Combination Sum

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

The **same** repeated number may be chosen from ***C*** unlimited number of times.

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 `2,3,6,7` and target `7`,
A solution set is:

```
[7]
[2, 2, 3]
```

Solution 1

Accepted 16ms c++ solution use backtracking for Combination Sum:

```
class Solution {
public:
    std::vector<std::vector<int> > combinationSum(std::vector<int> &candidates, i
nt target) {
        std::sort(candidates.begin(), candidates.end());
        std::vector<std::vector<int> > res;
        std::vector<int> combination;
        combinationSum(candidates, target, res, combination, 0);
        return res;
    }
private:
    void combinationSum(std::vector<int> &candidates, int target, std::vector<std
::vector<int> > &res, std::vector<int> &combination, int begin) {
        if  (!target) {
            res.push_back(combination);
            return;
        }
        for (int i = begin; i != candidates.size() && target >= candidates[i]; ++
i) {
            combination.push_back(candidates[i]);
            combinationSum(candidates, target - candidates[i], res, combination,
i);
            combination.pop_back();
        }
    }
};
```

Accepted 12ms c++ solution use backtracking for Combination Sum II:

```
class Solution {
public:
    std::vector<std::vector<int> > combinationSum2(std::vector<int> &candidates,
int target) {
        std::sort(candidates.begin(), candidates.end());
        std::vector<std::vector<int> > res;
        std::vector<int> combination;
        combinationSum2(candidates, target, res, combination, 0);
        return res;
    }
private:
    void combinationSum2(std::vector<int> &candidates, int target, std::vector<st
d::vector<int> > &res, std::vector<int> &combination, int begin) {
        if  (!target) {
            res.push_back(combination);
            return;
        }
        for (int i = begin; i != candidates.size() && target >= candidates[i]; ++
i)
            if (i == begin || candidates[i] != candidates[i - 1]) {
                combination.push_back(candidates[i]);
                combinationSum2(candidates, target - candidates[i], res, combinat
ion, i + 1);
                combination.pop_back();
            }
    }
};
```

Accepted oms c++ solution use backtracking for Combination Sum III:

```
class Solution {
public:
    std::vector<std::vector<int> > combinationSum3(int k, int n) {
        std::vector<std::vector<int> > res;
        std::vector<int> combination;
        combinationSum3(n, res, combination, 1, k);
        return res;
    }
private:
    void combinationSum3(int target, std::vector<std::vector<int> > &res, std::ve
ctor<int> &combination, int begin, int need) {
        if (!target) {
            res.push_back(combination);
            return;
        }
        else if (!need)
            return;
        for (int i = begin; i != 10 && target >= i * need + need * (need - 1) / 2
; ++i) {
            combination.push_back(i);
            combinationSum3(target - i, res, combination, i + 1, need - 1);
            combination.pop_back();
        }
    }
};
```

written by prime_tang original link here

Solution 2

Sort the candidates and we choose from small to large recursively, every time we add a candidate to our possible sub result, we subtract the target to a new smaller one.

```
public List<List<Integer>> combinationSum(int[] candidates, int target) {
    List<List<Integer>> ret = new LinkedList<List<Integer>>();
    Arrays.sort(candidates); // sort the candidates
    // collect possible candidates from small to large to eliminate duplicates,
    recurse(new ArrayList<Integer>(), target, candidates, 0, ret);
    return ret;
}

// the index here means we are allowed to choose candidates from that index
private void recurse(List<Integer> list, int target, int[] candidates, int index,
List<List<Integer>> ret) {
    if (target == 0) {
        ret.add(list);
        return;
    }
    for (int i = index; i < candidates.length; i++) {
        int newTarget = target - candidates[i];
        if (newTarget >= 0) {
            List<Integer> copy = new ArrayList<Integer>(list);
            copy.add(candidates[i]);
            recurse(copy, newTarget, candidates, i, ret);
        } else {
            break;
        }
    }
}
```

written by dylan_yu original link here

Solution 3

```
public class Solution {
    public List<List<Integer>> combinationSum(int[] candidates, int target) {
        Arrays.sort(candidates);
        List<List<Integer>> result = new ArrayList<List<Integer>>();
        getResult(result, new ArrayList<Integer>(), candidates, target, 0);

        return result;
    }

    private void getResult(List<List<Integer>> result, List<Integer> cur, int can
didates[], int target, int start){
        if(target > 0){
            for(int i = start; i < candidates.length && target >= candidates[i];
i++){
                cur.add(candidates[i]);
                getResult(result, cur, candidates, target - candidates[i], i);
                cur.remove(cur.size() - 1);
            }//for
        }//if
        else if(target == 0 ){
            result.add(new ArrayList<Integer>(cur));
        }//else if
    }
}
```

written by luckygxf original link here