Position Method: A Linear-Time Generation Algorithm for Permutations

Yusheng Hu
Independent Researcher, Beijing, 100085, China
dr.huyusheng@gmail.com

January 12, 2026

Abstract

This paper proposes a new permutation mapping algorithm based on the "Position Method", establishing a one-to-one correspondence between the factorial number system and permutations. Both ranking (calculating rank from permutations) and unranking (generating permutations from rank) achieve O(n) time and space complexity. Experimental results show that its efficiency is slightly better than the Myrvold-Ruskey [1] algorithm.

1 Introduction

A permutation refers to all ordered arrangements of n distinct elements. Although algorithms such as **Heap's algorithm** [5] and **Ives's algorithm** [6] efficiently generate permutations, they are limited to generation capabilities. Algorithms with bidirectional mapping capabilities for ranking and unranking, due to their support for parallel processing, demonstrate greater practical value in engineering applications. The mapping between factorial number system and permutations is the core technical pathway for achieving this bidirectional capability. This paper proposes a specific implementation scheme based on the factorial number system—the Position algorithm.

Reverse Factoradic Representation The reverse factoradic system (also known as the little-endian factorial number system) is a mixed-radix representation used to encode permutations, particularly for the Myrvold-Ruskey algorithm. A non-negative integer K is uniquely represented by a sequence of digits $\mathbf{c} = (c_0, c_1, \ldots, c_{n-1})$, where the i-th digit c_i (at 0-based index i) is associated with a base of $i + 1$ and satisfies the constraint:

$$0 \leq c_i \leq i \quad \text{for } i = 0, 1, \ldots, n - 1$$

Unlike the standard Lehmer code, the weights in this system increase from left to right. The value of K is given by:

$$K = \sum_{i=0}^{n-1} c_i \cdot (i!) = c_0 \cdot 0! + c_1 \cdot 1! + c_2 \cdot 2! + \cdots + c_{n-1} \cdot (n-1)!$$

In this notation, the sequence $(0,0,0,0)$ represents the first permutation (Rank 0), and the sequence grows by incrementing the rightmost digit first, e.g., $(0,0,0,1),(0,0,0,2),\ldots,(0,1,2,3)$, directly corresponding to the control vector required for linear-time unranking via swap operations.

Note: In this paper, permutations are ranked using the *reverse factoradic representation* (vector form), where the rank is maintained as a coefficient array $\mathbf{c} = (c_0, c_1, \ldots, c_{n-1})$ satisfying $0 \le c_i \le i$, rather than as a single large integer.

This paper presents an algorithm named **Position** with a time complexity of O(N), along with its improved variant, **Position pure**.

2 Position Algorithm

2.1 Algorithm Principles

Assume that the data for permutations in the algorithm starts from 0. Let P_n denote the set of all permutations of n elements. The permutation of 1 element is $P_1 = \{0\}$.

Derivation process for permutations of 2 elements: The permutation of 1 element is $\{0\}$, denoted as P1. When extending to 2 elements, a new position (position 1) is added, which can take values (0, 1).

By appending these values to P_1, we get $\{P_1, 0\}$ and $\{P_1, 1\}$, which are expressed as follows:

$$(\mathrm{P}_1)\begin{pmatrix}0\\1\end{pmatrix} = \begin{cases}\{\mathrm{P}_1,\ 0\} = \{\{0\},\ 0\} & \text{——(1)}\\ \{\mathrm{P}_1,\ 1\} = \{\{0\},\ 1\} & \text{——(2)}\end{cases}$$

The {0} in (1) is 1 element permutation, because the appearance of 0 afterwards, then change {0} to the **position** , that is {1}

There is no conflict in the formula (2) , therefore keep {0,1)}

Thus, the full permutation of 2 elements is obtained

$$\begin{pmatrix}0 \rightarrow 1 & 0\\ 1 & 1\end{pmatrix} \xrightarrow{\textit{After replace}} \begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix}$$

Therefore, the full permutation of 3 elements is generated based on the full permutation of 2 elements:

$$\begin{pmatrix}(P_2) & 0\\(P_2) & 1\\(P_2) & 2\end{pmatrix} \xrightarrow{\text{assume } P_2=\begin{pmatrix}1&0\\0&1\end{pmatrix} \text{ is known}} \begin{pmatrix}\begin{pmatrix}1&0\\0&1\end{pmatrix} & 0\\ \begin{pmatrix}1&0\\0&1\end{pmatrix} & 1\\ \begin{pmatrix}1&0\\0&1\end{pmatrix} & 2\end{pmatrix}$$

Replace element in P_2 that have already appeared to 2, also as n-1,

$$\begin{pmatrix}1&0\\0&1\end{pmatrix}\begin{pmatrix}\begin{pmatrix}1 & 0\to 2\\ 0\to 2 & 1\end{pmatrix} & 0\\ \begin{pmatrix}1\to 2 & 0\\ 0 & 1\to 2\end{pmatrix} & 1\\ \begin{pmatrix}1&0\\0&1\end{pmatrix} & 2\end{pmatrix} \xrightarrow{replace} \begin{pmatrix}\begin{pmatrix}1&2\\2&1\end{pmatrix}&0\\ \begin{pmatrix}2&0\\0&2\end{pmatrix}&1\\ \begin{pmatrix}1&0\\0&1\end{pmatrix}&2\end{pmatrix}\xrightarrow{final}\begin{pmatrix}1&2&0\\2&1&0\\2&0&1\\0&2&1\\1&0&2\\0&1&2\end{pmatrix}$$

the general generation method for the complete permutation P_n is

$$P_n=\begin{pmatrix}(P_{n-1}) & 0\\(P_{n-1}) & ...\\(P_{n-1}) & n-1\end{pmatrix}\xrightarrow{P_{n-1}\text{ is known}}\begin{pmatrix}\begin{pmatrix}0\to n-1 & ... & ...\\ ... & ... & ...\\ ... & ... & 0\to n-1\end{pmatrix} & 0\\ \begin{pmatrix}1\to n-1 & ... & ...\\ ... & ... & ...\\ ... & ... & 1\to n-1\end{pmatrix} & 1\\ ... & ...\\ (P_{n-1}) & n-1\end{pmatrix}$$

The replacement rule for the Position algorithm is, for step i, find former value same as C[i], then replace by the position value i.

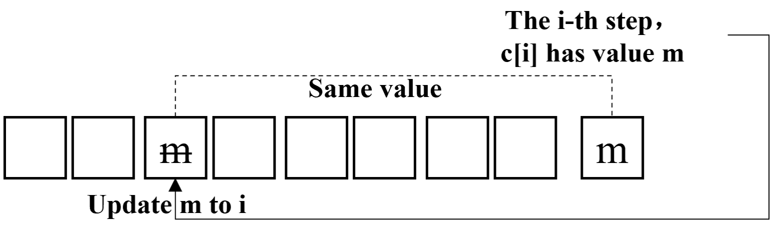

Figure 1: Replacement rule for the Position algorithm

2.2 Position algorithm and complex

The pseudocode for rank and unrank of the Position algorithm is provided below

Algorithm 1 Position unrank

```
Require: $C$, $n$
Ensure: $D$
1: for $i \leftarrow 0$ to $n-1$ do
2:     $D[i] \Leftarrow C[i]$
3:     $M[i] \Leftarrow M[C[i]]$
4:     $M[C[i]] \Leftarrow i$
5:     $D[M[i]] \Leftarrow i$
6: end for
```

Algorithm 2 Position rank

```
Require: $D$, $n$
Ensure: $C$
1: for $i \leftarrow 0$ to $n-1$ do
2:     $C[i] \Leftarrow D[i]$
3:     $M[C[i]] \Leftarrow i$
4: end for
5: for $i \leftarrow$n-1 down to 0 do
6:     $C[M[i]] \Leftarrow C[i]$
7:     $M[C[i]] \Leftarrow M[i]$
8: end for
```

- **Space Complexity** The algorithm uses an intermediate array $M[n]$ for computation, resulting in a space complexity of $O(n)$.

- **Time Complexity** To generate a permutation, the algorithm executes two loops each of length n. Thus, the time complexity is $O(n)$.

- **Algorithm Recursion** The complexity for generating a single permutation is $O(n)$. If permutations are output consecutively, generating the next full permutation can reuse most information from the previous one. Therefore, when generating all permutations, the complexity per permutation can be reduced to $O(1)$. The overall complexity of generating all permutations (excluding output) is $O(n!)$.

- **Parallel Algorithm** Since there is a strict one-to-one correspondence between the generation of full permutations and factoradic numbers, permutations can be generated by enumerating the factoradic numbers. The set of factoradic numbers can also be partitioned into subsets for parallel execution by specifying a start and end factoradic number for each task. This allows the task to be decomposed into parallel subtasks that can be executed across multiple devices, multiple CPU cores within a device, or on a device's GPU, thereby significantly improving the efficiency of full permutation generation. Partitioning is typically performed by fixing the first few digits of the factoradic number; specific details are omitted here.

2.3 Proof of Recursive Completeness

Assume that the permutation P_{n-1} of $n-1$ numbers is a complete permutation, i.e., it contains all permutations of the elements $\{0, 1, \ldots, n-2\}$, with a total of $(n-1)!$ permutations.

When recursively generating the n-element permutation P_n according to the algorithm proposed in this paper, a new element is appended to the end of each permutation in P_{n-1}. The value of this element ranges over $\{0, 1, \ldots, n-1\}$, and the specific recursive rule is:

$$P_n = P_{n-1} \cup \{x\}, \quad x \in \{0, 1, \ldots, n-1\}$$

Further derivation:

1. A single complete $(n-1)$-element permutation P_{n-1} can be extended to n distinct n-element permutations.

2. Since P_{n-1} contains $(n-1)!$ permutations, the total number of n-element permutations obtained after extension is:

$$(n-1)! \times n = n!$$

3. $n!$ is exactly the total number of full permutations of the n elements $\{0, 1, \ldots, n-1\}$. Therefore, the recursively generated P_n is a complete n-element permutation.

In conclusion, using the recursive rule of "appending a new element ranging from 0 to $n-1$ at the end" allows generating a complete set of n-element permutations P_n from a complete set of $(n-1)$-element permutations P_{n-1}. The recursive completeness of the algorithm is thus proven.

If the newly added digit is $n-1$, since P_{n-1} is a complete permutation set containing $(n-1)!$ distinct permutations, it directly forms $(n-1)!$ different new permutations.

When the newly added digit is among $0, \ldots, n-2$, the proof proceeds as follows: Different last digits inevitably produce different permutations. What remains to be proved is that permutations ending with the same digit m do not repeat.

It is known that there are $(n-1)!$ distinct permutations preceding m. According to the algorithm, every occurrence of the value m in the set P_{n-1} is replaced by $n-1$. Since $n-1$ does not appear in P_{n-1}, replacing m cannot cause a duplication of permutations. Therefore, after applying the algorithm, all permutations ending with suffix m retain the same number as P_{n-1}, i.e., $(n-1)!$ distinct permutations.

Since the last digit can take values from 0 to $n-1$, that's n possibilities in total. Hence, the total number of permutations in the new set P_n is n times P_{n-1}. i.e., $n \cdot (n-1)! = n!$. Proven.

3 Position pure Algorithm

Acceleration Approach It is observed that the $(n-1)$-th position needs to be swapped with the digits at positions $0, \ldots, n-2$. The swapping rule is applied when the digit at the $(n-1)$-th position already appears among the preceding digits. Since the digit $n-1$ itself occurs at position $n-1$, no swap is required at that stage. Ignoring the exact positions and considering sequential swapping in the order $0, \ldots, n-1$, we have:

Assume an existing $(n-1)$-permutation: $D[0]D[1]\ldots D[n-2]$. When extending it by appending $D[n-1]$ (where $D[n-1] \in \{0, \ldots, n-1\}$), the procedure is as follows:

Table 1: All possible exchange processes for the (n-1)th position

first n-2 position	**last position**	**process procedure**
[n-1]$D[1]\ldots D[n-2]$	$D[0]$	$D[n-1] \leftarrow D[0]$, $D[0] \leftarrow [n-1]$
$D[0]$ **[n-1]**$\ldots D[n-2]$	$D[1]$	$D[n-1] \leftarrow D[1]$, $D[1] \leftarrow [n-1]$
$\ldots$	$\ldots$	$\ldots$
$D[0]D[1]\ldots$ **[n-1]**	$D[n-2]$	$D[n-1] \leftarrow D[n-2]$, $D[n-2] \leftarrow [n-1]$
$D[0]D[1]\ldots D[n-2]$	$[n-1]$	$D[n-1] \leftarrow [n-1]$

Acceleration Strategy Therefore, an acceleration strategy can be employed in the process: From the 0-th position up to the $(n-2)$-th position, assign the value $n-1$ to each corresponding location in turn and change the element originally at that location to $n-1$. That is, first set $D[n-1] = n-1$, which itself constitutes a permutation; then successively interchange $D[i]$ (for $i = 0, \ldots, n-2$) with $D[n-1]$ to produce $n-2$ additional permutations. In this way a total of $n-1$ permutations are generated directly in a loop, without needing to explicitly check any positions.

New Mapping Insights from the Acceleration Approach The previous analysis reveals that the newly introduced element $n-1$ can occupy exactly $n-1$ possible positions when being inserted into the permutation. This coincides precisely with the range of values for the $(n-1)$-th digit of the factorial-number storage (denoted as C). Leveraging this correspondence, a mapping algorithm can be developed: at each insertion step, $k \in \{0, 1, \ldots, n-1\}$ chooses a position according to digit $C[k]$.

Assuming an $(n-2)$-order permutation $D[0] \ldots D[n-2]$ is given, to obtain an $(n-1)$-order permutation with $n-1$ inserted, we set $k = n-1$ and do: take $D[C[n-1]]$ and assign it to $D[n-1]$, then set $D[C[n-1]] \leftarrow n-1$.

This algorithm is both concise and elegant: each round of the loop selects an element from the existing permutation to be swapped out, and places k into the swapped position. Thus each permutation can be obtained by performing exactly $(n-1)$ swaps starting with $[0, 1, \ldots, n-1]$. Its time complexity is clearly $O(n)$, which maintains high efficiency.

algorithm: for the step i, C[i]=m, assign the m-th position C[m] value to C[i], then C[m] set to i.

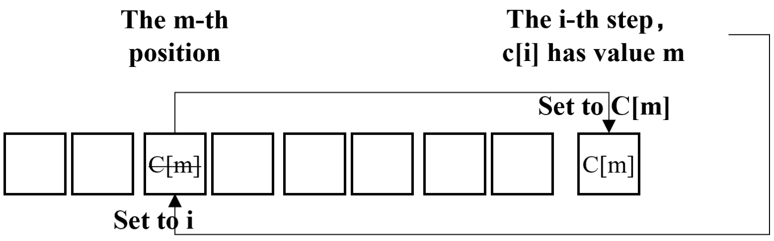

Figure 2: Replacement rule for the Position pure algorithm

pseudo-code for its **rank** and **unrank** operations is as follows:

Algorithm 3 PositionPure unrank

```
Require: $C$, $n$
Ensure: $D$
1: for $i \leftarrow 0$ to $n-1$ do
2:     $D[i] \Leftarrow D[C[i]]$
3:     $D[C[i]] \Leftarrow i$
4: end for
```

Algorithm 4 PositionPure rank

```
Require: $D$, $n$
Ensure: $C$
1: for $i \leftarrow 0$ to $n-1$ do
2:     $M[D[i]] \Leftarrow i$
3: end for
4: for $i \leftarrow$n-1 down to 0 do
5:     $C[i]] \Leftarrow M[i]$;
6:     $D[M[i]] \Leftarrow D[i]$;
7:     $M[D[i]] \Leftarrow M[i]$;
8: end for
```

4 summary

This paper presents a generation algorithm and its improved version for converting n-ary numbers into permutations. Both new algorithms achieve a time complexity of O(n) . The implementation code is concise and intuitive with a fixed number of lines and no redundancy.

Future Work: The **Position** and **Position pure** algorithms proposed in this paper, after iterative optimization, can serve as effective permutation generation methods. However, due to the inherent overhead of factorial number carry handling, their generation efficiency is slightly lower than that of pure swap-based algorithms. Further research is required to optimize the efficiency of permutation generation using mapping-based algorithms.

References

[1] Wendy Myrvold and Frank Ruskey. Ranking and unranking permutations in linear time. *Information Processing Letters*, 79(6):281–284, 2001.

[2] H. R. Parks and D. C. Wills. Improved linear-time ranking of permutations. *J. Appl. Math. Comput.*, 5(4):277–282, 2021.

[3] Robert Sedgewick. Permutation generation methods. *Computing Surveys*, 9:137–164, 1977.

[4] Donald E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 2: Generating All Tuples and Permutations.* Addison-Wesley Professional, Reading, MA, 2005.

[5] B. R. Heap. Permutations by interchanges. *The Computer Journal*, 6(3):293–294, 1963.

[6] F. Ives. Permutation enumeration: Four new permutation algorithms. *Communications of the ACM*, 19(2):68–72, 1976.

Appendix A Position algorithm procedure

unranking procedure Referring to the algorithm described above, for n-carry arrays, only one traversal is needed to generate the corresponding full permutation.

Table 2: Example: 4-digit number unranking process

position	**read**	**pending**	**conflict**	**identification**	**correction**	**result**
0	C[0] = 0	0	None	0	None	0
1	C[1] = 1	01	None	01	None	01
2	C[2] = 1	011	C[1] = 1	$0\underline{1}1$	$0\langle 1\rangle 21$	021
3	C[3] = 2	0212	C[1] = 2	$0\underline{2}12$	$0\langle 2\rangle 312$	0312

Taking the 4-digit number 0112 as an example, Traverse (according to sequence numbers 0-3):

- Check the 0th position value is 0, There is no number in front, no change. get 0
- Check the 1st position value is 1, There is no other 1 in front, no change. get 01
- Check the 2nd position value is 1, There is 1 in position 1, change to position value 2. get 02(1->2)1
- Check the 3rd position value is 2, There is 2 in position 1, change to position value 3. get 03(2->3)12

ranking procedure According to the Position algorithm, as long as the operation is reversed, a reverse Map mapping can be performed.

Traverse the variable i from n-1 to 0, first traversing the array to create a positional index. Then traverse the variable i from n-1 to 0, judgment: If there is a value D [i] in the D array that is the same as current position i, then this value is rewritten by repeating the number above base and needs to be restored to D [i]. Change D [i] to i.

Here are the corresponding tables for generating full permutations of 4 elements.

Table 3: Permutation mapping of 4 elements (Position unrank)

Dec	$C_0C_1C_2C_3$	$D_0D_1D_2D_3$	**Dec**	$C_0C_1C_2C_3$	$D_0D_1D_2D_3$	**Dec**	$C_0C_1C_2C_3$	$D_0D_1D_2D_3$
0	0 0 0 0	1 2 3 0	8	0 0 2 0	1 3 2 0	16	0 1 1 0	3 2 1 0
1	0 0 0 1	3 2 0 1	9	0 0 2 1	3 0 2 1	17	0 1 1 1	0 2 3 1
2	0 0 0 2	1 3 0 2	10	0 0 2 2	1 0 3 2	18	0 1 1 2	0 3 1 2
3	0 0 0 3	1 2 0 3	11	0 0 2 3	1 0 2 3	19	0 1 1 3	0 2 1 3
4	0 0 1 0	2 3 1 0	12	0 1 0 0	2 1 3 0	20	0 1 2 0	3 1 2 0
5	0 0 1 1	2 0 3 1	13	0 1 0 1	2 3 0 1	21	0 1 2 1	0 3 2 1
6	0 0 1 2	3 0 1 2	14	0 1 0 2	3 1 0 2	22	0 1 2 2	0 1 3 2
7	0 0 1 3	2 0 1 3	15	0 1 0 3	2 1 0 3	23	0 1 2 3	0 1 2 3

The mapping result of **Position pure algorithm** is easy to obtain.

Table 4: Permutation mapping of 4 elements (Position pure unrank)

Dec	$C_0C_1C_2C_3$	$D_0D_1D_2D_3$	**Dec**	$C_0C_1C_2C_3$	$D_0D_1D_2D_3$	**Dec**	$C_0C_1C_2C_3$	$D_0D_1D_2D_3$
0	0 0 0 0	3 0 1 2	8	0 0 2 0	3 0 2 1	16	0 1 1 0	3 2 1 0
1	0 0 0 1	2 3 1 0	9	0 0 2 1	1 3 2 0	17	0 1 1 1	0 3 1 2
2	0 0 0 2	2 0 3 1	10	0 0 2 2	1 0 3 2	18	0 1 1 2	0 2 3 1
3	0 0 0 3	2 0 1 3	11	0 0 2 3	1 0 2 3	19	0 1 1 3	0 2 1 3
4	0 0 1 0	3 2 0 1	12	0 1 0 0	3 1 0 2	20	0 1 2 0	3 1 2 0
5	0 0 1 1	1 3 0 2	13	0 1 0 1	2 3 0 1	21	0 1 2 1	0 3 2 1
6	0 0 1 2	1 2 3 0	14	0 1 0 2	2 1 3 0	22	0 1 2 2	0 1 3 2
7	0 0 1 3	1 2 0 3	15	0 1 0 3	2 1 0 3	23	0 1 2 3	0 1 2 3

Appendix B Algorithm Compare

Rank/Unrank implementations achieve bidirectional mapping between permutations and integers, with significant differences in efficiency, mapping capabilities, and applicability across various algorithms:

Heap's Algorithm: A classic permutation generation algorithm known for its efficiency, requiring amortized $O(1)$ time per generation. However, it does not support efficient Unrank (random access), and is difficult to parallelize directly. It is best suited for sequential traversal tasks where indexing is not required.

Traditional Lexicographical Algorithms: Standard mapping algorithms (e.g., based on the Lehmer code) typically enable Rank/Unrank operations but suffer from $O(N^2)$ time complexity due to the overhead of maintaining and shifting the list of available elements. Optimized variants using data structures like Order Statistic Trees (e.g., Fenwick Trees) can reduce the Unrank complexity to $O(N \log N)$, but this remains computationally expensive for large N.

Myrvold-Ruskey Algorithm: A breakthrough algorithm proposed in 2001 that solved the linear-time Unranking problem. Unlike traditional methods, it utilizes a specific non-lexicographical ordering (based on the Fisher-Yates shuffle logic) to achieve Rank and Unrank operations in strict $O(N)$ time without complex data structures. It is particularly valuable for parallel generation and random sampling tasks where lexicographical order is not mandatory.

The algorithms with a time complexity of $\mathrm{O(n^2)}$ include: inverse Cantor expansion algorithm, Factor mapping algorithm, and others. Because the performance gap is significant, they are not included in the comparison here. The following only compares the Myrvold-Ruskey algorithm, the Position algorithm, and the Position pure algorithm, all of which have a time complexity of O(n).

Experimental Setup All experiments were conducted on a machine equipped with an Intel Core i7-8550U (4 cores, 8 threads, 1.8–4.0 GHz), 16 GB DDR4 RAM, running Windows 11 Pro. The code was compiled using GCC 13.1.0 with `-O3` optimization. To ensure consistency, Turbo Boost was disabled and CPU affinity was fixed to a single core.

All tests were conducted with 10 independent runs, and the final results were reported as the average value. All algorithms were optimized with the same strategy and executed in a cross-run configuration to ensure the fairness and reliability of the comparison.

From practical testing, the Position and Position pure algorithm shows improved efficiency over MyrvoldRuskey algorithm.

Table 5: Algorithm unranking process time (second) compare for large n, still need optimization

n	Process Count	MyrvoldRuskey	Position pure
200	10^6	2.66	2.11
400	10^6	5.48	4.36
600	10^6	8.85	6.94
800	10^6	11.49	9.04
1000	10^6	14.76	11.38