A Simple Algorithm for Generating Combinations

Nitin Verma
mathsanew.com

December 22, 2021

For a set S of n elements, a *k-Combination* is any of its subset having k elements. It can be proved that, for any given k, the number of k-combinations of S must be $\binom{n}{k} = n!/((n-k)!k!)$.

Given an array a of n distinct elements and a non-negative integer k ($k \leq n$), suppose we need to generate all k-combinations of elements of a. In this article, we will discuss a simple recursive algorithm for the same and also write a non-recursive equivalent of that.

A Recursive Algorithm

We will use a single array c of k elements to generate each k-combination in it one by one. Let c_i denote the array element $c[i]$ for any index i. For any combination in c, say the elements $c_0, c_1, c_2, \ldots, c_{k-1}$ have come from array a's indices $j_0, j_1, j_2, \ldots, j_{k-1}$ respectively. One intuitive idea is to generate each combination in c such that, for that combination, $j_0 < j_1 < j_2 < \ldots < j_{k-1}$. That is, for each combination in c, the k elements appear in the same order in array a as they do in c.

Once we decide such ordering of elements in c, the following idea to generate any combination may come naturally to us. Pick c_0 from any of indices 0 to $n-k$ of a; we can't go further than $n-k$ because we still need to pick $(k-1)$ elements from higher indices of a. If c_0 was picked from index j_0 of a, c_1 can be picked from any of indices j_0+1 to $n-(k-1)$, say j_1. Further, c_2 can be picked from any of indices j_1+1 to $n-(k-2)$, say j_2. And so on.

To generate every possible combination, this process can be repeated for every possibility of $j_0, j_1, j_2, \ldots, j_{k-1}$ in the corresponding ranges.

The following recursive algorithm (implemented in C) is based on the above idea. *generate*(s, r) (s for "start", r for "remaining") generates all r-combinations of elements $a[s]$ to $a[n-1]$ over indices $(k-r)$ to $(k-1)$ of c. The initial call has $s = 0$ and $r = k$. *print*() prints the combination currently in c.

```
#define n 10
int a[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
int k;
int c[n];

void generate(int s, int r)
{
  int i;

  if(r == 0)
  {
    print();
    return;
  }

  for(i = s; i < n-r+1; i++)
  {
    c[k-r] = a[i];

    generate(i+1, r-1);
  }
}

void print()
{
  int i;

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

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

Consider the case when the elements of array a are in increasing order. We can prove that each combination generated by the above algorithm will have its elements in increasing order too. Further, these generated combinations are mutually ordered as per their lexicographical order. Note that the concept of a k-combination itself does not include any order for its k elements. Above we are referring to its representation in the array c, and any array has an intrinsic order of its elements.

Non-Recursive Equivalent

In the non-recursive equivalent of *generate*(), we will be simulating the recursive calls by ourselves. To do that, we will first make some observations about the *generate*() method and its execution flow.

First notice that, there can be up to $k+1$ active invocations of *generate*(), with argument r having values from k to 0. Further, for all $r > 0$, this method is simply a loop: a sequence of iterations for each i in the range s to $n - r$. To maintain this state i of (up to) k active invocations, we will use an array I of k integers. $I[k - r]$ will hold the value of i for invocation of $generate(s, r)$.

Now note that the method parameter s is used only to initialize i for the first iteration. Further, for every recursive invocation, the value of s is specified as "$(i+1)$" by the caller. So, while making the (simulated) recursive call, we will directly initialize 'i' of the new invocation using current 'i' plus 1, and avoid storing the argument s.

To keep track of the currently executing *generate*() call (top of the call-stack), we will use an integer r having value equal to argument r of the call. While making a (simulated) recursive call or returning, r will be decremented or incremented by 1 respectively.

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 either (a) a fresh *generate*() call is starting, or (b) a *generate*() call is resuming and starting a new iteration (the recursive call made by its earlier iteration has just returned). Case (b) implies $r > 0$.

```
void generate_nonrecur()
{
  int I[k], r;

  /* prepare to "execute" generate(0, k) */
  I[0] = 0;
  r = k;

  while(r <= k)
  {
    if(r == 0)
    {
      print();

      /* "return" to the caller */
      r = r + 1;
      continue;
    }

    /* for both cases (a) and (b), the 'i' of the iteration to
       execute is present in I[k-r] */

    if(I[k-r] < n-r+1)
    {
      c[k-r] = a[I[k-r]];

      /* prepare to "execute" generate(i+1, r-1) */
      if(r > 1)
        I[k-(r-1)] = I[k-r] + 1;      /* equivalent of "s = i+1" */

      /* if the new call has r = 0, no need to init its 'i' */

      I[k-r]++;

      /* make the recursive call */
      r = r - 1;
    }
    else
    {
      /* all iterations have already completed */
      /* "return" to the caller */
      r = r + 1;
    }
  }
}
```

■