Subsets II

Given a collection of integers that might contain duplicates, ***nums***, return all possible subsets.

Note:

- Elements in a subset must be in non-descending order.
- The solution set must not contain duplicate subsets.

For example,
If ***nums*** = `[1,2,2]`, a solution is:

```
[
  [2],
  [1],
  [1,2,2],
  [2,2],
  [1,2],
  []
]
```

Solution 1

To solve this problem, it is helpful to first think how many subsets are there. If there is no duplicate element, the answer is simply 2^n, where n is the number of elements. This is because you have two choices for each element, either putting it into the subset or not. So all subsets for this no-duplicate set can be easily constructed: num of subset

- (1 to 2^0) empty set is the first subset
- (2^0+1 to 2^1) add the first element into subset from (1)
- (2^1+1 to 2^2) add the second element into subset (1 to 2^1)
- (2^2+1 to 2^3) add the third element into subset (1 to 2^2)
- ....
- (2^(n-1)+1 to 2^n) add the nth element into subset(1 to 2^(n-1))

Then how many subsets are there if there are duplicate elements? We can treat duplicate element as a spacial element. For example, if we have duplicate elements (5, 5), instead of treating them as two elements that are duplicate, we can treat it as one special element 5, but this element has more than two choices: you can either NOT put it into the subset, or put ONE 5 into the subset, or put TWO 5s into the subset. Therefore, we are given an array (a1, a2, a3, ..., an) with each of them appearing (k1, k2, k3, ..., kn) times, the number of subset is (k1+1)*(k2+1)*...(kn+1). We can easily see how to write down all the subsets similar to the approach above.

```
    class Solution {
public:
    vector<vector<int> > subsetsWithDup(vector<int> &S) {
        vector<vector<int> > totalset = {{}};
        sort(S.begin(),S.end());
        for(int i=0; i<S.size();){
            int count = 0; // num of elements are the same
            while(count + i<S.size() && S[count+i]==S[i])  count++;
            int previousN = totalset.size();
            for(int k=0; k<previousN; k++){
                vector<int> instance = totalset[k];
                for(int j=0; j<count; j++){
                    instance.push_back(S[i]);
                    totalset.push_back(instance);
                }
            }
            i += count;
        }
        return totalset;
        }
};
```

written by mathsam original link here

Solution 2

If we want to insert an element which is a dup, we can only insert it after the newly inserted elements from last step.

```
vector<vector<int> > subsetsWithDup(vector<int> &S) {
    sort(S.begin(), S.end());
    vector<vector<int>> ret = {{}};
    int size = 0, startIndex = 0;
    for (int i = 0; i < S.size(); i++) {
        startIndex = i >= 1 && S[i] == S[i - 1] ? size : 0;
        size = ret.size();
        for (int j = startIndex; j < size; j++) {
            vector<int> temp = ret[j];
            temp.push_back(S[i]);
            ret.push_back(temp);
        }
    }
    return ret;
}
```

written by yuruofeifei original link here

Solution 3

The characteristics of C++ reference is an outstanding tool for backtracking algorithm!

let us use [1,2,3,4] as an example to explain my solution:

```
subsets([1,2,3,4]) = []
                     // push(1)
                     [1, subsets([2,3,4])] // if push N times in subsets([2,3,4])
, the pop times is also N, so vec is also [1] after backtrack.
                     // pop(), push(2)
                     [2, subsets([3,4])]
                     // pop(), push(3)
                     [3, subsets([4])]
                     // pop(), push(4)
                     [4, subsets([])]
                     // pop()
```

Accepted 10ms c++ solution use backtracking for Subsets

```
class Solution {
public:
    std::vector<std::vector<int> > subsets(std::vector<int> &nums) {
        std::sort(nums.begin(), nums.end());
        std::vector<std::vector<int> > res;
        std::vector<int> vec;
        subsets(res, nums, vec, 0);
        return res;
    }
private:
    void subsets(std::vector<std::vector<int> > &res, std::vector<int> &nums, std
::vector<int> &vec, int begin) {
        res.push_back(vec);
        for (int i = begin; i != nums.size(); ++i) {
            vec.push_back(nums[i]);
            subsets(res, nums, vec, i + 1);
            vec.pop_back();
        }
    }
};
```

Accepted 10ms c++ solution use backtracking for Subsets II

```
class Solution {
public:
    std::vector<std::vector<int> > subsetsWithDup(std::vector<int> &nums) {
        std::sort(nums.begin(), nums.end());
        std::vector<std::vector<int> > res;
        std::vector<int> vec;
        subsetsWithDup(res, nums, vec, 0);
        return res;
    }
private:
    void subsetsWithDup(std::vector<std::vector<int> > &res, std::vector<int> &nu
ms, std::vector<int> &vec, int begin) {
        res.push_back(vec);
        for (int i = begin; i != nums.size(); ++i)
            if (i == begin || nums[i] != nums[i - 1]) {
                vec.push_back(nums[i]);
                subsetsWithDup(res, nums, vec, i + 1);
                vec.pop_back();
            }
    }
};
```

written by prime_tang original link here