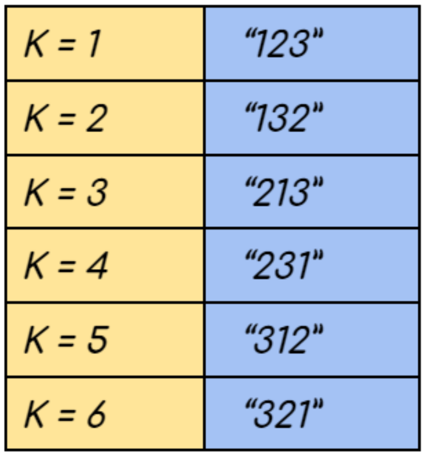
Find K-th Permutation Sequence

Problem Statement: Given **N** and **K**, where N is the sequence of numbers from **1 to N([1,2,3..... N])** find the **Kth permutation sequence**.

For N = 3 the 3! Permutation sequences in order would look like this:-



K = 1	"123"
K = 2	"132"
K = 3	"213"
K = 4	"231"
K = 5	"312"
K = 6	"321"

Note: 1<=K<=N! Hence for a given input its Kth permutation always exists

Examples:

```
Example 1:

Input: N = 3, K = 3

Output: “213”

Explanation: The sequence has 3! permutations as illustrated in the figure above. K = 3
corresponds to the third sequence.

Example 2:

Input: N = 3, K = 5
```

Result: "312"

Explanation: The sequence has 3! permutations as illustrated in the figure above. K = 5 corresponds to the fifth sequence.

Solution

***Disclaimer**: Don't jump directly to the solution, try it out yourself first.*

Solution 1: Brute Force Solution

Approach:

The extreme naive solution is to generate all the possible permutations of the given sequence. This is achieved using recursion and every permutation generated is stored in some other data structure (here we have used a vector). Finally, we sort the data structure in which we have stored all the sequences and return the Kth sequence from it.

Code:

```
public class Main {
    static void swap(char s[], int i, int j) {
        char ch = s[i];
        s[i] = s[j];
        s[j] = ch;
    }
    static void solve(char s[], int index, ArrayList < String > res) {
        if (index == s.length) {
            String str = new String(s);

            res.add(str);
            return;
        }
        for (int i = index; i < s.length; i++) {
            swap(s, i, index);
            solve(s, index + 1, res);
            swap(s, i, index);
        }
    }

    static String getPermutation(int n, int k) {
        String s = "";
        ArrayList < String > res = new ArrayList < > ();
        for (int i = 1; i <= n; i++) {
            s += i;
        }
        solve(s.toCharArray(), 0, res);
        Collections.sort(res);

        return res.get(k);
     }
   }
```

Time complexity: **O(N! * N) +O(N! Log N!)**

Reason: The recursion takes O(N!) time because we generate every possible permutation and another O(N) time is required to make a deep copy and store every sequence in the data structure. Also, *O(N! Log N!) time required to sort the data structure*

Space complexity: O(N)

Reason: Result stored in a vector*, we are auxiliary space taken by recursion*

Solution 2:(Optimal Approach)

Say we have N = 4 and K = 17. Hence the number sequence is {1,2,3,4}.

Intuition:

Since this is a permutation we can assume that there are four positions that need to be filled using the four numbers of the sequence. First, we need to decide which number is to be placed at the first index. Once the number at the first index is decided we have three more positions and three more numbers. Now the problem is shorter. We can repeat the technique that was used previously until all the positions are filled. The technique is explained below.

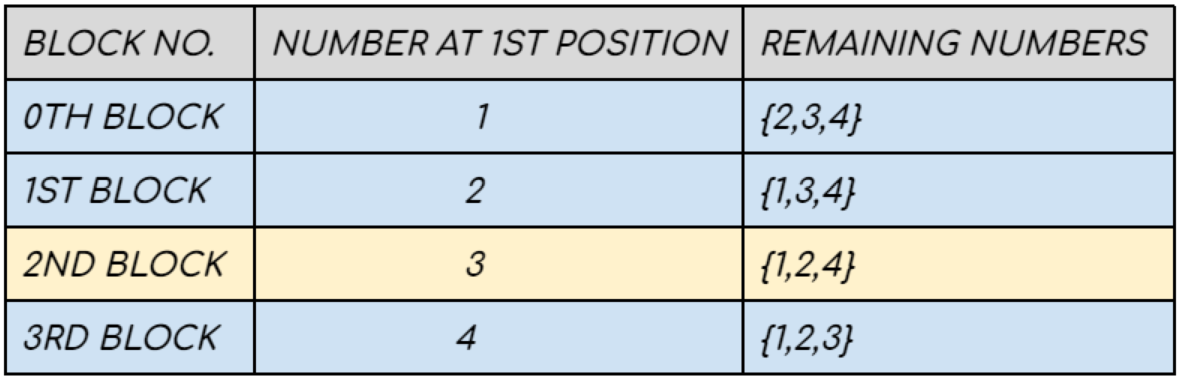
Approach:

STEP 1:

Mathematically speaking there can be 4 variations while generating the permutation. We can have our permutation starting with either 1 or 2 or 3 or 4. If the first position is already occupied by one number there are three more positions left. The remaining three numbers can be permuted among themselves while filling the 3 positions and will generate 3! = 6 sequences. Hence each block will have 6 permutations adding up to a total of 6*4 = 24 permutations. If we consider the sequences as 0-based and in the sorted form we observe:-

- The **0th – 5th permutation** will start with 1
- The **6th – 11th permutation** will start with 2
- The **12th – 17th permutation** will start with 3
- The **18th – 23rd permutation** will start with 4.

(For better understanding refer to the picture below.)



BLOCK NO.	NUMBER AT 1ST POSITION	REMAINING NUMBERS
0TH BLOCK	1	{2,3,4}
1ST BLOCK	2	{1,3,4}
2ND BLOCK	3	{1,2,4}
3RD BLOCK	4	{1,2,3}

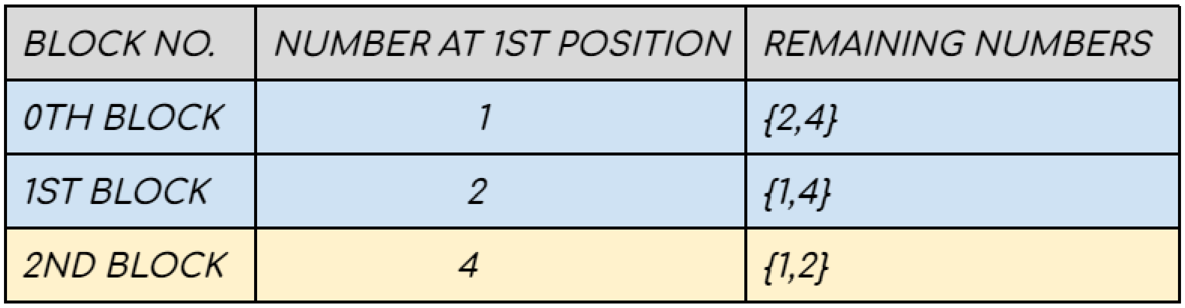
We make **K = 17-1 considering 0-based indexing**. Since each of the four blocks illustrated above comprises 6 permutations, therefore, the 16th permutation will lie in (16 / 6) = 2nd block, and our answer is the (16 % 6) = 4th sequence from the 2nd block. Therefore 3 occupies the first position of the sequence and **K = 4.**

_3_ ____ ____ ____

STEP 2:

Our new search space comprises three elements {1,2,4} where K = 4 . Using the previous technique we can consider the second position to be occupied can be any one of these 3 numbers. Again one block can start with 1, another can start with 2 and the last one can start with 4 . Since one position is fixed, the remaining two numbers of each block can form **2! = 2 sequences.** In sorted order :

- The **0th – 1st sequence** starts with 1
- The **2nd – 3rd sequence** starts with 2
- The **4th – 5th sequence** starts with 4



BLOCK NO.	NUMBER AT 1ST POSITION	REMAINING NUMBERS
0TH BLOCK	1	{2,4}
1ST BLOCK	2	{1,4}
2ND BLOCK	4	{1,2}

The 4th permutation will lie in **(4/2) = 2nd block** and our answer is the **4%2 = 0th sequence from the 2nd block**. Therefore **4** occupies the second position and **K = 0.**

3 4 ____ ____

STEP 3:

The new search space will have two elements **{1 ,2}** and **K = 0**. One block starts with 1 and the other block starts with 2. The other remaining number can form only one **1! = 1 sequence**. In sorted form –

- The **0th sequence** starts with 1
- The **1st sequence**. starts with 2

BLOCK NO.	NUMBER AT 1ST POSITION	REMAINING NUMBERS
0TH BLOCK	1	{2}
1ST BLOCK	2	{1}

The 0th permutation will lie in the **(0/1) = 0th block** and our answer is the 0%1 = **0th sequence from the 0th block**. Therefore **1** occupies the 3rd position and **K = 0.**

3 4 1 ____

STEP 4:

Now the only block has 2 in the first position and no **remaining number is present**.

BLOCK NO.	NUMBER AT 1ST POSITION	REMAINING NUMBERS
0Th BLOCK	2	{}

This is the point where we place 2 in the last position and stop.

The final answer is 3412.

```
public String getPermutation(int n, int k) {
        int fact=1;
        ArrayList<Integer> con=new ArrayList();
        for(int i=1;i<n;i++)
        {
            fact*=i;
            con.add(i);
        }
        k=k-1;
        con.add(n);
        String ans="";
        while(true)
        {
            int ind=k/fact;
            ans+=String.valueOf(con.get(ind));
            con.remove(ind);
            if(con.size()<=0)
                break;
            k=k%fact;
            fact=fact/con.size();
        }
        return ans;
    }
```