In [ ]:

```python
# 77 组合
def combine(n, k):
    # [1,2,3,...,n]取k个数
    res = []
    path = []
    def backtrack(n,k,startIndex):
        # startIndex控制了输入（且只往后取，和全排列有所不同）= pop
        # 终止条件
        if len(path) == k:
            res.append(path[:])
            return
        for i in range(startIndex,n-(k-len(path))+2):  # 层的可选项
            # n-(k-len(path))+2 举例子，n=4,k=3,path=[],最多是[2,3,4]，取2， 而4-(3-0)+2=3,但是
            # n是从1开始取，取相应的值时必须+1
            path.append(i)  # 选择
            backtrack(n,k,i+1) # n、k、i+1 共同控制了层的大小
            path.pop() # 回退,但是可选项不需要补全，因为不能往前面取了
    backtrack(n,k,1)
    return res
```

此处n相当于一个可选列表，

n+startIndex控制了选择列表

递归含义：

i+startIndex开始，再判断长度是否为k

for 选择列表：

选一个

回溯

退选

```python
# 216 组合求和III，求和问题I
def combinationSum3(k,n) :
    # k 个数，和为n, 取值[1,2,3,...,9]
    res = []
    path = []
    def backtrack(targetSum,Sum,k,startIndex):
        if Sum > n : return
        if len(path) == k and targetSum == Sum:
            return res.append(path[:])
        for i in range(startIndex,9-(k-len(path))+2):
            Sum += i
            path.append(i)
            backtrack(targetSum,Sum,k,i+1)
            Sum -= i
            path.pop()
    backtrack(n,0,k,1)
    return res
```

限制个数k，字从n中取

长为k，和为target.

backtrack(控制结束参数,

选择列表参数)

选择列表

```python
# 17 电话号码的字母组合
def letterCombinations(digits):
    result = []  # 可全局调用
    self.s = ''  # 不可全局调用
    letterMap = ["","","abc","def","ghi","jkl","mno","pqrs","tuv","wxyz"]
    if len(digits) == 0: return []
    def backtrack(digits,index):
        # index决定在哪一层,表示取digits的第index位开始取
        if index == len(digits): return result.append(self.s)
        digit = int(digits[index])
        letters = letterMap[digit]
        for i in range(len(letters)):
            # 排列问题不需要修补letters
            self.s += letters[i]
            backtrack(digits,index+1)
            self.s  = self.s[:-1]
    backtrack(digits,0)
    return result
```

→ 数字

digits: 数组

index: 第n个数

取字母

→ 层

做选择

```

# 46 全排列(没有重复数字)
def permute(self, nums: List[int]) -> List[List[int]]:
    if nums == []:return []
    results = []
    res = []
    def backtrack(num):
        if num == []:
            results.append(res[:])  # 非常重要
        for i in range(len(num)):
            c = num.pop(i)
            res.append(c)
            backtrack(num)
            num.insert(i,c)
            res.pop()
    backtrack(nums)
    return results

# 47 全排II(有重复数字)
def permuteUnique(nums):
    if nums == []: return []
    results = []
    res = []
    def backtrack(num):
        if num == [] and res[:] not in results: # 区别在去重
    return  results.append(res[:])   # 深拷贝，需要复制
        for i in range(len(num)):
            c = num.pop(i)
            res.append(c)
            backtrack(num)
            num.insert(i,c)
            res.pop()
    backtrack(nums)
    return results

# 字符排列
def Permutation(ss):
    if ss == '': return ''
    result = []
    res = []
    def backtrack(string):
        string = list(string)
        if string == []:
            result.append(''.join(res)) # 已经不可修改了
        for i in range(len(string)):
            c = string.pop(i)
            if c:
                res.append(c)
                backtrack(string)
                string.insert(i,c)
                res.pop()
    backtrack(ss)
    return sorted(list(set(result)))

                                          无重复，但可重复使用自己。
# 39 组合总和II  无限个数
def combinationSum(candidates, target):
    res = []
    path = []
                                                  nums
    def backtrack(target,Sum,startIndex,candidates):
        if Sum == target: return res.append(path[:])
        for i in range(startIndex,len(candidates)):
            if Sum + candidates[i]> target: return  # 经过排序后可以用，直接跳出
            Sum += candidates[i]
            path.append(candidates[i])
            backtrack(target,Sum,i,candidates) # i表示能 重复使用自己
            Sum -= candidates[i]
            path.pop()
    candidates = sorted(candidates)
    backtrack(target,0,0,candidates)
    return res

# 40 组合求和III
```

无限个数

```
def combinationSum2(candidates, target)
    res = []
    path  = []
    def backtrack(target,Sum,startIndex,candidates):
        if Sum == target: return res.append(path[:])
        for i in range(startIndex,len(candidates)):
            if Sum + candidates[i] > target: return
            if i > startIndex and candidates[i] == candidates[i-1]: continue  # 加了这样行去
            Sum += candidates[i]
            path.append(candidates[i])
            backtrack(target,Sum,i+1,candidates) # i+1 表示当前字符能否重复使用自己，i+1表示不能
            Sum -= candidates[i]
            path.pop()
    candidates = sorted(candidates)
    backtrack(target,0,0,candidates)
    return res
```

[ ]有重复
不能重复使用自己

```
# 131 分割回文字符
def partition(s):
    res = []
    path = []
    def backtrack(s,startIndex):
        if startIndex == len(s) : return res.append(path[:])
        for i in range(startIndex,len(s)):
            p = s[startIndex:i+1]
            if p == p[::-1]:   # 是回文字符，才递归，不是的直接扔掉
                path.append(p)
                backtrack(s,i+1)
                path.pop()
    backtrack(s,0)
    return res
```

选择

判断

i+1 startIndex

是之后再继续切分

```
# 93 复原IP
 def restoreIpAddresses(s):
    res = []
    path = []
    def backtrack(s,level):
        if level == 5 and s == '' and '.'.join(path) not in res:
            return res.append('.'.join(path))
        if level ==5 or s == '': return
        for i in range(1,4):
            x = s[:i]
            if int(x) < 256 and (x =='0' or x[0] !='0'):
                path.append(x)
                backtrack(s[i:],level+1)
                path.pop()
    backtrack(s,1)
    return res
```

→ 个数

取，选择

判断

```
# 78 子集I
def subsets(nums):
    res = []
    path = []
    def backtrack(startIndex,nums):
        res.append(path[:])  # [:] ,第一个为空, 加入, 来了先加入
        if startIndex >= len(nums): return
        for i in range(startIndex,len(nums)):
            path.append(nums[i])
            backtrack(i+1,nums)
            path.pop()
    backtrack(0,nums)
    return res

# 90 子集II
def subsetsWithDup(nums):
    res = []
    path = []
    def backtrack(startIndex,nums):
        res.append(path[:])
```

{}, {1}, {1,2}

不用递k的

```
        if startIndex == len(nums): return
        for i in range(startIndex,len(nums)):
            if i > startIndex and nums[i] == nums[i-1]: continue #我们要对同一树层使用过的元
            path.append(nums[i])
            backtrack(i+1,nums)
            path.pop()
    nums = sorted(nums)
    backtrack(0,nums)
    return res

# 491 递增子序列
```

从原数组中取

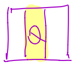
```
def findSubsequences(nums):
    # 给的例子是个坑，不能对序列进行排序，找这个顺序下的最长子序列
    res = []
    path = []
    def backtrack(startIndex,nums):
        repeat = [] # 同一层下不重复出现，树枝中可重复
        if len(path) > 1: res.append(path[:])
        for i in range(startIndex,len(nums)):
            if nums[i] in repeat: continue  # 往后走
            if len(path) > 0 and nums[i] < path[-1]: continue  # 往后走
            repeat.append(nums[i])
            path.append(nums[i])
            backtrack(i+1,nums)
            path.pop()
    backtrack(0,nums)
    return res

# 51 N皇后问题
```

一行一列填充

```
def solveNQueens(n):
    if not n: return []
    board = [['.']*n  for i in range(n)]  # 构建棋盘
    res = []
    def isVaild(board,row,col):
```

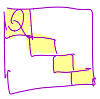
定义好游戏规则"

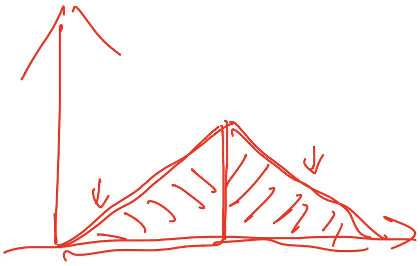
```
        # 简单
        # 判断同一列是否冲突
        for i in range(len(board)):
            if board[i][col] == 'Q': return False
        # 判断同一左斜线冲突
        i = row - 1
        j = col - 1
```



找3种
不符条件

```
        while i >=0 and j >= 0:
            if board[i][j] == 'Q': return False
            i -= 1
            j -= 1
        # 判断同一右斜线冲突
        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
    def backtrack(board,row,n):
        if row == ~~len(board)~~ n:  res.append([''.join(i) for i in board]) # 特殊格式
        for col in range(n):
            if not isVaild(board,row,col): continue
  for {     board[row][col] = 'Q'
            backtrack(board,row+1,n)
            board[row][col] = '.'
    backtrack(board,0,n)
    return res

# 数独判断 37
def solveSudoku(board):
    """
    Do not return anything, modify board in-place instead.只做方法修改即可
    """
    def isValid(row,col,val,board):   游戏规则
        for i in range(9): # 判断同行
            if board[row][i] == str(val): return False
        for i in range(9): # 判断 同列
            if board[i][col] == str(val): return False
        startRow = (row // 3) *3
        startCol = (col // 3) *3
        for i in range(startRow,startRow+3): # 判断同个9*9
            for j in range(startCol,startCol+3):
                if board[i][j] == str(val):return False
        return True

    def backtrack(board):
        # 直接修改board
        for i in range(len(board)):
            for j in range(len(board[0])):
                if board[i][j] != '.': continue
                for num in range(1,10):  选数
                    if isValid(i,j,num,board):
                        board[i][j] = str(num)
                        if backtrack(board) : return True
                        board[i][j] = '.' # 不填num,换一个数
                return False
        return True  →走完
    backtrack(board)

# 22 括号生成
def generateParenthesis(n):
    if n == 0: return []
    res = []
    self.path = ''  # 字符串只能这样定义全局变量
    def backtrack(left,right):
        # left 左边括号数,  right 右边括号数
        if left < 0 or right < 0: return
```





```
        if left > right: return
        if left == 0 and right == 0: return res.append(self.path)
        self.path += '('
        backtrack(left-1,right)
        self.path = self.path[:-1]
        self.path += ')'
        backtrack(left,right-1)
        self.path = self.path[:-1]
    backtrack(n,n)
    return res
```

for

```
for i in '()':
    self.path += i
    if i == '(': backtrack(left-1, right)
    else: backtrack(left, right-1)
    self.path = self.path[:-1]
```