回溯算法

回溯法标准框架:

```
def backtrack(path, selected):
    if 满足停止条件:
        res.append(path)
    for 选择 in 选择列表:
        做出选择
        递归执行backtrack
        撤销选择
```

- 组合问题: N个数里面按一定规则找出k个数的集合
- 排列问题: N个数按一定规则全排列，有几种排列方式
- 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
- 棋盘问题: N皇后，解数独等等

组合是不强调元素顺序的，排列是强调元素顺序

组合无序，排列有序

```
if not pth or nums[i]>=pth[-1]: # 需满足递增
    pth.append(nums[i])         # 选nums[i]
    bt(i+1, pth)
    pth.pop()                   # 回溯复原
    # bt(i+1, pth+[nums[i]])    # 与以上三行等价
```

组合

```
输入: n = 4, k = 2
输出:
[
  [2,4],
  [3,4],
  [2,3],
  [1,2],
  [1,3],
  [1,4],
]
```

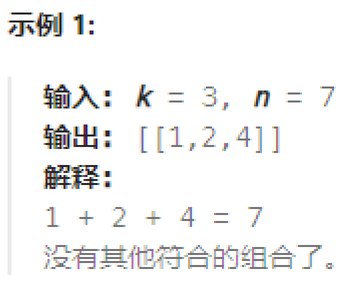
```
class Solution:
```

```python
    def combine(self, n: int, k: int) -> List[List[int]]:
        def backtrack(start, path):
            if len(path) == k:
                ans.append(path[:]) #别漏写[:]
                return
            #n+1可以优化n-(k-len(pth))+2
            for i in range(start, n+1):
                path.append(i)
                backtrack(i+1, path) #是path，别写错
                path.pop()

        ans = []
        backtrack(1, [])
        return ans
```

> *ans.append(path[:]) 中使用 path[:] 原因:*
>
> *path[:] 会创建一个 path 的副本。如果直接使用 ans.append(path)，则 ans 列表中的每个元素都会指向同一个 path 列表对象，而不是其副本。这意味着在后续的迭代过程中，当我们改变 path 的内容时，ans 列表中的元素也会随之改变，这可能不是我们想要的行为。*

组合总和 III



示例 1:

> **输入:** ***k*** = 3, ***n*** = 7
> **输出:** [[1,2,4]]
> **解释:**
> 1 + 2 + 4 = 7
> 没有其他符合的组合了。

```python
class Solution:
    def combinationSum3(self, k: int, n: int) -> List[List[int]]:
        def bt(tot, start, pth):
            if tot>n: # 剪枝
                return

            if len(pth)==k and tot == n:
                ans.append(pth[:])
                return

            for i in range(start, 9-(k-len(pth))+2): # 剪枝
                pth.append(i)
                tot+=i
                bt(tot, i+1, pth)
                tot-=i     # 回溯
                pth.pop()  # 回溯

        ans=[]
        bt(0,1,[])
        return ans
```

电话号码的字母组合

示例 1:

> **输入:** digits = "23"
> **输出:** ["ad","ae","af","bd","be","bf","cd","ce","cf"]

```python
class Solution:
    def letterCombinations(self, digits: str) -> List[str]:
        if not digits:return []

        mp = {
            "2": "abc",
            "3": "def",
            "4": "ghi",
            "5": "jkl",
            "6": "mno",
            "7": "pqrs",
            "8": "tuv",
            "9": "wxyz",
        }

        def bt(i,pth):
            if i == len(digits):
                ans.append("".join(pth[:]))
            else:
                d = digits[i]
                for s in mp[d]:
                    pth.append(s)
                    bt(i+1, pth)
                    pth.pop()

        ans = []
        bt(0, [])
        return ans
```

组合总和 【直接看下一题】[解题也可同下]

示例 1：

> **输入：** candidates = `[2,3,6,7]`, target = `7`
> **输出：** [[2,2,3],[7]]
> **解释：**
> 2 和 3 可以形成一组候选，2 + 2 + 3 = 7 。注意 2 可以使用多次。
> 7 也是一个候选， 7 = 7 。
> 仅有这两种组合。

> *无重复元素 的整数数组 candidates*
>
> *2 <= candidates[i] <= 40*

```python
class Solution:
    def combinationSum(self, candidates: List[int], target: int) -> List[List[int]]:
        def bt(target, start, pth):
            if target==0:
                ans.append(pth[:])
                return

            for i in range(start, len(candidates)):
                target -= candidates[i]
                if target < 0:
                    break
                pth.append(candidates[i])
                bt(target, i, pth) #重复使用元素，仍使用i
                target += candidates[i]
                pth.pop()

        candidates.sort()
        ans = []
        bt(target,0,[])
        return ans
```

组合总和 II

```python
class Solution:
    def combinationSum2(self, candidates: List[int], target: int) -> List[List[int]]:
        def bt(target, start, pth):
            if target == 0:
                ans.append(pth[:])
                return

            for i in range(start, len(candidates)):
                # 跳过同一树层使用过的元素
                if i>start and candidates[i] == candidates[i-1]:
                    continue
                target -= candidates[i]
                if target < 0:
                    break
                pth.append(candidates[i])
                bt(target, i+1, pth)
                target += candidates[i]
                pth.pop()

        candidates.sort()
        ans = []
        bt(target, 0, [])
        return ans
```

分割回文串

> **输入：** s = "aab"
> **输出：** [["a","a","b"],["aa","b"]]

```python
class Solution:
    def partition(self, s: str) -> List[List[str]]:
        def bt(start, pth):
            if start == len(s):
                ans.append(pth[:])
                return

            for i in range(start, len(s)):
                if s[start: i+1] == s[start: i+1][::-1]:
                    pth.append(s[start:i+1])
                    bt(i+1, pth)
                    pth.pop()
        ans = []
        bt(0, [])
        return ans
```

复原 IP 地址

```python
class Solution:
    def restoreIpAddresses(self, s: str) -> List[str]:
        def bt(start, pth):
            if start == len(s) and len(pth) == 4:
                ans.append(".".join(pth))
                return

            for i in range(start, min(start+3, len(s))):
                if len(pth) > 4:  # 剪枝
                    break
                if self.is_valid(s, start, i):
                    pth.append(s[start:i+1])
                    bt(i+1, pth)
                    pth.pop()
        ans = []
        bt(0, [])
        return ans

    def is_valid(self, s, start, end):
        if s[start] == '0' and start != end:  # 0开头的数字不合法
            return False
        num = int(s[start:end+1])
        return 0 <= num <= 255
```

子集 [不包含重复元素]

```python
class Solution:
    def subsets(self, nums: List[int]) -> List[List[int]]:
        def bt(start, pth):
            ans.append(pth[:])

            for i in range(start, len(nums)):
                pth.append(nums[i])
                bt(i+1, pth)
                pth.pop()

        ans = []
        bt(0, [])
        return ans
```

子集 II [包含重复元素]

```python
class Solution:
    def subsetsWithDup(self, nums: List[int]) -> List[List[int]]:
        def bt(start, pth):
            ans.append(pth[:])

            for i in range(start, len(nums)):
                # 比上题多一个判断逻辑
                if i > start and nums[i] == nums[i-1]:
                    continue
                pth.append(nums[i])
                bt(i+1, pth)
                pth.pop()

        ans = []
        nums.sort()
        bt(0, [])
        return ans
```

非递减子序列

```python
class Solution:
    def findSubsequences(self, nums: List[int]) -> List[List[int]]:
        def bt(nums, pth):
            if len(pth) > 1:
                ans.append(pth[:])

            tmp = set()
            for i, n in enumerate(nums):
                if n in tmp:
                    continue
                if not pth or n >= pth[-1]:
                    tmp.add(n)
                    bt(nums[i+1:], pth+[n])
        ans = []
        bt(nums, [])
        return ans
```

思路2

```python
class Solution:
    def findSubsequences(self, nums: List[int]) -> List[List[int]]:
        def bt(i, pth):
            if i == len(nums):
                if len(pth) > 1:
                    ans.append(pth[:])
                return

            # 【1】选 nums[i]
            if not pth or nums[i]>=pth[-1]: # 需满足递增
                pth.append(nums[i])         # 选nums[i]
                bt(i+1, pth)
                pth.pop()                   # 回溯复原
                # bt(i+1, pth+[nums[i]])   # 与以上三行等价

            # 【2】不选 nums[i]:
            # 只有在nums[i]不等于前一项tmp[-1]的情况下才考虑不选nums[i]
            # 即若nums[i] == pth[-1]，则必考虑选nums[i]，不予执行不选nums[i]的情况
            if not pth or (pth and nums[i] != pth[-1]): # 避免重复
                bt(i+1, pth)

        ans = []
        bt(0, [])
        return ans
```

全排列

```python
class Solution:
    def permute(self, nums: List[int]) -> List[List[int]]:
        def bt(nums, pth):
            if not nums:
                ans.append(pth[:])
                return
            for i in range(len(nums)):
                bt(nums[:i] + nums[i+1:], pth + [nums[i]])

        ans = []
        bt(nums, [])
        return ans
```

全排列 II

```python
class Solution:
    def permuteUnique(self, nums: List[int]) -> List[List[int]]:
        def bt(nums, pth):
            if not nums:
                ans.append(pth[:])
                return

            tmp = set()
            for i in range(len(nums)):
                if nums[i] in tmp:
                    continue
                bt(nums[:i]+nums[i+1:], pth+[nums[i]])
                tmp.add(nums[i])

        ans = []
        bt(nums, [])
        return ans
```

重新安排行程

```python
class Solution:
    def findItinerary(self, tickets: List[List[str]]) -> List[str]:
        from collections import defaultdict
        mp = defaultdict(list)

        for f, t in tickets:
            mp[f] += [t]
        for f in mp:
            mp[f].sort()
        print(mp)

        def bt(f):
            while mp[f]:
                bt(mp[f].pop(0))#路径检索
            ans.insert(0, f)     #放在最前

        ans = []
        bt('JFK') #题目必须从JFK开始
        return ans
```

N 皇后

正对角就是 (i,j) 相加之和一样的

负对角就是 (i,j) 相减只差一样的

```
class Solution:
    def solveNQueens(self, n: int) -> List[List[str]]:
                              #列，     正对角，    负对角
        def bt(i=0, pth=[], col=[], z=set(), f=set()):
            if i == n: #行
                ans.append(pth)
                return

            for j in range(n):
                if j not in col \
                and i-j not in z \
                and i+j not in f:
                    bt(i+1,
                    pth+[s[:j]+'Q'+s[j+1:]],
                    col+[j],
                    z|{i-j},
                    f|{i+j}) #并集
        ans = []
        s = '.' * n
        bt()
        return ans
```

解数独

```python
class Solution:
    def solveSudoku(self, board: List[List[str]]) -> None:
        """
        Do not return anything, modify board in-place instead.
        """
        def bt(i, j):
            """i, j代表遍历到的行、列索引"""
            if i == 9:   # 遍历完最后一行后，结束
                return True

            if j == 9:   # 遍历完最后一列后，转去遍历下一行
                return bt(i+1, 0)

            if board[i][j] != '.':  # 有数字
                return bt(i, j+1)

            for n in range(1, 10):  # 填空
                n = str(n)
                if not self.check(board, i, j, n):
                    continue
                board[i][j] = n
                # 直接return是因为只需要一个可行解，而不需要所有可行解

                if bt(i, j+1):
                    return True
                board[i][j] = '.'  # 撤销选择
        bt(0, 0)

    def check(self, board, row, col, n):
        for i in range(9):
            if board[row][i] == n:
                return False
            if board[i][col] == n:
                return False
            r = (row//3)*3 + i // 3
            c = (col//3)*3 + i % 3
            if board[r][c] == n:
                return False
        return True
```