入选理由

1. 熟悉了剪枝的形象化，接下来开始真正的剪枝

2. 回溯的题目基本都需要剪枝，这是回溯的一个考点

标签

- 剪枝
- 回溯

难度

- 中等

题目地址（39 组合总和）

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

题目描述

```
给定一个无重复元素的数组 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 中的数字可以无限制重复被选取。

说明：

所有数字（包括 target）都是正整数。
解集不能包含重复的组合。
示例 1:

输入: candidates = [2,3,6,7], target = 7,
所求解集为:
[
[7],
[2,2,3]
]
示例 2:

输入: candidates = [2,3,5], target = 8,
所求解集为:
[
```

```
  [2,2,2,2],
  [2,3,3],
  [3,5]
]


提示：

1 <= candidates.length <= 30
1 <= candidates[i] <= 200
candidate 中的每个元素都是独一无二的。
1 <= target <= 500
```

前置知识

- 剪枝
- 回溯

思路

读完题，首先自然考虑最容易想到的解决方案，遍历数组！但是发现这同一个元素能用无限次，这可咋遍历。

没错，遇到 for 循环解决不了的，我们自然的就会想到搜索（回溯递归解决）方法。一个搜索策略+合适的剪枝可以大大提高算法效率哦。

相信回溯方法大家也都不陌生了，直接上个回溯代码：

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

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

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

    if (cur < 0)
        return;

    if (cur == 0) {

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

    for (int i = 0; i < candidates.length; i++) {

        list.add(candidates[i]);
```

```
        backtrack(res, list, candidates, cur - candidates[i]);
        list.remove(list.size() - 1);
    }
}
```

开开心心提交，结果发现没过去，尴尬，问题也很直观，就是我们没有去重，比如 2，2，3 和 2，3，2 这种都会存在于结果集中，那么怎么办呢？我们直接对结果集去重嘛？其实很直观发现，对结果集去重复杂度可不低啊，那么我们可不可以在搜索过程中就把重复的解剪掉呢？

当然可以

- 我们可以发现每次递归数组都是从头遍历的，并没有对顺序进行任何限制，那么我们不妨就限制一下顺序，比如 3，4 就只能是 3，4 不能是 4，3。
- 那我们递归的时候每次只能在当前的位置往后拿，不就避免了这种无序导致的重复情况了嘛。
- 我们在参数中再传入一个 pos，来记录当前位置。
- 注意：因为一个元素可以重复多次，因此我们 pos 没必要每次递归+1，只限制不取之前的元素就好。

代码

代码支持： Java，Python，JS

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

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

public void backtrack(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]);
        backtrack(res, list, candidates, cur - candidates[i], i);
        list.remove(list.size() - 1);
```

```
    }
}
```

Python Code：

```python
class Solution:
    def combinationSum(self, candidates: List[int], target: int) -> List[List[int]]:
        def backtrack(ans,tempList, candidates, remain, start):
            if remain < 0: return
            elif remain == 0: return ans.append(tempList.copy()) # 将部分解空间合并到总体的解空间
            # 枚举所有以 i 为开始的部分解空间
            for i in range(start, len(candidates)):
                tempList.append(candidates[i])
                backtrack(ans, tempList, candidates, remain - candidates[i], i)#  数字可以重复使用
                tempList.pop()

        ans = [];
        backtrack(ans, [], candidates, target, 0);
        return ans;
```

JS Code:

```javascript
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++) {
    tempList.push(nums[i]);
    backtrack(list, tempList, nums, remain - nums[i], i); // 数字可以重复使用
    tempList.pop();
  }
}
/**
 * @param {number[]} candidates
 * @param {number} target
 * @return {number[][]}
 */
var combinationSum = function (candidates, target) {
  const list = [];
  backtrack(
    list,
    [],
    candidates.sort((a, b) => a - b),
    target,
    0
  );
  return list;
};
```

我们仅仅额外利用了一个 pos 参数，就完美的剪掉了重复解。

当然，上述解决方案可能只是把重复的解剪掉了，是否还可以继续剪，比如提前终止搜索？留给大家思考啦。

复杂度分析

- 时间复杂度：该题不是很好分析，我个人分析是最坏情况，也就是没有任何剪枝时$O(N^{target/min})$,其中 N 时候选数组的长度，min 时数组元素最小值，target/min 也就是递归栈的最大深度。

- 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