40-组合总和II

题述

40. 组合总和 II

难度 中等 952

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]
]
```

思路

本题和组合总和 I 的区别在于：

- 本题candidates 中的每个数字在每个组合中只能使用一次。
- 本题数组candidates的元素是有重复的

但是还不能有重复的组合

可能会想到，先求出所有的组合，再用set或者map去重，但是容易超时。

因此，要在搜索的过程中去掉重复组合。

所谓去重，就是使用过的元素不能重复选取。

组合问题可以抽象为树形结构，那么“使用过”在这个树形结构上是有两个维度的，一个维度是同一树枝上使用过，一个维度是同一树层上使用过。**没有理解这两个层面上的“使用过” 是造成大家没有彻底理解去重的根本原因。**

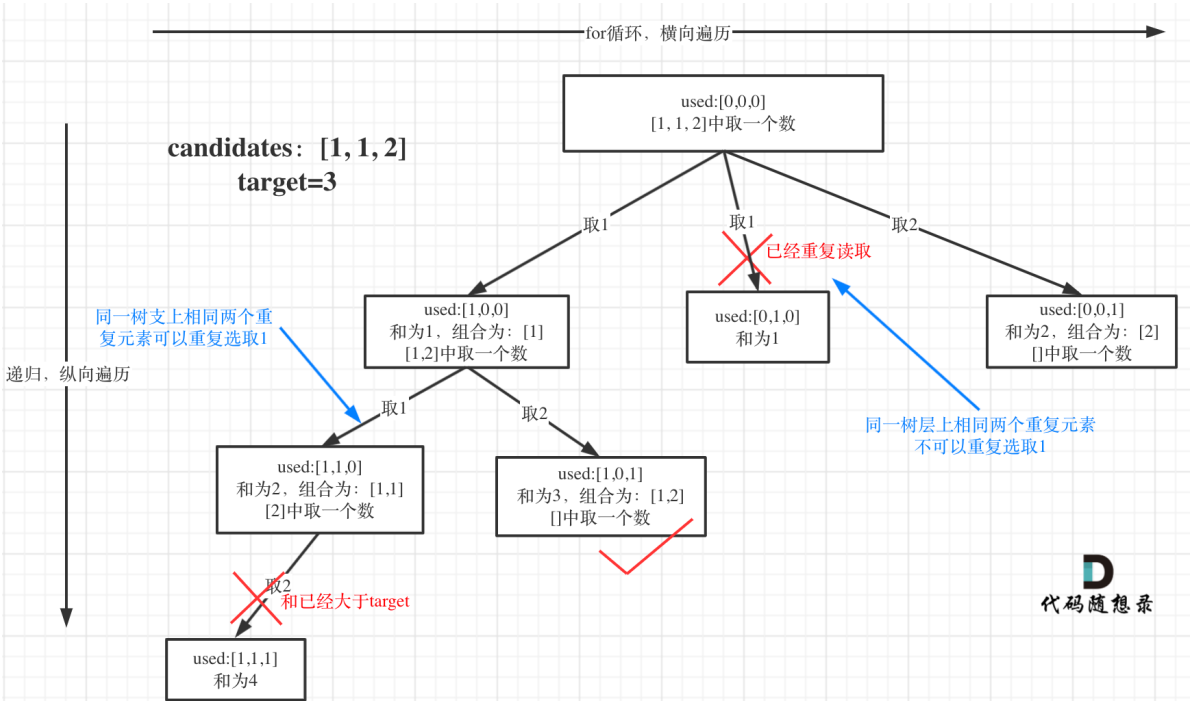
那么问题来了，我们是要同一树层上使用过，还是同一树枝上使用过呢？

回看一下题目，元素在同一个组合内是可以重复的，怎么重复都没事，但两个组合不能相同。

所以我们要去重的是同一树层上的“使用过”，同一树枝上的都是一个组合里的元素，不用去重。

为了理解去重我们来举一个例子，candidates = [1, 1, 2], target = 3，（方便起见candidates已经排序了）

强调一下，树层去重的话，需要对数组排序！



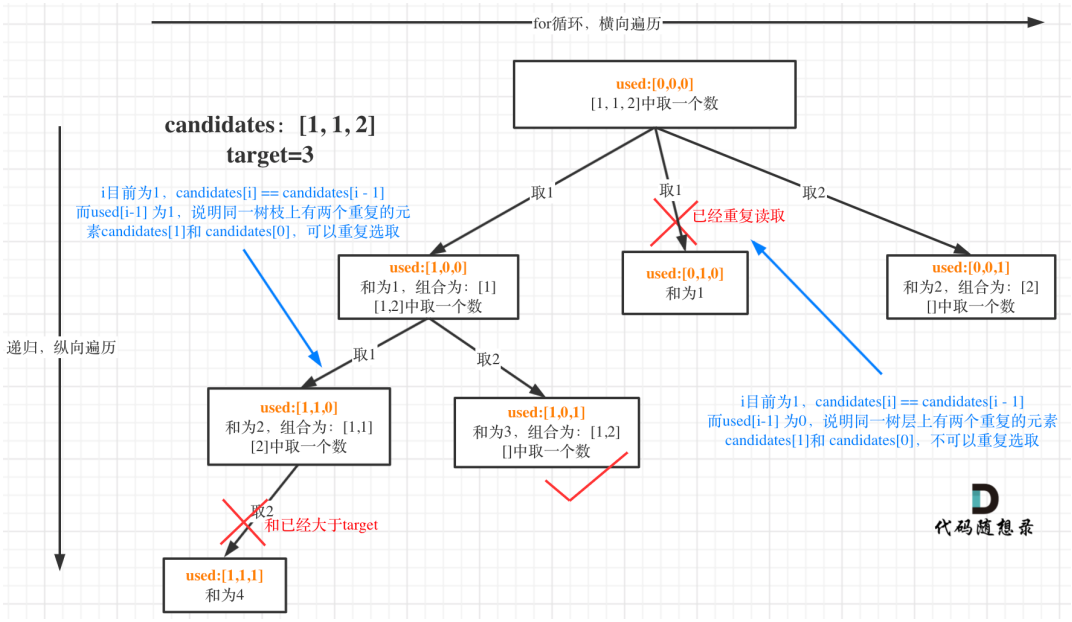
回溯三板斧

- 递归函数参数
  - 套路相同，此题还需要加一个bool型数组used，用来记录同一树枝上的元素是否使用过。

    这个集合去重的重任就是used来完成的。
  - ```
    vector<vector<int>> result; // 存放组合集合
    vector<int> path;           // 符合条件的组合
    void backtracking(vector<int>& candidates, int target, int sum, int
    startIndex, vector<bool>& used) {
    ```

- 递归终止条件
  - 终止条件为 `sum > target` 和 `sum == target`。
  - ```
    if (sum > target) { // 这个条件其实可以省略
        return;
    }
    if (sum == target) {
        result.push_back(path);
        return;
    }
    ```

- `sum > target` 这个条件其实可以省略，因为和在递归单层遍历的时候，会有剪枝的操作，下面会介绍到。

- **单层搜索的逻辑**
  - 最大的不同就是要去重了。
  - 前面我们提到：要去重的是“同一树层上的使用过”，如果判断同一树层上元素（相同的元素）是否使用过了呢。
  - **如果 `candidates[i] == candidates[i - 1]` 并且 `used[i - 1] == false`，就说明：前一个树枝，使用了candidates[i - 1]，也就是说同一树层使用过candidates[i - 1]。**
  - 此时for循环里就应该做continue的操作。
  - 

  - used[i - 1] == true，说明同一树枝candidates[i - 1]使用过
  - used[i - 1] == false，说明同一树层candidates[i - 1]使用过
  - ```
    for (int i = startIndex; i < candidates.size() && sum + candidates[i] <= target; i++) {
        // used[i - 1] == true，说明同一树枝candidates[i - 1]使用过
        // used[i - 1] == false，说明同一树层candidates[i - 1]使用过
        // 要对同一树层使用过的元素进行跳过
        if (i > 0 && candidates[i] == candidates[i - 1] && used[i - 1] == false) {
            continue;
        }
        sum += candidates[i];
        path.push_back(candidates[i]);
        used[i] = true;
        backtracking(candidates, target, sum, i + 1, used); // 和39.组合总和的区别1：这里是i+1，每个数字在每个组合中只能使用一次
        used[i] = false;
        sum -= candidates[i];
        path.pop_back();
    }
    ```
  - **注意sum + candidates[i] <= target为剪枝操作**

题解
    ```

C++回溯

```
class Solution {
private:
    vector<vector<int>> result;
    vector<int> path;
    void backtracking(vector<int>& candidates, int target, int sum, int
startIndex, vector<bool>& used) {
        if (sum == target) {
            result.push_back(path);
            return;
        }
        for (int i = startIndex; i < candidates.size() && sum + candidates[i] <=
target; i++) {
            // used[i - 1] == true，说明同一树枝candidates[i - 1]使用过
            // used[i - 1] == false，说明同一树层candidates[i - 1]使用过
            // 要对同一树层使用过的元素进行跳过
            if (i > 0 && candidates[i] == candidates[i - 1] && used[i - 1] ==
false) {
                continue;
            }
            sum += candidates[i];
            path.push_back(candidates[i]);
            used[i] = true;
            backtracking(candidates, target, sum, i + 1, used); // 和39.组合总和的
区别1，这里是i+1，每个数字在每个组合中只能使用一次
            used[i] = false;
            sum -= candidates[i];
            path.pop_back();
        }
    }

public:
    vector<vector<int>> combinationSum2(vector<int>& candidates, int target) {
        vector<bool> used(candidates.size(), false);
        path.clear();
        result.clear();
        // 首先把给candidates排序，让其相同的元素都挨在一起。
        sort(candidates.begin(), candidates.end());
        backtracking(candidates, target, 0, 0, used);
        return result;
    }
};
```

Python不用used数组

```
class Solution:
    def __init__(self):
        self.result = []
        self.path = []

    def backTracking(self, candidates: List[int], target: int, sum_: int,
start_index: int) -> None:
        if sum_ == target:
```

```
            self.result.append(self.path[:])
            return

        #单层递归
        for i in range(start_index,len(candidates)):
            #剪枝
            if sum_ + candidates[i] > target:
                return

            #跳过同一树层使用过的元素
            if i > start_index and candidates[i] == candidates[i-1]:
                continue

            sum_ += candidates[i]
            self.path.append(candidates[i])
            self.backTracking(candidates,target,sum_,i+1)
            self.path.pop()
            sum_ -= candidates[i]

    def combinationSum2(self, candidates: List[int], target: int) ->
List[List[int]]:
        candidates.sort()
        self.backTracking(candidates,target,0,0)
        return self.result
```

提交结果	执行用时	内存消耗	语言	提交时间	备
通过	44 ms	14.9 MB	Python3	2022/05/08 10:35	

思考