The Methodology of Teaching Algorithms of Combinatorics: Permutations, Combinations, Arrangements

George MANDARIA*

Abstract

In this article we have described the methodology for teaching the algorithms of combinatorics which are often used when solving tasks of informatics. These are the economic types of tasks in which we need to select different objects, sort selected objects in some order and choose the best selection from all possible selections. The formulas of calculating number of such selections are known from mathematics, but in informatics we are interested not only in number, but also in selections themselves, which can be generated by special algorithms. In general, the number of such selections is quite large, so we need to use optimal algorithms to find the desired answer in real time. The essence of combinatoric objects is explained. It is shown how to find the desired object in optimal way. The samples and description of the corresponding algorithms are presented using the programming language C ++.

Keywords: Combinatorics, Methodology, Permutations, Combinations, Arrangements, Combinatorial object, Selections

Introduction

Without the algorithms of combinatorics, it would be impossible to solve many problems in informatics. It is one of the most powerful and well-known algorithmic tools, and therefore there are different algorithms that handle the same task. However, some of these algorithms are more difficult to understand, and some are relatively simple. That is why it is of great importance how we present it to beginner programmers while teaching. The essence, purpose and build methods of these algorithms need to be clearly and comprehensibly explained and the most understandable and effective algorithms must be selected.

As with all other sciences, the combinatorics also has its terminology and one of its notions is a ***combinatorial object***, also known as ***selection***. When choosing an element of m from a different number of n elements we say they form a selection of m number of elements from n. Depending on whether you have order of elements in selection or if all elements of n are included in selection or only part of it , there are three types of sselection. These are: permutations, combinations, arrangements.

The Methodology

Permutations

Selections, each of which is composed of n different elements that are in a certain order, is called permutations. It should be noted that the order of

* Assoc. Prof., Dr. International Black Sea University, Faculty of Computer Technologies and Engineering, E-mail: gmandaria@ibsu.edu.ge

elements in the permutation is important and all the n elements (m = n) are involved in the formation of it. The number of all different permutations drawn from n different elements is denoted by P_n. From mathematics is known the formula for calculating P_n: $P_n=n!$

Example: How many six-digit numbers can be compiled by numbers: 1, 3, 4, 5, 7 and 9, if none of these digits are repeated in any number.

Solution: The last digit of an even number must be even. Therefore, the last digit of each lookup can be just 4, since all the other digits are odd. Each of the remaining five digits can be placed on the remaining five places in any sequence. Therefore, number of possible permutations is: $P_5 = 5! = 120$.

Sometimes we are interested not only in finding the number of possible permutations but also in finding each of them.

There are several algorithms for generating all possible permutations for n different elements. They mainly differ by the sequence of permutation acceptance (Cormen Thomas, Leiserson Charles, Rivest Ronald, Stein Clifford, 2009).

We discuss the algorithm in which we are getting lexicographically sorted permutations. That means that for comparison of two permutations we compare elements with same indexes from left to right and the greater is the one in which the greater element was found first (for example S = (3, 5, 4, 6, 7) and L = (3, 5, 6, 4, 7), then S < L because $S_3 < L_3$).

Let's consider the algorithm for n = 5, which we can then expand for any possible value of n. Then, for convenience, we will work not with the elements themselves, but with their indices (from 1 to n). After identifying the indexes, the problem is no longer the issue of releasing the relevant elements according to these indexes (Mandaria, Pertakhia, Shioshvili, 2000).

Let's discuss the working principle of algorithm using example. Suppose you need to find all possible permutations of numbers 1, 2, 3, 4, 5. Because they are lexicographically sorted, the first one will be: 1, 2, 3, 4, 5 and the last one is: 5, 4, 3, 2, 1.

Let's assume that at some step of the work of algorithm we get the P, where P = (2, 3, 5, 4, 1) = (p_1, p_2, p_3, p_4, p_5). In order to determine the actual permutation of the next step, the following steps are required:

Step 1: Let's consider this permutation from right to left until we find the first element in the array that will be less that the element on the right, and stop immediately as soon as we find that element. In our given replacement it will be element 3 < 5, or element in array on index 1 (Indices are starting with 0). We will remember the index of this element (2, 3, 5, 4, 1);

Step 2: Let's consider the permutation from right to left again, until we find first element which will be greater than the element we've on our saved index. This will be an array element with index 3. We will remember the index of this element (2, 3, 5, 4, 1);

Step 3: We will switch the found elements;

Step 4: The part of the array, which is located at the right of the first element found, will be sorted by ascending order. Because this part is always sorted by descending order, their sorting by ascending order is not a problem - they can simply be reversed. The permutation which we get after it is the lexicographically next permutation. Let's note it as Q = (q_1, q_2, q_3, q_4, q_5).

We have to save indices of the elements in P array. Let's number elements from 1 to n. So, in the beginning we'll have elements from 0 to n ordered by

ascending order in a way, that every element of the array will be equal to it's index. 0th element is fictional. We will use it to define when must the algorithm stops working. The generation of permutations will be finished when this element will become a number different from zero.

Written in C++, our algorithm for generating permutations will look like this:

```
cin >> n;
for (i = 0; i <= n; i++)
p[i] = i;
while (p[0] == 0)
{
for (i = 1; i <= n; i++)
   cout << p[i];
   cout << endl;
   j = n;
   while (p[j-1] > p[j])                (1)
   j--;
   k = n;
   while (p[j-1] > p[k])
   k--;
   swap (p[j-1], p[k]);
   for (i = j; i <= (n+j) / 2; i++)
   swap (p[i], p[n-i+j]);
}
```

Combinations

Selections which are different from each other by at least one element and each of which contains m elements (m ≤ n), which are chosen from n different elements, are called combinations of m elements from n elements. The order of elements in the combination is not a factor.

The number of combinations of m elements from n is indicated with C(n,m) and it can be calculated with the following formula:

$$C(n, m) = \frac{n!}{m!(n-m)!} \quad (2)$$

It must be mentioned, that

$$C(n, m) = \frac{n!}{m!(n-m)!} = C(n, n-m)$$

Example: In how many different ways we can choose groups of 4 people from 10 people?

Solution: The number of such ways is combination of 4 elements from 10 and is equal to

$$C(10, 4) = \frac{10!}{4!(10-4)!} = \frac{7*8*9*10}{1*2*3*4} = 210$$

When solving some problems of informatics we may need to calculate not only number of combinations, but find every one of them. We'll provide an algorithm for finding all combinations of m elements from n below. Also, as in every permutation finding algorithm, we'll work not with actual elements, but with their indices.

When working with combinations, the order of elements is not a factor, so when calculating them it is handy to use increasing indexes of used elements which are used on current step (Their overall number is equal to m). We'll have to save current combination in B array. Select the following configuration (1, 2, 3, … , m) as initial combination for which the following equation is correct: B[j] = j, where j = 1, 2, 3, … , m.

The combinations we'll get will be lexicographically ordered, so last combination will be: (n – m + 1, n – m + 2, …, n - 1, n).

For each element of last combination the following condition is right: B[j] = n – m + j. For all remaining combinations the condition will fail for at least one element.

To generate next combination let's find B[j] element with maximum j index for which the following condition is correct:

B[j] < n – m + j

Along with this we must consider current combination from right to left. After this we must increase B[j] element by 1 and for each k > j assign incremented element value to each (k-1) element: B[k] = B[k-1] + 1. If such B[j] element does not exist, it means, that generation of combinations of m elements is finished (Kotov, Lapo, 2000).

Written in C++, corresponding algorithm will look like this:

```
cin >> n >> m;
for (i = 1; i <= m; i++)
b[i] = i;
do
{
   for (i = 1; i <= m; i++)
   cout << b[i];
   cout << endl;
   j = m;
   while (j > 0 && b[j] >= n – m + j)          (3)
   j--;
   if (j != 0)
      {
        b[j] = b[j] + 1;
        for (k = j+1; k <= m; k++)
        b[k] = b[k-1] + 1;
      }
}
while (j != 0);
```

Arrangements

Selections which are different from each other in composition or in their order, each of which contains m elements (m≤n), which are chosen from n different elements are called arrangements of m elements from n elements. The order of the elements in the arrangement is a factor.

Number of arrangements of m element from n is marked with A(n, m). Let's count it's value, thus count in how many ways we can to pick and place m (which is picked from n) number of elements in m number of different positions. On the first position we can place any element from n number of elements, on the second position we can place any element from the remaining (n-1) number of elements, on the third position we can place one from the remaining (n-2) elements and so on. On the second-to-last position by index (m-1) we can place any element from the remaining (n-(m-2)) elements and on the last position one from (n-(m-1)). Finally we'll get:

$$A(n, m) = n(n-1)(n-2)\ldots(n-(m-2))(n-(m-1)) = \frac{n!}{(n-m)!}$$

According to definition: 0! = 1.

Example: How many seven-digit phone numbers are there in which none of the digits are repeated?

Solution: This problem is based on finding all arrangements of 7 from 10. So, the number of such phone numbers is A(10,7) = 10*9*8*7*6*5*4 = 604 800

Now let's prove all combination formulas (2) mentioned above: C(n, m) method is used to select m number of elements from n number of different elements.

m! is a number of permutations can be found in every found combination. So, m!*C(n,m) is the number of all possible arrangements of m elements from n. So:

*A(n, m) = m! * C(n, m)*

At this point we can say, that number of all possible combinations of m elements from n is less than of all possible arrangements of m elements from n by m!. So:

$$C(n, m) = \frac{A(n,m)}{m!} = \frac{n!}{m!(n-m)!}$$

In the case when it's needed to find all arrangements of m elements from n, we can find all possible combinations of m elements from n by using the algorithm mentioned above (3) and for each of those combinations find all possible permutations also by using the algorithm mentioned above (1). Finally we will get all possible arrangements of m elements from n (Berov, Lapunov, Matiukhin, Ponomarev, 2000).

Written in C++, corresponding algorithm will look like this:

```
cin >> n >> m;
for (i = 1; i <= m; i++)
b[i] = i;
do
{
   permutations (m);
   j = m;
   while (j > 0 && b[j] >= n – m + j)
      j--;
      if (j != 0)
        {
          b[j] = b[j] + 1;
          for (k = j + 1; k <= m; k++)
          b[k] = b[k-1] + 1;
        }
}
while (j != 0);
```

(4)

```
void permutations(int l)
{
for (i = 0; i <= l; i++)
   p[i] = i;
while (p[0] == 0)
{
   for (i = 1; i <= l; i++)
     cout << b[p[i]];
     cout << endl;
   j = l;
   while (p[j-1] > p[j])
     j--;
   k = l;
   while (p[j-1] > p[k])
     k--;
   swap (p[j-1], p[k]);
   for (i = j; i <= (l+j) / 2; i++)
   swap (p[i], p[l - i + j]);
}
```

Conclusion

When using algorithms of combinatorics for solving problems it is of great importance to form the task in a right way, to see the combinatoric nature in it and to rightly define the combinatoric object. After this we must try to select the algorithm which will allow us to solve the problem and to output the solution in most convenient form using minimum computer resources (memory, time). It is very important to select the optimal algorithm and build it correctly.

References

1. H. Cormen Thomas, E. Leiserson Charles, L. Rivest Ronald, Stein Clifford. (2009). *Introduction*

to Algorithms (3rd Edition). Boston: Massachusetts Institute of Technology.

2. V.M. Kotov, I.A. Lapo. (2000). *Algorithmic Methods.* Minsk.
3. V.I. Berov, A.V. Lapunov, V.A. Matiukhin, A.E. Ponomarev. (2000). *Features of National Tasks in Informatics.* kirov.
4. G. Mandaria, B. Pertakhia, B Shioshvili. (2000). *Tasks in Informatics (Tasks, Solutions, Programs).* Tbilisi.