回溯法：

```
# 回溯法
        result = []
        def backtrack(路径, 选择列表):
            if 满足结束条件:
                result.add(路径)
                return
            for 选择 in 选择列表:
                做选择
                backtrack(路径, 选择列表)
                 撤销选择
```

1 全排列：

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]
]
```

```
# def backtrack(路径, 选择列表): 路径path, 选择列表 nums
def dfs(nums, size, depth, used, path, res):
    # nums: 要进行全排列的序列
    # size: 序列大小
    # depth : 第几层
    # used: 序列，记录当前数是否被使用
    # path: 当前的选择除了的路径，就是一种选择结果
    # res: 结果
            # 回溯终结条件：把路径添加到结果
             # 层数 == 序列大小，  说明已经到了树的最底层，叶子节点，把路径加入到结果
             if depth == size:
                 res.append(path[:])
                 return
             # for 选择 in 选择列表
             # 从列表中做出选择
             for i in range(size):
                 # 做选择
```

```
                # 如果第i个数，没有添加过，就把这个数添加进了
                if not used[i]:
                    used[i] = True
                    path.append(nums[i])

                     # 在下一层去做选择，深度+1
                    dfs(nums, size, depth+1, used, path, res)

                    # 撤回选择
                    used[i] = False
                    path.pop()
```

47. 全排列 II

难度 中等 371

给定一个可包含重复数字的序列，返回所有不重复的全排列。

示例:

```
输入: [1,1,2]
输出:
[
  [1,1,2],
  [1,2,1],
  [2,1,1]
]
```

```
# 分析 ，全排列II和全排列的区别在于，全排列中有重复元素，那么先按全排列写出代码，除去重
def permuteUnique(self, nums):
        """
        :type nums: List[int]
        :rtype: List[List[int]]
        """
        def dfs(nums, size, depth, path, used, res):
            if depth == size:
                res.append(path[:])
                 return
             for i in range(size):
                if not used[i]:
                    # 在图中 ① 处，搜索的数也和上一次一样，但是上一次的 1 刚刚被
                    if i > 0 and nums[i] == nums[i - 1] and not used[i -
                        continue

                    used[i] = True
                    path.append(nums[i])
                    dfs(nums, size, depth + 1, path, used, res)
```

```
                    used[i] = False
                    path.pop()

        size = len(nums)
        if size == 0:
            return []

        # 注意：这里是关键，需要对nums进行一次排列，这样，如果后面的数再次出现，就可
        nums.sort()

        used = [False] * len(nums)
        res = []
        dfs(nums, size, 0, [], used, res)
        return res
```

39 组合总和

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]
]
```

解题思路如下图：

因为target = 7， 如果第一个先减2，找到 7 -2 =5 的所有组合，那么 +2，就是7的组合

如果第一个先减3，找到7-3 = 4 的所有组合，在+3，那就是7的组合

依次类推

说明这个可以用回溯法

1 确定递归终止条件：

因为taget一直减序列中的数，得到的差值就是新的target，那么一直到taget == 0（图中红色），在结果中添加那条路径

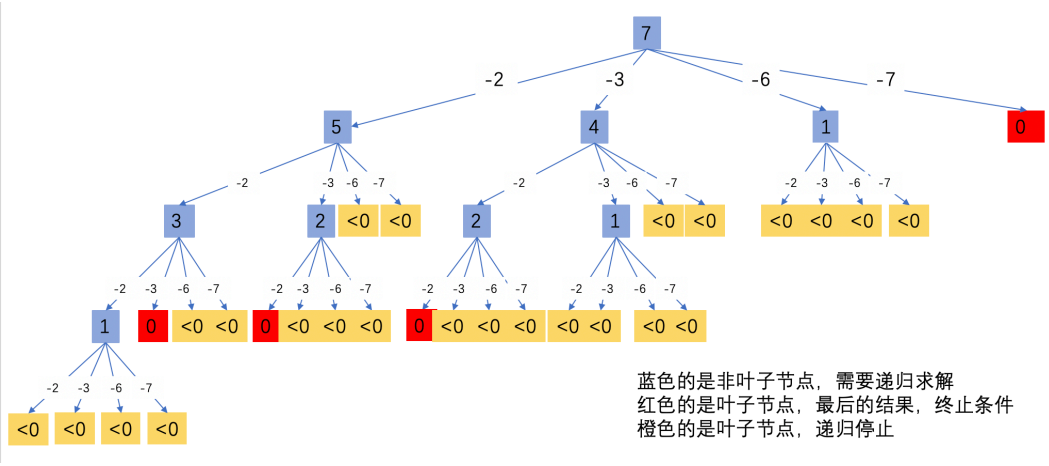
```
if target == 0:
```

```
                res.append(path[:])
                return
```

2 做递归求解，图中蓝色部分，就是target不为 0 的时候，从列表中找要减的数

```
for i in range(begin, size):
    #[2,3,6,7]
                residue = target - candidates[i]
                # 第二个终止的可能，那就是，差值比0小，也不用循环了
                if residue < 0 :
                    break
                # 把对应减的数添加到path里
                path.append(candidates[i])
                # 回溯 这里的begin 换成了 i，[3,6,7],
                backtrack(candidates, path, i, size, residue, res )

                path.pop()
```




```
def combinationSum(self, candidates, target):
        """
        :type candidates: List[int]
        :type target: int
        :rtype: List[List[int]]
        """
        size = len(candidates)
        if size < 0 :
            return
         res = []
         path = []
         candidates.sort()
```

```
        def dfs(path, begin,  target ):
            # 终止条件: target == 0
            if target == 0:
                res.append(path[:])
                return
            for i in range(begin, size):
                residue = target - candidates[i]
                if residue < 0 :
                    break
                path.append(candidates[i])
                dfs(path, i, residue)
                path.pop()

        dfs(path, 0, target)
        return res
```

40. 组合总和 II

难度 中等 331

给定一个数组 `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` 中的每个数字在每个组合中只能使用一次。

说明：

- 所有数字（包括目标数）都是正整数。
- 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,
所求解集为:
[
  [1, 7],
  [1, 2, 5],
  [2, 6],
  [1, 1, 6]
]
```

示例 2:

```
输入: candidates = [2,5,2,1,2], target = 5,
所求解集为:
[
  [1,2,2],
  [5]
]
```

```
#组合2 和组合的区别在于，组合2的每个数字在每个组合只能用一次，所以，我就想直接在在原来
即把 dfs(path, i, residue) == 》dfs(path, i+1, residue) --〉这样就能保证每个
但是，不完全对，因为说明中还要求，解集中不包含重复组合
candidates全排列后，当candidates[i] == candidates[i-1]: 退出此次循环，说明这个

def combinationSum2(self, candidates, target):
        """
        :type candidates: List[int]
        :type target: int
         :rtype: List[List[int]]
         """
         size = len(candidates)
         if size < 0 :
             return
         res = []
         path = []
         candidates.sort()
         def dfs(path, begin,  target ):
             # 终止条件: target == 0
             if target == 0:
                 res.append(path[:])
                 return

             for i in range(begin, size):
                 residue = target - candidates[i]
                 if residue < 0 :
                     break
                 if i > begin and candidates[i] == candidates[i-1]:
                     continue
                 path.append(candidates[i])
                 dfs(path, i+1, residue)
                 path.pop()

         dfs(path, 0, target)
         return res
```

子集：

78. 子集

难度 中等 705

给定一组**不含重复元素**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: nums = [1,2,3]
输出:
[
  [3],
  [1],
  [2],
  [1,2,3],
  [1,3],
  [2,3],
  [1,2],
  []
]
```

求解方法，三种：

第一种：库函数 itertools.combinations(nums, i)：返回值是，nums中，长度为 i的子序列

```
def subsets(self, nums):
        res = []
        for i in range(len(nums) + 1):
            # 返回nums中，长度为 i的所有子序列
            for tmp in itertools.combinations(nums, i):
                res.append(tmp)
        return res
```

第二种：迭代

```
def subsets(self, nums):
        res = [[]]
        for i in nums:
            res = res + [[i] + num for num in res]
        return res
```

第三章：递归/回溯

```
def subsets(self, nums):

        #递归法：
        # 求长度为k的子集
        def backtrack(first = 0, curr = []):
            # if the combination is done
            if len(curr) == k:
                output.append(curr[:])
            for i in range(first, n):
                 # add nums[i] into the current combination
                 curr.append(nums[i])
                 # use next integers to complete the combination
```

```
                backtrack(i + 1, curr)
                # backtrack
                curr.pop()

        output = []
        n = len(nums)
        # 遍历 k的所有可能
        for k in range(n + 1):
            backtrack()
        return output
```

90 子集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],
  []
]
```

最简单的想法就是利用子集，在结果中，把重复元素去掉

```
def subsetsWithDup(self, nums):
        """
        :type nums: List[int]
        :rtype: List[List[int]]
        """
        if not nums:
            return []
        res = []
        n = len(nums)
         nums.sort()
         # 求长度为k的子集
         def backtrack(first = 0, curr = []):
             # if the combination is done
             if len(curr) == k:
                 # 去重
                 if curr not in output:
                     output.append(curr[:])
             for i in range(first, n):
                 # add nums[i] into the current combination
```

```
                curr.append(nums[i])
                # use next integers to complete the combination
                backtrack(i + 1, curr)
                # backtrack
                curr.pop()

        output = []
        n = len(nums)
        # 遍历 k的所有可能
        for k in range(n + 1):
            backtrack()
        return output
```

递归法二：

```
def subsetsWithDup(self, nums):
        if not nums:
            return []
        res = []
        n = len(nums)
        nums.sort()
        def helper(i, tmp):
            if tmp not in res:
                res.append(tmp)
             for j in range(i, n):
                helper(j + 1,tmp + [nums[j]] )
        helper(0, [])
        return res
```