Some experiences with the Johnson–Trotter permutation generation algorithm

Yaakov L. Varol* and Doron Rotem†

Some experiences with the Johnson–Trotter permutation generation algorithm are reported. A version which generates strings containing permutations as subsequences and has the offset maintenance removed is proposed. This version is faster than any other known algorithm and has indexing properties which make it of practical use.

(Received April 1978; revised April 1979)

In a survey paper, Sedgewick (1977) has pointed out that permutation generation is not an end in itself, but rather a first step of a more global process which seeks to find a permutation satisfying a set of criteria. The set of criteria may be formulated simply in terms of a partial order on the set of N marks to be permuted, or may involve a complex cost reducing requirement, or in some other way. The time taken by this global process is in general much higher than the task of generating all permutations. This might be relatively simple for some applications if consecutive permutations differed in exactly two adjacent locations. It may then be sufficient to calculate the effect of an exchange rather than reprocessing the entire new permutation (Sedgewick, 1977). A further characteristic of the generation which may simplify the search for a particular permutation is the presence of a hierarchical structure of the subsets of permutations differing by a cyclic rotation of a subsequence. Often, if a permutation does not satisfy the requirements, then the same is true for any other permutation obtained by a cyclic rotation of a subsequence. Thus, it is possible to achieve great savings if subsets of cyclically related permutations were properly nested with respect to the generation (Tompkins, 1956). The first condition above is satisfied by the Johnson–Trotter (J–T) type algorithms (Johnson, 1963; Trotter, 1962), while the second one is satisfied by algorithms based on nested cycling such as the one by Langdon (1967). Clearly, to have both conditions satisfied all the time is not feasible, since rotating a subsequence of i elements introduces changes in i locations. However, a compromise can be achieved.

Assume that a string which contained all permutations as subsequences (Koutas and Hu, 1975) could be constructed. An algorithm to extract from it the individual permutations might be extremely complicated. Nevertheless, one could conceive of a string containing a large class of permutations as substrings, from which extracting them might be cheaper than generating them one at a time. Our algorithm is based on the generation of such keys differing from each other in two adjacent locations. Each key is then cyclically rotated in both directions to obtain all related permutations.

Consider the string $P \equiv 1234123$. By using a pointer to indicate the start and imply a direction, one could easily extract from P the substrings

1234	4321
2341	1432
3412	2143
4123	3214

which are eight distinct permutations of the marks 1, 2, 3 and 4. In general, each string P of length $2N - 1$, constructed by concatenating to the right of a permutation the first $N - 1$ elements of itself, will be the key for $2N$ permutations. For instance, the three strings needed to generate all 24 permutations of four marks are 1234123, 2134213 and 2314231.

We now need a method for generating the $(N - 1)!/2$ keys. We shall accomplish this by improving a modified version of the J–T algorithm (see Sedgewick, 1977). This algorithm is based on:

(*a*) a counter $c(i)$, $1 \leqslant c(i) \leqslant N + 1 - i$, to indicate the number of various positions mark i has occupied within the current configuration of the subsequence containing only the marks greater than i;

(*b*) a Boolean variable $d(i)$ to indicate the direction of motion of i, where

$$d(i) = \begin{cases} \text{true} & \text{if } i \text{ is moving from left to right} \\ \text{false} & \text{if } i \text{ is moving from right to left} \end{cases}$$

(*c*) an offset quantity x to take care of the fact that when i is to be moved some of the lesser marks may be to the left of i, while others may be to its right.

If the marks 1, 2, . . . $N - 4$, $N - 3$ are allowed to occupy $N - 1$, $N - 2$, . . . 4, 3 distinct locations respectively, then

$$(3)\,(4)\ldots(N-2)\,(N-1) = \frac{(N-1)!}{2}$$

distinct permutations would be generated. In the J–T algorithm this can be accomplished by the requirements that $1 \leqslant i \leqslant N - 3$ and $1 \leqslant c(i) \leqslant N - i$. Furthermore, if any two of these permutations were used as keys, they would give two non-intersecting sets of $2N$ permutations each. If the two sets had a common permutation then the sets would be equal due to the cyclic nature of their derivation, contradicting the fact that the initial keys were distinct. These facts, together with the logic of Johnson's original proof (Johnson, 1963), constitute the outline of a proof that all $N!$ permutations would be generated exactly once.

In deriving a new key if mark i is to be moved (by this we mean that a decision to transpose mark i with one of its adjacent marks is reached in the algorithm) this would necessarily be with a mark greater than i, and it would effectively introduce a cyclic rotation on the current configuration of the set $S_i = \{j | i + 1 \leqslant j \leqslant N\}$. This configuration of S_i remains fixed for all the following $2N(N - 1)!/(N - i)!$ permutations, and the moving of any of the lesser marks will not change it. Thus, given two permutations, the size of the common subset of marks which are fixed in their relative position except possibly for a cyclic rotation, is a non-decreasing function of the distance between the two in the generation sequence. In other words, there is a proper nesting of cyclically related permutations.

*Computer Science Department, Southern Illinois University, Carbondale, Illinois 62901, USA.
†Computer Science Department, University of Waterloo Waterloo, Ontario N2L 3E5, Canada.

CCC-0010-4620/81/0024-0173 $02.00

In our generation of keys, since $1 \leqslant c(i) \leqslant N - i$, when mark i is to be moved, the location k of the first (left) mark in the pair of adjacent marks to be transposed is given by

$$k = \begin{cases} c(i) + x(i) = \text{loc}(i) & \text{if } d(i) \text{ is true} \\ N - i - c(i) + x(i) = \text{loc}(i) - 1 & \text{otherwise} \end{cases}$$

where $x(i)$ is the offset. The offset can be written as

$$x(i) = \sum_{j=1}^{i-1} \delta(j) \quad \text{where} \quad \delta(j) = \begin{cases} 1 & \text{if } d(j) \text{ is true} \\ 0 & \text{otherwise} \end{cases}$$

Assume that mark i is being considered and let T_j, $1 \leqslant j \leqslant i - 1$, stand for the number of times mark j has completed a full traversal from left to right, or right to left extremes. By definition, $d(j)$ will be true if and only if T_j is even. Recall also that a mark can only move following a full traversal of its predecessor mark, and that for any mark a traversal means $N - j$ moves by it. When mark $i - 1$ has completed T_{i-1} traversals, i.e. mark i has occupied T_{i-1} different positions, mark $i - 2$ will have done $T_{i-1}[N - (i - 1)]$ traversals, requiring $T_{i-1}[N - (i - 1)]\,[N - (i - 2)]$ traversals by mark $i - 3$ etc. In general, we have

$$\delta(i - 1) = \begin{cases} 1 & \text{if } T_{i-1} \text{ is even} \\ 0 & \text{otherwise} \end{cases}$$

and for $j \leqslant i - 2$

$$T_j = T_{i-1}[N - (i - 1)]\,[N - (i - 2)] \ldots [N - (j + 1)]$$

For $j \leqslant i - 3$, the above product will involve two consecutive integers, and thus

$$\delta(j) = 1 \quad \text{for} \quad j \leqslant i - 3$$

Finally, since $T_{i-2} = T_{i-1}[N - (i - 1)]$

$$\delta(i - 2) = \begin{cases} 1 & \text{if } d(i - 1) \text{ is true or } N - i \text{ is odd} \\ 0 & \text{otherwise} \end{cases}$$

Combining these equations we can write: for $i > 2$,

$$x(i) = i - \begin{cases} 1 & \text{if } d(i - 1) \text{ is true} \\ 2 & \text{if } (N - i) \text{ is odd} \\ 3 & \text{otherwise} \end{cases}$$

and for $i = 2$,

$$x(2) = \begin{cases} 1 & \text{if } d(i - 1) \text{ is true} \\ 0 & \text{otherwise} \end{cases}$$

Since it is known in advance if $N - i$ is even or odd, the offset $x(i)$ is determined by simply testing $d(i - 1)$. We can thus compute $k[= \text{loc}(i) \text{ or } \text{loc}(i) - 1]$ as follows

if $d(i)$ **then** $k := c(i) + i - 1$
else $k := N + 1 - c(i)$;
if not $d(i - 1)$ **then** $k := k - X(i)$

where $X(i)$ is precomputed as $X(2) = 1$ and for $i > 2$

$$X(i) = \begin{cases} 1 & \text{if } N - i \text{ is odd} \\ 2 & \text{otherwise} \end{cases}$$

In this way, we add an offset to k only half as many times as before, and we eliminate completely the resetting of the offset to zero as well as its updating.

In the algorithm below we make use of a well known optimisation and handle mark 1 separately since it is moved most often $[(N - 2)\,(N - 2)!/2$ times]. Note also that its motion does not require any offset. The variable V is a pointer to the first element of a permutation. The two processes 1 and 2 differ in that they process N marks from left to right or from right to left respectively starting with the mark at location V. This may seem to be more expensive than handling a vector of N elements where the start and end as well as the direction of processing are fixed. However, it is easy to see that the only extra work which is not shown explicitly in the algorithm below, and that process 1 and process 2 need to do, is to compute an end pointer as $V + (N - 1)$ or $V - (N - 1)$, respectively. Finally, we assume that initially the vector **P** contains the objects corresponding to the marks 1, 2, . . . N, 1, 2, . . . $N - 1$ in that order.

```
PROCEDURE PERMUTGEN(VAR N:INTEGER;
P:LIST1);
(* OBJECT IS A DATA TYPE DEFINED IN A *)
(* CALLING PROGRAM, LIST1 IS A DATA TYPE OF *)
(* THE FORM ARRAY [1 . . 2U] OF OBJECT, LIST2 *)
(* HAS THE FORM ARRAY [1 . . U] OF INTEGER, *)
(* LIST3 HAS THE FORM ARRAY [1 . . U] OF *)
(* BOOLEAN, WHERE U IS AN INTEGER ≥ N *)
VAR I,J,V,K,K1,C1:INTEGER; D1:BOOLEAN;
    C,X:LIST2; D:LIST3;
BEGIN
I:= N; WHILE I > 3 DO BEGIN I:= I - 2; J := I - 1;
                            C[I]:= 1; C[J] := 1;
                            D[I]:= TRUE;
                            D[J]:= TRUE;
                            X[I]:= 2; X[J]:= 1
                      END;
C1:= 1; D[1]:= TRUE; D1:= TRUE; X[2]:= 1; I := 1;
WHILE I < = N - 3 DO
 BEGIN FOR V:= 1 TO N DO PROCESS1(V);
  FOR V:= N TO 2*N - 1 DO PROCESS2(V);
  IF C1 < N - 1 THEN BEGIN IF D1 THEN K:= C1
                           ELSE K:= N - 1 - C1;
                           C1:= C1 + 1 END
                ELSE BEGIN C1:= 1; D1:= NOT D1;
                D[1]:= D1; I:=2;
                 WHILE C[I] > = N - I DO
                       BEGIN D[I]:= NOT D[I];
                       C[I]:= 1; I:= I + 1;
                       END;
                 IF D[I] THEN K:= C[I] + I - 1
                         ELSE K:= N - 1 - C[I];
                 IF NOT D[I - 1] THEN K:= K - X[I];
                 C[I]:= C[I] + 1
                 END;
  K1:= K + 1; P[K]:= P[K1]; P[K1]:= P[N + K];
  P[N + K]:= P[K];
  P[N + K1]:= P[K1]
 END
END
```

Following Sedgewick's (1977) approach we can directly translate this algorithm into a simple assembly language program using only load, store, add, subtract, compare and jump instructions. Assuming that instructions referencing data in memory take two units, while all jump instructions and those not referencing data in memory take one time unit, we can record execution frequencies of all instructions and establish that our algorithm requires

$$4N! + 12(N - 1)! + O((N - 2)!)$$

time units, which compares favourably with any of the time estimates given for all the algorithms surveyed by Sedgewick (1977). Note that the improvement on the calculation of the offset contributes a reduction in the time complexity of order $O((N - 2)!)$, while the more significant reduction in the coefficient of $N!$ is due to the use of keys. Empirical performance statistics obtained from stand-alone executions of our algorithm as well as the major algorithms given in Sedgewick (1977), agreed with the theoretical time estimates.

Needless to say, our improvements do not make the problem any more practical, since when considering $N > 25$ the time required is either equivalent to the Earth's age or twice as much. Recall, however, that permutation generation algorithms would in general be used to obtain a specific permutation

satisfying certain specified conditions. When it is not possible to construct directly the required permutation and one must rely on extensive search methods, then one must incorporate some heuristics into the search, in order to be able to do selective generation. Some backtracking applications use implicit instructions like 'don't generate any more permutations starting with these three marks'. However, this approach is not suitable for all applications. In general one needs a generation-order-preserving indexing of all permutations (a one to one correspondence between the set of permutations and the set of integers [0, $N! - 1$]) suitable for efficient implementation of the tasks:

1. Given the index K find the permutation π_K corresponding to it.
2. Given a permutation π find its index K.
3. Given $\pi = \pi_K$ start the generation from this point onwards.

Efficient routines for performing these three tasks would provide the capability of jumping back and forth in the sequence of all permutations. The first two are readily available for most permutation generation algorithms, and they would also cater for the random generation of a permutation or given $\pi = \pi_K$ finding π_{K+L} where L is an integer such that $0 \leqslant K + L \leqslant N! - 1$. The third task, however, can be quite complicated for some algorithms. For the generation algorithm presented in this paper, routines designed to perform all three tasks are given in the Appendix.

Acknowledgement
The authors are indebted to the referee who helped to improve the presentation and eliminate certain errors.

Appendix

Using mixed radix representation of integers, it is easy to construct a one to one correspondence between the set of integers in [0, $N! - 1$] and the set of all permutations of N marks. The correspondence we consider is generation-order-preserving, i.e. if π precedes π' in the generation sequence then their corresponding indices K and K' are such that $K < K'$.

Given an integer IND $\in$ [0, $N! - 1$] let K_{key} and K_{per} be the quotient and remainder resulting from dividing IND by $2N$. K_{key} will be used to construct the key and K_{per} to obtain from it the permutation whose index is IND. Consider the sequence of generated keys starting from $12 \ldots N$ and ending with the one whose index is K_{key}. In this sequence, for $1 \leqslant i \leqslant N$, let t_i be the total number of times mark i has been considered for motion, and s_i be the number of times it has been moved following the last time it occupied a left or right extreme position. Clearly, $t_1 = K_{key}$ and

$$s_i = t_i = 0 \quad \text{for} \quad i = N, N-1, N-2$$

Since there are $N - i$ distinct positions mark i occupies before mark $i + 1$ is considered for motion we can easily compute the values of t_{i+1} and s_i respectively as the quotient and remainder resulting from dividing t_i by $N - i$, i.e.

$$\frac{t_i}{N-i} = t_{i+1} + \frac{s_i}{N-i}$$

The value t_{i+1} also stands for the number of complete traversals mark i has undergone. Thus

$$d(i) = \begin{cases} \text{true} & \text{if } t_{i+1} \text{ is even} \\ \text{false} & \text{otherwise} \end{cases}$$

The sequence $s_1, s_2, \ldots s_{N-3}, K_{per}$ uniquely defines K, and by definition $c(i) = s_i + 1$, where $c(i)$ is the counter used in our algorithm. Furthermore, $J(i)$, defined as the number of marks greater than i and to its left in the key permutation, is related to s_i as follows:

$$J(i) = \begin{cases} s_i & \text{if } d(i) \text{ is true} \\ N - i - 1 - s_i & \text{if } d(i) \text{ is false} \end{cases}$$

Note that to find the key with index K_{key}, we do not need to construct the vectors **d** or **c**. In fact, we can eliminate the vectors **t** and **s** as well, since their entries can be used one at a time as they are computed. To construct the key in a vector KEY having initially $2N - 1$ blanks we start from $i = 1$ and replace the $[J(i) + 1]$st remaining blank in it by i. We also copy the first $N - 1$ marks of KEY in sequence into KEY($N + 1$), ... KEY($2N - 1$). Having constructed the key, set $v = 1 + K_{per}$ if $K_{per} < N$, and the permutation with the given index is KEY(v) KEY($v + 1$) ... KEY($v + N - 1$). If $K_{per} \geqslant N$ then the desired permutation is KEY(v)KEY($v - 1$)...KEY($v - N + 1$) where $v = K_{per}$. An example listing the values of all relevant variables follows.

$N = 7$, IND = 2361
$K_{key} = 168$, $K_{per} = 9$
$t \equiv \langle 168, 28, 5, 1, 0, 0, 0 \rangle$
$s \equiv \langle 0, 3, 1, 1, 0, 0, 0 \rangle$
$c \equiv \langle 1, 4, 2, 2, 1, 1, 1 \rangle$
$d \equiv \langle$true, false, false, true, true, true, true$\rangle$
$J \equiv \langle 0, 1, 2, 1, 0, 0, 0 \rangle$
KEY $\equiv$ 1524367152436
$v = 9$
PERM $\equiv$ 5176342

The procedure below generates in PERM, the permutation corresponding to a given index. It assumes that **P** contains the permutation whose index is 0.

```
PROCEDURE PERMFROMIND(VAR N, IND:INTEGER;
P,PERM:LIST1);
(* OBJECT IS A DATA TYPE DEFINED IN A *)
(* CALLING PROGRAM, LIST1 IS A DATA TYPE OF *)
(* THE FORM ARRAY [1 . . U] OF OBJECT, LIST2 *)
(* HAS THE FORM ARRAY [1 . . 2U] OF OBJECT, *)
(* WHERE U IS AN INTEGER ⩾ N.                *)
VAR KKEY,KPER,TI,SI,I,JI,NMI,L,M,V:INTEGER;
 KEY:LIST2;
BEGIN
 KKEY:= IND DIV (2*N); KPER:= IND MOD (2*N);
 TI:= KKEY;
 FOR I:= 1 TO N - 1 DO
 BEGIN NMI := N - I;SI:= TI MOD NMI;
 TI:= TI DIV NMI;
  IF ODD(TI) THEN JI := NMI - 1 - SI
             ELSE JI := SI;
  L := 0;M := 0;
  WHILE L < = JI DO
   BEGIN M := M + 1;
    IF KEY[M] = ' ' THEN L := L + 1
   END;
  KEY[M] := P[I]; KEY[N + M]:= P[I]
 END;
 KEY[N] := P[N];
  IF KPER < N THEN FOR V:= 1 TO N DO
                    PERM[V]:= KEY[V + KPER]
                   ELSE FOR V:= 1 TO N DO
                    PERM[V]:= KEY [KPER + 1 - V]
END
```

We now consider the converse problem of finding the index of a given permutation π. Let us first find the key permutation from which π was derived. To do so, we make use of the facts that in a key permutation the rightmost mark must be N and the marks $N - 2$, $N - 1$ and N must appear in this relative order. In any given π(not necessarily the key) these three marks

seen from left to right can appear in one of the six relative orders

(*a*) $N-2, N, N-1$	or	$N, N-1, N-2,$
	or	$N-1, N-2, N$
(*b*) $N-2, N-1, N$	or	$N, N-2, N-1$
	or	$N-1, N, N-2$

We can determine which case is true by finding the locations of the marks in question and testing for the conditions above. In case (*a*), start with the mark to the left of N and read cyclically into KEY,N marks from right to left. Set $K_{per} = N + \text{loc}(N) - 1$. In case (*b*), start with the mark to the right of N and read cyclically into KEY,N marks from left to right. Set $K_{per} = N - \text{loc}(N)$.

Having found the key permutation, we proceed to find $K_{key} = t_1$ by performing in reverse order the computations in PROCEDURE PERMFROMIND starting with $J(N-3) = s_{N-3} = t_{N-3}$. Finally, we compute the index as $t_1 \times 2N + K_{per}$. As before, we can also set the variables $c(i)$ and $d(i)$ along the way. An example listing the values of all relevant variables follows.

$\pi \equiv 5216473$ KEY $\equiv 3521647352164$
$K_{per} = 7 - 6 = 1$
$J \equiv \langle 3, 2, 0, 2, 0, 0, 0\rangle$
$c \equiv \langle 4, 3, 1, 3, 1, 1, 1\rangle$
$d \equiv \langle$true, true, true, true, true, true, true$\rangle$
$s \equiv \langle 3, 2, 0, 2, 0, 0, 0\rangle$
$t \equiv \langle 255, 42, 8, 2, 0, 0, 0\rangle$
IND $= 255 \times 14 + 1 = 3571$.

The procedure below assumes that **P** contains the permutation whose index is 0, and computes the index corresponding to a permutation given in PERM.

```
PROCEDURE INDFROMPERM(VAR N,INDEX:
INTEGER; P,PERM:LIST);
(* OBJECT IS A DATA TYPE DEFINED IN A *)
(* CALLING PROGRAM, LIST IS A DATA TYPE OF *)
(* THE FORM ARRAY [1 . . U] OF OBJECT, WHERE *)
(* U IS AN INTEGER ≥ N *)
VAR J,L,M,V,KPER,TI,SI,I,JI,K:INTEGER;
 KEY:LIST;
BEGIN
 J:= 0; REPEAT J:= J + 1 UNTIL PERM[J] = P[N];
 L:= 0;
 REPEAT L:= L + 1 UNTIL PERM[L] = P[N - 1];
 M:= 0;
 REPEAT M:= M + 1 UNTIL PERM[M] =P[N - 2];
 IF (M < J) AND (J < L) OR (J < L) AND (L < M) OR
  (L < M) AND (M < J)
 THEN BEGIN FOR V:= J - 1 DOWNTO 1 DO
                KEY [J - V]:= PERM[V];
              FOR V:= N DOWNTO J DO
               KEY [J + N - V]:= PERM[V];
              KPER:= N + J - 1
       END
 ELSE BEGIN FOR V:= J + 1 TO N DO
               KEY[V - J] := PERM[V];
             FOR V := 1 TO J DO
              KEY[N - J + V] := PERM[V];
              KPER: = N - J
      END;
 TI:= 0;
 FOR I:= N - 3 DOWNTO 1 DO
 BEGIN JI:= 0; K:= 0;
  REPEAT K:= K + 1;
   IF KEY[K] > P[I] THEN JI:= JI + 1
  UNTIL KEY[K] = P[I];
  IF ODD(TI) THEN SI:= N - I - 1 - JI ELSE SI:= JI;
  TI:= TI*(N - I) + SI
 END;
 INDEX:= TI*2*N + KPER
END
```

To start the generation scheme from the key of a given permutation or a given index, we begin by finding the missing index or permutation, as the case may be. At the same time we generate the key, set the values of $d(i)$ and $c(i)$ as outlined above. Assuming that the values of $X(i)$ were prefixed we then enter the main loop of the generation algorithm with $i:= 1$.

References

JOHNSON, S. M. (1963). Generation of permutations by adjacent transposition, *Math. Comp.*, Vol. 17, pp. 282–285.

KOUTAS, P. J. and HU, T. C. (1975). Shortest string containing all permutations, *Discrete Mathematics*, Vol. 11, pp. 125–132.

LANGDON, G. G. Jr (1967). An algorithm for generating permutations, *CACM*, Vol. 5, pp. 298–299.

SEDGEWICK, R. (1977). Permutation generation methods, *Computing Surveys*, Vol. 9, pp. 137–164.

TOMPKINS, C. (1956). Machine attacks on problems whose variables are permutations, *Proc. Symposium in Appl. Math. Numerical Analysis*, Vol. 6, McGraw-Hill, New York, pp. 195–211.

TROTTER, H. F. (1962). Perm (algorithm 115), *CACM*, Vol. 5, pp. 434–435.