入选理由

1.上一题的进阶版，如果会了上一题，稍微拓展一下你会么?

标签

- 剪枝
- 回溯

难度

- 中等

题目地址（40 组合总数 II）

https://leetcode-cn.com/problems/combination-sum-ii/

题目描述

```
给定一个数组 candidates 和一个目标数 target ，找出 candidates 中所有可以使数字和为 target 的组合。

candidates 中的每个数字在每个组合中只能使用一次。

说明：

所有数字（包括目标数）都是正整数。
解集不能包含重复的组合。
示例 1:

输入: candidates = [10,1,2,7,6,1,5], target = 8,
所求解集为:
[
[1, 7],
[1, 2, 5],
[2, 6],
[1, 1, 6]
]
示例 2:

输入: candidates = [2,5,2,1,2], target = 5,
所求解集为:
[
  [1,2,2],
```

```
  [5]
]
```

前置知识

- 剪枝
- 数组
- 回溯

思路

套娃题，既然大家都做过了 39，这个题也不难理解，肯定是要用搜索了，那么看一下区别吧：

- 39 中数组无重复元素，40 数组中可能有重复元素。
- 39 一个元素可以用无数次，40 一个元素只能用一次

首先我们大致的搜索过程其实和上一个题没有啥太大差距，把上一个题基础上加个限制，就是每次搜索指针后移一位，这样保证一个元素只用了一次，看代码

```
public List<List<Integer>> combinationSum(int[] candidates, int target) {

    List<List<Integer>> res = new ArrayList<>();
    List<Integer> list = new LinkedList<>();
    helper(res, list, candidates, target, 0);
    return res;
}

public void helper(List<List<Integer>> res, List<Integer> list, int[] candidates, int cur, int pos) {

    if (cur < 0)
        return;

    if (cur == 0) {

        res.add(new LinkedList<>(list));
        return;
    }

    for (int i = pos; i < candidates.length; i++) {

        list.add(candidates[i]);
    // 变化在下面这行呢
        helper(res, list, candidates, cur - candidates[i], i + 1);
        list.remove(list.size() - 1);
```

```
    }
}
```

没问题，提交，发现又错了。。。。。。结果一看，怎么还有重复的，我不是都限制 pos 了么：

- 我们限制的 pos 只是限制了元素出现的先后顺序，由于 39 无重复元素，因此可行。

- 在看 40，如果有重复元素，那限制元素出现顺序就不能将重复解剪干净。

- 下面所说的方法是搜索中常用的去重策略：
  - 先将整个数组排好序
  - 在搜索（dfs）过程中，若该元素和前一个元素相等，那么因为前一个元素打头的解都已经搜所完毕了，因此没必要在搜这个元素了，故 pass

```
if (i > start && candidates[i] == candidates[i - 1])
    continue;
```

这样我们就把重复的解给剪干净了。

代码

代码支持：Java，Python，JS

Java Code:

```
public List<List<Integer>> combinationSum2(int[] candidates, int target) {

    Arrays.sort(candidates);
    List<List<Integer>> res = new ArrayList<>();
    List<Integer> list = new LinkedList<>();
    helper(res, list, target, candidates, 0);
    return res;
}

public void helper(List<List<Integer>> res, List<Integer> list, int target, int[] candidates, int start) {

    if (target == 0) {
        res.add(new LinkedList<>(list));
        return;
```

```
    }

    for (int i = start; i < candidates.length; i++) {

        if (target - candidates[i] >= 0) {

            if (i > start && candidates[i] == candidates[i - 1])
                continue;

            list.add(candidates[i]);
            helper(res, list, target - candidates[i], candidates, i + 1);
            list.remove(list.size() - 1);
        }
    }
}
```

Python Code：

```
class Solution:
    def combinationSum2(self, candidates, target):
        lenCan = len(candidates)
        if lenCan == 0:
            return []
        candidates.sort()
        path = []
        res = []
        self.backtrack(candidates, target, lenCan, 0, 0, path, res)
        return res

    def backtrack(self, curCandidates, target, lenCan, curSum, indBegin, path, res):
            # 终止条件
            if curSum == target:
                res.append(path.copy())
            for index in range(indBegin, lenCan):
                nextSum = curSum + curCandidates[index]
                # 减枝操作
                if nextSum > target:
                    break
                # 通过减枝避免重复解的出现
                if index > indBegin and curCandidates[index-1] == curCandidates[index]:
                    continue
                path.append(curCandidates[index])
                # 由于元素只能用一次，所以indBegin = index+1
                self.backtrack(curCandidates, target, lenCan, nextSum, index+1, path, res)
                path.pop()
```

JS Code:

```
function backtrack(list, tempList, nums, remain, start) {
  if (remain < 0) return;
  else if (remain === 0) return list.push([...tempList]);
  for (let i = start; i < nums.length; i++) {
    if (i > start && nums[i] == nums[i - 1]) continue; // skip duplicates
    tempList.push(nums[i]);
    backtrack(list, tempList, nums, remain - nums[i], i + 1); // i + 1代表不可以重复利用，  i 代表数字可以重复使用
    tempList.pop();
  }
}
/**
 * @param {number[]} candidates
 * @param {number} target
 * @return {number[][]}
 */
var combinationSum2 = function (candidates, target) {
  const list = [];
  backtrack(
    list,
    [],
    candidates.sort((a, b) => a - b),
    target,
    0
  );
  return list;
};
```

可能我的代码剪的并不是最优，大家可以自行按照思路修改。

复杂度分析

- 时间复杂度：在最坏的情况下，数组中的每个数都不相同，数组中所有数的和不超过 target，那么每个元素有选和不选两种可能，一共就有 2^n种选择，又因为我们每一个选择，**最多**需要 O(n) 的时间 push 到结果中。因此一个粗略的时间复杂度上界为 $O(N * 2^N)$，其中 N 是数组长度。更加严格的复杂度意义不大，不再分析。

- 空间复杂度：递归调用栈的长度不大于$target/min$，同时用于记录路径信息的 list 长度也不大于 $target/min$，因此空间复杂度为 $O(target/min)$

上一页 下一页