Generating Permutations Lexicographically

Nitin Verma
mathsanew.com

July 11, 2020

Given an array a of N distinct elements, we want to write a program to generate all permutations of the N elements. There will be $N!$ such permutations.

As $N!$ is very large even for small N, we will never want to store all the generated permutations in memory. So, we can process each permutation as soon as it is generated, before constructing another permutation.

In mathematical expressions, a_i will be used to represent element $a[i]$ of the array a. Since contents of array a may change during such a program, we will refer to the initial sequence of elements in array a, $(a_0, a_1, a_2, \ldots, a_{N-1})$, as elements $(A_0, A_1, A_2, \ldots, A_{N-1})$.

If an algorithm needs to generate all possible permutations of N elements, it should not generate one permutation after other in random order. Because, then it has no efficient way to remember which permutation it has already generated and which ones are remaining. The algorithm needs to generate them in a definite order: the i^{th} permutation generated is definite for a given input. An algorithm to generate permutations has this order defined, and provides a method to generate permutations in that order.

In this article, we discuss an algorithm which generates permutations in *Lexicographical Order*. The algorithm has been discovered multiple times independently, but is known to be first published in 1812 by L. L. Fischer and K. C. Krause [1]. So, it is sometimes referred as *Fischer-Krause Algorithm*.

Lexicographical Order

Consider two permutations of input array a:

$$P_1 = (A_{i_0}, A_{i_1}, A_{i_2}, \ldots, A_{i_{N-1}})$$
$$P_2 = (A_{j_0}, A_{j_1}, A_{j_2}, \ldots, A_{j_{N-1}})$$

Say, A_{i_m} is the first element at which these two permutations differ, that is $A_{i_m} \neq A_{j_m}$. We assume that there is an order defined among elements of array a, and suppose we have $A_{i_m} < A_{j_m}$ under that order. Then we say that permutation P_1 is *Lexicographically* smaller than P_2. We will write $P_1 < P_2$.

For example, with array $a = (1, 2, 3, 4, 5)$, the permutation $(2, 4, 1, 5, 3)$ is smaller than $(2, 4, 5, 3, 1)$.

Since elements of array a can be ordered, suppose we have their order as:

$$A_{k_0} < A_{k_1} < A_{k_2} < \ldots < A_{k_{N-1}}$$

Then, we can reason that the following permutation is the smallest permutation in lexicographic ordering of all permutations:

$$P_s = (A_{k_0}, A_{k_1}, A_{k_2}, \ldots, A_{k_{N-1}})$$

Say, any other permutation differ from above permutation first at A_{k_i}, with another element d at that position. We know that A_{k_i} is the smallest of all elements to be placed at this position onwards. So, we must have $A_{k_i} < d$.

And with similar reasoning, following is the largest permutation:

$$P_l = (A_{k_{N-1}}, A_{k_{N-2}}, A_{k_{N-3}}, \ldots, A_{k_0})$$

Next in Lexicographical Order

Suppose, in our program we are generating permutations in array a itself, and at certain point have generated some permutation P in it:

$$P = (a_0, a_1, a_2, \ldots, a_{N-1})$$

What is the permutation P' which is the very next permutation of P in the lexicographic ordering? That is, among all permutations larger than P, P' is the smallest.

Say, the first element at which P and P' differ is their m^{th} element. Call the m^{th} element in P' as b. So, $a_{m-1} \neq b$ and since $P < P'$, $a_{m-1} < b$.

Note that, among all permutations having their first m elements' sequence same as P, P must be the largest. Because, if P is not the largest,

then there is a permutation Q larger than P sharing first m elements' sequence with it. That is, $P < Q$. The initial m elements of Q being same as of P (till a_{m-1}) implies $Q < P'$. So, $P < Q < P'$. That is a contradiction, since P' is the very next permutation after P.

Since P is the largest among all permutations having their first m elements' sequence same as P, we can conclude this about the remaining $N-m$ elements of P:

$$a_m > a_{m+1} > a_{m+2} > \ldots > a_{N-1} \tag{1}$$

Since P' shares its first $m-1$ elements with P, and its m^{th} element b differs from a_{m-1}, we can say that b is one of the remaining $N-m$ elements of P, referred as set S:

$$S = \{a_m, a_{m+1}, a_{m+2}, \ldots, a_{N-1}\}$$

Note that elements of S can be ordered as in (1). Since P' is the smallest of all permutations larger than P, b has to be the smallest of the elements in set S such that $b > a_{m-1}$. Say, $b = a_k$. Since a_m is the largest in set S, such b exists only if:

$$a_{m-1} < a_m \tag{2}$$

The elements in P' after its m^{th} element b are actually this set of $N-m$ elements: $S' = (S \setminus \{a_k\}) \cup \{a_{m-1}\}$.

Again, since P' is the smallest of all permutations larger than P, all $N-m$ elements in P' after b must be in increasing order. That is, elements of set S' in increasing order.

We already know the order of elements of set S from ordering (1). That ordering can be modified by removing a_k and adding a_{m-1}. Since a_k is the smallest element in set S which is larger than a_{m-1}, so, in that ordering, a_k can be removed and a_{m-1} added at the same position, while maintaining the decreasing order. This modification in (1) gives us a decreasing order of elements of set S', which can be reversed to obtain increasing order.

Fischer-Krause Algorithm

Now we will use the relationship among elements of P and P' to generate P' from P. Properties (1) and (2) can be used to locate the first element a_{m-1} in P at which P and P' differ. a_k can be searched in the subsequent elements of P, with its desired property. Below is an outline of how we can generate P' from P:

1. By reading elements in P backwards starting at a_{N-1}, find m such that:

   $$a_m > a_{m+1} > a_{m+2} > \ldots > a_{N-1}$$

   and $a_{m-1} < a_m$.

   Note, it is possible that $m = N - 1$. Also, when we have $m = 0$, we must have the largest permutation already in a, and we can terminate.

2. Again, by reading elements in P backwards starting at a_{N-1}, find first (hence smallest in set S) element a_k such that: $a_k > a_{m-1}$.

3. Swap a_{m-1} and a_k.

4. We must be having all elements after the $m - 1$ index in decreasing order. Make them in increasing order simply by reversing this portion of the array (last $N - m$ elements). We have now arrived at the next permutation P' in array a.

Now the algorithm to generate all permutations of array a is straightforward. Generate the smallest permutation by sorting a in increasing order. Starting with this permutation, keep generating the next permutation by above method. Terminate upon reaching the largest permutation as detected in step (1) above.

Below C program shows the `next()` function which generates the next permutation of the current permutation in array a. If a already contains the largest permutation, it returns 0, otherwise 1. Function to sort array a initially is not shown; the sample array is already sorted.

```
#include <stdio.h>

int a[] = {1,2,3,4,5,6,7,8,9};
int N = 9;

void print(void);
int next(void);

int main(void)
{
  /* sort a[] in increasing order */

  print();

  while(next())
    print();

  return 0;
}

#define SWAP(i, j) {int t; t = a[i]; a[i] = a[j]; a[j] = t;}

int next(void)
{
  int m, k;

  /* Find m */
  m = N-1;

  while(m > 0 && a[m-1] > a[m])
    m--;

  /* (m == 0 OR a[m-1] < a[m]) is TRUE */

  if(m == 0)
  {
    /* Permutation in a[] is the largest one */
    return 0;
  }

  /* (a[m-1] < a[m]) is TRUE */

  /* Find k. Since a[m] is larger than a[m-1], we are sure
   * to find required a[k] within the index range [m, N-1].
   * Hence, no lower bound checking for variable k is needed
   * in the below loop.
   */

  k = N-1;
```

```
  while(a[k] < a[m-1])
    k--;

  /* (a[k] > a[m-1]) is TRUE */

  SWAP(m-1, k);

  /* Reverse the order of elements from a[m] to a[N-1] */
  int s = m;
  int e = N-1;

  while(s < e)
  {
    SWAP(s, e);
    s++;
    e--;
  }

  return 1;
}

void print(void)
{
  int i;

  for(i = 0; i < N; i++)
    printf("%d ", a[i]);

  printf("\n");
}
```

References

[1] R. Sedgewick. *Permutation Generation Methods.* ACM Comput. Surv., Vol 9 (2) (1977), 137–164.