Combination Sum II

Given a collection of candidate numbers (***C***) and a target number (***T***), find all unique combinations in ***C*** where the candidate numbers sums to ***T***.

Each number in ***C*** may only be used **once** in the combination.

Note:

- All numbers (including target) will be positive integers.
- Elements in a combination (a_1, a_2, … , a_k) must be in non-descending order. (ie, $a_1 \le a_2 \le \ldots \le a_k$).
- The solution set must not contain duplicate combinations.

For example, given candidate set `10,1,2,7,6,1,5` and target `8`,
A solution set is:
`[1, 7]`
`[1, 2, 5]`
`[2, 6]`
`[1, 1, 6]`

Solution 1

```
 public List<List<Integer>> combinationSum2(int[] cand, int target) {
    Arrays.sort(cand);
    List<List<Integer>> res = new ArrayList<List<Integer>>();
    List<Integer> path = new ArrayList<Integer>();
    dfs_com(cand, 0, target, path, res);
    return res;
}
void dfs_com(int[] cand, int cur, int target, List<Integer> path, List<List<Integer>> res) {
    if (target == 0) {
        res.add(new ArrayList(path));
        return ;
    }
    if (target < 0) return;
    for (int i = cur; i < cand.length; i++){
        if (i > cur && cand[i] == cand[i-1]) continue;
        path.add(path.size(), cand[i]);
        dfs_com(cand, i+1, target - cand[i], path, res);
        path.remove(path.size()-1);
    }
}
```

written by lchen77 original link here

Solution 2

At the beginning, I stuck on this problem. After careful thought, I think this kind of backtracking contains a iterative component and a resursive component so I'd like to give more details to help beginners save time. The revursive component tries the elements after the current one and also tries duplicate elements. So we can get correct answer for cases like [1 1] 2. The iterative component checks duplicate combinations and skip it if it is. So we can get correct answer for cases like [1 1 1] 2.

```
class Solution {
public:
    vector<vector<int> > combinationSum2(vector<int> &num, int target)
    {
        vector<vector<int>> res;
        sort(num.begin(),num.end());
        vector<int> local;
        findCombination(res, 0, target, local, num);
        return res;
    }
    void findCombination(vector<vector<int>>& res, const int order, const int target, vector<int>& local, const vector<int>& num)
    {
        if(target==0)
        {
            res.push_back(local);
            return;
        }
        else
        {
            for(int i = order;i<num.size();i++) // iterative component
            {
                if(num[i]>target) return;
                if(i&&num[i]==num[i-1]&&i>order) continue; // check duplicate combination
                local.push_back(num[i]),
                findCombination(res,i+1,target-num[i],local,num); // recursive componenet
                local.pop_back();
            }
        }
    }
};
```

written by luming.zhang.75 original link here

Solution 3

I also did it with recursion, turns out the DP solution is 3~4 times faster.

```
def combinationSum2(self, candidates, target):
    candidates.sort()
    table = [None] + [set() for i in range(target)]
    for i in candidates:
        if i > target:
            break
        for j in range(target - i, 0, -1):
            table[i + j] |= {elt + (i,) for elt in table[j]}
        table[i].add((i,))
    return map(list, table[target])
```

written by ChuntaoLu original link here

From Leetcoder.