International Journal of Applied Mathematics

Volume 37 No. 4 2024, 479–483
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)
doi: http://dx.doi.org/10.12732/ijam.v37i4.7

AN ALGORITHM FOR LISTING ALL PERMUTATIONS OF N ELEMENTS USING MODULO ARITHMETIC

Dubravko Sabolic

University of Zagreb, FER

Unska 3

10000 Zagreb, CROATIA

e-mail: dubravko.sabolic@gmail.com

Abstract

This article presents a novel algorithm that can list all of the N-element permutations in a straightforward and easily automated manner. If N is fed as input, the algorithm will find all the permutations of N elements using modulo arithmetic, which makes it different from earlier algorithms. We do not present rigorous mathematical proofs for the algorithm or its time complexity. Instead, relying on the elementary rules of modulo arithmetic, we provide intuitive justification of the rather obvious algorithm and numerical examples to show that it has the time complexity of the order $O(N \times N!)$ or simply $O(N!)$, as expected based on the elementary analysis of the computational procedure.

Math. Subject Classification: 05A05, 68R99, 68Q25

Key Words and Phrases: permutations, algorithm, modulo arithmetic

1. Introduction

Suppose we have N elements and want to list all possible ways to arrange them in a definite order. For instance, we can have billiard balls marked with

Received: January 18, 2024

consequent natural numbers: $1, 2, 3, \ldots, N$, or we can have them in N different colors. It is common knowledge that we can arrange such a set in $N!$ different ways. However, listing all those permutations may be challenging, especially when N increases even over modest quantities. This article presents a novel algorithm that can identify all the N-element simple permutations (meaning the permutations of all the N elements).

Heap's algorithm's [1] critical characteristic is its efficiency regarding the minimal element swaps needed to create all permutations. It operates in place without requiring additional storage space for storing permutations, making it space-efficient. Its overall time complexity is $O(N \times N!)$ or simply $O(N!)$.

Knuth's recursive approach [2] for generating permutations is based on systematically swapping elements to explore all possible arrangements. The overall time complexity is also $O(N \times N!)$ or simply $O(N!)$.

Johnson-Trotter algorithm [3, 4] is another approach for generating permutations with the same overall time complexity. Its advantage lies in efficiently generating permutations with fewer changes than other algorithms.

The Steinhaus–Johnson–Trotter algorithm [5], like the Johnson-Trotter algorithm, focuses on generating permutations that avoid consecutive ascents or descents. It, too, has the time complexity of $O(N!)$.

Lexicographic ordering [6] is a method of arranging permutations in the order defined by lexicography. This algorithm exhibits the same time complexity as all the ones mentioned above.

2. **The Algorithm**

First, we will establish the notation rule for different elements to permute. We will mark them with natural numbers. Let us start from the most elementary permutation: a single element that we decided to name "1". So, we begin with:

$$1$$

Let us introduce the next element, "2", to the left and mark its addition with the symbol "|":

$$2 \mid 1$$

Let us add 1 to both elements but in modulo 2. However, because we have started to count from 1 instead of zero, we need to prepare the "name numbers" and then carry on with the procedure. In our example, we need to perform the following steps: First, subtract 1 from both elements, then add 1 in modulo 2 arithmetic to both elements, and then add back 1 to all the elements to return to the original natural-number notation. (Not working

with the number 0 is a matter of convenience; for example, we may not like zeros but only natural numbers. Then, we must perform these trivial steps.)

Here is the complete sequence of these elementary operations performed on the sequence 2 | 1:

- Subtracting 1 from both: 1 | 0
- Adding 1 (mod 2) to both: 0 | 1
- Adding back 1 to both: 1 | 2

The last line contains the new permutation, and now the set of all two-element permutations is complete:

$$\begin{array}{cc} 2 & 1 \\ 1 & 2 \end{array}$$

Following the same logic, we can append this complete set of permutations for $N = 2$ with the new element to start constructing the list of permutations for $N = 3$:

$$\begin{array}{ccc} 3 & | & 2 \quad 1 \\ 3 & | & 1 \quad 2 \end{array}$$

Then, subtract 1 from all the numbers and make two additions in modulo 3 arithmetic, by 1 and 2, to generate new rows. We will keep the starting two lines and add all the newly generated ones. Finally, after adding back 1 to return to natural numbers, we will get the complete set of three-element permutations. The whole sequence reads:

- Subtract 1:

$$\begin{array}{ccc} 2 & | & 1 \quad 0 \\ 2 & | & 0 \quad 1 \end{array}$$

- Add 1 (mod 3):

$$\begin{array}{ccc} 0 & | & 2 \quad 1 \\ 0 & | & 1 \quad 2 \end{array}$$

- Add 2 (mod 3):

$$\begin{array}{ccc} 1 & | & 0 \quad 2 \\ 1 & | & 2 \quad 0 \end{array}$$

- Add back 1 to all:

$$\begin{array}{ccc} 3 & 2 & 1 \\ 3 & 1 & 2 \end{array}$$

$$\begin{array}{ccc} 1 & 3 & 2 \\ 1 & 2 & 3 \end{array}$$

$$\begin{array}{ccc} 2 & 1 & 3 \\ 2 & 3 & 1 \end{array}$$

The next step follows the same procedure:

$$\begin{array}{llll} 4 \mid 3 & 2 & 1 & \\ 4 \mid 3 & 1 & 2 & \\ 4 \mid 1 & 3 & 2 & (*) \\ 4 \mid 1 & 2 & 3 & \\ 4 \mid 2 & 1 & 3 & \\ 4 \mid 2 & 3 & 1 & \end{array}$$

After trivial subtraction of 1 from all the numbers, we add 1, then 2, and then 3, in modulo 4 arithmetic, to all of the values from the above list, and finally add 1 back to every number obtained. The number of permutations will now be increased by a factor of 4. In principle, this algorithm can be iterated forever.

We can effortlessly extract any lower-order M permutation table directly from the N-th order one, $M < N$, by taking the upper-right corner of the N-order matrix, which has the dimensions of $M!$ rows and M columns.

We should demonstrate why this algorithm works with $N-1$ stepwise additions in modulo N arithmetic. A proper proof would stem directly from the basic properties of modular arithmetic [7], but we shall leave it to the reader. The following properties apply for shifting the elements by more than zero and less than N steps:

- None of the elements is mapped back onto itself.
- Shifting the same element by different numbers of steps maps it to different elements.
- Different elements, when shifted by the same number of steps, map to other different elements.

Now, recall $(*)$. In the first line, we have:

$$4 \mid 3\ 2\ 1$$

After subtracting 1 from all the elements, it becomes:

$$3 \mid 2\ 1\ 0$$

It is a 4-permutation as it has four distinct elements. Adding any natural number smaller than 4 to all of them, for example, 1, in modulo 4 arithmetic by necessity generates another legitimate permutation, in this case, 0 3 2 1. It happens for any eligible step sizes (1,2,3). Since in modulo 4, there are $4-1=3$ eligible step sizes, we can produce $6 \times 3 = 18$ additional lines and list them below those already printed in (*). All of them are sure to be legitimate permutations. In this way, we have generated a total of $6 + 18 = 24 = 4!$ permutations.

If we have more than ten elements (or more than nine if we do not use the number 0), we may need to replace numbers larger than 9 with suitable

"symbolic numbers." The most obvious way is to work in the number system base corresponding to N.

3. **Time Complexity**

If the N-1-permutations are already known, one must at least append $N-1$ new elements to the already known permutations, add 1, then 2, then 3, ..., then $N-1$ (mod N) to each element in each of the such created permutations, and store all the permutations created.

Those actions require the number of operations proportionate to, respectively: $N-1$; $(N-1)! \times N \times (N-1) = N! \times (N-1)$; and $N! \times (N-1)$. For large N, the latter two tend to $N \times N!$ and become much larger than $N-1$. Therefore, the computation load increases with $N \times N!$.

To test this, we made a short Matlab script to repeat the algorithm many times for each $N < 12$, record the computation time in milliseconds, and compute the average. The relationship between average computing time and $N \times N!$ was almost perfectly linear, with Pearson's R^2 larger than 0.99999, which justifies the case at least for $N < 12$. The regression between average computing time and $N!$ was slightly worse, with R^2 larger than 0.99996.

4. **Conclusions**

In this article, we presented an algorithm for listing all the permutations of N elements using modular arithmetic. We also deduced that the algorithm's time complexity is $O(N \times N!)$, which we tested numerically for $N < 12$.

References

[1] B.R. Heap, Permutations by Interchanges, *Communications of the ACM*, **6** (1963), 293-298.

[2] D.E. Knuth, *The Art of Computer Programming, Volume 1*, Addison-Wesley Professional, Boston (1997).

[3] S.M. Johnson, The problem of arranging alliances, *Proceedings of the American Mathematical Society*, **14** (1963), 799-806.

[4] H.F. Trotter, Perm groups and algebras in complexity theory, *Algorithmica*, **1** (1986), 393-413.

[5] R. Sedgewick, Permutation generation methods, *Computing Surveys*, **9** (1977), 137-164.

[6] R. Sedgewick and K. Wayne, *Algorithms*, Addison-Wesley Professional, Boston (2011).

[7] E. Lehman, F.T. Leighton and A.R. Meyer, *Mathematics for Computer Science*, Retrieved 1 Dec. 2023. MIT OpenCourseWare, URL: https://ocw.mit.edu/resources/res-18-001-mathematics-for-computer-science-fall-2005/.