Generation of permutation sequences: Part 2

R. J. Ord-Smith

Computing Laboratory, University of Bradford, Bradford, Yorkshire BD7 1DP

The main purpose of Part 2 is to present explicitly the six fastest general permutation algorithms in an improved and standardised form. A comparison shows the kind of difficulties which can arise in the implementation of a high level language. An appendix contains an extended bibliography. A brief account of continuing work is included.

(Received July 1970)

1. Introduction

Delay in the appearance of Part 2 of this paper has enabled the author to incorporate a few details arising from a fruitful correspondence with a number of other workers. The names of the relevant correspondents are given in context but, since the work referred to is in development and unpublished, the best that can be offered at this stage is to put anyone interested in particular details into touch with the originator.

In optimisation applications each arrangement has to be evaluated. A transposition algorithm may permit the value of the current arrangement to be obtained by a small correction to the immediately previous value, rather than *ab initio*. As J. D. Murchland points out, this may be an important consideration. He also discusses the pseudo-random sampling of arrangements when the enumeration of all is impracticable. To this end J. Lions had developed a generalised permutation algorithm allowing many arrangements to be skipped but with a considerable degree of combinatorial control. A. D. Woodall appears to have discovered an explicit transposition algorithm different again from those of Wells, Johnson, or Trotter which promises to have a form which may lead to a very fast algorithm. Miss E. Brahler has modified the lexicographic algorithm Bestlex in a manner similar to that used in Boothroyd's production of Algorithm 30 (see Section 2.1.3 below). She claims a speed increase of more than three times. It is clear that, although it was judged that this paper appeared at a suitably stabilised point for a general review, there continues to be important improvement in techniques still taking place.

2. The fastest published general permutation algorithms

There are six published permutation algorithms which are at present the fastest existent. Three of these are transposition sequence algorithms and three are lexicographic. All have undergone some improvements since their original publication. They are, therefore, reproduced here for the first time in their improved versions.

2.1. *Transposition sequence algorithms*

2.1.1. *ACM Algorithm* 115 *slightly improved* (*ACM* 115*A*)

Historically this is the first of the six algorithms. Though Wells (1961) had already proved the existence of transposition sequences with his algorithm, Algorithm ACM 115A defines a different transposition sequence to that of Wells. It differs also from the adjacent transposition sequence described by Johnson (1963). Due to Trotter (1962) it remained for a number of years the fastest permutation algorithm of any kind. It retains a combinatorial interest because it generates a bell-ringing sequence. This algorithm is also noteworthy for the subtle manner of its skilful organisation. In general array access dominates in an ALGOL procedure of this kind. Notwithstanding the extra array accesses caused by the use of both the signature vector *d* and an auxiliary vector *p* the elegant control exercised in the use of these is worthy of study by anyone who wishes to be known as a programmer. The very minor improvement is due to Boothroyd.

```
procedure perm (x, n); value n; integer n; array x;
begin own integer array p, d[2: 10]; integer k, q; real t;
         if first then begin for k := 2 step 1 until n do
         begin p[k] := 0; d[k] := 1 end;
         first := false
         end;
         k := 0;
index: p[n] := q := p[n] + d[n];
         if q = n then begin d[n] := −1;
                          goto loop
                          end;
         if q ≠ 0 then goto transpose;
         d[n] := 1; k := k + 1;
loop: if n > 2 then begin n := n − 1;
                          goto index
                          end;
         q := 1; first := true;
transpose: q := q + k; k := q + 1; t := x[q]; x[q] := x[k]; x[k] := t
end of procedure perm;
```

2.1.2. *British Computer Journal Algorithm* 29 *improved* (*BCJ* 29*A*)

This algorithm, due to Boothroyd (1967) is a direct implementation of Wells' sequence, though the algorithm modifies the rules given by Wells. This version has been modified by Boothroyd from that originally published in order to bring the specification of the parameter list and the use of a global boolean variable into line with 'standard practice'. It is improved to capitalise the fact that only $x[1]$ and $x[2]$ are involved in half the transpositions carried out. See also comment on this in Section 2.2.2.

```
procedure perm (x, n); value n; integer n; array x;
begin integer k, k less 1, dk; real temp;
        own boolean odd; own integer array d[3:10];
        if first then begin odd := false;
                          if n > 2 then begin first := false;
                          for k := 3 step 1 until n do d[k] := 0
                                          end
                    end;
        if odd then begin k less 1 := 2; k := 3;
              count: dk := d[k];
                          if dk ≠ k less 1 then goto swap;
                          d[k] := 0;
                          if k ≠ n then begin k less 1 := k;
                                                   k := k + 1;
                                                   goto count
                                          end;
                         first := true; goto exit;
              swap: dk := d[k]: = dk + 1;
                         if dk > 2 then begin
                         if k − k ÷ 2 × 2 = 0 then
                         k less 1 := k − dk
                                          end;
                         temp := x[k]; x[k] := x[k less 1];
                         x[k less 1] := temp
                          end
        else begin temp := x[1]; x[1] := x[2]; x[2] := temp
              end;
        odd := ¬odd;
exit: end of procedure perm;
```

2.1.3. *British Computer Journal Algorithm* 30 *improved* (*BCJ* 30*A*)

Boothroyd has standardised and improved his original Algorithm 30 (1967) to provide a general transposition sequence algorithm which only possesses the restriction $n \geqslant 5$. It exploits a repeated pattern in 23 successive entries involving only elements $x[1]$ to $x[4]$. Unless some special sequence of arrangements is demanded, this is the fastest permutation algorithm to have been published (but see next section). Its efficiency arises from the considerable use of explicitly named array elements $x[1]$ to $x[4]$ reducing the use of the slow subscript mechanism of the implementation of high level languages.

```
procedure perm (x, n); value n; integer n; array x;
begin integer j, k, k less 1, dk; real xk;
      own real x1, x2, x3, x4;
      own integer i; own integer array d[5:10];
      switch s := s1, s2, s1, s2, s1, s3, s1, s2, s1, s2, s1, s3, s1, s2, s1, s2,
            s1, s4, s1, s2, s1, s2, s1, s5;
      switch ss := ss1, ss2, ss3, ss4;
      if first then begin for k := 5 step 1 until n do d[k] := 0;
                     x1 := x[1]; x2 := x[2];
                     x3 := x[3]; x4 := x[4];
                     i := 0; first := false
                  end;
      i := i + 1; goto s[i];
   s1: xk := x1; x1 := x[1] := x2;
      x2 := x[2] := xk; goto exit;
   s2: xk := x2; x2 := x[2] := x3;
      x3 := x[3] := xk; goto exit;
   s3: xk := x3; x3 := x[3] := x4;
      x4 := x[4] := xk; goto exit;
   s4: xk := x4; x4 := x[4] := x1;
      x1 := x[1] := xk; goto exit;
   s5: k less 1 := 4; k := 5; i := 0;
count: dk := d[k]; if dk ≠ k less 1 then goto swap;
      d[k] := 0; if k ≠ n then begin k less 1 := k;
                                     k := k + 1;
                                     goto count
                              end;
      first := true; goto exit;
 swap: dk := d[k] := dk + 1;
      if dk > 2 then begin if k - k ÷ 2 × 2 = 0 then
                              k less 1 := k - dk
                     end;
      xk := x[k]; x[k] := x[k less 1];
      x[k less 1] := xk;
      goto if k less 1 ≤ 4 then ss[k less 1] else exit;
     ss1 : x1 := xk, goto exit;
     ss2 : x2 := xk; goto exit;
     ss3 : x3 := xk; goto exit;
     ss4 : x4 := xk;
exit: end of procedure perm;
```

2.2. *Lexicographic and pseudo lexicographic algorithms*

2.2.1. *ACM Algorithm* 308 *improved* (*ACM* 308*A*)

Original publication by the author (Ord-Smith, 1967) of this algorithm, which produces a pesudo lexicographic sequence, preceded that of the truly lexicographic algorithm ACM 323. It has been subsequently noted that it is best described as a modified form of the lexicographic algorithm. See Part 1 of this paper for further description and also Ord-Smith (1969). Boothroyd noted the failure of the algorithm in the trivial case $n = 2$ and his recommended modification, included here, removes this restriction.

```
procedure perm (x, n); value n; integer n; array x;
begin own integer array q[3:10]; integer k, m;
      real t; own boolean odd;
      if first then begin odd := true;
                      if n > 2 then begin first := false;
                      for m := 3 step 1 until n do
                      q[m] := 1     end
                   end;
      if odd then begin odd := false;
                      t := x[1]; x[1] := x[2]; x[2] := t;
                      goto finish
                   end;
      odd := true; k := 3;
 oop: m := q[k]; if m = k then
      begin if k < n then begin q[k] := 1;
                                k := k + 1;
                                goto loop
                          end
            else begin first := true; goto trinit end
      end;
                      q[k] := m + 1;
                      trinit : m := 1;
                      transpose: t := x[m]; x[m] := x[k]; x[k] := t;
                            m := m + 1; k := k - 1;
                            if m < k then goto transpose;
finish: end of procedure perm;
```

2.2.2. *ACM Algorithm* 323 *improved* (*ACM* 323*A*)

This algorithm was originally proposed by the author. An improvement in speed was proposed to the author by Trotter which involved explicit transposition of the element $x[1]$ and $x[2]$ in half the cases. This was incorporated into the version published as ACM Algorithm 323 Ord-Smith (1968). The idea has subsequently also been used by Boothroyd in his improved Algorithm BCJ 29A. The version below also incorporates Boothroyd's control of the $x[1]$, $x[2]$ transpositions by means of a local boolean variable and his suggestion to remove the failure in the case $n = 2$ in the original.

```
procedure perm(x, n);
      {This is identical to the previous algorithm of 2.2.1 except that the
      line which was:
            q[k] := m + 1;
      now becomes
            t := x[m]; x[m] := x[k]; x[k] := t;
            q[k] := m + 1; k := k - 1;          }
finish: end of procedure perm;
```

2.2.3. *British Computer Journal Algorithm* 28 *corrected* (*BCJ* 28*A*)

This algorithm by Phillips was published together with algorithms 29 and 30 in *The Computer Journal* (Phillips, 1967). It is reproduced here in corrected and 'standardised' form. Standardisation can include regeneration of the identity when *first* is reset **true**. See **comment** and optional line in the body of the procedure.

The algorithm dispenses with the need for a signature by making use of the numerical values of the elements of the given array. All permutations will be generated only if all marks are distinct and the lowest lexicographic permutation starts the process. If the marks are not all distinct, only unique permutations are generated. If other than lowest lexicographic ordering starts the process, only higher orderings are generated. These conditions may often be advantageous to the user. Where they are not restrictive, the algorithm provides both the fastest lexicographic algorithm and the fastest general algorithm for all $n > 1$.

```
procedure perm (x, n); value n; integer n; array x;
begin integer i, j, k; real xi, xj, xk, xn, t;
      own integer n1, n2; own real x1, x2, x3;
      if first then begin n1 := n - 1;
                     if n1 = 0 then goto exit;
                     n2 := n - 2;
                     x1 := x[n]; x2 := x3 := x[n1];
                     if n2 ≠ 0 then
                     begin first := false;
                           x3 := x[n2]
                     end
                  end;
      t := x1;
      if x2 < x1 then begin x1 := x[n] := x2;
                          x2 := x[n1] := t
                      end
      else begin if x3 < x2 then
                 begin if x1 > x3 then
                       begin x1 := x[n] := x2;
                             x2 := x[n1] := x3;
                             x3 := x[n2] := t
                       end
                 else begin x1 := x[n] := x3;
                            x3 := x[n2] := x2;
                            x2 := x[n1] := t
                      end
                 end
      else begin i := n2; xi := x3; j := n - 3;
           search: if j ≠ 0 then begin xj := x[j];
                     if xj < xi then goto swap;
                     i := j; xi := xj; j := j - 1;
                     goto search
                     end;
                     first := true;
                     comment the next line is optional. If included the final
                     arrangement is reset to the original one at the end of
                     the factorial nth call; i := 1; j := n; goto reset
           end;
```

Table 1 A time comparison between the six algorithms

	TIMES IN SECONDS FOR $n = 8$						
ALGORITHM	COLUMN 1 Total time Elliott 503 with subscript checks, driving program time deducted	COLUMN 2 Total time Elliott 503 without subscript checks, driving program time deducted	COLUMN 3 Time Elliott 503 without subscript checks with procedure call time deducted	COLUMN 4 Total time ICL 1905 with subscript checks, driving program time deducted	COLUMN 5 Time ICL 1905 with subscript checks with procedure call time deducted	COLUMN 6 Ratio of Times in Column 3	COLUMN 7 Ratio of Times in Column 5
BCJ 30A	36·1	32·9	22·8	44	20	1·00	1·00
BCJ 28A	53·7	47·4	37·3	44	20	1·64	1·00
BCJ 29A	62·3	54·0	43·9	48	24	1·93	1·20
ACM 308A	68·0	58·8	48·7	48	24	2·14	1·20
ACM 115A	78·1	66·7	56·6	54	30	2·48	1·50
ACM 323A	80·0	68·4	58·3	52	28	2·56	1·40

```
        goto exit;
swap: k := n;
        for xk := x[k] while xk ⩽ xj do k := k − 1;
        x[k] := xj; x[j] := xk; j := n;
reset: xi := x[i]; x[i] := x[j]; x[j] := xi;
        i := i + 1; j := j − 1;
        if j >  i then goto reset;
        x1 := x[n]; x2 := x[n1]; x3 := x[n2]
         end;
exit: end of procedure perm;
```

3. Implementation overheads

A time comparison (see **Table 1**) between the six algorithms probably serves little purpose in practical decision concerning which to use. The difference in time is sufficiently small to make the major criterion that of the combinatorial advantage of the particular permutation sequence generated. The times on different machines do serve to show that certain features of a high level language suffer differing penalties in different implementations. One cannot, therefore, make absolute judgements in time comparison of algorithms, particularly when written in a high level language.

Columns 5 and 6 give the fairest comparisons of time that can be obtained, and Columns 6 and 7 the respective ratios of speed. It is not possible to control the built in array bound checks provided in compilers available to the author as it is for the Elliott 503 ALGOL operating system in use at the Hydro-University Computing centre of the University of Tasmania.*

The improvement in algorithm 308,323 relative to 29,115 for ICL 1905 arises because 1900 ALGOL implementation particularly benefits subscripted variables with explicit subscripts, which are treated as simple variables. The proportions of such variables assigned during use is higher for these algorithms. On the other hand, the very noticeable discrepancy in ratio for algorithms 28, 30 arises from slow implementation of switches in 1900 ALGOL. Algorithm 30 relies heavily on a multiple switch. Elliott 503 implementation does not carry so obvious a burden because goto statements and switches share a common mechanism. Implementation for both computers showed a worthwhile improvement for algorithms 29, 308, 323 by using an **own boolean** variable as described in Section 2.2.2. above, and this is incorporated in the versions given here. This may not, however, be an improvement for all compilers.

4. Conclusions and acknowledgements

The author is encouraged to believe, from the reaction to Part 1 already published, that this review has been of value, and that it might serve both to take stock of the situation and to stimulate and promote further studies.

It is clear from literature search and from comments received that book references are insufficiently used or quoted in periodical references. Though textbooks suffer worse from being outdated in a quickly moving subject, they contain important work for all that and it is hoped that the bibliography given in appendix will help to remedy the situation.

The author is indebted to many persons, some named and some not, for helping him to produce this paper. Essential tribute must be paid again for the mammoth work of Mr. Boothroyd throughout the exercise. The appendix was compiled by Mr. K. Boateng as part of his M.Sc. dissertation submitted for work in this field at Bradford University.

Appendix

Related periodical references other than those already given in Part 1.

Permutation algorithms

Bratley, P. (1967). Permutations with Repetitions, Algorithm 306, *CACM*, Vol. 10, p. 450.

Eaves, B. C. (1962). Permute, Algorithm 130, *CACM*, Vol. 5, p. 551.

Heap, B. R. (1963). Permutations by Interchanges, *The Computer Journal*, Vol. 6, p. 293.

Howell, J. R. (1962). Permutation Generator, Algorithm 87, *CACM*, Vol. 5, p. 209.

Sag, T. W. (1964). Permutations of a Set with Repetitions, Algorithm 242, *CACM*, Vol. 7, p. 585.

Schrak, G. F., and Shimrat, M. (1962). Permutation in Lexicographic Order, Algorithm 102, *CACM*, Vol. 5, p. 346.

Random permutation generators

Durstenfield, R. (1964). Random Permutation, Algorithm 235, *CACM*, Vol. 7, p. 420.

Robinson, C. L. (1967). Permutation, Algorithm 317, *CACM*, Vol. 10, p. 729.

Permutation inverse

Boonstra, B. H. (1965). Inverse Permutation, Algorithm 250, *CACM*, Vol. 8, p. 105.

Medlock, C. W. (1965). Remark on Inverse Permutation. *CACM*, Vol. 8, p. 670.

Permutation representations

Snapper, E. (1968). The polynomial of a Permutation Representation, *Jour. Combinatorial Theory*, Vol. 5, p. 105.

*The operating system was written by W. G. Warne, formerly officer-2nd-in-charge, HUCC, later with Radio-physics Division CSIRO, currently with UNIVAC (Australia) Pty Ltd.

Combinations

KURTZBERG, J. (1962). Combination, Algorithm 94, *CACM*, Vol. 5, p. 344.

MIFSUD, C. J. (1963). Combination in Lexicographic and in any Order, Algorithms 154, 155. *CACM*, Vol. 6, p. 103.

WOLFSON, M. L., and WRIGHT, H. V. (1963). Combinatorial of *m* Things. Algorithms 160, 161. *CACM*, Vol. 6, p. 161.

Related algorithm

FENICHEL, R. R. (1968). Distribution of Indistinguishable Objects into Distinguishable Slots, Algorithm 329. *CACM*, Vol. 11, p. 430.

Review of permutation techniques

Proceedings of Symposium in Applied Mathematics, Vol. 10. Particularly papers by Hoffman, A. J. and Lehmer, D. H.

Book references

Permutation theory

Combinatorial Mathematics, Ryser, H. J. Mathematical Association of America, 1963.

Introduction to the Theory of Groups, Alexandroff P. S. Blackie, 1959.

Theory of Groups, Marshall Hall. Macmillan, New York, 1969.

Background to Set and Group Theory, Mansfield, D. E., and Bruckheimer, M., Chatto and Windus, 1965.

Survey of Modern Algebra, Birkhoff, G., and Maclane, S. Macmillan, New York, Revised, 1953.

Elements of Abstract Algebra, Moore, J. T. Macmillan, New York, 1962.

Linear Algebra and Group Theory, Smirnov, V. I. McGraw-Hill, New York, 1961.

Algebra, Volume 1, Redei, L. Pergamon, 1967.

Modern Algebra, Ayres, F. Schaum, 1965.

Algebra, Archbold, J. W. Pitman, 1961.

A University Algebra, Littlewood, D. E. Heinemann, 1950.

Permutation theory (Statistical context)

Elements of Statistical Inference, Huntsberger, D. V. Prentice-Hall, 1962.

Introduction to Statistical Analysis, Dixon, W. J., and Massey, F. J. McGraw-Hill, 1957.

Brief accounts in many other books.

Permutation sequence algorithms

Applied Combinatorial Mathematics, Edited by Beckenbach, E. F. John Wiley (1964). Contains an important chapter entitled 'The machine tools of Combinatorics' by Lehmer, D. H.

Book review

Switching and Finite Automata Theory (Computer Science Series). by Z. Kohavi, 1970; 592 pages. (*McGraw-Hill*, £7·90)

Some weeks of devoted work would be needed to give a fair review of this book, and many months to study it properly. For theoretically oriented Computer Scientists, Logic Designers or Control Engineers, the book must be a very valuable one, and one feels that it will probably be from such investigations that fundamental discoveries of value may well spring. It is hard to believe that it can rank as a text for either under- or postgraduate courses, in spite of the preface. One would have difficulty in incorporating more than a small fraction of the contents into such courses unless they were very specialised ones. For this reason it seems unnecessary to attempt to make the book free standing and some of the first two chapters dealing with binary arithmetic, codes, and sets and lattices, might well be assumed knowledge. Certainly some of these early sections are the least well written part of the book.

The second part of the book introduces Switching Functions, AND, OR and NOT operations, and Boolean Algebras. A neat and unified structure is exposed if in the rather refined terminology of set theory. The fourth chapter goes into considerable detail concerning algorithms for the minimisation of switching functions, and explains very clearly some of the techniques which have been developed. However, it is very much oriented towards hand manipulation which, Kohavi admits, become cumbersome for more than six variables. Though the Quine-McCluskey method is systematic enough for computer implementation, we are told on page 90, no explicit references to such work is given. The methods are further weakened when NAND and NOR logic is introduced in Chapter 5 and one is told that the methods prove less effective in this case. However, the discussion of NAND and NOR logic in this Chapter and of Threshold logic in Chapter 7 are very valuable, perhaps more than the network logic of Chapter 6. Part 2 ends with a useful chapter on design and testing of faults. I approve of the use of the word 'fault' in this context even though the word 'error' is used instead in the introductory Chapter 1.

Chapters 9 to 13 of Part 3 are devoted to applications of the techniques of Part 2 to sequential circuits and finite state machines. Again there is a wealth of information which most computer scientists will find is marred by the combination of terse theoretical vocabulary and hand based methods. Although the same may be said for Chapter 14, most will find here a valuable survey of memory and information loss and retention in finite machines which it is hard to find given so concisely elsewhere. Chapter 15 returns to an important subset of finite machines, those which are linear. The last chapter is a rather abstruse one concerned with the characterisation of finite state machines.

R. J. ORD-SMITH (Bradford)