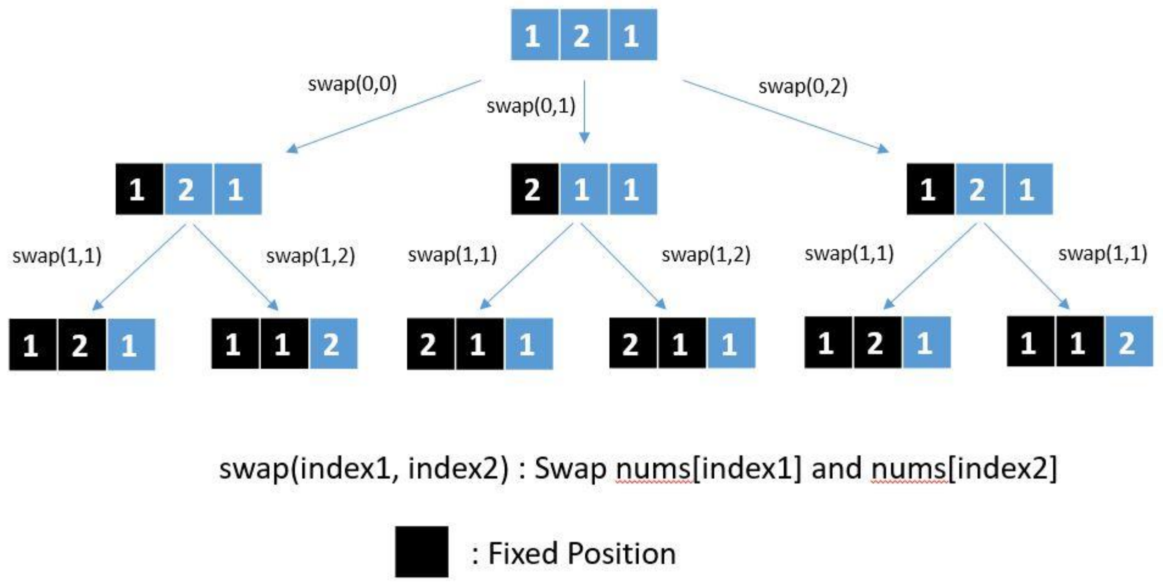
Problem Link : https://leetcode.com/problems/permutations-ii/

Approach 1 :
The easiest way is to generate all the possible permutation of the numbers without caring for duplicates. Finally inserting all the permutation in the set to eliminate duplicate entries. The entries present in the set represent unique permutations. To get all the permutation we make use of the function permute(int pos, vector<int> nums), where pos represent the index we are currently at in the vector. At each step, we swap all the elements which are to the right of the current position. Then we fix the current position and recursively do the same thing with the next index until we reach the end of the array. Reaching the end will give us a permutation (not necessarily unique).

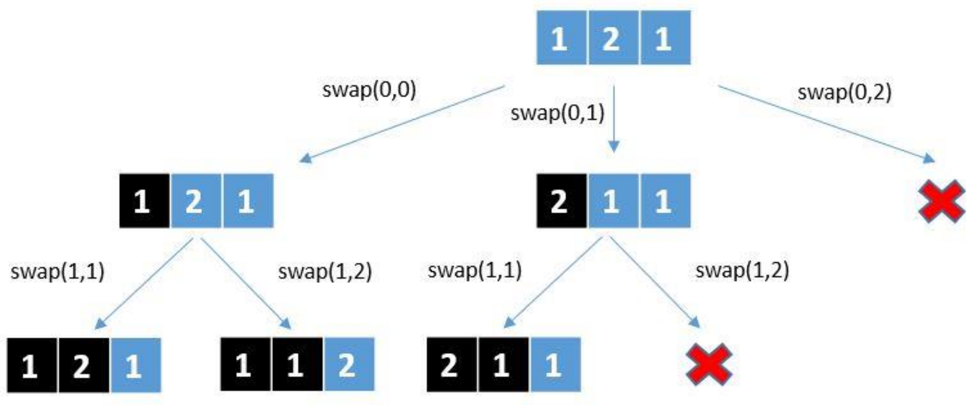


```
void permute(int pos,vector<int> &nums,set<vector<int>> &ans,int n) {
   if(pos==n) {
      ans.insert(nums);
      return;
   }

   for(int next_pos=pos ; next_pos<n ; next_pos++) {

      swap(nums[pos],nums[next_pos]);
      permute(pos+1,nums,ans,n);
      swap(nums[pos],nums[next_pos]);

   }
}
class Solution {
public:
    vector<vector<int>> permuteUnique(vector<int>& nums) {

        set<vector<int>> res;
        int n = nums.size();
        permute(0,nums,res,n);
        vector<vector<int>> ans(res.begin(), res.end());
        return ans;
    }
};
```

Time Complexity: O(n! * n * log(n!))
If we don't care about duplicate permutations then there will be n! permutations. Insertion in the set will be of the order (Size of Permutation * log(Number of elements in the set)). So in the worst case, the insertion will be of the order n*log(n!)

Space Complexity: O(n! * n)
There will be n! permutations and the size of the permutation is n. Hence the space complexity will be O(n! * n)

Approach 2 :
The idea is similar to the previous approach but we at each step we ensure that we do not swap the current index with two different indices with the same values, otherwise it will lead to two branches which will produce exactly the same permutations at the end. Thus we create a set of integers at each step which will tell us whether the element at any index has not been previously swapped or not. If not then we insert that value in the set so that in future we do not swap the same value (present at different index) with the current index. Hence we make only necessary calls to ensure that we only have unique permutations.



swap(index1, index2) : Swap nums[index1] and nums[index2]

■ : Fixed Position

```
void permute(int pos,vector<int> &nums,vector<vector<int>> &ans,int n) {
   if(pos==n) {
      ans.push_back(nums);
      return;
   }

   unordered_set<int> us;
   for(int next_pos=pos ; next_pos<n ; next_pos++) {
      if(us.count(nums[next_pos])>0)
         continue;

      us.insert(nums[next_pos]);

      swap(nums[pos],nums[next_pos]);
      permute(pos+1,nums,ans,n);
      swap(nums[pos],nums[next_pos]);

   }
}

class Solution {
public:
    vector<vector<int>> permuteUnique(vector<int>& nums) {
        vector<vector<int>> ans;
        int n = nums.size();
        permute(0,nums,ans,n);
        return ans;
    }
};
```

Time Complexity: O(n! * n)
In the worst-case, there can be n! unique permutations and inserting each permutation in the final result will take O(n) time.

Space Complexity: O(n! * n)
There will be n! permutations and the size of the permutation is n. Hence the space complexity will be O(n! * n)