Subset – II | Print all the Unique Subsets

Problem Statement: Given an array of integers that **may contain duplicates** the task is to return all possible subsets. Return only **unique subsets** and they can be in any order.
Examples:

Example 1:

Input: array[] = [1,2,2]

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

Explanation: We can have subsets ranging from length 0 to 3. which are listed above. Also the subset [1,2] appears twice but is printed only once as we require only unique subsets.

Input: array[] = [1]

Output: [[ ], [1]]

Explanation: Only two unique subsets are available

Solution

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

Solution 1: Brute Force
Approach:
At every index, we make a decision whether to pick or not pick the element at that index. This will help us in generating all possible combinations but does not take care of the duplicates. Hence we will use a set to store all the combinations that will discard the duplicates.
Code:

- C++ Code
- Java Code
- Python Code

```cpp
#include <bits/stdc++.h>

using namespace std;
void printAns(vector < vector < int >> & ans) {
  cout << "The unique subsets are " << endl;
  cout << "[ ";
  for (int i = 0; i < ans.size(); i++) {
    cout << "[ ";
    for (int j = 0; j < ans[i].size(); j++)
      cout << ans[i][j] << " ";
    cout << "]";
  }
  cout << " ]";
}
class Solution {
  public:
    void fun(vector < int > & nums, int index, vector < int > ds, set < vector < int >> & res) {
      if (index == nums.size()) {
        sort(ds.begin(), ds.end());
        res.insert(ds);
        return;
      }
      ds.push_back(nums[index]);
      fun(nums, index + 1, ds, res);
      ds.pop_back();
      fun(nums, index + 1, ds, res);
    }
  vector < vector < int >> subsetsWithDup(vector < int > & nums) {
    vector < vector < int >> ans;
    set < vector < int >> res;
    vector < int > ds;
    fun(nums, 0, ds, res);
    for (auto it = res.begin(); it != res.end(); it++) {
      ans.push_back( * it);
```

```
        }
        return ans;
    }
};
int main() {
    Solution obj;
    vector < int > nums = {1, 2, 2};
    vector < vector < int >> ans = obj.subsetsWithDup(nums);
    printAns(ans);
    return 0;
}
```

Output:

The unique subsets are

[[ ][1][1 2][1 2 2][2][2 2]]

Time Complexity: O(2^n *(k log (x))). 2^n for generating every subset and **k* log(x)** to insert every combination of average length k in a set of size x. After this, we have to convert the set of combinations back into a list of list /vector of vectors which takes more time.

Space Complexity: O(2^n * k) to store every subset of average length k. Since we are initially using a set to store the answer another **O(2^n *k)** is also used.

Solution 2: Optimal

Approach:

In the previous method, we were taking extra time to store the unique combination with the help of a set. To make the solution efficient we will have to decide on a method that will consider only the unique combinations without the help of additional data structure.

Lets understand with an example where arr = [1,2,2].

Initially start with an empty data structure. In the first recursion, call make a subset of size one, in the next recursion call a subset of size 2, and so on. But first, in order to make a subset of size one what options do we have?

We can pick up elements from either the first index or the second index or the third index. However, if we have already picked up two from the second index, picking up two from the third index will make another duplicate subset of size one. Since we are trying to avoid duplicate subsets we can avoid picking up from the third index. This should give us an intuition that whenever there are duplicate elements in the array we pick up only the first occurrence.

The next recursion calls will continue from the point the previous one ended.

Let's summarize:-

- Sort the input array.Make a recursive function that takes the input array ,the current subset,the current index and a list of list/ vector of vectors to contain the answer.
- Try to make a subset of size n during the nth recursion call and consider elements from every index while generating the combinations. Only pick up elements that are appearing for the first time during a recursion call to avoid duplicates.
- Once an element is picked up, move to the next index.The recursion will terminate when the end of array is reached.While returning backtrack by removing the last element that was inserted.

Code:

- C++ Code
- Java Code
- Python Code

```
#include <bits/stdc++.h>

using namespace std;
void printAns(vector < vector < int >> & ans) {
    cout<<"The unique subsets are "<<endl;
    cout << "[ ";
    for (int i = 0; i < ans.size(); i++) {
        cout << "[ ";
        for (int j = 0; j < ans[i].size(); j++)
            cout << ans[i][j] << " ";
        cout << "]";
    }
    cout << " ]";
}
class Solution {
    private:
        void findSubsets(int ind, vector < int > & nums, vector < int > & ds, vector < vector < int >> & ans) {
```

```
            ans.push_back(ds);
            for (int i = ind; i < nums.size(); i++) {
                if (i != ind && nums[i] == nums[i - 1]) continue;
                ds.push_back(nums[i]);
                findSubsets(i + 1, nums, ds, ans);
                ds.pop_back();
            }
        }
    public:
        vector < vector < int >> subsetsWithDup(vector < int > & nums) {
            vector < vector < int >> ans;
            vector < int > ds;
            sort(nums.begin(), nums.end());
            findSubsets(0, nums, ds, ans);
            return ans;
        }
};
int main() {
    Solution obj;
    vector < int > nums = {1,2,2 };
    vector < vector < int >> ans = obj.subsetsWithDup(nums);
    printAns(ans);
    return 0;
}
```

Output:

The unique subsets are

[[ ][1][1 2][1 2 2][2][2 2]]

Time Complexity: O(2^n) for generating every subset and **O(k)** to insert every subset in another data structure if the average length of every subset is **k.** Overall **O(k * 2^n).**

Space Complexity: O(2^n * k) to store every subset of average length k. Auxiliary space is **O(n)** if n is the depth of the recursion tree.