BIT 15 (1975), 158–164

A SIMPLIFIED LOOP-FREE ALGORITHM FOR GENERATING PERMUTATIONS

NACHUM DERSHOWITZ

Abstract.

A simple loop-free algorithm for generation of all permutations of a set of elements is presented and its validity is proved. It is a simplification of Ehrlich's loop-free version of Johnson and Trotter's algorithm. Each permutation is generated by exchanging two adjacent elements of the preceding permutation. A very simple data structure obviates the need for looping during the generation of each successive permutation.

Key Words and Phrases: permutations, permutations and combinations, permutation generation, loop-free algorithms, ordering.

Introduction.

To the growing number of algorithms for generating the $n!$ permutations of n items we add yet another. The loop-free[1] T-algorithm presented here is significantly simpler than the loop-free version of an algorithm by Johnson [1] and Trotter [2] recently published by Ehrlich [3]. (See also Even [5].)

All three algorithms generate the same sequence of permutations. Each successive permutation is obtained by transposing two adjacent items of the preceding permutation. The algorithms differ in the manner in which they determine which is the pair of items to be interchanged next.

The n items are numbered $1, 2, \ldots, n$ so that we may speak of permuting a set of consecutive integers, and of some items as being larger or smaller than others.

Received November 4, 1974.

[1] A permutation algorithm is defined to be loop-free if for any set of n elements, the transition from one permutation to the next, and the test for the last permutation, require no more than a fixed number of operations (independent of n) [3].

The Johnson and Trotter Algorithm.

The following conventions and definitions will be of use (see Figures 1 and 2).

a) The current "direction"—left or right—of an integer is symbolized by an appropriate arrow above the integer.
b) An integer is said to "face" the integers in the direction of its arrow; an integer's "neighbor" is the adjacent integer it is facing. (If it faces none, it has no neighbor.)
c) An integer can be in either an "active" or an "inactive" state; when active, it is underlined.
d) The largest active integer has a double underline and is referred to as the "cursor".
e) The cursor is considered "stuck" if it has no neighbor or if its neighbor is greater than itself.

What follows is essentially the Johnson and Trotter algorithm [1, 2].

JT-algorithm.

1. initially the n integers are in their natural sequence, all facing left, and all but the smallest are active.
2. output the current permutation.
3. if no integer is active, terminate.
4. find the cursor i.
5. transpose the cursor with its neighbor.
6. reactivate all integers greater than i.
7. if i is stuck, reverse its direction and deactivate it.
8. continue with Step 2.

By induction on n, the validity of the above algorithm is proved [1]. For $n=2$ it can easily be verified (see Figure 1). Initially n is active and being the largest integer moves left through all positions until stuck with no neighbor. Now n is inactive and at an extreme position. Assuming that the algorithm works for $n-1$ integers, the next permutation of $1,2,\ldots,n-1$ is generated, with n playing no part. Then n is reactivated and passes unobstructed to the opposite extreme. The process continues, n reversing direction for each permutation of $1,2,\ldots,n-1$, until the last permutation of $1,2,\ldots,n-1$ is generated, leaving all but n inactive. After the final traversal of n through the integers, it is also deactivated and the algorithm terminates. Thus each of the $n!$ permutations is generated once and only once.

Sequence number	Permutation	Sequence number	Step 2 permutation	Step 4 cursor
			< < <	
		(0)	1 2 3	3
	< <		< < <	
(0)	1 2	(1)	1 3 2	3
			> < <	
		(2)	3 1 2	2
			> > <	
		(3)	3 2 1	3
	> <		> > <	
(1)	2 1	(4)	2 3 1	3
			> < <	
		(5)	2 1 3	–
(*a*)	$n = 2$	(*b*)	$n = 3$	

Figure 1. Examples of the JT-algorithm for $n=2$ and $n=3$. Active integers are underlined.

Improved Algorithm.

Note that the cursor follows a definite pattern. For $n=4$ (see Figure 2) the cursor takes on the values: 44434443444244434443444. 4 repeats thrice until getting stuck, while 3 repeats twice; once 3 is stuck, 2 will follow 4 the next time 4 gets stuck. In general, if i is stuck, the next largest active integer j, $j<i$, will replace i as the cursor when again all k, $k>i$, are stuck. After j is moved, the pattern of cursors greater than j repeats itself.

This provides the motivation for designing a data structure which will keep track of the cursor (by keeping track of all active integers), thereby eliminating the need to search for it (through a maximum of n elements). At the same time, the reactivation loop of Step 6 will be removed. We pay with the updating of the structure.

Accordingly, the T-algorithm below employs a vector $T=(t_2,t_3,\ldots,t_{n+1})$ within which the linked list $t_{n+1},t_{t_{n+1}},t_{t_{t_{n+1}}},\ldots$ is embedded. We shall prove in the next section that this is a list of the active integers in descending order, and it follows that t_{n+1} is the cursor. In all other respects the T-algorithm parallels the JT-algorithm.

T-algorithm.

1. initially the n integers are in their natural sequence, facing left, and for $j=2,3,\ldots,n+1$: $t_j=j-1$.
2. output the current permutation.
3. if $t_{n+1}<2$, terminate.

4. set the cursor: $i \leftarrow t_{n+1}$.
5. transpose the cursor with its neighbor.
6. set: $t_{n+1} \leftarrow n$.
7. if i is stuck,
   reverse its direction and set:
   a) $t_{i+1} \leftarrow t_i$
   b) $t_i \leftarrow i-1$.
8. continue with Step 2.

Figure 2 illustrates the algorithm for $n=4$. Arrows are shown pointing from t_j to t_{t_j}, except for $t_j=1$ which points to the square at the left. When $t_{n+1}=1$ the algorithm terminates.

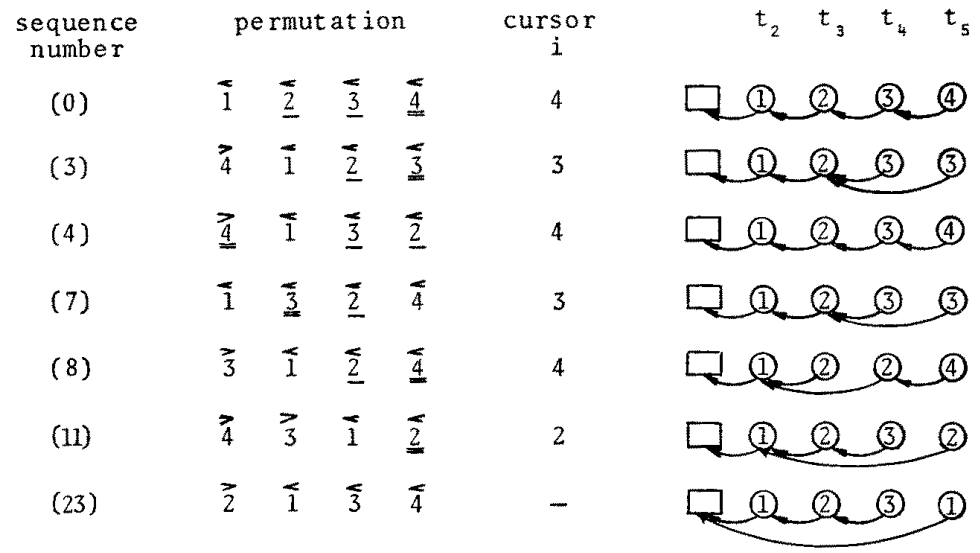

Figure 2. Selected stages of the T-algorithm for $n=4$. The arrows under the list T are added in order to emphasize T's structure.

Validity.

Before showing the equivalence of the T and JT algorithms, we prove two lemmas by induction on the execution of the T-algorithm. They are shown to be true upon initialization and that once true, they remain true after executing Steps 6 and 7, the only steps that alter T. For notational convenience, new values are primed. Step 6 becomes: $t'_{n+1} \leftarrow n$ and for Step 7 we have

a) $t'_{i+1} \leftarrow t_i$
b) $t_i' \leftarrow i-1$.

LEMMA 1. *$t_j < j$ for $j=2,3,\ldots,n+1$.*

PROOF. Initially $t_j = j-1 < j$.

Hypothesizing $t_j < j$, the execution of Steps 6 and 7 results in:

Step 6. $t'_{n+1} = n < n+1$

Step 7. a) $t'_{i+1} = t_i < i < i+1$

b) $t_i' = i-1 < i$.

LEMMA 2. *For $k = 2,3,\ldots,n+1$, if j satisfies $t_k < j < k$, then $t_j = j-1$.*

PROOF. Initially all $t_k = k-1$ and the lemma is trivially true.

Assuming that $t_k < j < k$ implies $t_j = j-1$, executing Steps 6 and 7 yields:

Step 6. $t'_{n+1} = n$ and the lemma holds.

Step 7. a) for j such that $t'_{i+1} = t_i < j < i$, we have, by assumption, $t_j' = t_j = j-1$, and

b) for $j = i$, $t_j' = j-1$.

Together, a) and b) give: for j, $t'_{i+1} < j < i+1$, $t_j = j-1$. Since there is no k such that $t_k < i+1 < k$, setting t'_{i+1} has no other effect.

THEOREM. *The ordered set S defined by the T-algorithm, where $S = \{t_{n+1}, t_{t_{n+1}}, t_{t_{t_{n+1}}}, \ldots, t_l\}$, $t_{n+1} > t_{t_{n+1}} > \ldots > t_l$, and l is such that $t_l \geqq 2$ but $t_{t_l} < 2$, is equal to the set A of active integers, as defined by the JT-algorithm. t_{n+1} is therefore the cursor.*

PROOF. Initially $S = \{t_{n+1} = n, t_n = n-1, \ldots, t_3 = 2\} = A$.

Assume that at some point in the execution $S = A \neq \emptyset$ and the cursor $i = t_{n+1}$.

We show that Step 6 implements the activation of all integers greater than i. Let $A' = A \cup \{i+1, i+2, \ldots, n\}$. The preceding lemma ensures that $t_j = j-1$ for $i = t_{n+1} < j < n+1$. Setting $t'_{n+1} = n$ results in

$$S' = \{t'_{n+1} = n, t_{t'_{n+1}} = t_n = n-1, t_{n-1} = n-2, \ldots, t_{i+2} = i+1, t_{i+1} = i = t_{n+1}, t_i, t_{t_i}, \ldots, t_l\}.$$

For all j $(i < j \leqq n)$, $j \in S'$, and $S' = A'$.

Step 7 removes i from S'. By Lemma 1, $j > t_j$ and $t_{n+1} > t_{t_{n+1}} > \ldots > t_l$. Setting $t'_{i+1} = t_i < i$ gives, in strictly descending order:

$$S'' = \{t'_{n+1} = n, t_n = n-1, \ldots, t_{i+2} = i+1, t'_{i+1} = t_i, t_{t_i}, \ldots, t_l\} = A' - \{i\} = A''.$$

(For $i = n$, $S'' = \{t'_{n+1} = t_i, t_{t_i}, \ldots, t_l\}$.) Since $i \notin S''$, Step 7. b) $t_i' \leftarrow i-1$ leaves S'' unaltered. This completes the proof.

Thus the T-algorithm is equivalent to the JT-algorithm whose validity has already been shown.

Efficiency.

The TN-algorithm given in the Appendix is a modification of the T-algorithm. It takes advantage of the fact that n is by far the most frequent cursor and deals separately with that case (with a corresponding loss in elegance), as does Ehrlich's fast permutation generator, PERMU [4]. Ehrlich [private communication], by separating $n, n-1$ and $n-2$, found his generator to run twice as fast as the next fastest algorithm. Revising PERMU to incorporate the mechanism of the TN-algorithm, by removal of three comparisons and one assignment, and reordering the steps, yields a shorter and simpler program, with no loss in speed.

In the worst case, PERMU executes six comparisons and twenty assignments for one permutation; the revised version requires only three of the six comparisons and eighteen of the assignments. In the best and most frequent case, the two algorithms are equivalent.

Appendix.

This Appendix contains a variation of the T-algorithm which, following Ehrlich [3, 4], takes advantage of the fact that the largest integer n is active $(n!-(n-1)!)/n!=(n-1)/n$ of the time.

TN-algorithm.

1. initially the n integers are in their natural sequence, facing left, for $j=2,3,\ldots,n$: $t_j=j-1$, and the cursor $i=n$.
2. output the current permutation.
3. transpose the cursor with its neighbor.
4. if the cursor i is not stuck, reset the cursor: $i \leftarrow n$.
5. if the cursor i is stuck and $i=n$, then reverse n's direction,
   set: $i \leftarrow t_n$
   $t_n \leftarrow n-1$
   test: if $i<2$, output and terminate.
6. if the cursor i is stuck and $i<n$, then reverse i's direction and set:
   $t_{i+1} \leftarrow t_i$
   $t_i \leftarrow i-1$
   $i \leftarrow n$.
7. continue with Step 2.

Ehrlich [3] also suggests dealing separately with transposition for $i=n$.

Acknowledgement.

I wish to thank Professor Shimon Even for introducing me to the problem and for continued encouragement and guidance.

REFERENCES

1. S. M. Johnson, *Generation of permutations by adjacent transpositions*, Math. Comput. 17 (1963), 282–285.
2. H. F. Trotter, *Algorithm 115, Perm.* Comm. ACM 5, 8 (Aug. 1962), 434–435.
3. G. Ehrlich, *Loopless algorithms for generating permutations, combinations, and other combinatorial configurations*, J. ACM 20, 3 (July 1973), 500–513.
4. G. Ehrlich, *Algorithm 466, Permu.* Comm. ACM 16, 11 (Nov. 1973), 690–691.
5. S. Even, *Algorithmic Combinatorics*, Macmillan, New York, 1973, pp. 2–11.

WEIZMANN INSTITUTE OF SCIENCE
REHOVOT, ISRAEL