Esc101 Lecture

1st April, 2008

Generating Permutation

In this class we will look at a problem to write a program that takes as input 1,2,...,N and prints out all possible permutations of the numbers 1,2,...,N. For e.g. given 1 2 3 it will print all

(1 2 3)
(1 3 2)
(2 1 3)
(2 3 1)
(3 1 2)
(3 2 1)

How do we break this problem up into smaller problems? One way to do it is the following.

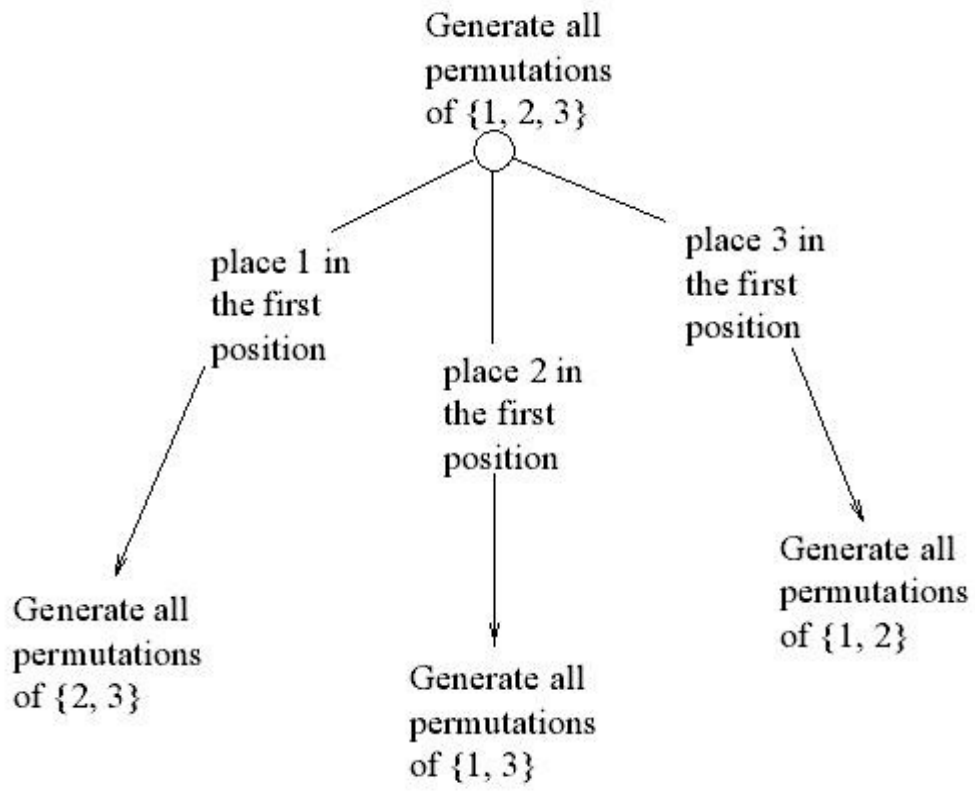
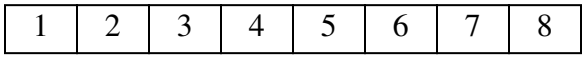


Figure 1: The problem of generating all permutations of {1,2,3} has been reduced to the problems of generating all permutations of {1,2}, {1,3}, and {2,3}.

In general, all permutations of ***N*** elements can be expressed in terms of all permutations of ***N - 1*** elements. Having a set of ***N*** elements, we take one of them and append it to all the permutations of the remaining ***N - 1*** elements. Repeating it for every element of N gives us all permutations of ***N*** elements. Unsurprisingly, it fits well into the known formula of the amount of permutations of ***N*** elements: ***N! = N(N-1)!***

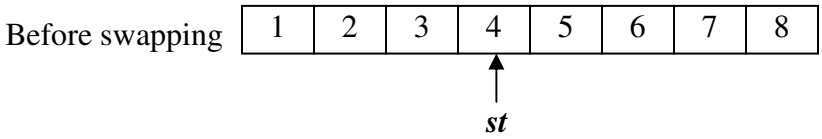
Consider we are given all the elements in an array of size N. The indexing of array starts from 0.

.



Array

For a given instance consider we have reached at position ***st*** , (i.e we have fixed the position of elements at index up to ***st-1***),in the array, where $0 \leq st < N$. For all ***i***, such that $st < i < N$, we will pick element from index ***i*** and swap it with element at index ***st*** then generate all the possible permutation of elements from index ***st+1*** to ***N***.



After swapping the element at position ***st*** with the element at position ***i*** , the array looks like following.

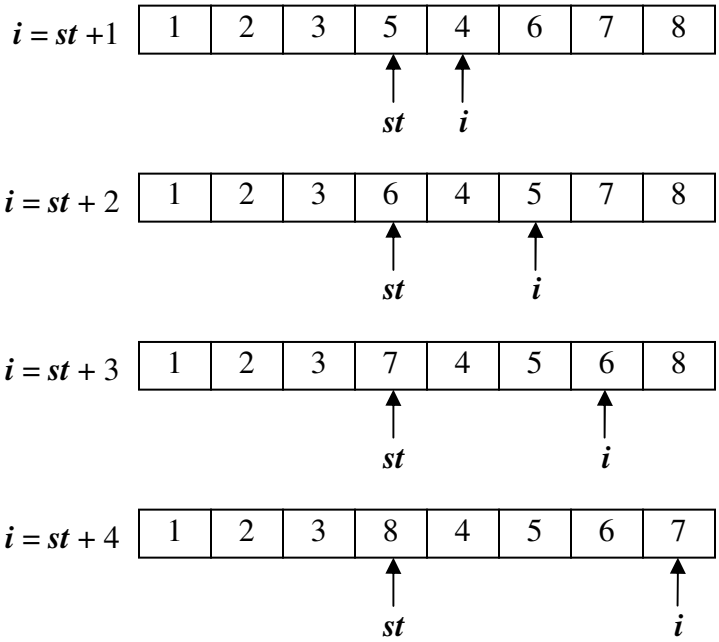


Figure 2: For $N = 8$ $st=3$

Once we have iterated through all the values of ***i,*** from ***st+1*** up to ***N .*** Note that we have altered the position of elements. We need to put the elements back to its place as they were before this swapping.

If we look at the above figure 2 we would notice that the elements are right shifted by 1 position, starting from ***st***, whereas the last element is now at position ***st***. So in order to get the elements back to their place where they where before the swap, we need to left shift the data starting from ***st+1*** up to ***(N -1)***, the element in ***st*** will be moved to (***N-1)***th position.

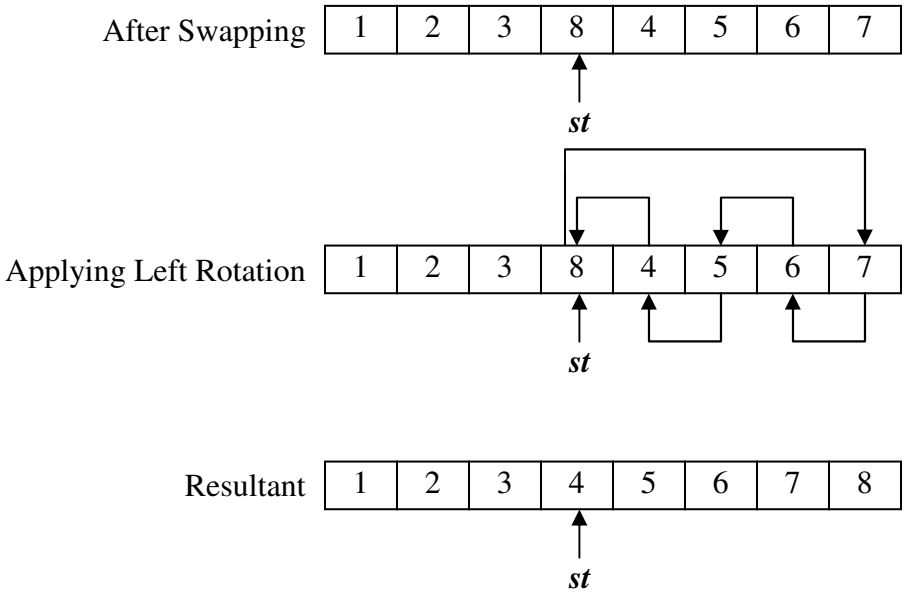


Figure 3: Left Rotation

We still need to figure out the base cases. We know that 1! = 1 thus when ***st*** reaches the end of array we are done i.e ***st > N-1***, then what left is to print the elements.

The code that implements the above idea is below.

```
class Permutation
{
        static int numPermutation=0;
        static void Permute_items(int a[],int st)
        {
                int i,t;
```

```
        /* Base case */
        if(st >a.length-1)
        {
                Print_Permutation(a);
                return;
        }

        /* Generate the permutaion of st+1 to N */
        Permute_items(a,st+1);


        /*
         Generating Permutaion after swapping of element at position i with
        element at  position st
        */
        for(i=st+1;i<a.length;i++)
        {
                /* Swapping element at position st with element at position i */
                t=a[st];
                a[st]=a[i];
                a[i]=t;

                /* Generating the permutation st+1 to N*/
                Permute_items(a,st+1);
        }

        /* Applying Left rotation to put the elements back to its correct place */

        t=a[st+1]; // Temporarily copying the element at st position.

        /* Left rotating */
        for(i=st+1;i<a.length;i++)
        {
                a[i-1]=a[i];
        }

        a[a.length-1]=t; //copying the element at st position to (N-1)th position
}

/* Printing the content of the array which represents a permutation */
public static void Print_Permutation(int a[])
{
        int i;

        System.out.print("( ");
```

```
            for(i=0;i<a.length;i++)
            {
                    System.out.print(" "+a[i] );
            }
            System.out.print(" )");

            /*  Incrementig the total count of permutation */
            numPermutation +=1;
        }


        public static void main(String args[])
        {
            int a[]=new int[args.length];
            int i;

            try
            {
                    for(i=0;i<args.length;i++)
                            a[i]=Integer.parseInt(args[i]);
            }
            catch (Exception e)
            {
                    System.err.println(" Error in input "+ e);
                    System.exit(-1);
            }
            Permute_items(a,0);
            System.out.println(" Total no of Permutation "+numPermutation);

        }
}
```

In the above code we have used try -catch block because the method parstInt(String s) of class Integer throws an instance of NumberFormatException which needs to be handle.

For e.g.

```
int a=Interger.parseInt("07A");
```

It will throw "NumberFormatException" at runtime. Hence such things need to be handled. When the exceptions are caught, the code in the corresponding catch block is executed.

Here when we encounter such an exception we need to stop the execution of the program for that we use the method exit(int rv) of class System. The parameter -1 is to indicate that the program terminated due to exceptions.

Time Complexity

The recurrence relation of the permutation is as follows
$T(i) = (i) * T(i-1) + k_1$;
$T(0) = k_2$;
Thus, the time complexity would be O(n!).

Data Structure
A **data structure** is a way of representing data in a computer so that it can be used efficiently.

For e.g. Stack, Queue, Linked List.

Linked List

Consider a problem of reading an array of elements from user.

What we usually do is to ask how many elements user wants to enter and then create an array of that much size. Then read the elements one by one and store them in the created array. The disadvantage of such design is that we need to specify the no. of elements before specifying the elements.

The other way could be to assume some large size of array and create and array of that size. Now just read the elements one by one and store them in the array. There may be situations where the users may enter elements more than the array could handle. The other situation could be that the size of the array declared is too large, and most of the time the user is utilizing only minor fractional part of the whole array.

Hence, even this scheme is also not efficient.

The efficient way of handling such scenario is with the use of Linked List. Linked list is a kind of data structure used for dynamic size of input.

In link list consists of chain of Nodes. Each node consists of two fields; one is called the data field and other the reference field. There may be more than one data and reference field in a Node.

The reference field contains the reference of the next element in the List. Hence the elements are linked through the reference field, hence the name Link List. Each Link List has special reference called head to access the initial element in the list and the reference of the last element is ground (null).

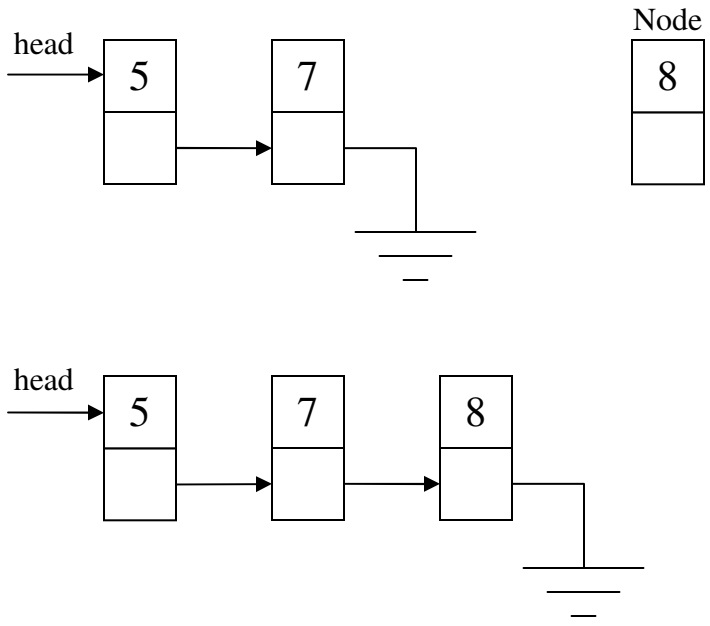


Figure 4: Insertion of a Node in the Linked List

We will carry on with the details of Linked List in next lecture.