A NEW ALGORITHM
FOR GENERATION OF PERMUTATIONS

by

Shmuel Zaks

Technical Report # 220

September 1981
ABSTRACT

A new algorithm for generating permutations is presented, that generates the next permutation by reversing a certain suffix of its predecessor. The average size of this suffix is less than 2.8. It is also shown how to find the position of a given permutation and how to construct the permutation of a given position, where the position refers to the order in which the permutations are generated, and is also new.
1. INTRODUCTION

A new algorithm for generating the \( n! \) permutations of the numbers \( 1, 2, \ldots, n \) is presented. It generates the permutations in a new order. For literature about the subject the reader may consult Even [4], Reingold et al. [6], and the survey paper by Sedgewick [7].

The algorithm mentioned in Johnson [5], Trotter [8], Ehrlich [2,3] and Dershowitz [1] generates each successive permutation by transposing two adjacent items of the preceding permutation; these four algorithms differ only in the manner in which they determine the candidate pair of elements to be exchanged. In our algorithm each successive permutation is generated by reversing a suffix of the preceding permutation. We show how the size of this suffix can be determined in constant time; also, the ranking and unranking algorithms are quite simple.

The idea to this new algorithm for generating permutations comes from the "stack of pancakes" folk problem (see [9]): A waiter carries in his left hand a plate stacked with \( n \) pancakes of \( n \) different sizes; in how many steps can he arrange them on the plate - from largest to smallest - where in each step he can take any number of pancakes from the top - using his right hand - and turn them over? Using our algorithms the poor waiter will be able to generate, in \( n! \) such steps, all possible \( n! \) stacks (returning to the original one), to tell which stack will be the \( 1 \)-th, and to tell when a given stack will show. Moreover, in \( 1/2 \) of these steps he will reverse the top 2 pancakes, in \( 1/3 \) of them - the top 3, and, in general, in \( \frac{k-1}{k!} \) of them he will reverse the top \( k \) pancakes, which amounts to less than 2.8 pancakes reversed on the average.

The generating procedure is discussed in Section 2, the ranking and unranking procedures are the subject of Section 3, and Section 4 shows an efficient way of generating the sequence of suffix sizes.
2. THE GENERATING ALGORITHM

We start with the permutation $1, 2, \ldots, n$ and in each step reverse a certain suffix. The sequence of sizes of these suffixes is denoted by $s_n$, and is defined recursively as follows (a sequence is written as a concatenation of its elements):

$$s_2 = 2;$$

$$s_n = (s_{n-1})^{n-1} s_{n-1} \quad n > 2. \quad (*)$$

Examples

$n = 2 \quad s_2 = 2$

$$1 \quad 2 \quad 2 \quad 1$$

$n = 3 \quad s_3 = 23232$

$$1 \quad 2 \quad 3 \quad 2 \quad 3 \quad 1 \quad 3 \quad 1 \quad 2$$

$$1 \quad 3 \quad 2 \quad 2 \quad 1 \quad 3 \quad 3 \quad 2 \quad 1$$

$n = 4 \quad 232324232423242324232423242324232$

$$1 \quad 2 \quad 3 \quad 4 \quad 2 \quad 3 \quad 4 \quad 1 \quad 3 \quad 4 \quad 1 \quad 2 \quad 4 \quad 3 \quad 2 \quad 1 \quad 4 \quad 3 \quad 1 \quad 2 \quad 4 \quad 3 \quad 2 \quad 1 \quad 4 \quad 3 \quad 2 \quad 1$$

Proof of validity

We prove by induction on $n$ that, starting with the permutation $1, 2, \ldots, n$ and applying the sequence $s_n$ of suffix reversals, we generate all $n!$ permutations ending with the permutation $n, n-1, \ldots, 1.$
The assertion holds for $n = 2$. Assuming it holds for $n-1$, we prove it for $n$: $s_n$ starts with $s_{n-1}$. Hence, starting with $1,2,\ldots,n$, we first generate the $(n-1)!$ permutations that start with $1$ and, by the induction hypothesis, the last one is $1,n,n-1,\ldots,2$.

The next element in $s_n$ is $n$, hence the next permutation generated is $2,3,\ldots,n,1$. The following elements thus generated are all permutations starting with $2$. Continuing in this manner, we then generate the permutations starting with $3,\ldots,n$.

Moreover, because the first permutation starting with $2 = 2,3,\ldots,n,1$ is obtained from the first permutation starting with $1 = 1,2,\ldots,n$ by increasing each element by $1$ (while $n$ becomes $1$), and because both the permutations starting with $1$ and those starting with $2$ are generated by the sequence $s_{n-1}$, therefore the last permutation starting with $2$ is derived from $1,n,n-1,\ldots,2$ (last permutation starting with $1$) in the same manner, namely, it is $2,1,n,\ldots,3$. Continuing in this manner it is easy to show that the first permutation starting with $i$, $1 < i < n$, is $i,i+1,\ldots,n,1,2,\ldots,i-1$ and the last one is $i,i-1,\ldots,1,n,n-1,\ldots,i+1$ (it is $n,n-1,\ldots,1$ for $i = n$).

The proof is thus completed.

The algorithm for generating the permutations goes as follows:

**Generation Algorithm**

\[
p_1p_2\ldots p_n := 1,2,\ldots,n \quad \text{(start)}
\]

repeat

\[
i := \text{next element of } s_n \quad \text{(starting with the first one)}
\]

\[
p_{n-i+1}\ldots p_n := p_n \ldots p_{n-i+1}
\]

\[
\text{(reverse last } i \text{ elements)}
\]

until $i$ is the last element of $s_n$.
As for the operation of reversing a suffix: first, if one is using a permutation network, this operation can be easily implemented on such a network by \[
\left\lfloor \frac{1}{2} \right\rfloor \text{ "exchange modules" (see Sedgewick [7]) connecting } n+1-t \text{ and } n-1+t \text{ for } t = 1, 2, \ldots, \left\lfloor \frac{1}{2} \right\rfloor \]; second, if one is interested in outputting the permutations sequentially, this will take an \(O(n)\) time anyhow.

**Lemma:** The average size of a suffix reversed by the generation algorithm is less than 2.8.

**Proof:** Suppose we apply the sequence of suffix reversals \(s_n\) (starting and ending with the permutation \(1, 2, \ldots, n\)). \(n\) appears in this sequence \(\frac{n!}{(n-1)!} = n\) times. \(n-1\) appears in the positions that are multiples of \((n-2)!\) but not of \((n-1)!\), namely \(\frac{n!}{(n-2)!} - \frac{n!}{(n-1)!}\) times.

By an easy induction the number \(i, 2 \leq i < n\), appears in the sequence \(\frac{n!}{i!} - \frac{n!}{(i+1)!}\) times. Therefore, the expected suffix size is

\[
\frac{1}{n!} \left[ n \cdot \frac{n!}{(n-1)!} + \sum_{i=2}^{n-1} \left( \frac{n!}{i!} - \frac{n!}{(i+1)!} \right) \right] < \frac{n!}{(n-1)!} + \sum_{i=2}^{n-1} \frac{1}{(i-1)!} = \frac{1+(n-1)}{(n-1)!} + \sum_{i=1}^{n-2} \frac{1}{i!} < \sum_{i=1}^{n} \frac{1}{i!} < e < 2.8. 
\]

\(\square\)
3. RANKING AND UNRANKING

We first show how to find the position \( \text{index}(p) \) of a given permutation \( p = p_1p_2 \ldots p_n \). If \( p_1 > 1 \) then all \( (p_1-1)(n-1)! \) permutation starting with \( 1, 2, \ldots, p-1 \) precede it. Therefore, \( \text{index}(p) = (p_1-1)(n-1)! + \text{position of } p \text{ among the permutations starting with } p_1. \)

But, following the previous proof of validity, the position of \( p \) among the permutations starting with \( p_1 \) is equal to

\[
\text{index}(p_2-p_1, p_3-p_1, \ldots, p_n-p_1),
\]

where the subtraction is modulo \( n \). Note that the new permutation contains \( n-1 \) elements. (and we use the same function name to rank these permutations), and therefore, the following algorithm is valid:

**Ranking Algorithm**

\[
\text{index}(1) = 1 \\
\text{index}(p_1p_2 \ldots, p_n) = (p_1-1)(n-1)! + \text{index}(p_2-p_1, p_3-p_1, \ldots, p_n-p_1)
\]

with subtraction mod \( n \).

**Example:**

\[
\text{index}(3 4 2 1) = 2 \cdot 3! + \text{index}(1 3 2) \\
= 2 \cdot 3! + 0 \cdot 2! + \text{index}(2 1) \\
= 2 \cdot 3! + 0 \cdot 2! + 1 \cdot 1! + \text{index}(1) \\
= 2 \cdot 3! + 0 \cdot 2! + 1 \cdot 1! + 1 \\
= 14
\]

As for unranking, it is done in a reverse order. We look for a permutation \( p = p_1, \ldots, p_n \) such that \( \text{index}(p) = t \), for a given \( t \) and \( n \). For this we first decompose \( t \) as
\[ t = 1 + a_1 \cdot 1! + a_2 \cdot 2! + \ldots + a_{n-1} \cdot (n-1)! \]

where \(0 \leq a_i \leq 1\) (see Reingold et al. [6]).

We then proceed as follows:

**Unranking Algorithm**

\[
\begin{align*}
p_1 &= 1; \\
\text{for} \ j = 1 \text{ to } n-1 \text{ do} \\
\text{begin} \\
\text{for} \ k = j+1 \text{ to } 2 \text{ do} \ p_k := (p_k + a_j) \mod (j+1); \\
p_1 := a_j + 1; \\
\text{end;}
\end{align*}
\]

**Example:** To find the 14th permutation for \(n = 4\), we first have:

\[14 = 1 + 1 \cdot 1! + 0 \cdot 2! + 2 \cdot 3!\]

We then set \(p_1 = 1\).

- for \(j = 1\), \(a_1 = 1\), we get \(p_1p_2 = 21\)
- for \(j = 2\), \(a_2 = 0\), we get \(p_1p_2p_3 = 132\)
- for \(j = 3\), \(a_3 = 2\), we get \(p_1p_2p_3p_4 = 3421\)

Both the ranking and the unranking algorithms are of \(O(n^2)\),

when each basic operation requires a constant time.
4. EFFICIENT GENERATION OF $s_n$

It is simple to generate the elements of $s_n$ recursively, following (\*). Using the explanation in Section 3, an iterative algorithm might be proposed. Suppose the variable $x$ holds the current element of $s_n$, and the variable $\text{count}[i], \ 3 \leq i \leq n$, holds the number of times the variable $i$ has appeared since the last time $i+1$ appeared; when $x$ gets the value $i+1$, $\text{count}[i]$ is set to 0. It follows that $\text{count}[i]$ gets the values $0, 1, \ldots, i-1, 0, 1, \ldots, i-1, \ldots$ When $x$ gets a value $i$, all previous counts, namely $\text{count}[j]$ for $j < i$, has to be set to 0, and this operation is proportional in time to $j$ (i.e., it is of $O(n)$ time). Since the expected value of $j$ is less than 2.8 (see Section 3), this algorithm will behave quite well on the average. We leave its details to the reader. A more efficient algorithm, that generates the next elements of $s_n$ in $O(1)$ time and $O(n)$ space is now presented. (A similar idea is found in Dershowitz [1].)

$x$ is the current value of $s_n$, initialized to 0.

$\text{follow}[i]$ is the number that will be generated in the next position which is a multiple of $i$, initialized to $i+1, 2 \leq i \leq n$.

$\text{count}[i]$ is the number of times $i$ has appeared since the last time $i+1$ appeared, $3 \leq i \leq n-1$.

$\text{count}[n]$ is the number of times $n$ has been generated, and $\text{count}[n+1]$ is a sentinel.
**NEXT Algorithm**

1. if $x \neq 2$ then $x := 2$

   else begin
2. $x := \text{follow}[2]$;
3. $\text{follow}[2] := 3$;
4. $\text{count}[x] := \text{count}[x] + 1$;
5. if $\text{count}[x] = x - 1$ then begin
6. $\text{count}[x] := 0$;
7. $\text{follow}[x-1] := \text{follow}[x]$;
8. $\text{follow}[x] := x + 1$
      end
   end

**Lemma:** The program

$x := 0$;

while $x \leq n$ do begin output$(x)$; NEXT end;

outputs the sequence $s_n$.

**Proof:** We show by induction on $n$, $n \geq 2$, the following: the above program outputs $s_n$, and when it stops we have $x = n + 1$, $\text{follow}(i) = i + 1$ for all $i$, $\text{count}(n+1) = 1$, and $\text{count}(i) = 0$ for all other $i$, which we term a fine configuration.

$n = 2$: it can be checked that in this case the program outputs $s_2 = 2$, and the final configuration is fine (see Figure 1).

![Figure 1 - Final configuration for $n = 2$.](image)

(follow[i] is denoted by an arrow).
Suppose the program outputs \( s_{n-1} \) for \( n-1 \) with a fine final configuration as described in Figure 2.

![Figure 2 - Final configuration for n-1.](image)

Now, as for \( n \): when first \( x = n \) we have, in the beginning of the while loop, \( x = n \), with \( s_{n-1} \) output so far, and a configuration as described in Figure 3.

![Figure 3 - Configuration for n, after generating the prefix \( s_{n-1} \).](image)

Now \( s_{n-1} \) is generated again, and in the last statement (line 4) we have \( \text{count}[n] = 2 \), therefore we have generated so far \((s_{n-1} n)^{n-1}\) with a final configuration like in Figure 3, with the only difference that \( \text{count}[n] = 2 \) rather than 1. After \( n \) has been generated \( n-1 \) times, with \( n \) have generated so far \((s_{n-1} n)^{n-1}\). Then in line 5 we have \( \text{count}[n] = n-1 \), and the changes made in lines 6-8 yield the configuration in Figure 4.

![Figure 4 - Configuration for n, after generating the prefix \( (s_{n-1} n)^n \).](image)
Now, by the induction hypothesis, the algorithm will generate $s_{n-1}$, stopping when $x = n+1$, and therefore we have generated $(s_{n-1})^{n-1}s_{n-1}$, with $x' = n+1$ at the end, which stops the while loop. It is also an easy matter to verify that the final configuration is fine. $\square$
REFERENCES


