Generating Permutations with Recursion

Nitin Verma
mathsanew.com

September 16, 2021

Given an array a of N elements which are all distinct, we want to generate all the $N!$ permutations of its elements. Since $N!$ grows very fast with N (e.g. $15! = 1307674368000$), it is generally not practical to store all the permutations in memory. The approach we take is to generate each permutation in array a itself, process it (e.g. print), then generate a new permutation in a, and so on.

R. Sedgewick provided a very good survey about permutation generation algorithms in [1]. In this article, we will discuss a few recursive algorithms for this problem. All of them are based on a common recursive structure which is described below.

We will be denoting any subarray of a from some index i to some index j ($j \geq i$) as $a[i:j]$. All implementation will be in C language, and assumes elements of array a to be of type "int".

Common Recursive Structure

Consider a subarray $a[s : N-1]$ for some s. Let us call its set of $N-s$ elements as set B. Now, to generate all the permutations of elements of this subarray, we can proceed as follows.

Pick some element e from set B and place it at index s of array a, while keeping rest of the elements of B (set $B \setminus \{e\}$) in subarray $a[s+1 : N-1]$. Now solve the subproblem of permuting the subarray $a[s+1 : N-1]$. After that, repeat the above process by picking some other element from B not already picked and placing it at index s, and permuting the rest of B in subarray $a[s+1 : N-1]$. Repeat this by picking each of the $N-s$ elements of B. This process must generate all the $(N-s)!$ permutations of the subarray $a[s : N-1]$.

Note that, the subproblem of permuting the smaller subarray $a[s+1 : N-1]$ can be solved in a similar way. Repeating this process recursively will eventually give us a subproblem with subarray $a[N-1 : N-1]$, which is trivial.

The question yet to be answered is, how to pick elements of B for placing at index s without skipping or repeating any? To appreciate the challenge in this, notice that the subproblem of $a[s+1 : N-1]$ will reorder the elements of the subarray while permuting them. The algorithms we discuss have some defined strategy to locate the next element of B to pick, from some index x of the subarray. Such element is then swapped with the existing element at index s to proceed for solving the subproblem of $a[s+1 : N-1]$.

Below method depicts the above recursive structure and provides a skeleton in which all of our recursive algorithms will fit. *generate*(s) is supposed to generate all permutations of the subarray $a[s : N-1]$. The initial call is *generate*$(s = 0)$. *print*$()$ prints the permutation currently in array a.

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

void generate(int s)
{
  int i;

  if(s == N-1)
  {
    print();
    return;
  }

  for(i = 0; i < N-s; i++)
  {
    /* Setup Block */
    if(i > 0)
      SWAP(s, x);

    generate(s+1);

    /* Cleanup Block */
  }
}
```

Note that there are $N-s$ iterations (size of set B), each with one recursive call. The very first recursive call (in first iteration with $i = 0$) is made without performing any swap — the element initially at index s remains at index s. The “Setup Block” and “Cleanup Block” are general placeholders for any steps the algorithm needs to perform before and after the recursive call respectively. When the base-case of $s = N-1$ is reached, a permutation of the complete array a is ready for processing; here we just print it.

We will now discuss the recursive algorithms individually.

Fike's Algorithm

R. Sedgewick in [1] has attributed this algorithm to C. T. Fike [2]. Following is an implementation of this algorithm:

```
void fike(int s)
{
  int i;

  if(s == N-1)
  {
    print();
    return;
  }

  for(i = 0; i < N-s; i++)
  {
    /* Setup Block */
    if(i > 0)
      SWAP(s, s+i);

    fike(s+1);

    /* Cleanup Block */
    if(i > 0)
      SWAP(s, s+i);
  }
}
```

It is based on the idea that if each recursive call *fike*$(s+1)$ keeps the subarray $a[s+1 : N-1]$ intact upon return, we can pick each element of B correctly (without skipping or repeating any) one by one from indices $s, s+1, s+2, \ldots, N-1$.

Now, to ensure this, it enforces upon method *fike*(s) the specification that it must keep the input subarray $a[s : N-1]$ intact, for each s. For $s = N-1$, this specification is trivially met. For any other s, it is implemented by reversing the swap done before the recursive call *fike*$(s+1)$, once that call returns. So, if *fike*$(s+1)$ meets the specification, each iteration of *fike*(s) will also leave the subarray intact. This leads to the complete *fike*(s) meeting the specification. The correctness of this algorithm is thus established by Mathematical Induction.

An optimization is possible in above method. Note that all iterations (except the first) do two swaps which involve the element initially at $a[s]$. We can exploit this fact and save some writes by assigning elements as below. The 6 writes of two swaps are now replaced by the 3 writes in an iteration:

```
initial_a_s = a[s];

for(i = 0; i < N-s; i++)
{
  if(i > 0)
  {
    a[s] = a[s+i];
    a[s+i] = initial_a_s;
  }

  fike(s+1);

  if(i > 0)
    a[s+i] = a[s];
}

a[s] = initial_a_s;
```

Heap's Algorithm

This algorithm is due to B. R. Heap [3]. Following is an implementation of this algorithm.

```
void brheap(int s)
{
  int i;

  if(s == N-1)
  {
    print();
    return;
  }

  for(i = 0; i < N-s; i++)
  {
    /* Setup Block */
    if(i > 0)
    {
      if((N-s)%2 != 0)
        SWAP(s, N-1)
      else
        SWAP(s, N-i)
    }

    brheap(s+1);
  }
}
```

Notice how the next element of B is picked based on whether $N - s$, the number of elements in subarray $a[s : N - 1]$, is even or odd. What is most impressive about this algorithm is that it performs exactly one swap for each permutation generated (do we see why?). Thus it involves total $N! - 1$ swaps. This makes the algorithm one of the efficient algorithms, because for moving from one permutation to any other, at least one swap is always required.

When executing this algorithm, we can notice that an invocation of this method for any s leaves the subarray $a[s : N-1]$ altered in a definite pattern (compare the subarray upon method entry and exit). This pattern only depends upon the parity of $N - s$, the number of elements in the subarray. Consider the subarray:

$$(e_1\ e_2\ e_3\ e_4\ e_5\ \ldots\ e_{n-1}\ e_n)$$

For odd n, we find it becoming:

$$(e_n\ e_2\ e_3\ e_4\ e_5\ \ldots\ e_{n-1}\ e_1)$$

For even n, we find it becoming:

$$(e_n\ e_1\ e_4\ e_5\ \ldots\ e_{n-1}\ e_2\ e_3)$$

The original paper [3] of this algorithm does not include a proof of correctness. D. E. Knuth in his book [4], asks to prove it as an exercise which hints at using the "Sims Table" (section "Generating All Permutations", exercise 40). (I am still searching for an elegant proof of this algorithm and of the pattern observed above.)

Intact-Rotate Algorithm

Consider the following pair of methods:

```
void intact(int s)
{
  int i;

  if(s == N-1)
  {
    print();
    return;
  }

  for(i = 0; i < N-s; i++)
  {
    rotate(s+1);

    SWAP(s, N-1);
  }
}

void rotate(int s)
{
  int i;

  if(s == N-1)
  {
    print();
    return;
  }

  for(i = 0; i < N-s; i++)
  {
    if(i > 0)
      SWAP(s, N-i);

    intact(s+1);
  }
}
```

Consider the iterations of $\mathit{intact}(s)$. Assume that any call to $\mathit{rotate}(s+1)$ leaves the subarray $a[s+1 : N-1]$ left-rotated by 1 place upon return. That is, if the subarray before the call is: $(e_{s+1}\ e_{s+2}\ \ldots\ e_{N-1})$, it will become after the call: $(e_{s+2}\ e_{s+3}\ \ldots\ e_{N-1}\ e_{s+1})$.

Suppose the subarray $a[s : N-1]$ at the start of an iteration is:

$$(e_s\ e_{s+1}\ e_{s+2}\ \ldots\ e_{N-1})$$

Due to the call to $\mathit{rotate}(s+1)$ followed by the swap, the subarray will become, at the end of the iteration:

$$(e_{s+1}\ e_{s+2}\ \ldots\ e_{N-1}\ e_s)$$

This is nothing but a left-rotation of the subarray $a[s : N-1]$. Thus the side-effect of each iteration is to left-rotate $a[s : N-1]$. So the total of $N-s$ such iterations will left-rotate this subarray (of $N-s$ elements) by $N-s$ places, leaving it what it was prior to the loop. So we can conclude that $\mathit{intact}()$ leaves the subarray $a[s : N-1]$ intact upon return.

Now consider the iterations of $\mathit{rotate}(s)$ and assume that any call to $\mathit{intact}(s+1)$ leaves the subarray $a[s+1 : N-1]$ intact upon return. To see the side-effect of the iterations, we can ignore the call to $\mathit{intact}(s+1)$ and simply consider the effect of executing the swaps. We can figure out that when the loop terminates, the subarray $a[s : N-1]$ must have left-rotated by 1 place.

Also notice that the base-case $(s = N-1)$ of both the methods leaves the subarray $a[N-1 : N-1]$ intact which can also be seen as a left-rotation of the single element subarray.

Suppose we associate a specification to $\mathit{intact}()$ and $\mathit{rotate}()$ each that they leave the subarray $a[s : N-1]$ intact and left-rotated respectively upon return. In above paragraphs, we proved that $\mathit{intact}()$ and $\mathit{rotate}()$ follow their specification assuming that their recursive call to the other one also follows its own specification. As we saw, their base-case also follows the specification. So, by applying Mathematical Induction over $N-s$, we can conclude that both methods do follow their specification.

Now that we have established how a call to $\mathit{intact}()$ and $\mathit{rotate}()$ leaves the subarray upon return, it is easy to see why the swapping with index $N-1$ and $N-i$ by these methods will pick all elements of B correctly. In conclusion, invoking any of these two methods with $s = 0$ must generate all permutations of array a.

We can notice some similarity among these two methods and the *brheap*() shown earlier: each of them involves $N-s$ iterations where *almost all* iterations perform one swap each. The iterations of *intact*() do total $N-s$ swaps but those of *rotate*() and *brheap*() do total $N-s-1$ swaps. Thus there is one extra swap for each invocation of *intact*(), making this algorithm less efficient than the Heap's Algorithm.

Since the two methods call each other, any execution of this algorithm must involve their alternate invocations: one particular method for even s and the other one for odd s. Also, the structure of these methods are similar. So we can combine these into a single method as below, which adheres to the common recursive structure:

```
void intact_rotate(int s)
{
  int i;

  if(s == N-1)
  {
    print();
    return;
  }

  for(i = 0; i < N-s; i++)
  {
    /* Setup Block (for rotate()) */
    if(i > 0 && ((N-s)%2 == 0))
      SWAP(s, N-i);

    intact_rotate(s+1);

    /* Cleanup Block (for intact()) */
    if((N-s)%2 != 0)
      SWAP(s, N-1);
  }
}
```

Ord-Smith's Algorithm

R. Sedgewick in [1] has attributed this algorithm to R. J. Ord-Smith [5]. This algorithm generates the permutations in *Lexicographical Order*. (Article [6] discussed the definition of this order, some of its properties, and a non-recursive algorithm to generate permutations in this order.)

Following is an implementation of the Ord-Smith's Algorithm:

```
#define REVERSE(s1, e1) { \
  int s2 = s1;            \
  int e2 = e1;            \
                          \
  while(s2 < e2)          \
  {                       \
    SWAP(s2, e2);         \
    s2++;                 \
    e2--;                 \
  }                       \
}

void ordsmith(int s)
{
  int i;

  if(s == N-1)
  {
    print();
    return;
  }

  for(i = 0; i < N-s; i++)
  {
    /* Setup Block */
    if(i > 0)
    {
      REVERSE(s+1, N-1);
      SWAP(s, s+i);
    }

    ordsmith(s+1);
  }
}
```

For generating permutations in lexicographical order, the initial call *ordsmith*(0) must be done with array a sorted in increasing order.

Let us denote by predicates $I(s)$ and $D(s)$ (for any s) the fact that subarray $a[s : N-1]$ is in increasing and decreasing order respectively. The algorithm works by enforcing the following specification P upon method *ordsmith*$()$: if *ordsmith*(s) is called with $I(s)$ true, then $D(s)$ must hold upon its return.

Note that specification P holds trivially for $s = N-1$. Now suppose, for some other s, it holds for $s+1$. Consider the invocation *ordsmith*(s) and assume that $I(s)$ holds at its beginning. After its iteration of $i = 0$, it is clear that $D(s+1)$ must hold (since the call *ordsmith*$(s+1)$ holds P). We claim that at the end of each iteration, $D(s+1)$ holds. This is because, any iteration with $i > 0$ which begins with $D(s+1)$ holding performs:

(a) reversal of $a[s+1 : N-1]$ resulting in $I(s+1)$ holding,
(b) swap which still maintains $I(s+1)$ (see below),
(c) call to *ordsmith*$(s+1)$ which would leave $D(s+1)$ holding again.

While making the recursive call, the algorithm places at index s the elements of set B (subarray $a[s : N-1]$) in increasing order. For $i = 0$ iteration, we already have the smallest element at index s (since $I(s)$ holds). For all later iterations, the swap picks the next larger element for placing at index s. Since $I(s+1)$ holds after the reversal, such larger element is easily located by the swap at index $s+i$. This swap won't alter the increasing order of $a[s+1 : N-1]$, because what replaces the element at index $s+i$ is the element just smaller than it.

It is this placing of increasingly larger elements at index s, which gives the generated permutations the lexicographical order.

We now come back to the claim about $D(s+1)$. Since $D(s+1)$ holds at the end of each iteration and the last iteration would place the largest element of $a[s : N-1]$ at index s, so the loop must terminate with $a[s : N-1]$ in decreasing order. That is, $D(s)$ must hold upon the method's return.

Thus, by Mathematical Induction, specification P must hold for method *ordsmith*(s) for all s.

It is interesting to note that this method, though giving a lexicographical result, does not perform any comparison among elements or inspection of their values. These are needed only for the initial sorting of array a. Since this method is thus a pure structural one, it must generate all permutations of a whatever be its initial order. Though, the order of such permutations will be based on the initial order of a, not the lexicographical order of its elements.

$\blacksquare$

References

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

[2] C. T. Fike. *A Permutation Generation Method.* Comput. J., Vol 18 (1) (1975), 21–22.

[3] B. R. Heap. *Permutations by Interchanges.* Comput. J., Vol 6 (3) (1963), 293–294.
`https://academic.oup.com/comjnl/article/6/3/293/360213`.

[4] D. E. Knuth. *The Art of Computer Programming*, Vol 4A, First Edition. Addison-Wesley (2011).

[5] R. J. Ord-Smith. *Algorithm 323: Generation of Permutations in Lexicographic Order*. Commun. ACM, Vol 11 (2) (1968), 117.

[6] Nitin Verma. *Generating Permutations Lexicographically.*
`https://mathsanew.com/articles/generating_permutations.pdf`.