Generating Permutations without 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.

In an earlier article titled *Generating Permutations with Recursion* [2], we discussed a few recursive algorithms for this problem. All of those algorithms were based on a common recursive structure, as depicted by the following skeleton method. Please refer to that article for details about this recursive structure (section "Common Recursive Structure").

```
#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 */
  }
}
```

In this article, we will find out how we can implement those recursive algorithms without using recursive calls. What makes our task simpler is that all of them follow the above common recursive structure. From the above skeleton, we will design an equivalent non-recursive skeleton, which itself will serve as a common skeleton for writing non-recursive variants of the algorithms.

We will now attempt to convert the above *generate*() method into its non-recursive equivalent.

Common Non-Recursive Structure

A generic approach to convert a recursive method into a non-recursive one is to simulate the recursive calls on our own stack within the method. This stack would contain the arguments and local variables for each recursive call. Such an approach was discussed in an earlier article titled *Maintaining the Stack for Recursion* [3].

But we will follow a different approach below, which makes use of some specific observations about the *generate*() method. We will now inspect closely the execution flow during the call *generate*(0).

First notice that for all $s < N - 1$, the *generate*() method is simply a loop: a sequence of iterations for each of $i = 0, 1, 2, \ldots, N - s - 1$. Further, whenever a permutation is printed by *generate*($s = N - 1$), each of the callers *generate*(0), *generate*(1), *generate*(2),..., *generate*($N - 2$) are inside one of their iterations with certain i. The range to which this i can belong depends upon the respective s: $0 \leq i \leq N - s - 1$.

To maintain this state i of (up to) $N-1$ active invocations of the method, we will use an array I of $N - 1$ integers. $I[s]$ will hold the value of i for invocation of *generate*(s). We don't need to maintain i for $s = N - 1$.

Now notice that for any $s < N - 1$, an execution of *generate*(s) simply recurses to *generate*($s + 1$) as the very first step (after setting $i = 0$ in its first iteration). The execution of *generate*($s + 1$) would do the same thing and recurse to *generate*($s + 2$). This goes on till execution of *generate*($N-1$) which prints the permutation. This suggests that, whenever an iteration has to make a recursive call, we can simply execute the equivalent of *generate*($N - 1$): print the permutation. Though, this "execution" of *generate*($N - 1$) must still "return" to its caller *generate*($N - 2$).

To keep track of the currently executing *generate*() call (top of the call-stack), we will use an integer s which would correspond to the argument s

of the call. Once *generate*(s) completes its last iteration (for $i = N - s - 1$), it should return to its caller. This is done by decrementing the tracking integer s.

These were some of the ideas on which the following non-recursive equivalent of *generate*() is based. In the while loop below, the beginning of any iteration can be seen as the point when an iteration of *generate*(s) resumes its operation after the recursive call *generate*($s + 1$) has returned.

```
void generate_nonrecur()
{
  int I[N-1], s;

  for(s = 0; s < N-1; s++)
    I[s] = 0;

  /* "execute" generate(N-1) */
  print();

  /* "return" to the caller of generate(N-1) */
  s = N-2;

  while(s >= 0)
  {
    /* "resuming" after return from recursive call generate(s+1) */

    /* Cleanup Block */

    if(I[s] < N-s-1)
    {
      I[s]++;

      /* Setup Block */
      SWAP(s, x);

      /* "execute" generate(N-1) */
      print();

      /* "return" to the caller of generate(N-1) */
      s = N-2;
    }
    else
    {
      /* last iteration completed */
      I[s] = 0;

      /* "return" to the caller */
      s = s-1;
```

```
    }
  }
}
```

The above method will serve as a common skeleton for writing non-recursive variants of the recursive algorithms. For each algorithm, we only need to put the Setup Block and Cleanup Block (if any) in the place indicated in above method. The sections below will list the source code for the non-recursive variants thus created. Please refer article [2] for the description and recursive methods of the algorithms.

Fike's Algorithm

```
void fike_nonrecur()
{
  int I[N-1], s;

  for(s = 0; s < N-1; s++)
    I[s] = 0;

  print();

  s = N-2;

  while(s >= 0)
  {
    /* Cleanup Block */
    if(I[s] > 0)
      SWAP(s, s+I[s]);

    if(I[s] < N-s-1)
    {
      I[s]++;

      /* Setup Block */
      SWAP(s, s+I[s]);

      print();
      s = N-2;
    }
    else
    {
      I[s] = 0;
      s = s-1;
    }
  }
}
```

Heap's Algorithm

```
void brheap_nonrecur()
{
  int I[N-1], s;

  for(s = 0; s < N-1; s++)
    I[s] = 0;

  print();

  s = N-2;

  while(s >= 0)
  {
    if(I[s] < N-s-1)
    {
      I[s]++;

      /* Setup Block */
      if((N-s)%2 != 0)
        SWAP(s, N-1)
      else
        SWAP(s, N-I[s])

      print();
      s = N-2;
    }
    else
    {
      I[s] = 0;
      s = s-1;
    }
  }
}
```

Intact-Rotate Algorithm

```
void intact_rotate_nonrecur()
{
  int I[N-1], s;

  for(s = 0; s < N-1; s++)
    I[s] = 0;

  print();

  s = N-2;

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

    if(I[s] < N-s-1)
    {
      I[s]++;

      /* Setup Block (for rotate()) */
      if((N-s)%2 == 0)
        SWAP(s, N-I[s]);

      print();
      s = N-2;
    }
    else
    {
      I[s] = 0;
      s = s-1;
    }
  }
}
```

Ord-Smith's Algorithm

```
void ordsmith_nonrecur()
{
  int I[N-1], s;

  for(s = 0; s < N-1; s++)
    I[s] = 0;

  print();

  s = N-2;

  while(s >= 0)
  {
    if(I[s] < N-s-1)
    {
      I[s]++;

      /* Setup Block */
      REVERSE(s+1, N-1);
      SWAP(s, s+I[s]);

      print();
      s = N-2;
    }
    else
    {
      I[s] = 0;
      s = s-1;
    }
  }
}
```

■

References

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

[2] Nitin Verma. *Generating Permutations with Recursion.* `https://mathsanew.com/articles/permutations_with_recursion.pdf`.

[3] Nitin Verma. *Maintaining the Stack for Recursion.* `https://mathsanew.com/articles/own_stack_for_recursion.pdf`.