest terms.

M. M. Lehman (London)