# 题目

给定一个候选人编号的集合 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 。

注意：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,

输出:

[

[1,1,6],

[1,2,5],

[1,7],

[2,6]

]

示例 2:

输入: candidates = [2,5,2,1,2], target = 5,

输出:

[

[1,2,2],

[5]

]

提示:

1 <= candidates.length <= 100

1 <= candidates[i] <= 50

1 <= target <= 30

# 分析

## 方法一：回溯法

思路：

这个问题可以使用回溯法来解决。

具体步骤如下：

1、对候选人编号集合进行排序，以便后续处理。

2、使用一个标记数组来记录每个数字是否被使用过。

3、定义一个回溯函数backtrack，用来生成组合。

4、在回溯函数中，遍历候选人编号集合中的每个数字，如果该数字已被使用过（或者该数字等于前一个数字且前一个数字未被使用过），则跳过；否则，将该数字添加到当前组合中，并标记为已使用，然后递归调用回溯函数；最后在递归结束后，将该数字重新标记为未使用，以便下一次回溯时使用。

代码：

class Solution {

public:

vector<vector<int>> combinationSum2(vector<int>& candidates, int target) {

vector<vector<int>> result;

vector<int> path;

vector<bool> used(candidates.size(), false);

sort(candidates.begin(), candidates.end());

backtrack(candidates, target, 0, result, path, used);

return result;

}

private:

void backtrack(vector<int>& candidates, int target, int start, vector<vector<int>>& result, vector<int>& path, vector<bool>& used) {

if (target == 0) {

result.push\_back(path);

return;

}

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

if (used[i] || (i > start && candidates[i] == candidates[i - 1] && !used[i - 1])) {

continue;

}

if (target - candidates[i] < 0) {

break;

}

path.push\_back(candidates[i]);

used[i] = true;

backtrack(candidates, target - candidates[i], i + 1, result, path, used);

path.pop\_back();

used[i] = false;

}

}

};

在这个实现中，我们首先对候选人编号集合进行排序，然后定义了一个回溯函数 backtrack，用来生成组合。在回溯函数中，我们遍历候选人编号集合中的每个数字，如果该数字已被使用过（或者该数字等于前一个数字且前一个数字未被使用过），则跳过；否则，将该数字添加到当前组合中，并标记为已使用，然后递归调用回溯函数；最后在递归结束后，将该数字重新标记为未使用，以便下一次回溯时使用。

或：

要解决这个问题，可以使用回溯算法。回溯算法通过递归和剪枝的方式，搜索所有可能的组合。对于本题，我们需要在搜索过程中确保每个数字只被使用一次，并且避免产生重复的组合。

首先，我们需要对候选数字进行排序，以便在回溯过程中进行剪枝操作。排序后，我们可以确保相同的数字是相邻的，从而避免产生重复的组合。

接下来，我们可以编写回溯函数。回溯函数需要接收当前的目标数、当前的组合列表以及当前候选数字的索引。在回溯函数中，我们首先需要检查当前的目标数是否为零，如果是，则将当前的组合列表添加到结果集中。

然后，我们遍历候选数字列表，从当前索引开始。对于每个数字，我们首先检查它是否小于等于当前的目标数，并且是否与上一个选择的数字相同且索引也相同（以避免重复使用相同的数字）。如果满足这些条件，我们将该数字添加到当前的组合列表中，并递归调用回溯函数，将目标数减去该数字，索引加一，继续搜索剩余的数字。

最后，在递归返回之前，我们需要将当前组合列表中的最后一个数字移除，以便进行下一轮的组合搜索。

代码：

class Solution {

public:

// 主函数，接收候选数字列表和目标数，返回所有组合

vector<vector<int>> combinationSum2(vector<int>& candidates, int target) {

vector<vector<int>> result; // 存储最终结果的集合

vector<int> currentCombo; // 存储当前正在构建的组合

// 对候选数字进行排序，以便剪枝和避免重复组合

sort(candidates.begin(), candidates.end());

// 调用回溯函数开始搜索

backtrack(candidates, target, currentCombo, 0, result);

// 返回最终结果

return result;

}

private:

// 回溯函数，递归地搜索所有可能的组合

void backtrack(vector<int>& candidates, int target, vector<int>& currentCombo, int start, vector<vector<int>>& result) {

// 如果目标数减为零，说明当前组合是一个解，将其添加到结果集中

if (target == 0) {

result.push\_back(currentCombo);

return;

}

// 遍历候选数字列表，从当前索引开始

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

// 剪枝操作：避免重复使用相同的数字

// 如果当前数字与前一个数字相同，并且索引也相同，则跳过

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

continue;

}

// 如果当前数字小于等于目标数，则考虑将其添加到组合中

if (candidates[i] <= target) {

// 将当前数字添加到组合中

currentCombo.push\_back(candidates[i]);

// 递归调用回溯函数，继续搜索剩余的数字

// 注意，索引从i+1开始，确保不会重复使用相同的数字

// 目标数减去当前数字，继续搜索剩余的目标数

backtrack(candidates, target - candidates[i], currentCombo, i + 1, result);

// 回溯，将当前数字从组合中移除，尝试其他可能性

currentCombo.pop\_back();

} else {

// 如果当前数字已经大于目标数，无需继续搜索，直接退出循环

break;

}

}

}

};