Kth Permutation sequence

BRUTE FORCE :

Use recursion to generate all permutation sequence. choose an index and perform swap with every other index while backtracking place it to original by again swapping.

```
#include <bits/stdc++.h>
void helper(int ind,int n,string& s,vector<string>& ans)
{
    if(ind==s.size())
    {
        ans.push_back(s);
        return;
    }
    for(int i=ind;i<s.size();i++)
    {
        swap(s[ind],s[i]);
        helper(ind+1,n,s,ans);
        swap(s[ind],s[i]);
   }

}
string kthPermutation(int n, int k) {
    // Write your code here.
    string s;
    int cnt=0;
    vector<string> ans;
    for(int i=1;i<=n;i++)
    {
        s+=to_string(i);
    }
    helper(0,n,s,ans);
    sort(ans.begin(),ans.end());
    auto it=ans.begin()+(k-1);
    return *it;


}
```

- Time Complexity :O(N*N!) + O(N!*N log (N!*N))
- Space Complexity : O(N)

Optimal Approach :

Using simple maths fix a number position and find out the permutation that can be made using rest of the numbers.

for ex: 1234 total permutation possible are 24=4!

but if we fix 1 in first position then 3! permutation are possible and so on we are given k if we divide it by n-1! we will get the number which needs to be fixed now to find the next number to be fixed we reduce k and fact because now remaining numbers are 3 so k will now be k%fact and for 2 numbers fact will be fact/number.size() we keep on deleting a number from number array when its position gets fixed so dividing fact by size will reduce fact.

```
string kthPermutation(int n, int k) {
    // Write your code here.
    vector<int> numbers;
    string ans="";
    int fact=1;
    for(int i=1;i<n;i++)
    {
        fact=fact*i;
        numbers.push_back(i);
    }
    numbers.push_back(n);
    k=k-1;
    while(true)
    {


        ans+=to_string(numbers[k/fact]);
        numbers.erase(numbers.begin()+k/fact);
        if(numbers.size()==0)
            break;
        k=k%fact;
        fact=fact/numbers.size();


    }
    return ans;
}
```

- Time Complexity : O(N*N)

Reason: We are placing N numbers in N positions. This will take O(N) time. For every number, *we are reducing the search space by removing the element already placed in the previous step. This takes another O(N) time.*

- Space Complexity : O(N)