Permutations II

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

For example,
`[1,1,2]` have the following unique permutations:
`[1,1,2]`, `[1,2,1]`, and `[2,1,1]`.

Solution 1

```
class Solution {
public:
    void recursion(vector<int> num, int i, int j, vector<vector<int> > &res) {
        if (i == j-1) {
            res.push_back(num);
            return;
        }
        for (int k = i; k < j; k++) {
            if (i != k && num[i] == num[k]) continue;
            swap(num[i], num[k]);
            recursion(num, i+1, j, res);
        }
    }
    vector<vector<int> > permuteUnique(vector<int> &num) {
        sort(num.begin(), num.end());
        vector<vector<int> >res;
        recursion(num, 0, num.size(), res);
        return res;
    }
};
```

written by guoang original link here

Solution 2

```
class Solution {
public:
    vector<vector<int> > permuteUnique(vector<int> &S) {
        // res.clear();
        sort(S.begin(), S.end());
        res.push_back(S);
        int j;
        int i = S.size()-1;
        while (1){
            for (i=S.size()-1; i>0; i--){
                if (S[i-1]< S[i]){
                    break;
                }
            }
            if(i == 0){
                break;
            }

            for (j=S.size()-1; j>i-1; j--){
                if (S[j]>S[i-1]){
                    break;
                }
            }
            swap(S[i-1], S[j]);
            reverse(S, i, S.size()-1);
            res.push_back(S);
        }
        return res;
    }
    void reverse(vector<int> &S, int s, int e){
        while (s<e){
            swap(S[s++], S[e--]);
        }
    }

    vector<vector<int> > res;
};
```

Basically, assume we have "1234", the idea is to increase the number in ascending order, so next is "1243", next is "1324", and so on.

written by TransMatrix original link here

Solution 3

I see most solutions are using next permutation. That's great and only uses O(1) space.

Anyway I am sharing backtracking solution which uses O(n) space. This is actually a typical backtracking problem. We can use hash map to check whether the element was already taken. However, we could get TLE if we check vector num every time. So we iterate the hash map instead.

```
class Solution {
public:
vector<vector<int> > permuteUnique(vector<int> &num) {
    vector<vector<int>> v;
    vector<int> r;
    map<int, int> map;
    for (int i : num)
    {
        if (map.find(i) == map.end()) map[i] = 0;
        map[i]++;
    }
    permuteUnique(v, r, map, num.size());
    return v;
}

void permuteUnique(vector<vector<int>> &v, vector<int> &r, map<int, int> &map, in
t n)
{
    if (n <= 0)
    {
        v.push_back(r);
        return;
    }
    for (auto &p : map)
    {
        if (p.second <= 0) continue;
        p.second--;
        r.push_back(p.first);
        permuteUnique(v, r, map, n - 1);
        r.pop_back();
        p.second++;
    }
}
};
```

written by oujiafan original link here