Description

46. Permutations

Given a collection of distinct integers, return all possible permutations.

```
Example:
Input: [1,2,3]
Output:
[
  [1,2,3],
  [1,3,2],
  [2,1,3],
  [2,3,1],
  [3,1,2],
  [3,2,1]
]
```

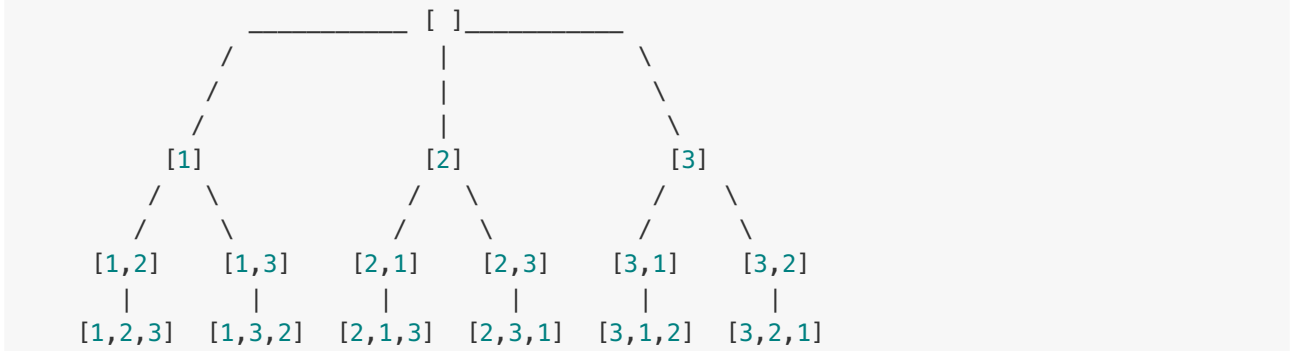
47. Permutations II

Given a collection of numbers that might contain duplicates, return all possible unique permutations.

```
Example:
Input: [1,1,2]
Output:
[
  [1,1,2],
  [1,2,1],
  [2,1,1]
]
```

Idea

For a input [1,2,3], we can draw the solution space tree:



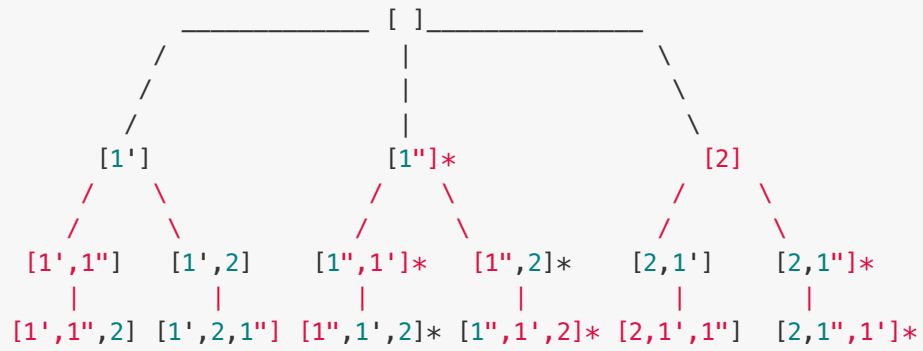
So this becomes a tree traversal problem. For each level, we need to add one element that has not already added, so we can have a hash or visited array to store which indices have already been added. We can use DFS (preorder) to find out all the leave nodes and add them in the result. We can have a List permu to store the numbers added which going from top to bottom, when we reaches leave node, the permu has all the elements from input array, so we can add it into the result. The time complexity is n factorial O(n!), and extra space is O(n) for permu to store the path, and recursion stack is also O(n) deep.

```
class Solution {
    public List<List<Integer>> permute(int[] nums) {
        List<List<Integer>> res = new ArrayList<>();
        dfsHelper(nums, res, new ArrayList<>(), new int[nums.length]);
        return res;
    }

    private void dfsHelper(int[] nums, List<List<Integer>> res,
                           List<Integer> permu, int[] visited) {
        if (permu.size() == nums.length) {
            res.add(new ArrayList<>(permu));
            return;
        }

        for (int i = 0; i < nums.length; i++) {
            if (visited[i] == 0) {
                permu.add(nums[i]);
                visited[i] = 1;
                dfsHelper(nums, res, permu, visited);
                permu.remove(permu.size() - 1);
                visited[i] = 0;
            }
        }
    }
}
```

For the input with duplication, we need to find out how the duplication occurs. Take [1',1",2] for example, so [1',1",2] and [1",1',2] is the same. We may want to sort first so that it's easier to find out the same numbers (1' and 1"). We can again draw the solution space tree. We can see child [1'] and [1"] and exactly the same, so for any number, if it is equal to the number before itself, the that number is not in the permu List, we know that's duplicated and skip them.




```java
class Solution {
    public List<List<Integer>> permuteUnique(int[] nums) {
        Arrays.sort(nums);
        List<List<Integer>> res = new ArrayList<>();
        dfsHelper(nums, res, new ArrayList<>(), new int[nums.length]);
        return res;
    }

    private void dfsHelper(int[] nums, List<List<Integer>> res,
                           List<Integer> permu, int[] visited) {
        if (permu.size() == nums.length) {
            res.add(new ArrayList<>(permu));
            return;
        }

        for (int i = 0; i < nums.length; i++) {
            if (i > 0 && nums[i] == nums[i - 1] && visited[i - 1] == 0) {
                continue;
            }
            if (visited[i] == 0) {
                permu.add(nums[i]);
                visited[i] = 1;
                dfsHelper(nums, res, permu, visited);
                permu.remove(permu.size() - 1);
                visited[i] = 0;
            }
        }
    }
}
```

Summary

- DFS.
- Backtrack.
- Deep copy.

C++ implementation

46. Permutations

```
class Solution {
public:
    vector<vector<int>> permute(vector<int>& nums) {
        vector<vector<int>> res;
        vector<int> permu;
        vector<int> visited(nums.size(), 0);
        dfsHelper(nums, res, permu, visited);
        return res;
    }

private:
    void dfsHelper(vector<int>& nums, vector<vector<int>>& res,
                   vector<int>& permu, vector<int>& visited) {
        if (permu.size() == nums.size()) {
            res.push_back(permu);
            return;
        }

        for (int i = 0; i < nums.size(); i++) {
            if (visited[i] == 0) {
                permu.push_back(nums[i]);
                visited[i] = 1;
                dfsHelper(nums, res, permu, visited);
                permu.pop_back();
                visited[i] = 0;
            }
        }
    }
};
```

47. Permutations II

```cpp
class Solution {
public:
    vector<vector<int>> permuteUnique(vector<int>& nums) {
        sort(nums.begin(), nums.end());
        vector<vector<int>> res;
        vector<int> permu;
        vector<int> visited(nums.size(), 0);
        dfsHelper(nums, res, permu, visited);
        return res;
    }

private:
    void dfsHelper(vector<int>& nums, vector<vector<int>>& res,
                   vector<int>& permu, vector<int>& visited) {
        if (permu.size() == nums.size()) {
            res.push_back(permu);
            return;
        }

        for (int i = 0; i < nums.size(); i++) {
            if (i > 0 && nums[i] == nums[i - 1] && visited[i - 1] == 0) {
                continue;
            }
            if (visited[i] == 0) {
                permu.push_back(nums[i]);
                visited[i] = 1;
                dfsHelper(nums, res, permu, visited);
                permu.pop_back();
                visited[i] = 0;
            }
        }
    }
};
```