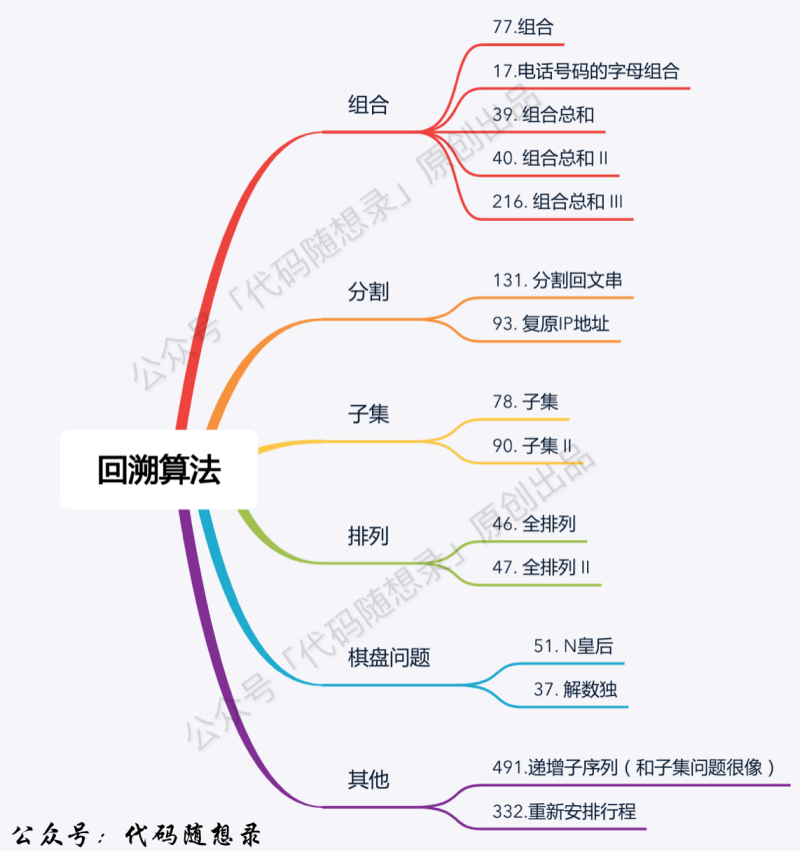
回溯算法(backtracking)

方法论



回溯法，一般可以解决如下几种问题：

- 组合问题：N个数里面按一定规则找出k个数的集合
- 切割问题：一个字符串按一定规则有几种切割方式
- 子集问题：一个N个数的集合里有多少符合条件的子集
- 排列问题：N个数按一定规则全排列，有几种排列方式
- 棋盘问题：N皇后，解数独等等

本质：N叉树！！

```
void backtracking(参数) {
    if (终止条件) {
        存放结果;
        return;
    }
    for (选择：本层集合中元素（树中节点孩子的数量就是集合的大小）) {
        处理节点;
        backtracking(路径，选择列表); // 递归
        回溯，撤销处理结果
    }
}
```

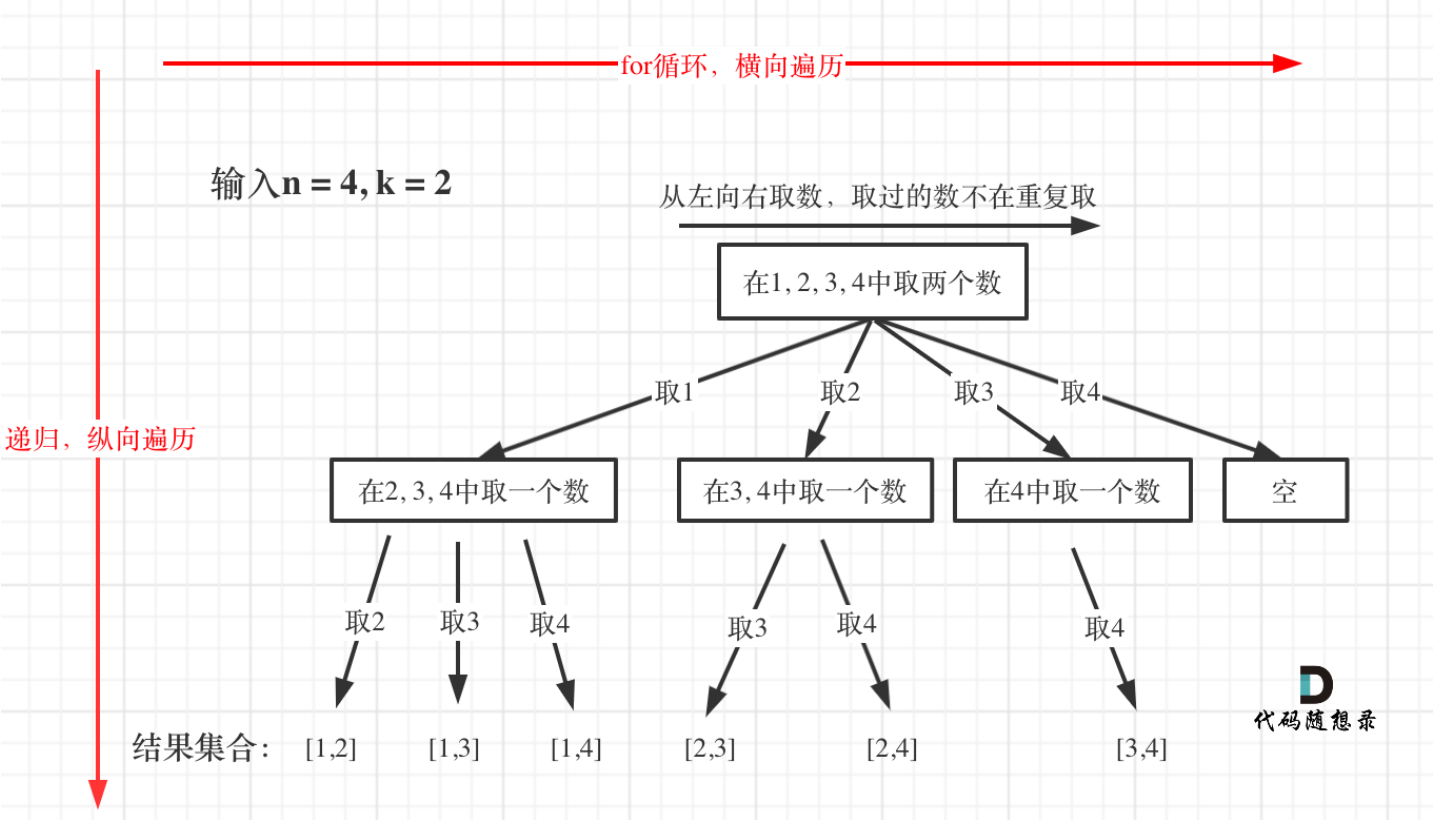
一些要注意的点：

- 什么时候使用startindex：如果是一个集合来求组合的话，就需要startIndex，例如：77.组合，216.组合总和III。如果是多个集合取组合，各个集合之间相互不影响，那么就不用startIndex，例如：[17.电话号码的字母组合]
- 什么时候回溯时要i+1？： 元素不能重复选取
- res.append(path[:]) **记得浅复制！！！**
- **for控制的是横向，递归控制的是纵向(观察77题，39题，46题区别)**

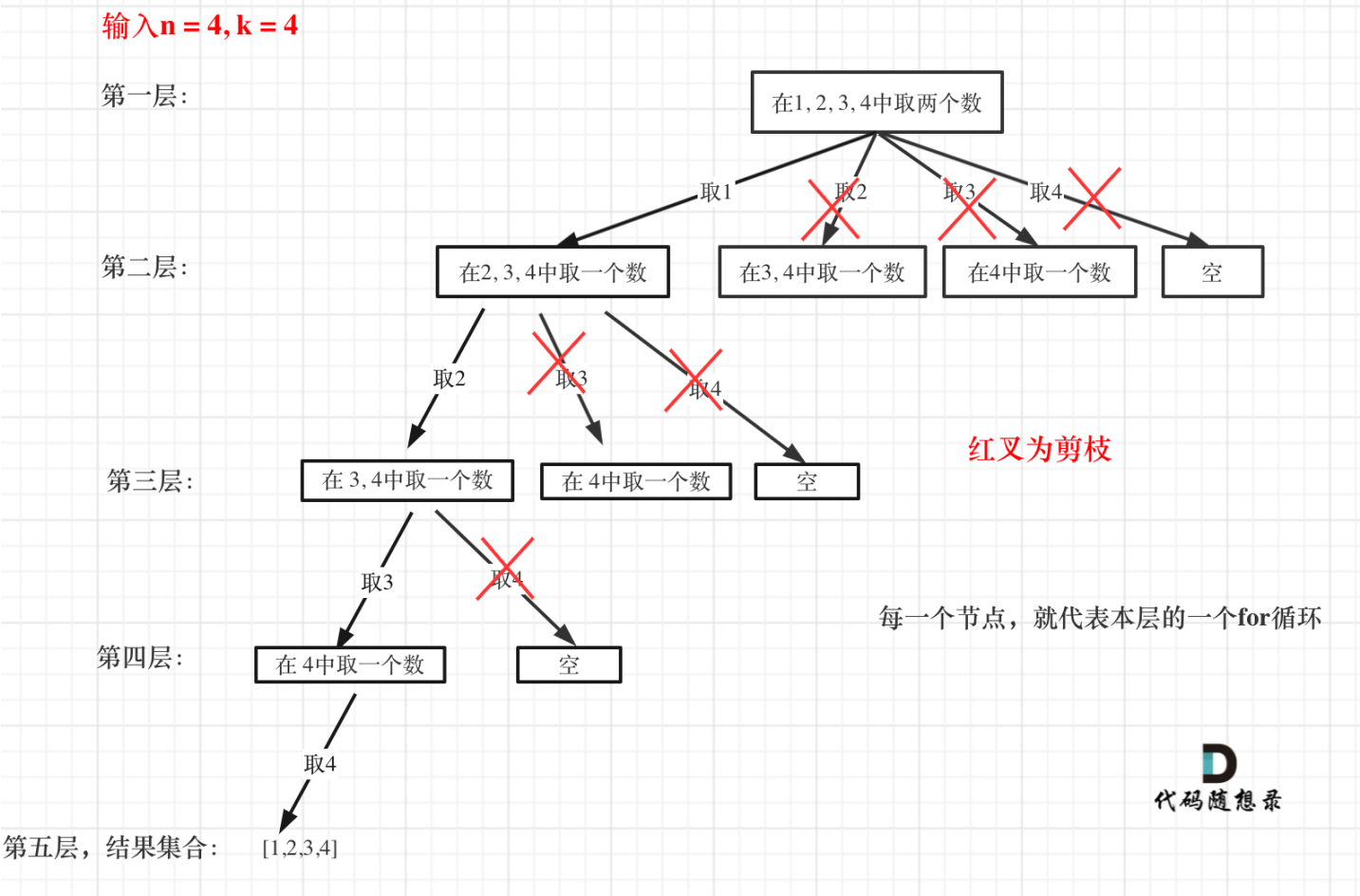
题目

组合(combinations)

77. Combinations




```python
class Solution:
    def combine(self, n: int, k: int) -> List[List[int]]:
        res = []
        path = []
        def backtracking(n,k,startindex):
            if(len(path)==k):
                res.append(path[:]) #[:]为浅拷贝
                return
            for i in range(startindex,n+1):
                path.append(i)
                backtracking(n,k,i+1)
                path.pop()
        backtracking(n,k,1)
        return res
```



可以剪枝的地方就在递归中每一层的for循环所选择的起始位置。

如果for循环选择的起始位置之后的元素个数 已经不足 我们需要的元素个数了，那么就没有必要搜索了。

```
class Solution:
    def combine(self, n: int, k: int) -> List[List[int]]:
        res=[]  #存放符合条件结果的集合
        path=[]  #用来存放符合条件结果
        def backtrack(n,k,startIndex):
            if len(path) == k:
                res.append(path[:])
                return
            for i in range(startIndex,n-(k-len(path))+2):  #优化的地方
                path.append(i)  #处理节点
                backtrack(n,k,i+1)  #递归
                path.pop()  #回溯，撤销处理的节点
        backtrack(n,k,1)
        return res
```

216. Combination Sum III

```
class Solution:
    def combinationSum3(self, k: int, n: int) -> List[List[int]]:
        path = []
        res = []
        def backtracking(k,n,sum_val,startindex):
            #pruning1(对和的pruning)
            if sum_val>n:
                return

            if len(path)==k:
                if sum_val == n:
                    res.append(path[:])
                    return

            #for i in range(startindex,10): #before pruning
            for i in range(startindex,10 - (k - len(path)) + 1): #pruning2(对k的pruning, 同
77题)
                sum_val += i
                path.append(i)
                backtracking(k,n,sum_val,i+1)
                path.pop()
                sum_val -= i

        backtracking(k,n,0,1)
        return res
```

17.Letter Combinations of a Phone Number

```
class Solution:
    def letterCombinations(self, digits: str) -> List[str]:
        path = []
        res = []
        dic = {
            '2':'abc',
            '3':'def',
            '4':'ghi',
            '5':'jkl',
            '6':'mno',
            '7':'pqrs',
            '8':'tuv',
            '9':'wxyz'
        }
```

```
        def backtracking(length):
            if not digits:
                return

            if len(digits) == length:
                res.append(''.join(path))
                return

            digit = digits[length] # 当前数字的英文
            for letter in dic[digit]:
                path.append(letter)
                backtracking(length+1)
                path.pop()
        backtracking(0)
        return res
```

注意字典的使用

注意字符串添加的方法，还是一开始使用path数组，但是可以使用 `''.join(path)` 将path中的字母变成字符串，见下面例子

```
path = []
string = 'abc'
path.append(string[0])
print(path)
#['a']

path2 = ['a','b','c']
s = ''.join(path2)
print(s)
print(type(s))

#abc
#<class 'str'>
```

本题每一个数字代表的是不同集合，也就是求不同集合之间的组合，所以不是从startindex开始遍历

39.Combination Sum

```
class Solution:
    def combinationSum(self, candidates: List[int], target: int) -> List[List[int]]:
        path = []
        res = []
        def backtracking(start_index,sum_val):
```

```
            if sum_val == target:
                res.append(path[:])
                return
            if sum_val > target: #pruning
                return
            for i in range(start_index,len(candidates)):
                sum_val += candidates[i]
                path.append(candidates[i])
                backtracking(i,sum_val) #可以重复选择 不用i+1
                sum_val -= candidates[i]
                path.pop()
        backtracking(0,0)
        return res
```

可以重复选择，但仍然需要startindex

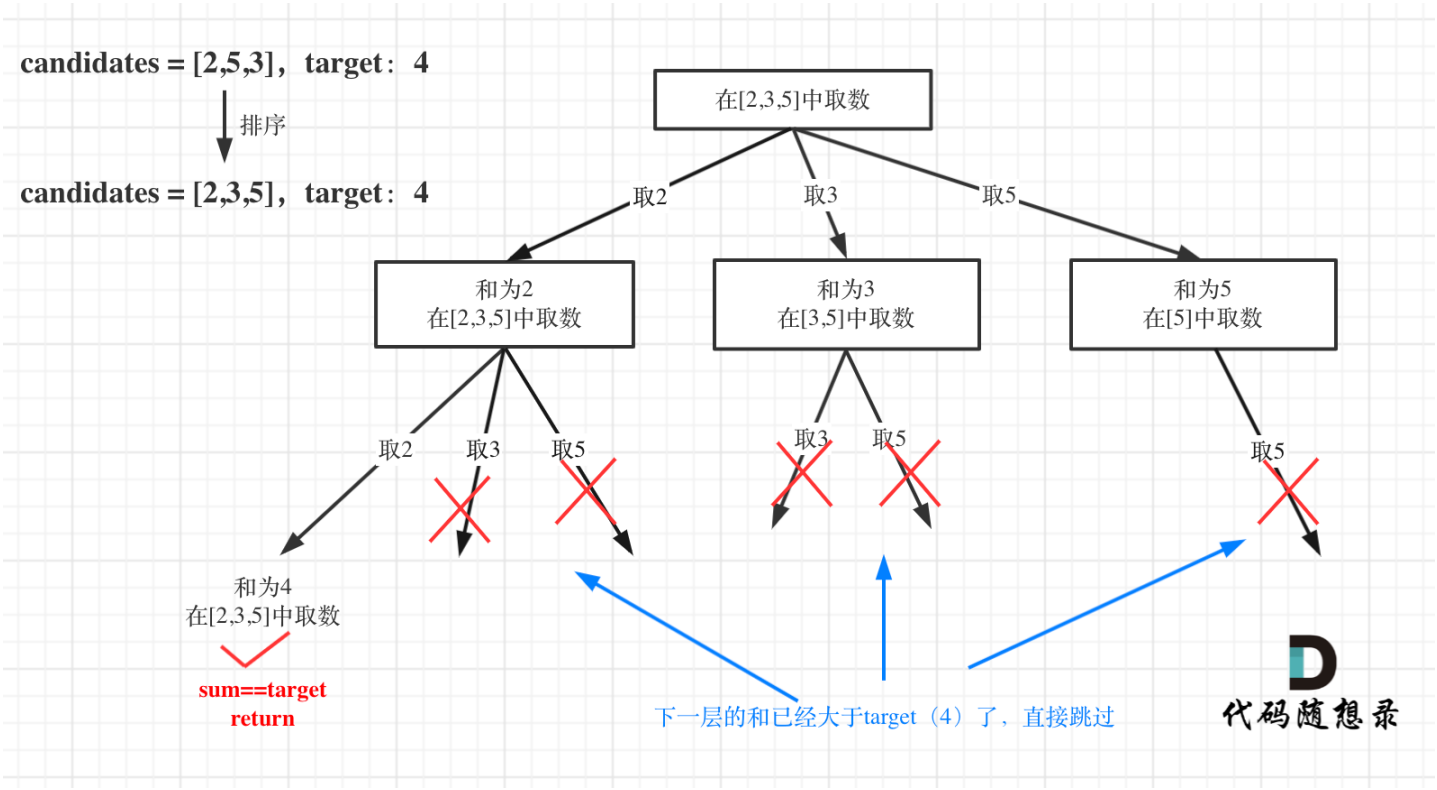
本题还需要startIndex来控制for循环的起始位置，对于组合问题，什么时候需要startIndex呢?

我举过例子，如果是一个集合来求组合的话，就需要startIndex，例如：77.组合，216.组合总和III。

如果是多个集合取组合，各个集合之间相互不影响，那么就不用startIndex，例如：17.电话号码的字母组合

注意以上我只是说求组合的情况，如果是排列问题，又是另一套分析的套路，后面我再讲解排列的时候就重点介绍。

Pruning



在求和问题中，排序之后加剪枝是常见的套路！

```python
class Solution:
    def combinationSum(self, candidates: List[int], target: int) -> List[List[int]]:
        path = []
        res = []
        def backtracking(start_index,sum_val):
            if sum_val == target:
                res.append(path[:])
                return
            if sum_val > target:
                return

            candidates.sort()
            for i in range(start_index,len(candidates)):
                if sum_val + candidates[i] > target:
                    return
                sum_val += candidates[i]
                path.append(candidates[i])
                backtracking(i,sum_val)
                sum_val -= candidates[i]
                path.pop()
        backtracking(0,0)
        return res
```

40. Combination Sum II

去重！

都知道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。**

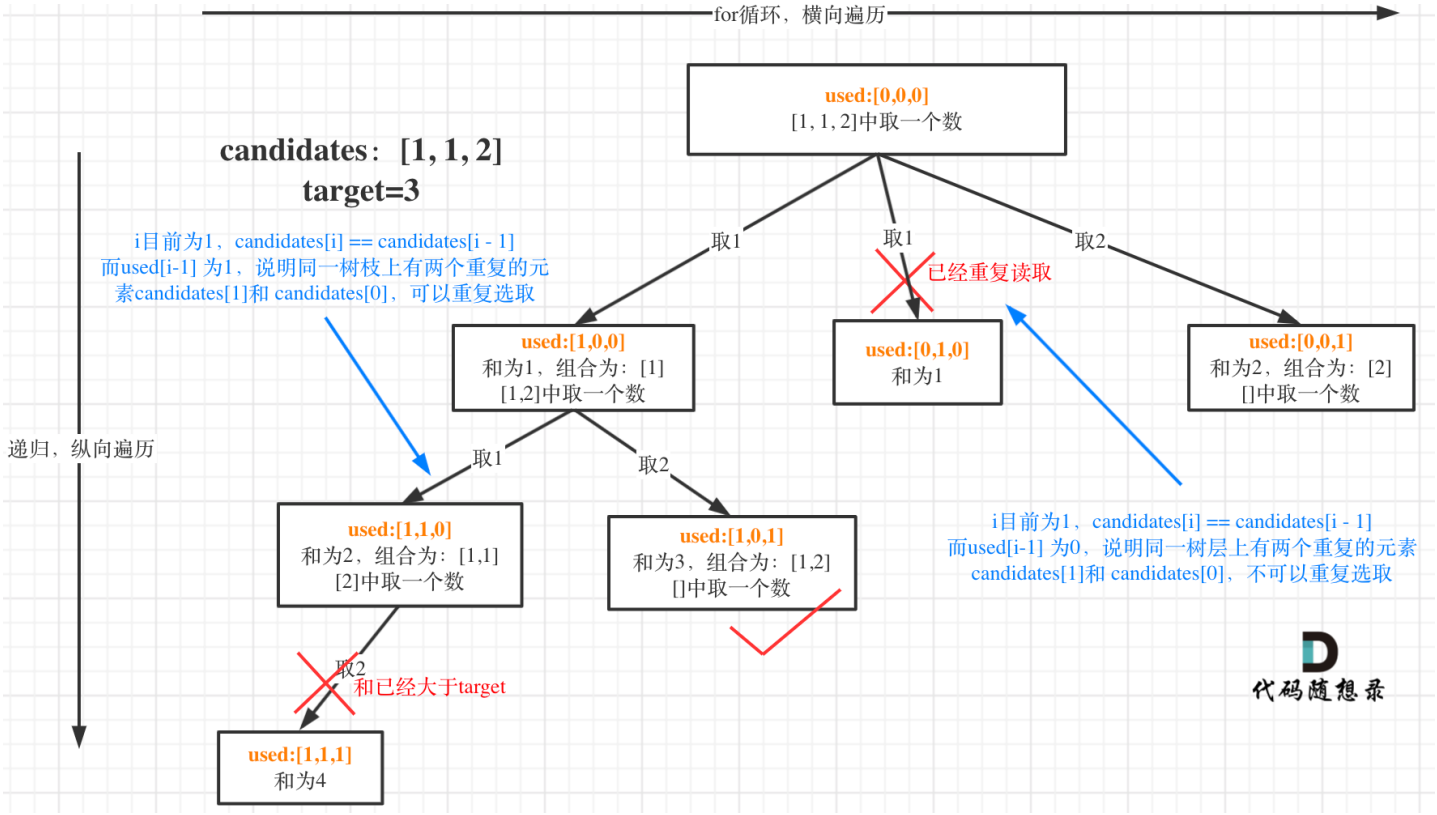
那么问题来了，我们是要同一树层上使用过，还是同一树枝上使用过呢?

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！




```python
class Solution:
    def combinationSum2(self, candidates: List[int], target: int) -> List[List[int]]:
        path = []
        res = []
        used = [False]*len(candidates)
        def backtracking(start_index,sum_val):
            if sum_val == target:
                res.append(path[:])
                return
            if sum_val > target:
                return

            candidates.sort()
            for i in range(start_index,len(candidates)):
                if sum_val + candidates[i] > target:
                    return

                #去重 used[i-1]==False 说明该元素没有被使用，后果就是跟当前元素出现在同一树层上面造成重复，所以这种情况时直接退出循环。
                if i > 0 and candidates[i]==candidates[i-1] and used[i-1]==False:
                    continue

                sum_val += candidates[i]
                used[i] = True

                path.append(candidates[i])
```

```
                path.append(candidates[i])
                backtracking(i+1,sum_val)
                sum_val -= candidates[i]
                used[i] = False
                path.pop()
        backtracking(0,0)
        return res
```

分割(segmentation)

131.Palindrome Partitioning

难点：

1. 切割问题中递归如何终止
2. **在递归循环中如何截取子串: 截取的时候也是左闭右开[) ！ 所以右边是i+1**

```
class Solution:
    def partition(self, s: str) -> List[List[str]]:
        path = []
        res = []
        def backtracking(startindex):
            #递归中止条件
            if startindex >= len(s):
                res.append(path[:])
                return
            for i in range(startindex,len(s)):
                tmp = s[startindex:i+1] #substring 左闭右开!
                if tmp == tmp[::-1]: #是回文
                    path.append(tmp)
                    backtracking(i+1)
                    path.pop()
                else: #不是回文
                    continue
        backtracking(0)
        return res
```

93. Restore IP Addresses

难点：

1. 是否符合条件的判断。要考虑到['0','0','0','0'].
2. 递归的终止条件：已经有了三个，是否要加第四个？和上一题不同，上一题可以无限分，但是这题限制了只能分4个。
3. 回溯时候遍历要从i=1开始，因为[].

```python
class Solution:
    def restoreIpAddresses(self, s: str) -> List[str]:
        path = []
        res = []

        if len(s)<4 or len(s)>12:
            return

        def isValid(string):
            if string == "0":
                return True

            if string[0] == "0":
                return False

            if 0 < int(string) <= 255:
                return True

            return False


        def backtrack(string,count):
            # 已经有三个字段了，检查最后一个字段是否合法，如果合法可以加入答案
            if count == 3:
                if isValid(string):
                    path.append(string)
                    res.append(".".join(path))
                    path.pop()
                return

            # 如果合法，把string[:i]部分作为一段加入path，在string[i:]上进行回溯
            for i in range(1, len(string)):
                if isValid(string[:i]):
                    path.append(string[:i])
                    backtrack(string[i:], count + 1)
                    path.pop()

        backtrack(s, 0)
        return res
```

子集(subsets)

子集(subsets)

78. Subsets

难点：

1. 递归终止条件和之前不同，这里是每个都能放入，直到startindex到了最后。

   子集是收集树形结构中树的所有节点的结果。

   而组合问题、分割问题是收集树形结构中叶子节点的结果。

```
class Solution:
    def subsets(self, nums: List[int]) -> List[List[int]]:
        #递归终止条件是什么?
        path = []
        res = []
        def backtracking(startindex):
            #递归终止条件和之前不同
            res.append(path[:])
            if startindex == len(nums):
                return

            for i in range(startindex,len(nums)):
                path.append(nums[i])
                backtracking(i+1)
                path.pop()
        backtracking(0)
        return res
```

90. Subsets II

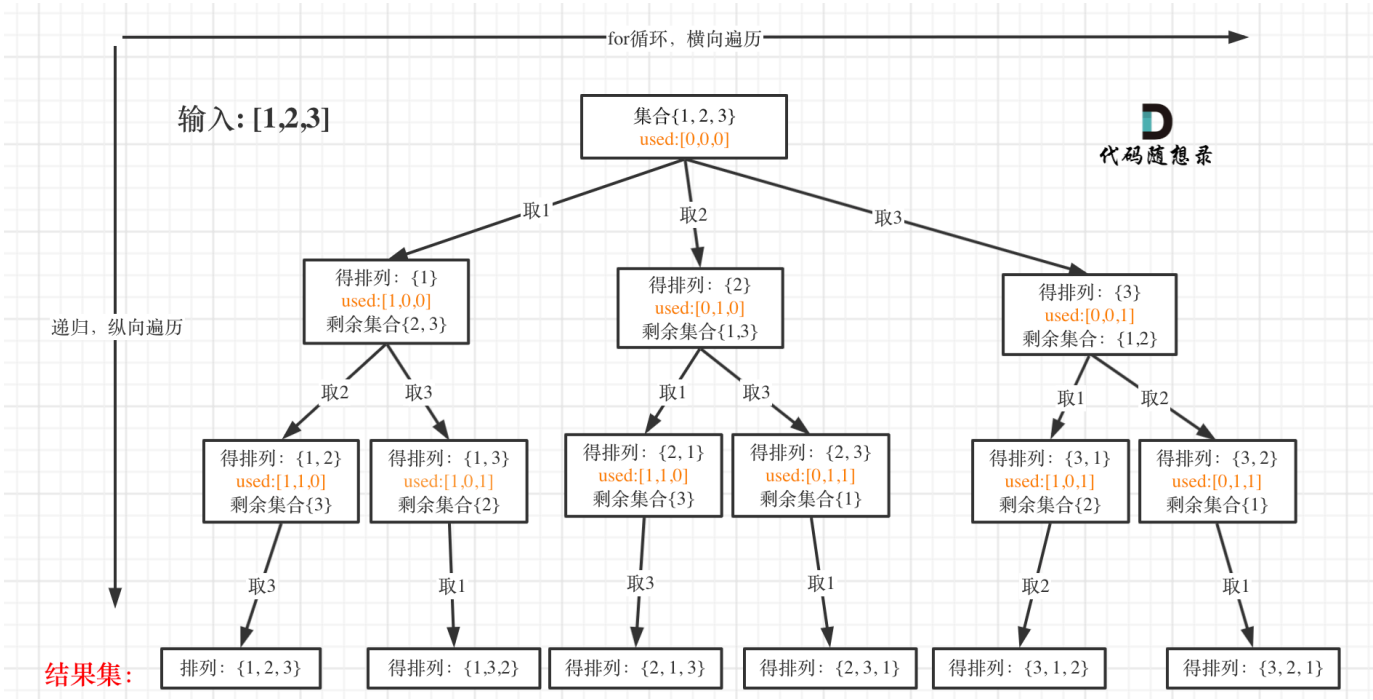
又涉及到去重 用之前去重的套路就行

```
class Solution:
    def subsetsWithDup(self, nums: List[int]) -> List[List[int]]:
        #又涉及到去重 用之前去重的套路就行
        path = []
        res = []
        used = [False]*len(nums)
        def backtracking(startindex):
            res.append(path[:])
            if startindex == len(nums):
                return

            nums.sort()
            for i in range(startindex,len(nums)):

                if i>0 and nums[i]==nums[i-1] and used[i-1]==False:
```

```
                continue
            path.append(nums[i])
            used[i] = True
            backtracking(i+1)
            path.pop()
            used[i] = False
        backtracking(0)
        return res
```

排列(Permutations)

1. **排列问题不需要用到startindex，for循环从0开始遍历。原因是每层都要从头开始搜索，而组合问题是每层从当前结点开始往后搜索，这是组合问题和排列问题的最大区别！**
2. **需要用到used数组去重**



46. Permutations

```
class Solution:
    def permute(self, nums: List[int]) -> List[List[int]]:
        path = []
        res = []
        used = [False]*len(nums)
        def backtracking(nums):
            if len(path)==len(nums):
                res.append(path[:])
                return
            for i in range(0,len(nums)):
```

```
                # 在path中遇到使用过的元素则退出
                if used[i] == True:
                    continue

                used[i]=True
                path.append(nums[i])
                backtracking(nums)
                path.pop()
                used[i]=False
        backtracking(nums)
        return res
```

break和continue都是用来控制循环结构的，主要是停止循环。

1.break

有时候我们想在某种条件出现的时候终止循环而不是等到循环条件为false才终止。

这是我们可以使用break来完成。break用于完全结束一个循环，跳出循环体执行循环后面的语句。

2.continue

continue和break有点类似，区别在于continue只是终止本次循环，接着还执行后面的循环，break则完全终止循环。

可以理解为continue是跳过当次循环中剩下的语句，执行下一次循环。

47.Permutations II

同90题，增加去重代码模版即可。

```
class Solution:
    def permuteUnique(self, nums: List[int]) -> List[List[int]]:
        # 需要去重
        path = []
        res = []
        used = [False]*len(nums)
        def backtracking(nums):
            if len(path)==len(nums):
                res.append(path[:])
                return

            nums.sort()
            for i in range(0,len(nums)):
                # 在path中遇到使用过的元素则退出
                if used[i] == True:
                    continue
```

```
                if i > 0 and nums[i]==nums[i-1] and used[i-1]==False:
                    continue

                used[i]=True
                path.append(nums[i])
                backtracking(nums)
                path.pop()
                used[i]=False
        backtracking(nums)
        return res
```

棋盘问题(chess board)

51. N-Queens

难点（新点）：

1. 棋盘初始化带'.'
2. is_valid函数的row和col是当前值，此函数的意义是判断当前点board[row][col] 是否合法.
3. 判断是否合法只需要判断是否在一列和斜线位置（两个：45度和135度），且只需从当前位置往上判断。因为回溯算法的特性，只会从上往下走，每一步都肯定符合条件。**所以 回溯函数的参数只有row，col通过for循环来控制。**
4. is_valid中的边界.

```
class Solution:
    def solveNQueens(self, n: int) -> List[List[str]]:
        board = [['.'] * n for _ in range(n)]
        res = []

        # 查看是否可以在board[row][col]的位置放置皇后
        def is_valid(board, row, col):
            # 查看上方是否有Q
            for i in range(len(board)):
                if board[i][col]=='Q':
                    return False
            # 查看45度角是否有Q（左上角）
            i = row - 1
            j = col - 1
            while i>=0 and j>=0:
                if board[i][j]=='Q':
                    return False
                i -= 1
                j -= 1
            #查看135度角是否有Q（右上角）
            i = row - 1
            j = col + 1
```

```
            while i>=0 and j<len(board):
                if board[i][j]=='Q':
                    return False
                i -= 1
                j += 1
            return True

        #回溯
        def backtracking(row):
            if row == n: #遍历到最后一行，证明有解
                tmp = [''.join(i) for i in board]
                res.append(tmp)
                return
            # 添加Q
            for i in range(n):
                #查看当前位置是否合法
                if not is_valid(board,row,i):
                    continue
                board[row][i]='Q'
                backtracking(row+1)
                board[row][i]='.'
        backtracking(0)
        return res
```

37.Sudoku Solver

其他(others)

79. Word Search

重点：如何遍历上下左右，其实是一个dfs+回溯

```
class Solution(object):

    # 定义上下左右四个行走方向
    directs = [(0, 1), (0, -1), (1, 0), (-1, 0)]

    def exist(self, board, word):
        """
        :type board: List[List[str]]
        :type word: str
        :rtype: bool
        """

        m = len(board)
```

```
        if m == 0:
            return False
        n = len(board[0])
        mark = [[0 for _ in range(n)] for _ in range(m)]

        for i in range(len(board)):
            for j in range(len(board[0])):
                if board[i][j] == word[0]:
                    # 将该元素标记为已使用
                    mark[i][j] = 1
                    if self.backtrack(i, j, mark, board, word[1:]) == True:
                        return True
                    else:
                        # 回溯
                        mark[i][j] = 0
        return False


    def backtrack(self, i, j, mark, board, word):
        if len(word) == 0:
            return True

        #回溯模版
        #如何遍历上下左右的做法需要学习
        for direct in self.directs:
            cur_i = i + direct[0]
            cur_j = j + direct[1]

            if cur_i >= 0 and cur_i < len(board) and cur_j >= 0 and cur_j < len(board[0])
and board[cur_i][cur_j] == word[0]:
                # 如果是已经使用过的元素，忽略
                if mark[cur_i][cur_j] == 1:
                    continue
                # 将该元素标记为已使用
                mark[cur_i][cur_j] = 1
                if self.backtrack(cur_i, cur_j, mark, board, word[1:]) == True:
                    return True
                else:
                    # 回溯
                    mark[cur_i][cur_j] = 0
        return False
```

329.Longest Increasing Path in a Matrix

上一题的位置技巧可以运用到这一题，不过这题要求最长，本质是dfs（记忆化搜索）

```
class Solution:
    def longestIncreasingPath(self, matrix: List[List[int]]) -> int:
        m, n = len(matrix), len(matrix[0])
        flag = [[-1] * n for _ in range(m)] #存储从（i，j）出发的最长递归路径

        def dfs(i, j):
            if flag[i][j] != -1: # 记忆化搜索，避免重复的计算
                return flag[i][j]
            else:
                d = 1
                for (x, y) in [[-1, 0], [1, 0], [0, 1], [0, -1]]:
                    x, y = i + x, j + y
                    #在里面判断
                    if 0 <= x < m and 0 <= y < n and matrix[x][y] > matrix[i][j]:
                        d = max(d, dfs(x, y) + 1) # 取四个邻接点的最长
                flag[i][j] = d
                return d

        res = 0
        for i in range(m): # 遍历矩阵计算最长路径
            for j in range(n):
                if flag[i][j] == -1:
                    res = max(res, dfs(i, j))
        return res
```

140. Word Break II

282. Expression Add Operators

60. Permutation Sequence

31. Next Permutation