排列、条件组合算法总结

排列、组合问题作为一种经常出现在各类算法考核中的编程题，其思路较为固定，但是变形比较多，结合一些自身学习、思考，现总结如下。

1. 排列算法总结

1.1 LeetCode 46. Permutations（排列）

Given a collection of **distinct** integers, return all possible permutations.

Example:

```
Input: [1,2,3]
Output:
[
  [1,2,3],
  [1,3,2],
  [2,1,3],
  [2,3,1],
  [3,1,2],
  [3,2,1]
]
```

对这种排列题，则无需再设定起始点 `start`，不然排列总数不够（因为加上起始点后，每次再从 `nums` 挑选数字到 `temp_res` 时，不能再回过去挑了，如上例，当 `start = 1` 时，只能从 `input` 的 `[2, 3]` 中挑选，不能再回过去选 1），但排列的情况则是输入中的任意数字均可作为开始起点，故而，每次，均是从 `start = 0` 开始遍历； 其次，因为这道题输入无重复数字，故而每次遍历时候只需要判定一次该数字是否已经遍历过，可直接使用 `if nums[i] not in temp_res:` 进行判定，无需再设置标记数组（用于处理输入含重复数字情况）便可得到结果。

> 有重复数字的输入情况可参考下倒例题方式

```
class Solution(object):
    def permute(self, nums):
        """
        :type nums: List[int]
        :rtype: List[List[int]]
        """


        if not nums:
            return []
        self.res = []
        temp_res = []
        self.helper(nums, [])
```

```
        # self.dfs(nums,0,[])
        return self.res
        '''
        res = []
        self.dfs(nums, res, [])
        return res
        '''

    def helper(self, nums, temp_res):

        if len(temp_res) == len(nums):
            self.res.append(temp_res[:])
            return
        if len(temp_res) > len(nums):
            return
        for i in range(len(nums)):
            if nums[i] not in temp_res:
                self.helper(nums, temp_res + [nums[i]])
```

对应 C++ 版本：

```
class Solution {
public:
    vector<vector<int>> permute(vector<int>& nums) {
        vector<int> temp_res;
        permuteCore(nums, {});
        return res;
    }
    void permuteCore(vector<int>& nums, vector<int> temp_res) {
        if (temp_res.size() == nums.size()) {
            res.push_back(temp_res);
            return;
        }
        if (temp_res.size() > nums.size()) {
            return;
        }
        for (int i = 0; i < nums.size(); ++i) {
            if (find(temp_res.begin(), temp_res.end(), nums[i]) ==
temp_res.end()) {
                temp_res.push_back(nums[i]);
                permuteCore(nums, temp_res);
                temp_res.pop_back();
            }
        }
    }
private:
vector<vector<int>> res;
};
```

注意：当传入的参数格式为 `vector<int>& temp_res` 时，必须在调用函数前定义 `vector<int> temp_res`，然后使用 `permuteCore(nums, temp_res);`

如果设定传入的参数格式为 `vector<int> temp_res` 时，则无需提点定义变量，直接传入 `permuteCore(nums, {});` 即可。

再提供一种递归法，比较容易理解与操作：

```
class Solution(object):
    def permute(self, nums):
        """
        :type nums: List[int]
        :rtype: List[List[int]]
        """
        if not nums:
            return []
        self.res = []
        temp_res = []

        res = []
        self.dfs(nums, res, [])
        return res


    def dfs(self, nums, res, path):
        if not nums:
            res.append(path)
        else:
            for i in range(len(nums)):
                self.dfs(nums[:i] + nums[i+1:], res, path + [nums[i]])
```

1.2 LeetCode 47. Permutations II（排列2）

Given a collection of numbers that might contain duplicates, return all possible unique permutations.

Example:

```
Input: [1,1,2]
Output:
[
  [1,1,2],
  [1,2,1],
  [2,1,1]
]
```

这道题与上道题最大区别点就在于输入含有重复数字，如果还按照上述题解法，则不可能得到结果（对应重复数字只能处理一次）；

- 但设想，是否直接去掉重复判定 `if nums[i] not in temp_res:` 就可以了，答案依旧是不行，去掉了重复判定，每次都从 0 开始重新遍历，会出现每个元素直接重复 `len(nums)` 结果，如 `[2,2,2], [1,1,1]` 之类；
- 再设想，是否可以通过添加 `for` 循环中遍历开始点 `start`，通过 `for i in range(start, nums)` 每次遍历过程 `start + 1` 来处理呢，结果依旧是不行，问题同上一个例题所述，这种情况无法出现输入后面元素作为起始位置的排列，如本题的 `[2,1,1]`；
- 因此，需要借助标记列表 `mark` 用于标记一个输入元素是否已经遍历

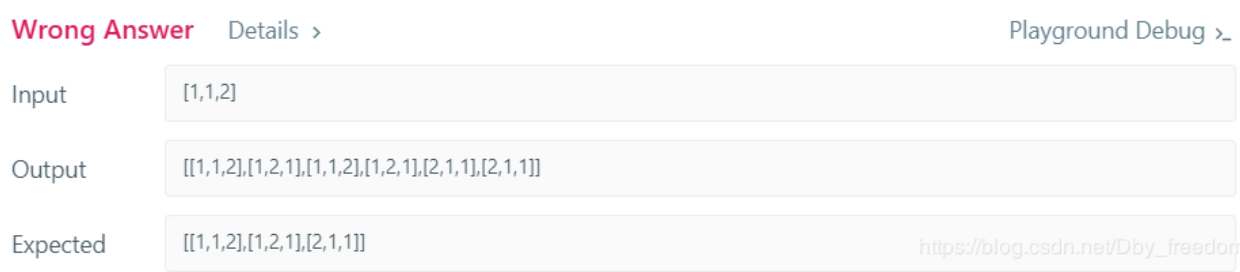
下面先据此思想，写出程序如下：

错误版本：

```
class Solution(object):
    def permuteUnique(self, nums):
        """
        :type nums: List[int]
        :rtype: List[List[int]]
        """
        if not nums: return []
        self.res = []
        # self.permute(nums, res, [])
        # self.helper(nums, [])
        mark = [False] * len(nums)
        self.dfs(nums, [], mark)
        return self.res

    def dfs(self, nums, temp_res, mark):
        if len(temp_res) > len(nums):
            return
        if len(temp_res) == len(nums):
            self.res.append(temp_res)
            return
        for i in range(len(nums)):
            if mark[i]: continue
            mark[i] = True
            self.dfs(nums, temp_res + [nums[i]], mark)
            mark[i] = False
```

结果如下：



通过检查结果不难得知，即便加了 `mark` 列表，只能保证元素不管是否重复，都会完成全排列（因此，这道题的结果完全可以对应上一道题的答案，因为上一道题输入不重复，全遍历就是正确结果）；

此时，需要在判定过程中排除出重复的 `temp_res`，排除方法有两种：

- 一种是在添加 `temp_res` 时，判定 `and temp_res not in self.res` 的被动后期结果过滤法：
- 一种是 `for` 训练中直接提前过滤，即先对数组进行排序，保证相等的数字放在一起，然后当我们遇到的不是第一个数字，并且现在的数字和前面的数字相等，同时前面的数字还没有访问过，我们是不能搜索的，需要直接返回。原因是，这种情况下，必须是由前面搜索到现在的这个位置，而不能是由现在的位置向前面搜索。

方式一：

```
class Solution(object):
    def permuteUnique(self, nums):
        """
        :type nums: List[int]
        :rtype: List[List[int]]
        """
        if not nums: return []
        self.res = []
        # self.permute(nums, res, [])
        # self.helper(nums, [])
        mark = [False] * len(nums)
        self.dfs(nums, [], mark)
        return self.res

    def dfs(self, nums, temp_res, mark):
        if len(temp_res) > len(nums):
            return
        if len(temp_res) == len(nums) and temp_res not in self.res:
            self.res.append(temp_res)
            return
        for i in range(len(nums)):
            if mark[i]: continue
            mark[i] = True
            self.dfs(nums, temp_res + [nums[i]], mark)
            mark[i] = False
```

方式二：

```
class Solution(object):
    def permuteUnique(self, nums):
        """
        :type nums: List[int]
        :rtype: List[List[int]]
        """
        if not nums: return []
        self.res = []
        # self.permute(nums, res, [])
```

```
        # self.helper(nums, [])
        mark = [False] * len(nums)
        nums.sort()
        self.dfs(nums, [], mark)
        return self.res


    def dfs(self, nums, temp_res, mark):
        if len(temp_res) > len(nums):
            return
        if len(temp_res) == len(nums):
            self.res.append(temp_res)
            return
        for i in range(len(nums)):
            # if mark[i]: continue
            if mark[i] or (i > 0 and nums[i] == nums[i-1] and mark[i-1]):
continue
            mark[i] = True
            self.dfs(nums, temp_res + [nums[i]], mark)
            mark[i] = False
```

对应 C++ 代码：

```
class Solution {
public:
    vector<vector<int>> permuteUnique(vector<int>& nums) {
        int length = nums.size();
        sort(nums.begin(), nums.end());
        vector<char> mark(length, false);
        permuteUniqueHelper(nums, {}, mark);
        return res;

    }
    void permuteUniqueHelper(vector<int>& nums, vector<int> temp_res, vector<char>
mark){
        if (temp_res.size() == nums.size()) {
            res.push_back(temp_res);
            return;
        }
        for (int i = 0; i < nums.size(); ++i) {
            if (mark[i] || ( i > 0 && nums[i] == nums[i-1] && mark[i-1])) {
                continue;
            }
            mark[i] = true;
            temp_res.push_back(nums[i]);
            permuteUniqueHelper(nums, temp_res, mark);
            temp_res.pop_back();
            mark[i] = false;
        }
```

```
    }
private:
vector<vector<int>> res;
};
```

这种主动过滤方式效率更高，运行时间快了十倍，但是代码相对麻烦；

最后，再提供一种简便思路，依旧采用后验式，代码如下：

```
class Solution(object):
    def permuteUnique(self, nums):
        """
        :type nums: List[int]
        :rtype: List[List[int]]
        """
        if not nums: return []
        self.res = []
        # self.permute(nums, res, [])
        # self.helper(nums, [])
        mark = [False] * len(nums)
        self.dfs(nums, [], mark)
        return self.res


    def permute(self, nums, res, temp_res):
        if not nums and temp_res not in res:
            res.append(temp_res[:])

        for i in range(len(nums)):
            self.permute(nums[:i] + nums[i+1:], res, temp_res + [nums[i]])
```

2. 组合

2.1 LeetCode 39. Combination Sum（组合总数）

Given a set of candidate numbers (`candidates`) (**without duplicates**) and a target number (`target`), find all unique combinations in candidates where the `candidate` numbers sums to `target`.

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

Note:

- All numbers (including target) will be positive integers.
- The solution set must not contain duplicate combinations.

题目： 给定一组候选数字 (候选) (不带重复项) 和目标数字 (目标), 可以在候选数字总和为目标的候选项中查找所有唯一的组合。

同样重复的数字可以从候选者无限次数中选择。

Example 1:

```
Input: candidates = [2,3,6,7], target = 7,
A solution set is:
[
  [7],
  [2,2,3]
]
```

Example 2:

```
Input: candidates = [2,3,5], target = 8,
A solution set is:
[
  [2,2,2,2],
  [2,3,3],
  [3,5]
]
```

按照前述的套路走一遍：

```
public class Solution {
   List<List<Integer>> result=new ArrayList<List<Integer>>();
   public List<List<Integer>> combinationSum(int[] candidates,int target) {
       Arrays.sort(candidates);
       List<Integer> list=new ArrayList<Integer>();
       return result;
    }
   public void backtracking(int[] candidates,int target,int start,){
    }
}
```

1. 全局List<List> result先定义
2. 回溯backtracking方法要定义，数组candidates 目标target 开头start 辅助链表List list都加上。
3. 分析算法：以[2,3,6,7] 每次尝试加入数组任何一个值，用循环来描述，表示依次选定一个值

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

           list.add(candidates[i]);

       }
```

接下来回溯方法再调用。比如第一次选了2，下次还能再选2是吧，所以每次start都可以从当前 i 开始（ps：如果不允许重复，从i+1开始）。第一次选择2，下一次要凑的数就不是7了，而是7-2，也就是5，一般化就是`remain = target - candidates[i]`，所以回溯方法为：

```
backtracking(candidates, target-candidates[i], i, list);
```

然后加上退回语句：`list.remove(list.size()-1);` 那么什么时候找到的解符合要求呢？自然是remain（注意区分初始的target）= 0 了，表示之前的组合恰好能凑出target。如果 `remain < 0` 表示凑的数太大了，组合不可行，要回退。当remain>0 说明凑的还不够，继续凑。 所以完整方法如下：

```
publicclass Solution {
    List<List<Integer>> result=newArrayList<List<Integer>>();
    public List<List<Integer>>combinationSum(int[] candidates, int target) {
        Arrays.sort(candidates);//所给数组可能无序，排序保证解按照非递减组合
        List<Integer> list=newArrayList<Integer>();
        backtracking(candidates,target,0,list);//给定target，start=0表示从数组第一个开
始
        return result;//返回解的组合链表
    }
    public void backtracking(int[]candidates,int target,int start,List<Integer>
list){

            if(target<0)    return;//凑过头了
            else if(target==0){

                result.add(newArrayList<>(list));//正好凑出答案，开心地加入解的链表

            }else{
                for(inti=start;i<candidates.length;i++){//循环试探每个数
                    list.add(candidates[i]);//尝试加入
          //下一次凑target-candidates[i]，允许重复，还是从i开始
                   backtracking(candidates,target-candidates[i],i,list);
          list.remove(list.size()-1);//回退
                }
            }

    }
}
```

其对应的python版本如下：

```
class Solution:
    def combinationSum(self, candidates, target):
        """
        :type candidates: List[int]
        :type target: int
        :rtype: List[List[int]]
        """
        candidates.sort()
        Solution.res = []
        self.DFS(candidates, target, 0, [])
```

```
        return Solution.res

    def DFS(self, candidates, target, start, temp_res):
        if target == 0:
            Solution.res.append(temp_res[:])
            return
        for i in range(start, len(candidates)):
            if candidates[i] > target:
                return
            self.DFS(candidates, target - candidates[i], i, temp_res +
[candidates[i]])
```

这里一定很迷惑，为什么转到了python版本之后就不用后退了呢？（对应`list.remove(list.size()-1);`） 事实上，问题出在了`self.DFS(candidates, target - candidates[i], i, temp_res + [candidates[i]])`

由于python解决方案中直接将`temp_res + [candidates[i]]`放在了递归语句中，则在递归遇到不满足条件跳出时，对应的`temp_res`也会将之前输入时加上的`[candidates[i]]`去掉；而上面java程序采用的是先将`[candidates[i]]`加到了`temp_res`，再传入递归程序DFS中，故而即便是跳出递归，`temp_res`并不会去除之前加上的`temp_res`，需要手动再加上后退程序：`temp_res.pop()`.

故而也可看与java对应的python版本：

```
class Solution:
    def combinationSum(self, candidates, target):
        """
        :type candidates: List[int]
        :type target: int
        :rtype: List[List[int]]
        """
        candidates.sort()
        Solution.res = []
        self.DFS(candidates, target, 0, [])

        return Solution.res

    def DFS(self, candidates, target, start, temp_res):
        if target == 0:
            Solution.res.append(temp_res[:])
            return
        for i in range(start, len(candidates)):
            if candidates[i] > target:
                return
            temp_res.append(candidates[i])
            self.DFS(candidates, target - candidates[i], i, temp_res)
            temp_res.pop()
```

注：还是推荐采用下面这种方式，因为直接将对`temp_res`的操作放在递归程序的输入函数中，容易出现一些问题；本题之所以成果是因为`temp_res + [candidates[i]]`的妥当使用，事实上，如果将其换为`temp_res.append(candidates[i])`，程序就会出现错误：

从debug的过程发现，出错原因在使用`temp_res + [candidates[i]]`会立刻对`temp_res`进行转换；而采用`temp_res.append(candidates[i])`，`temp_res`并不会立刻发生变化，而是直到下次达到此语句时候才进行了变化，与希望过程不符。

2.2 LeetCode 77. Combinations（组合）

Given two integers *n* and *k*, return all possible combinations of *k* numbers out of 1 ... *n*.

Example:

```
Input: n = 4, k = 2
Output:
[
  [2,4],
  [3,4],
  [2,3],
  [1,2],
  [1,3],
  [1,4],
]
```

这道题自以为按照 java 那种格式直接在 `i` 遍历时候加上 `start` 起始变量即可，事实上却走不通

递归法：

这个题要找到组合，组合和排列的不同之处在于组合的数字出现是没有顺序意义的。

剑指offer的做法是找出n个数字中m的数字的组合方法是，把n个数字分成两部分：第一个字符和其他的字符。如果组合中包括第一个字符，那么就在其余字符中找到m-1个组合；如果组合中不包括第一个字符，那么就在其余字符中找到m个字符。所以变成了递归的子问题。

我的做法如下，这个之中用到了if k > len(array)的做法，防止数组过界陷入死循环（其作用主要是对第二个递归而言的）。

```
class Solution(object):
    def combine(self, n, k):
        """
        :type n: int
```

```
        :type k: int
        :rtype: List[List[int]]
        """
        nums = [i+1 for i in range(n)]
        self.res = []
        temp_res = []
        # self.dfs(nums, k, 0, temp_res)
        self.recusion(nums, k, temp_res)
        return self.res

    def recusion(self, nums, k, temp_res):
        if k > len(nums):
            return
        if k == 0:
            self.res.append(temp_res)
            return

        self.recusion(nums[1:], k - 1, temp_res + [nums[0]])
        self.recusion(nums[1:], k, temp_res)
```

回溯法：

```
class Solution(object):
    def combine(self, n, k):
        """
        :type n: int
        :type k: int
        :rtype: List[List[int]]
        """

        nums = [i+1 for i in range(n)]
        self.res = []
        temp_res = []
        self.dfs(nums, k, temp_res)
        # self.recusion(nums, k, temp_res)
        return self.res


    def dfs(self, nums, k, temp_res):
        if k < 0:
            return
        if k == 0:
            self.res.append(temp_res[:])
            return
        for i in range(len(nums)):
            self.dfs(nums[i+1:], k-1, temp_res + [nums[i]])
```

回溯法中值得注意的是，每次回溯，修改的是 `array` 的输入长度，以及 `k`，没有设置 `start` 变量，实验了很多次，不好使，上述写法更实用！

> 上述约束中，使用 `if k > len(nums)` 要比使用 `if k < 0:` 约束更强，回溯更快！

C++ 版：

```
class Solution {
public:
    vector<vector<int>> combine(int n, int k) {
        vector<int> nums(n);
        for (int i = 1; i <= n; ++i) {
            nums[i - 1] = i;
        }
        combineCore(nums, k, {}, 0);
        return res;
    }


    void combineCore(vector<int>& nums, int k, vector<int> temp_res, int start) {
        if (temp_res.size() == k) {
            res.push_back(temp_res);
            return;
        }
        for (int i = start; i < nums.size(); ++i) {
            temp_res.push_back(nums[i]);
            combineCore(nums, k, temp_res, i + 1);
            temp_res.pop_back();
        }
    }
private:
vector<vector<int>> res;
};
```

一定要注意的是：回溯时候使用的是 `combineCore(nums, k, temp_res, i + 1);`

如果使用 `combineCore(nums, k, temp_res, start + 1);` 就错误了！

2.3 LeetCode 39. Combination Sum

Given a **set** of candidate numbers (`candidates`) **(without duplicates)** and a target number (`target`), find all unique combinations in `candidates` where the candidate numbers sums to `target`.

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

Note:

- All numbers (including `target`) will be positive integers.
- The solution set must not contain duplicate combinations.

Example 1:

```
Input: candidates = [2,3,6,7], target = 7,
A solution set is:
[
  [7],
  [2,2,3]
]
```

Example 2:

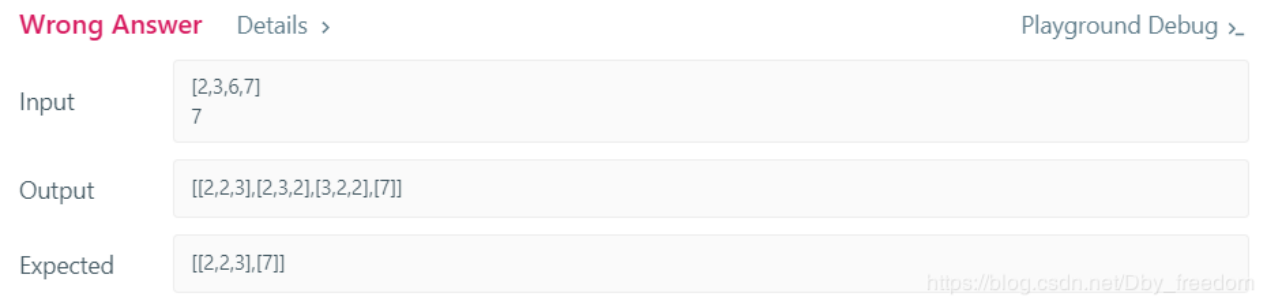
```
Input: candidates = [2,3,5], target = 8,
A solution set is:
[
  [2,2,2,2],
  [2,3,3],
  [3,5]
]
```

这道题也是一种变形，很值得注意的是，这道题就需要 `start` 变量，不然最终的结果会出现重复，如下：

错误版本：

```
class Solution(object):
    def combinationSum(self, candidates, target):
        """
        :type candidates: List[int]
        :type target: int
        :rtype: List[List[int]]
        """
        self.res = []
        self.dfs(candidates, target, [])
        return self.res


    def dfs(self, candidates, target, temp_res):
        if target == 0:
            self.res.append(temp_res)
            return
        if target < 0:
            return
        for i in range(len(candidates)):
            if candidates[i] > temp_res:
                continue
            self.dfs(candidates, target - candidates[i], temp_res +
[candidates[i]])
```



从错误中不难发现，错误原因在于出现重复项，对于这种情况，明显是希望当遍历一项之后，不再取前面的项，但可以取本项，故而要加上 `start` 来约束遍历范围，正确代码如下：

正确版本：

```
class Solution(object):
    def combinationSum(self, candidates, target):
        """
        :type candidates: List[int]
        :type target: int
        :rtype: List[List[int]]
        """
        self.res = []
        self.dfs(candidates, 0, target, [])
        # ret =  set(map(lambda x: sorted(x), self.res))
        return self.res


    def dfs(self, candidates, start, target, temp_res):
        if target == 0:
            self.res.append(temp_res)
            return
        if target < 0:
            return
        for i in range(start, len(candidates)):
            if candidates[i] > temp_res:
                continue
            self.dfs(candidates, i, target - candidates[i], temp_res +
[candidates[i]])
```

对应 C++ 版本：

```
class Solution {
public:
    vector<vector<int>> combinationSum(vector<int>& candidates, int target) {
        sort(candidates.begin(), candidates.end());
        combinationSumCore(candidates, target, {}, 0);
        return res;
```

```
    }


    void combinationSumCore(vector<int>& nums, int target, vector<int> temp_res,
int start) {
        if (target == 0) {
            res.push_back(temp_res);
            return;
        }
        if (target < 0) {
            return;
        }
        for (int i = start; i < nums.size(); ++i) {
            if (nums[i] > target) {
                return;
            }
            temp_res.push_back(nums[i]);
            combinationSumCore(nums, target - nums[i], temp_res, i);
            temp_res.pop_back();
        }
    }
private:
vector<vector<int>> res;
};
```

这里有个细节就是，位置 `start` 变量循环回溯的时候是用的 `combinationSumCore(nums, target - nums[i], temp_res, i);` 而不是 `combinationSumCore(nums, target - nums[i], temp_res, i + 1);`，前者对应本题，即可以重复采用同一位置样本，后者对应不能重复使用同一位置样本的情况。

2.4 LeetCode 40. Combination Sum II

Given a collection of candidate numbers (`candidates`) and a target number (`target`), find all unique combinations in `candidates` where the candidate numbers sums to `target`.

Each number in `candidates` may only be used **once** in the combination.

Note:

- All numbers (including `target`) will be positive integers.
- The solution set must not contain duplicate combinations.

Example 1:

```
Input: candidates = [10,1,2,7,6,1,5], target = 8,
A solution set is:
[
  [1, 7],
  [1, 2, 5],
  [2, 6],
  [1, 1, 6]
]
```

Example 2:

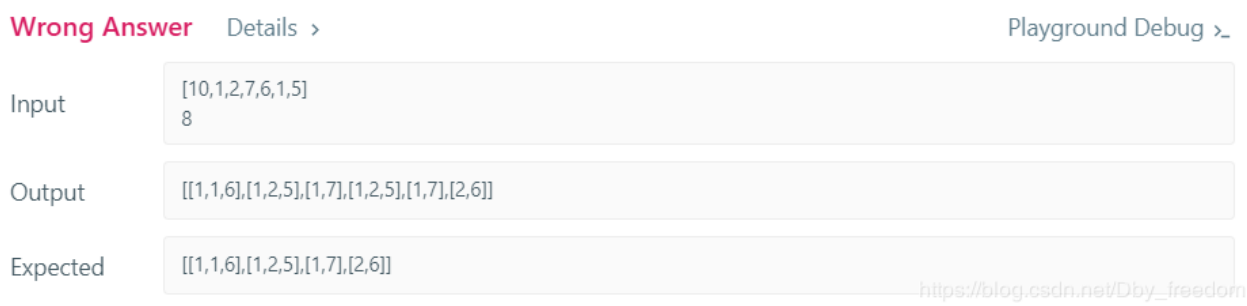
```
Input: candidates = [2,5,2,1,2], target = 5,
A solution set is:
[
  [1,2,2],
  [5]
]
```

这道题与上一道类似，但是又有不同，区别点在于不允许出现过的词再出现，故而，需要 `start` 每次 `+1`，但实验会发现，这样依旧存在问题，错误代码如下：

错误版本：

```
class Solution(object):
    def combinationSum2(self, candidates, target):
        """
        :type candidates: List[int]
        :type target: int
        :rtype: List[List[int]]
        """
        self.res = []
        candidates.sort()
        self.dfs(candidates, 0, target, [])
        return self.res


    def dfs(self, candidates, start, target, temp_res):
        if target == 0:
            self.res.append(temp_res)
            return
        if target < 0:
            return
        for i in range(start, len(candidates)):
            # if i > start and candidates[i] == candidates[i-1]:
            #     continue
            self.dfs(candidates, i + 1, target - candidates[i], temp_res +
[candidates[i]])
```



分下发现，其错误的主要原因是出现重复结果，这时又是跟上面一样，有两种解决思路，一种先找再过滤；一种是直接在遍历过程中直接忽略掉；

先找再过滤版本：

```
class Solution(object):
    def combinationSum2(self, candidates, target):
        """
        :type candidates: List[int]
        :type target: int
        :rtype: List[List[int]]
        """
        self.res = []
        candidates.sort()
        self.dfs(candidates, 0, target, [])
        return self.res


    def dfs(self, candidates, start, target, temp_res):
        if target == 0:
            if temp_res not in self.res:
                self.res.append(temp_res)
            return
        if target < 0:
            return
        for i in range(start, len(candidates)):
            # if i > start and candidates[i] == candidates[i-1]:
            #     continue
            self.dfs(candidates, i + 1, target - candidates[i], temp_res +
[candidates[i]])
```

直接在回溯过程中过滤：

这时候，要注意的是，过滤条件的确定，由答案可知，过滤是想过滤掉每次，以后回溯时，后面出现跟前面重复的情况（对应 `i > start and candidates[i] == candidates[i-1]`），尤其注意是 `i > start`，不是 `i > 1`，因为是向后遍历的！

代码如下：

```
class Solution(object):
    def combinationSum2(self, candidates, target):
        """
```

```
        :type candidates: List[int]
        :type target: int
        :rtype: List[List[int]]
        """
        self.res = []
        candidates.sort()
        self.dfs(candidates, 0, target, [])
        return self.res


    def dfs(self, candidates, start, target, temp_res):
        if target == 0:
            if temp_res:
                self.res.append(temp_res)
            return
        if target < 0:
            return
        for i in range(start, len(candidates)):
            if i > start and candidates[i] == candidates[i-1]:
                continue
            self.dfs(candidates, i + 1, target - candidates[i], temp_res +
[candidates[i]])
```

对应 C++ 版本：

```
class Solution {
public:
    vector<vector<int>> combinationSum2(vector<int>& candidates, int target) {
        sort(candidates.begin(), candidates.end());
        // vector<char> mark(candidates.size(), false);
        combinationSum2Core(candidates, target, {}, 0);
        return res;
    }


    void combinationSum2Core(vector<int>& nums, int target, vector<int> temp_res,
int start) {
        if (target == 0) {
            res.push_back(temp_res);
            return;
        }
        if (target < 0) return;
        for (int i = start; i < nums.size(); ++i) {
            if (nums[i] > target || (i > start && nums[i] == nums[i - 1]))
continue;
            temp_res.push_back(nums[i]);
            combinationSum2Core(nums, target - nums[i], temp_res, i + 1);
            temp_res.pop_back();
```

```
        }
    }
private:
vector<vector<int>> res;
};
```

这道题其实改动就一点，区别点在于不允许出现过的词再出现。

其改动点有两处值得注意：

1. 回溯时候需要 `start` 每次 `+1`；
2. 去重判定时候用的是 `if (nums[i] > target || (i > start && nums[i] == nums[i - 1]))` 而非 `i > 0`，这点要区别于上面 LeetCode 47 题，因为 LeetCode 47 是做排列，同时有 `mark` 列表辅助，只需要保证遍历过的元素不再重复进行遍历即可，而本题，如果使用 `i > 0` 则会泄露如 `[1, 1, 6]` 这种情况，自己开始做的时候还尝试加上 `mark` 数组来辅助，反倒麻烦了，这里只需要设定为 `i > start` 即可保证当第一次使用了元素值 `1` 之后，回溯时候依旧是 `i = start` 开始，这样，依旧不能满足 `(i > start && nums[i] == nums[i - 1])`，这时候可以保证不会漏掉如 `[1, 1, 6]` 这种情况，同时可以保证不会出现两次 `[1,7]`；
3. 当然也可以采用后验方式，即加入 `temp_res` 时候判定下 `temp_res` 是否符合要求：`if find(res.begin(), res.end(), temp_res)== res.end()`，只是这种方式相对效率较低。

2.5 LeetCode 216. Combination Sum III

Find all possible combinations of ***k*** numbers that add up to a number ***n***, given that only numbers from 1 to 9 can be used and each combination should be a unique set of numbers.

Note:

- All numbers will be positive integers.
- The solution set must not contain duplicate combinations.

Example 1:

```
Input: k = 3, n = 7
Output: [[1,2,4]]
```

Example 2:

```
Input: k = 3, n = 9
Output: [[1,2,6], [1,3,5], [2,3,4]]
```

依据上面思路不难分析，结果如下：

```
class Solution(object):
    def combinationSum3(self, k, n):
        """
        :type k: int
        :type n: int
```

```
        :rtype: List[List[int]]
        """
        self.res = []
        nums = [i + 1 for i in range(9)]
        self.dfs(nums, 0, k, n, [])
        return self.res


    def dfs(self, nums, start, k, target, temp_res):
        if target == 0 and len(temp_res) == k:
            self.res.append(temp_res)
            return
        if target < 0 or len(temp_res) > k:
            return
        for i in range(start, len(nums)):
            self.dfs(nums, i + 1, k, target - nums[i], temp_res + [nums[i]])
```

对应 C++ 版本：

```
class Solution {
public:
    vector<vector<int>> combinationSum3(int k, int n) {
        combinationSum3Core(k, n, {}, 1);
        return res;
    }
    void combinationSum3Core(int k, int target, vector<int> temp_res, int start) {
        if (k == 0 && target == 0) {
            res.push_back(temp_res);
            return;
        }
        if (k < 0 || target < 0) return;
        for (int i = start; i <= 9; ++i) {
            if (i > target) return;
            temp_res.push_back(i);
            combinationSum3Core(k - 1, target - i, temp_res, i + 1);
            temp_res.pop_back();
        }
    }

private:
vector<vector<int>> res;
};
```

2.6 LeetCode 22. Generate Parentheses（括号生成）

原题

Given n pairs of parentheses, write a function to generate all combinations of well-formed parentheses.

For example, given n = 3, a solution set is:

题目： 给出 n 代表生成括号的对数，请你写出一个函数，使其能够生成所有可能的并且有效的括号组合。

Example:

```
[
  "((()))",
  "(()())",
  "(())()",
  "()(())",
  "()()()"
]
```

Solution:

```
class Solution:
    def generateParenthesis(self, n):
        """
        :type n: int
        :rtype: List[str]
        """
        Solution.res = []
        n, start, temp_res, len_left, len_right, candidate_list = n, 0, "", 0, 0,
["(", ")"]
        self.backtracking(n, len_left, len_right, temp_res, candidate_list)
        return Solution.res

    def backtracking(self, n, len_left, len_right, temp_res, candidate_list):

        # len_left = len_right = 0

        if len_left == n and len_right == n:
            Solution.res.append(temp_res[:])

        if len_left > n or len_right > n or len_right > len_left:
            return

        if len(temp_res) < 2 * n:
            for i in range(len(candidate_list)):
                temp_res += candidate_list[i]
                if candidate_list[i] == "(":
                    self.backtracking(n, len_left + 1, len_right, temp_res,
candidate_list)
                else:
                    self.backtracking(n, len_left, len_right + 1, temp_res,
```

```
candidate_list)
                temp_res = temp_res[: len(temp_res) - 1]
```

依旧采用的回溯法，只是针对字符串进行操作需要注意：

1. 对于字符串进行回溯，一个很好的方法是直接写入回溯表达式（对于列表格式慎用，append不支持这种事实上赋值），如上述for循环表达式，可以替换为：

```
        if len(temp_res) < 2 * n:
            for i in range(len(candidate_list)):
                # temp_res += candidate_list[i]
                if candidate_list[i] == "(":
                    self.backtracking(n, len_left + 1, len_right, temp_res +
candidate_list[i], candidate_list)
                else:
                    self.backtracking(n, len_left, len_right + 1, temp_res +
candidate_list[i], candidate_list)
                # temp_res = temp_res[: len(temp_res) - 1]
```

2. 对于回溯法字符串删除，可采用 `temp_res = temp_res[: len(temp_res) - 1]` 曲线救国。

对应 C++ 代码：

```
class Solution {
public:
    vector<string> generateParenthesis(int n) {
        generateParenthesisCore("", n, n);
        return res;
    }


    void generateParenthesisCore(string temp_res, int left, int right) {
        if (left > right) return;
        if (left == 0 && right == 0) {
            res.push_back(temp_res);
            return;
            // return;
        }
        else {
            if (left > 0) {
                generateParenthesisCore(temp_res + '(', left - 1, right);
            }
            if (right > 0) {
                generateParenthesisCore(temp_res + ')', left, right - 1);
            }
        }
    }
private:
    vector<string> res;
```

```
};
```

注意：这道题的 C++ 代码有两点值得注意：

1. 对于这种带有约束的条件组合问题，采用 `if (left > 0)` 进行判定回溯更方便，不用再采用 `for` 循环；
2. 最需要注意的是，对于回溯调用传参时候，不能使用 `--left` 来代替 `left - 1`，因为前者表示 `left = left - 1`，`left` 等于永久性地少了1，而后者只是将 `left - 1` 作为参数传入调用（自己开始在这里卡了很久）。

参考文献：

回溯详解及其应用：Leetcode 39 combination sum [python]回溯法模板 手把手教你中的回溯算法——多一点套路