算法-回溯

回溯算法概述

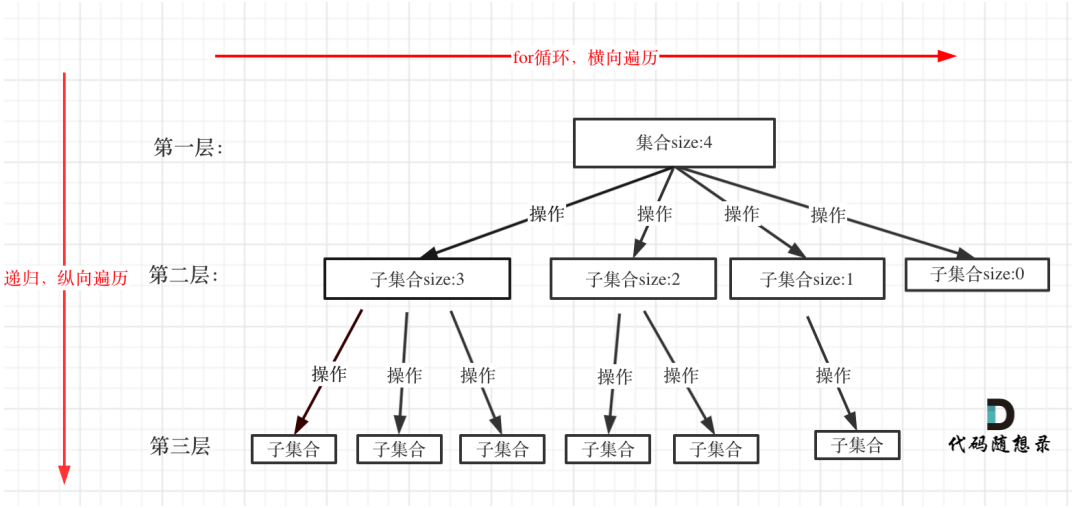
回溯法的本质和依赖仍旧是递归，优化的方式是进行剪枝。

解题过程中最好先想出树型结构。

回溯时一般遍历分为两个，一个是子树间的纵向遍历，一个是筛选时的横向遍历。

回溯法模板

回溯三部曲：

- 回溯函数模板返回值及其参数
  - 在回溯算法中，按个人习惯为函数起名字为backtracking，这个起名随意
  - 回溯算法中函数返回值一般为void。
  - 参数，因为回溯算法需要的参数可不像二叉树递归的时候那么容易一次性确定下来，所以一般是先写逻辑，然后需要什么参数，就填什么参数。
  - ```
    void backtracking(参数)
    ```
- 回溯函数终止条件
  - 般来说搜到叶子节点了，也就找到了满足条件的一条答案，把这个答案存放起来，并结束本层递归。
  - ```
    if (终止条件) {
        存放结果;
        return;
    }
    ```
- 回溯搜索的遍历过程
  - 回溯法一般是在集合中递归搜索，集合的大小构成了树的宽度，递归的深度构成的树的深度。
  - 

  - 图中，特意举例集合大小和孩子的数量是相等的！

- ```
  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
  }
  ```
- for循环就是遍历集合区间，可以理解一个节点有多少个孩子，这个for循环就执行多少次
- backtracking这里自己调用自己，实现递归。
- 可以从图中看出**for循环可以理解是横向遍历，backtracking（递归）就是纵向遍历**，这样就把这棵树全遍历完了，一般来说，搜索叶子节点就是找的其中一个结果了。

实验三-N皇后

```python
class Solution:
    def __init__(self):
        self.result =[]

    def isVaild(self,board,row, col):
            #判断同一列是否冲突
            for i in range(len(board)):
                if board[i][col] == 'Q':
                    return False
            # 判断左上角是否冲突
            i = row -1
            j = col -1
            while i>=0 and j>=0:
                if board[i][j] == 'Q':
                    return False
                i -= 1
                j -= 1
            # 判断右上角是否冲突
            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 backTracking(self,chessBoard,row,n):
        #如果走到最后一行，就说明已经找到了一个可行解
        if row == n:
            tempResult = []
            for temp in chessBoard:
                tempStr = "".join(temp)
                tempResult.append(tempStr)
            self.result.append(tempResult)
            return
        for col in range(n):
                if self.isVaild(chessBoard, row, col) == False:
                    continue
                chessBoard[row][col] = 'Q'
```
```

```
                self.backTracking(chessBoard, row+1, n)
                chessBoard[row][col] = '.'

    def solveNQueens(self, n: int) -> List[List[str]]:
        self.chessBoard = [['.'] * n for i in range(n)]
        self.backTracking(self.chessBoard,0,n)
        return self.result
```

实验三-单词拆分

140. 单词拆分 II

难度 困难 596

给定一个字符串 `s` 和一个字符串字典 `wordDict` ，在字符串 `s` 中增加空格来构建一个句子，使得句子中所有的单词都在词典中。**以任意顺序** 返回所有这些可能的句子。

注意：词典中的同一个单词可能在分段中被重复使用多次。

示例 1：

```
输入:s = "catsanddog", wordDict = ["cat","cats","and","sand","dog"]
输出:["cats and dog","cat sand dog"]
```

示例 2：

```
输入:s = "pineapplepenapple", wordDict = ["apple","pen","applepen","pine","pineapple"]
输出:["pine apple pen apple","pineapple pen apple","pine applepen apple"]
解释: 注意你可以重复使用字典中的单词。
```

示例 3：

```
输入:s = "catsandog", wordDict = ["cats","dog","sand","and","cat"]
输出:[]
```

```
class Solution:
    # 带备忘录的记忆化搜索
    def wordBreak(self, s: str, wordDict: List[str]) -> List[str]:
        result = []
        memo = [1] * (len(s)+1)
        wordDict = set(wordDict)

        def BackTracking(wordDict,temp,pos):
            num = len(res)                  # 回溯前先记下答案中有多少个元素
            if pos == len(s):
                res.append(" ".join(temp))
                return
            for i in range(pos,len(s)+1):
```

```
                if memo[i] and s[pos:i] in wordDict: # 添加备忘录的判断条件
                    temp.append(s[pos:i])
                    dfs(wordDict,temp,i)
                    temp.pop()
            # 答案中的元素没有增加，说明s[pos:]不能分割，修改备忘录
            memo[pos] = 1 if len(res) > num else 0

        BackTracking(wordDict,[],0)
        return result
```