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

Given a collection of numbers, nums, that might contain duplicates, return *all possible unique permutations* ***in any order***.

Example 1:

Input: nums = [1,1,2]

Output:

[[1,1,2],

[1,2,1],

[2,1,1]]

Example 2:

Input: nums = [1,2,3]

Output: [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]

Constraints:

- 1 <= nums.length <= 8
- -10 <= nums[i] <= 10

Attempt 1: 2022-10-20

Solution 1: Backtracking style 1 (10min)

Style 1: With redundant 'index' parameter

```
class Solution {
    public List<List<Integer>> permuteUnique(int[] nums) {
        List<List<Integer>> result = new ArrayList<List<Integer>>();
        boolean[] visited = new boolean[nums.length];
        Arrays.sort(nums);
        helper(nums, result, new ArrayList<Integer>(), visited, 0);
        return result;
    }

    private void helper(int[] nums, List<List<Integer>> result, List<Integer> tmp, boolean[] visited, int index) {
        if(tmp.size() == nums.length) {
            result.add(new ArrayList<Integer>(tmp));
            return;
```

```
        }
        for(int i = index; i < nums.length; i++) {
            if(visited[i] || (i > 0 && !visited[i - 1] && nums[i] == nums[i - 1])) {
                continue;
            }
            tmp.add(nums[i]);
            visited[i] = true;
            helper(nums, result, tmp, visited, index);
            tmp.remove(tmp.size() - 1);
            visited[i] = false;
        }
    }
}

Time Complexity: O(N * N!)
Space Complexity: O(N)
```

Style 2: Without redundant 'index' parameter

```
class Solution {
    public List<List<Integer>> permuteUnique(int[] nums) {
        List<List<Integer>> result = new ArrayList<List<Integer>>();
        boolean[] visited = new boolean[nums.length];
        Arrays.sort(nums);
        helper(nums, result, new ArrayList<Integer>(), visited);
        return result;
    }

    private void helper(int[] nums, List<List<Integer>> result, List<Integer> tmp, boolean[] visited) {
        if(tmp.size() == nums.length) {
            result.add(new ArrayList<Integer>(tmp));
            return;
        }
        for(int i = 0; i < nums.length; i++) {
            if(visited[i] || (i > 0 && !visited[i - 1] && nums[i] == nums[i - 1])) {
                continue;
            }
```

```
            tmp.add(nums[i]);
            visited[i] = true;
            helper(nums, result, tmp, visited);
            tmp.remove(tmp.size() - 1);
            visited[i] = false;
        }
    }
}

Time Complexity: O(N * N!)
Space Complexity: O(N)
```

Refer to

https://leetcode.com/problems/permutations-ii/discuss/18594/Really-easy-Java-solution-much-easier-than-the-solutions-with-very-high-vote/121098

The worst-case time complexity is O(n! * n).

For any recursive function, the time complexity is O(branches^depth) * amount of work at each node in the recursive call tree. However, in this case, we have n*(n-1)*(n*2)*(n-3)*...*1 branches at each level = n!, so the total recursive calls is O(n!)

We do n-amount of work in each node of the recursive call tree, (a) the for-loop and (b) at each leaf when we add n elements to an ArrayList. So this is a total of O(n) additional work per node.

Therefore, the upper-bound time complexity is O(n! * n).

Refer to

https://leetcode.com/problems/permutations/discuss/1527929/Java-or-TC%3A-O(N*N!)-or-SC%3A-O(N)-or-Recursive-Backtracking-and-Iterative-Solutions

Time Complexity: O(N * N!). Number of permutations = P(N,N) = N!.

Each permutation takes O(N) to construct

T(n) = n*T(n-1) + O(n)

T(n-1) = (n-1)*T(n-2) + O(n-1)

...

T(2) = (2)*T(1) + O(2)

T(1) = O(N) -> To convert the nums array to ArrayList.

Above equations can be added together to get:

T(n) = n + n*(n-1) + n*(n-1)*(n-2) + ... + (n....2) + (n....1) * n

= P(n,1) + P(n,2) + P(n,3) + ... + P(n,n-1) + n*P(n,n)

= (P(n,1) + ... + P(n,n)) + (n-1)*P(n,n)

= Floor(e*n! - 1) + (n-1)*n!

= O(N * N!)

Space Complexity: O(N). Recursion stack.

N = Length of input array.

Refer to

https://leetcode.com/problems/permutations-ii/discuss/18594/Really-easy-Java-solution-much-easier-than-the-solutions-with-very-high-vote/324818

The difficulty is to handle the duplicates. With inputs as [1a, 1b, 2a]. If we don't handle the duplicates, the results would be: [1a, 1b, 2a], [1b, 1a, 2a] ..., so we must make sure 1a goes before 1b to avoid duplicates by using nums[i - 1] == nums[i] && !used[i - 1], we can make sure that 1b cannot be chosen before 1a

http://www.jiuzhang.com/solutions/permutations-ii/

```java
public class Solution {
    public List<List<Integer>> permuteUnique(int[] nums) {
        List<List<Integer>> result = new ArrayList<List<Integer>>();
        if(nums == null || nums.length == 0) {
            return result;
        }
        Arrays.sort(nums);
        List<Integer> combination = new ArrayList<Integer>();
        helper(nums, result, combination, new boolean[nums.length]);
        return result;
    }

    private void helper(int[] nums, List<List<Integer>> result, List<Integer> combination, boolean[] used) {
        if(combination.size() == nums.length) {
            result.add(new ArrayList<Integer>(combination));
        }
        for(int i = 0; i < nums.length; i++) {
            /*
            判断主要是为了去除重复元素影响。
            比如，给出一个排好序的数组，[1,2,2]，那么第一个2和第二2如果在结果中互换位置，
            我们也认为是同一种方案，所以我们强制要求相同的数字，原来排在前面的，在结果
```

```
            当中也应该排在前面，这样就保证了唯一性。所以当前面的2还没有使用的时候，就
            不应该让后面的2使用。
            */
            if(used[i] || (i > 0 && !used[i - 1] && nums[i] == nums[i - 1])) {
                continue;
            }
            used[i] = true;
            combination.add(nums[i]);
            helper(nums, result, combination, used);
            combination.remove(combination.size() - 1);
            // Don't forget to reset 'used' boolean flag back to false
            used[i] = false;
        }
    }
}
```

Solution 2: Backtracking style 2 (10min, no Arrays.sort(), no boolean visited, but frequency Hash Map)

```
class Solution {
    public List<List<Integer>> permuteUnique(int[] nums) {
        List<List<Integer>> result = new ArrayList<List<Integer>>();
        //boolean[] visited = new boolean[nums.length];
        //Arrays.sort(nums);
        Map<Integer, Integer> freq = new HashMap<Integer, Integer>();
        for(int num : nums) {
            freq.put(num, freq.getOrDefault(num, 0) + 1);
        }
        helper(nums, result, new ArrayList<Integer>(), freq, 0);
        return result;
    }

    private void helper(int[] nums, List<List<Integer>> result, List<Integer> tmp, Map<Integer, Integer> freq, int index) {
        if(tmp.size() == nums.length) {
            result.add(new ArrayList<Integer>(tmp));
            return;
```

```
        }
        // Instead of loop on 'nums' array, loop on key set of frequency map since
        // we only iterate over the unique value to pick up
        for(int k : freq.keySet()) {
            // Only when unique value's frequency > 0, we are allowed to choose
            if(freq.get(k) > 0) {
                tmp.add(k);
                freq.put(k, freq.get(k) - 1);
                helper(nums, result, tmp, freq, index);
                // Why backtrack?
                // After DFS done and hit base case to store current combination, we
                // have to restore the statistics, prepare for next for loop
iteration
                // which start from new unique value to build new combination
                tmp.remove(tmp.size() - 1);
                freq.put(k, freq.get(k) + 1);
            }
        }
    }
}

```

Refer to

https://leetcode.com/problems/permutations-ii/discuss/18594/Really-easy-Java-solution-much-easier-than-the-solutions-with-very-high-vote/121098

The worst-case time complexity is O(n! * n).

For any recursive function, the time complexity is O(branches^depth) * amount of work at each node in the recursive call tree. However, in this case, we have n*(n-1)*(n*2)*(n-3)*...*1 branches at each level = n!, so the total recursive calls is O(n!)

We do n-amount of work in each node of the recursive call tree, (a) the for-loop and (b) at each leaf when we add n elements to an ArrayList. So this is a total of O(n) additional work per node.

Therefore, the upper-bound time complexity is O(n! * n).

Refer to

https://leetcode.com/problems/permutations/discuss/1527929/Java-or-TC%3A-O(N*N!)-or-SC%3A-O(N)-or-Recursive-Backtracking-and-Iterative-Solutions

Time Complexity: O(N * N!). Number of permutations = P(N,N) = N!.

Each permutation takes O(N) to construct

T(n) = n*T(n-1) + O(n)

T(n-1) = (n-1)*T(n-2) + O(n-1)

...

T(2) = (2)*T(1) + O(2)

T(1) = O(N) -> To convert the nums array to ArrayList.

Above equations can be added together to get:

T(n) = n + n*(n-1) + n*(n-1)*(n-2) + ... + (n....2) + (n....1) * n

= P(n,1) + P(n,2) + P(n,3) + ... + P(n,n-1) + n*P(n,n)

= (P(n,1) + ... + P(n,n)) + (n-1)*P(n,n)

= Floor(e*n! - 1) + (n-1)*n!

= O(N * N!)

Space Complexity: O(N). Recursion stack.

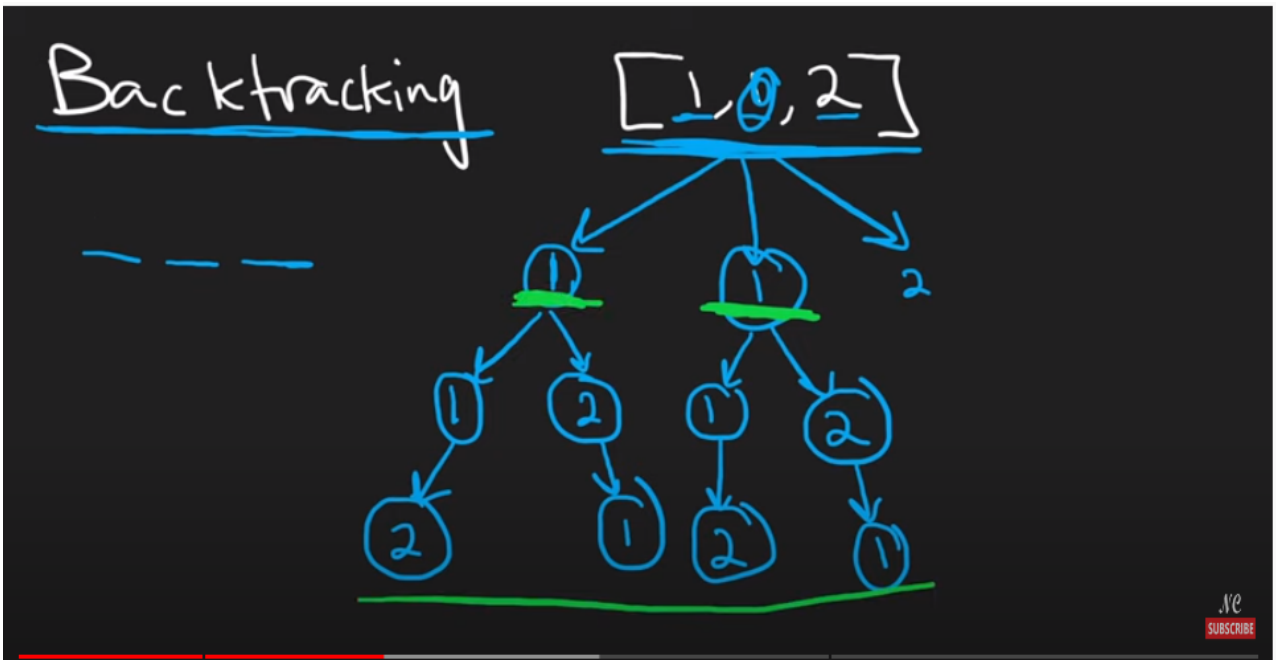
N = Length of input array.

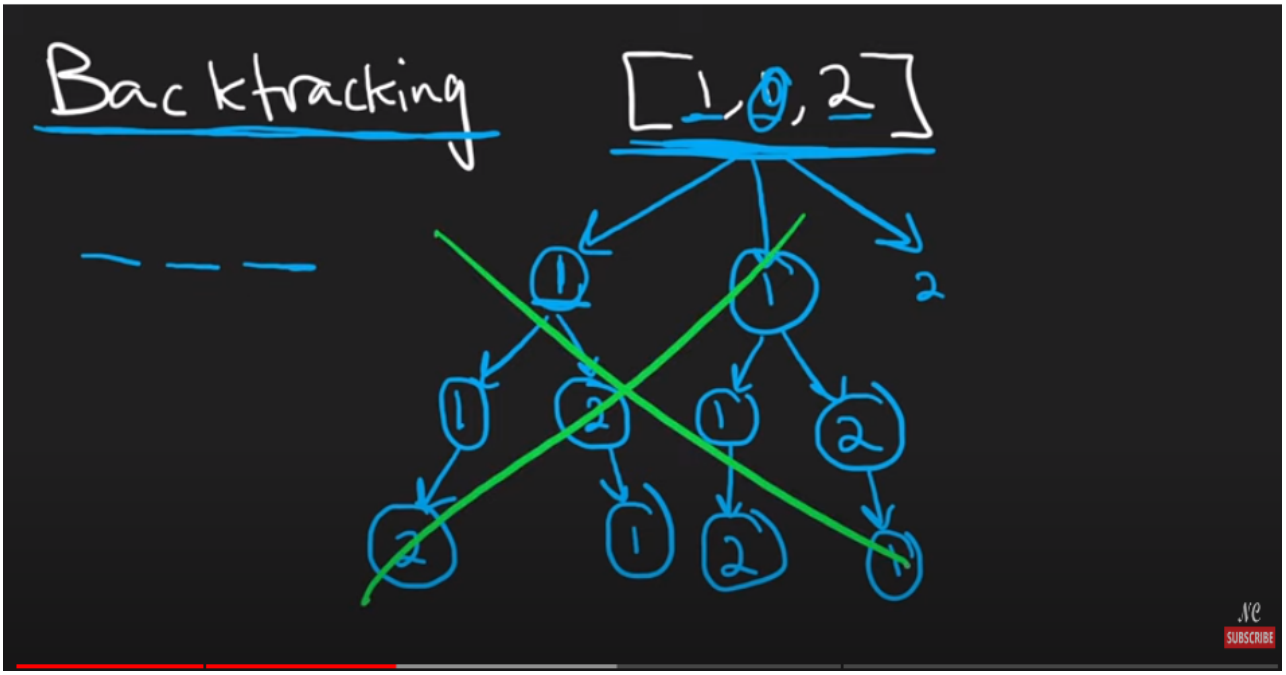
Video explain for Solution 2: Backtracking style 2 how to work with HashMap

Permutations II - Backtracking - Leetcode 47

https://www.youtube.com/watch?v=qhBVWf0YafA

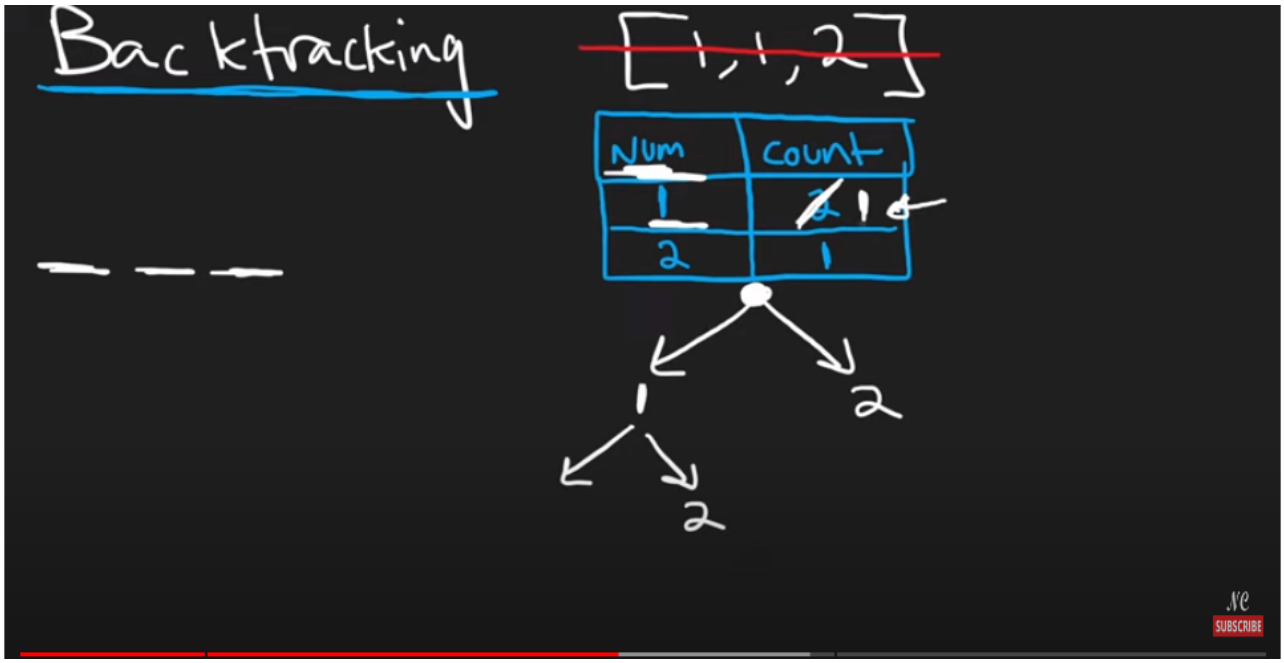
We cannot use duplicate elements in the same position which will result into same permutation



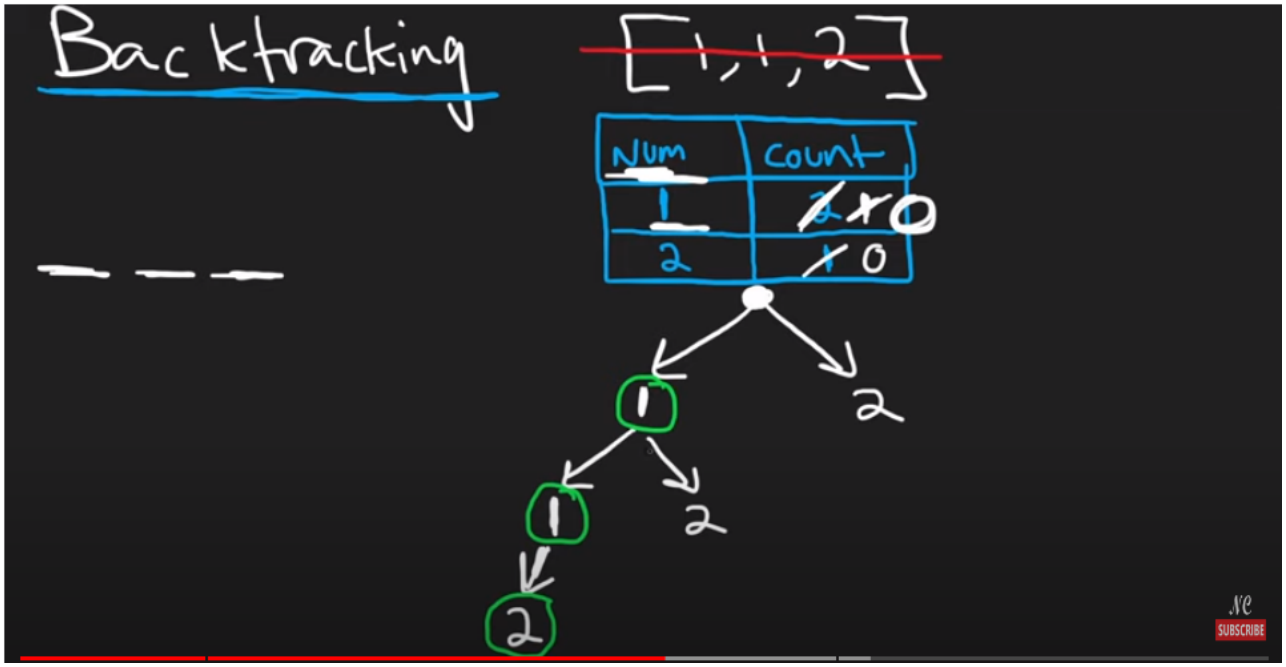


Create frequency table based on given input, when use an element decrease the frequency of that element

e.g initially element 1 frequency is 2, after using one of them, decrease 1 from frequency as 2 - 1 = 1



After using two 1 and one 2 to build one permutation {1,1,2}, both element 1 and 2's frequency drop to 0



Refer to

https://leetcode.com/problems/permutations-ii/discuss/1768113/Java-Backtracking-HashMap

```java
class Solution {
    public List<List<Integer>> permuteUnique(int[] nums) {
        List<List<Integer>> result = new ArrayList();
        Map<Integer, Integer> map = new HashMap();
        for (int num:nums)
            map.put(num, map.getOrDefault(num,0)+1);
        backtracking(nums, result, map, new ArrayList<Integer>());
        return result;
    }

    private void backtracking(int[] nums, List<List<Integer>> result, Map<Integer, Integer> map, List<Integer> list){
        if (list.size() == nums.length){
            result.add(new ArrayList<Integer>(list));
                return;
        }
        for (Integer key: map.keySet()){
            if (map.get(key)>0){
                list.add(key);
                map.put(key, map.get(key) -1);
                backtracking(nums, result, map, list);
                map.put(key, map.get(key) +1);
```

```
                list.remove(list.size() -1);
            }
        }
    }
}
```

No "Not pick" and "Pick" branch available for this problem yet (normal decision tree not gonna work)

Because in L47. we have to use all numbers in the given array, not like L77. we just pick k out of n numbers, so there is no chance for a number in L47 to 'Not pick', hence no "Not pick" and "Pick" strategy here

Refer to

L46.P11.3.Permutations (Ref.L77)

L90.P11.2.Subsets II (Ref.L491,L78)