# 求子集II

## 第90题.子集II

题目链接：https://leetcode-cn.com/problems/subsets-ii/

给定一个可能包含重复元素的整数数组 nums，返回该数组所有可能的子集（幂集）。

说明：解集不能包含重复的子集。

示例:

输入: [1,2,2]

输出:

[

[2],

[1],

[1,2,2],

[2,2],

[1,2],

[]

]

对于可能包含重复元素的整型数组nums
思路:
首先求子集的时候可以先sort完后,
用一个used数组表示同一树枝上的使用情况,
used数组的大小为nums.size()
used[i]=1表示**下标为i的字符**已经使用了,而used[i]=0表示**下标为i的字符**尚未使用.
单层逻辑中:

```
for(int i=startIndex;i<nums.size();i++){
    if(i>0&&used[i-1]==0&&nums[i]==nums[i-1]) countinue;
     path.push_back(nums[i]);
     used[i]=1;
     递归函数
     used[i]=0;
     path.pop_back();
}
```

```
1 class solution{
2  vector<vector<int>> result;
```

```
vector<int> path;
void backtracking(vector<int>& nums,int startIndex,vector<bool>& used){
result.push_back(path);
for(int i=startIndex;i<nums.size();i++){
if(i>0&&used[i-1]==false&&nums[i-1]==nums[i]) continue;
path.push_back(nums[i]);
used[i]=true;
 backtracking(nums,i+1,used);
 used[i]=false;
 path.pop_back();
 }
 }
  vector<vector<int>> subsetsWithDup(vector<int>& nums){
 vector<bool> used(nums.size(),false);
 sort(nums.begin(),nums.end());
 backtracking(nums,0,used);
 return result;
 }

};
```

# 递增子序列

> 和**子集问题**有点像，但**又处处是陷进**。

## 491.递增子序列

题目链接：https://leetcode-cn.com/problems/increasing-subsequences/

给定一个整型数组，你的任务是找到所有该数组的递增子序列，递增子序列的长度至少是2。

示例：

输入：[4, 6, 7, 7] 输出：[[4, 6], [4, 7], [4, 6, 7], [4, 6, 7, 7], [6, 7], [6, 7, 7], [7,7], [4,7,7]]

- 给定数组中可能包含重复数字，相等的数字应该被视为递增的一种情况。

> 本题目也是一个**标准的去重**的**求子集**问题,由于本题目求的是子序列**不能对nums数组进行排序**
> **递归函数参数:**
> vector<vector<int>> result;
> vector<int> path;
> void backtracking(vector<int>& nums,int startIndex)

```
终止条件：
由于递增子序列记录的是每一棵树上的每个节点,要求递增子序列的长度至少是2
if(path.size()>1){
    result.push_back(path);
    //此时不需要用return 因为记录的是树上的每个节点
}
单层逻辑：
for(int i=startIndex;i<nums.size();i++){
    if((path.size()>0&&nums[i]<path.back())||(uset.find(nums[i])!=uset.end()))
            continue;
//记录这个元素在本层用过了，本层后面不能再用了
//为什么不采用之前求子集的方法去重因为本道题不能排序
//为什么可以用set的方法去重因为nums序列为ABCD其中C,D相同
//当C,D同层且满足A<=B<=C时那么会有重复子集ABC==ABD
    uset.insert(nums[i]);
    path.push_back(nums[i]);
    backtracking(nums,i+1);
    path.pop_back();
}
```

由于map、set以及数组都是哈希的表示，如果数值表示范围比较小时则采用数组解决问题即可.

代码：

```cpp
class Solution {
public:
 vector<vector<int>> result;
 vector<int> path;
 void backtracking(vector<int>& nums,int startIndex){
 //由于递增子序列要求元素个数至少两个
 if(path.size()>1){
 result.push_back(path);
 //此处不能用return
  }
  //利用数组代替map和set来达到去重的目的，不能设置将数组容量正好设置为200
  vector<int> used(202,0);
  for(int i=startIndex;i<nums.size();i++){
  //去重逻辑
  if((!path.empty()&&nums[i]<path.back())||(used[nums[i]+100]==1))
  continue;
  used[nums[i]+100]=1;
  path.push_back(nums[i]);
  backtracking(nums,i+1);
  path.pop_back();
  }
  }
```

```
    vector<vector<int>> findSubsequences(vector<int>& nums)    {
backtracking(nums,0);
return result;
    }
};
```

求子集法(排序后)的去重和该问题的去重都是本层去重(**同一个父节下的本层,而不是整棵树的本层**)

# 为什么求子集法的时候需要排序才能去重达到?

求子集法再排序前的去重是整棵树的同一层去重，由于整棵树的同一层是否出现重复元素是我们无法控制的，所以子集问题才会通过排序之后，判断相邻两个分支的同一层是否出现相同元素来达到去重目的

理解以上内容，就知道了：

- 回溯算法：递增子序列去重用set的定义为什么放在单层搜索的逻辑里，而不是放在全局变量里。
- 为什么回溯算法：求子集问题（二）的去重要排序。

# 排列问题来了

## 46.全排列

题目链接：https://leetcode-cn.com/problems/permutations/

给定一个 没有重复 数字的序列，返回其所有可能的全排列。

示例:

输入: [1,2,3]

输出:

[

[1,2,3],

[1,3,2],

[2,1,3],

[2,3,1],

[3,1,2],

[3,2,1]

]

排列问题就是需要**同一树枝上去重,所以需要一个used数组来标记是否使用过**

```
class Solution {
public:
```

```
    vector<vector<int>> result;
    vector<int> path;
    void backtracking (vector<int>& nums, vector<bool>& used) {
        // 此时说明找到了一组
        if (path.size() == nums.size()) {
            result.push_back(path);
            return;
        }
        for (int i = 0; i < nums.size(); i++) {
            if (used[i] == true) continue; // path里已经收录的元素，直接跳过
            used[i] = true;
            path.push_back(nums[i]);
            backtracking(nums, used);
            path.pop_back();
            used[i] = false;
        }
    }
    vector<vector<int>> permute(vector<int>& nums) {
        result.clear();
        path.clear();
        vector<bool> used(nums.size(), false);
        backtracking(nums, used);
        return result;
    }
};
```

# 总结：

组合问题和排序问题的区别在于**单层逻辑时的遍历**

**组合问题的输入参数为一个startIndex则单层逻辑的遍历从startIndex开始**

**排序问题的输入参数为一个数组标记该一数枝的使用情况则单层逻辑的遍历从0开始，要去除使用过的元素**

## 对以上的排列问题进一步升级如下：

# 47.全排列 II

题目链接：https://leetcode-cn.com/problems/permutations-ii/

给定一个可包含重复数字的序列 nums ，按任意顺序 返回所有不重复的全排列。

示例 1：

输入：nums = [1,1,2]

输出：

[[1,1,2],

[1,2,1],

[2,1,1]]

示例 2：

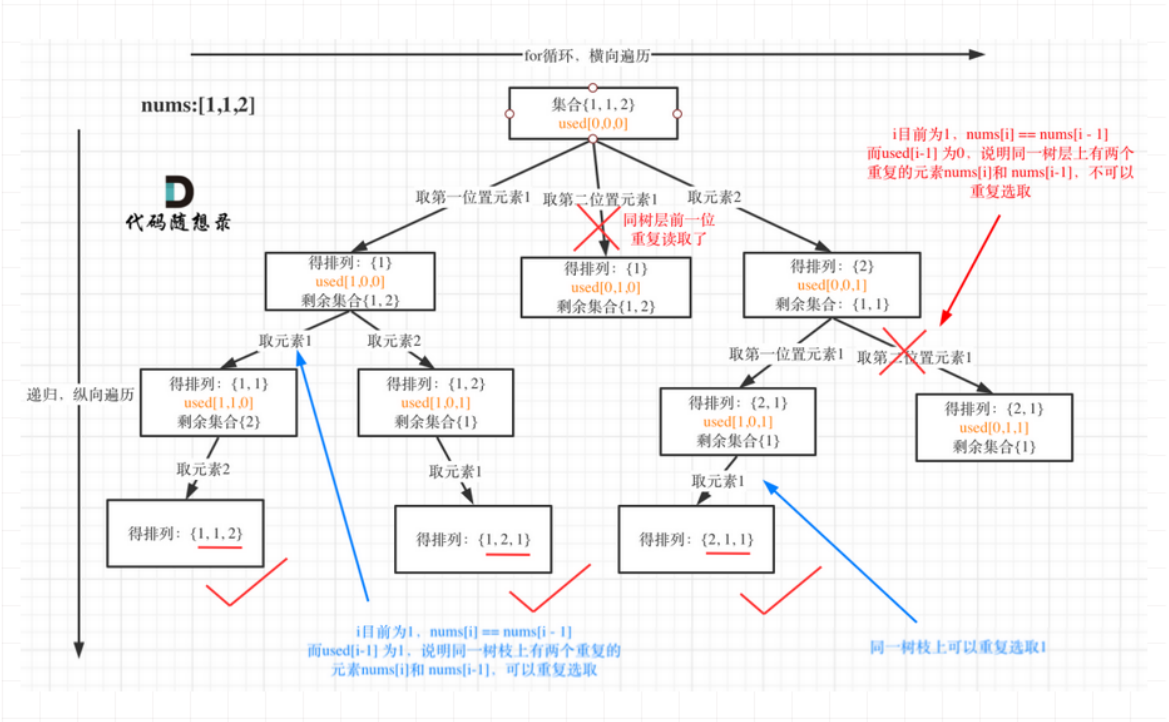
输入：nums = [1,2,3]

输出：[[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]

提示：

- 1 <= nums.length <= 8
- -10 <= nums[i] <= 10

> 这道题目和上面一道题目的区别在于“给定一个可包含重复数字的序列”，要返回**「所有不重复的全排列」**
> 还要强调的是**「还要强调的是去重一定要对元素进行排序，这样我们才方便通过相邻的节点来判断是否重复使用了」**。




**「一般来说：组合问题和排列问题是在树形结构的叶子节点上收集结果，而子集问题就是取树上所有节点的结果」。**

```
class Solution {
public:
  vector<vector<int>> result;
    vector<int> path;
    void backtracking (vector<int>& nums, vector<bool>& used) {
        // 此时说明找到了一组
        if (path.size() == nums.size()) {
            result.push_back(path);
            return;
         }
         for (int i = 0; i < nums.size(); i++) {
           //used[i-1]==false表示同一树层使用过，则跳过
//used[i-1]==true表示同一树枝使用过.
            if (i > 0 && nums[i] == nums[i - 1] && used[i - 1] == false)
                continue;
            }
            if (used[i] == false) {//表示同一树枝没有使用过
                used[i] = true;
                path.push_back(nums[i]);
                backtracking(nums, used);
                path.pop_back();
                used[i] = false;
            }
         }
```

```
25      }
26      vector<vector<int>> permuteUnique(vector<int>& nums) {
27  result.clear();
28          path.clear();
29          sort(nums.begin(), nums.end()); // 排序
30          vector<bool> used(nums.size(), false);
31          backtracking(nums, used);
32          return result;
33      }
34  };
```