Leetcode

2022年5月17日 15:18

1. 子集
   - 输入：nums = [1,2,3]
     输出：[[],[1],[2],[1,2],[3],[1,3],[2,3],[1,2,3]]
   - 无重复元素+不可重复选择
   - ```
     class Solution:
         def subsets(self, nums: List[int]) -> List[List[int]]:
             if not nums:
                 return []
             res = []
             cur = []
             def recur(start):
                 res.append(cur[:])
                 if len(cur) == len(nums):
                     return
                 for i in range(start, len(nums)):
                     cur.append(nums[i])
                     recur(i+1)
                     cur.pop()
             recur(0)
             return res
     ```
2. 子集II
   - 输入：nums = [1,2,2]
     输出：[[],[1],[1,2],[1,2,2],[2],[2,2]]
   - 有重复元素+不可重复选
   - ```
     class Solution:
         def subsetsWithDup(self, nums: List[int]) -> List[List[int]]:
             if not nums:
                 return []
             res = []
             cur = []
             nums.sort()
             def recur(start):
                 res.append(cur[:])
                 if len(cur) == len(nums):
                     return
                 for i in range(start, len(nums)):
                     if i > start and nums[i] == nums[i-1]:
                         continue
                     cur.append(nums[i])
                     recur(i+1)
                     cur.pop()
             recur(0)
             return res
     ```
3. 组合
   - 输入：n = 4, k = 2
     输出：
     [
       [2,4],
       [3,4],
       [2,3],
       [1,2],
       [1,3],
       [1,4],
     ]
   - 无重复元素+不可重复选
   - ```
     class Solution:
         def combine(self, n: int, k: int) -> List[List[int]]:
             if n < k:
                 return []
             res = []
             cur = []
             def recur(start):
                 if len(cur) == k:
                     res.append(cur[:])
                     return
                 for i in range(start, n+1):
                     cur.append(i)
                     recur(i+1)
                     cur.pop()
             recur(1)
             return res
     ```
4. 组合总和
   - 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+元素可以重复选择
   - ```
     class Solution:
         def combinationSum(self, candidates: List[int], target: int) -> List[List[int]]:
             res = []
             cur = []
             def recur(start, diff):
                 if diff < 0:
                     return
                 if diff == 0:
                     res.append(cur[:])
                     return
                 for i in range(start, len(candidates)):
                     cur.append(candidates[i])
                     recur(i, diff-candidates[i])
                     cur.pop()
             recur(0, target)
     ```

```
        return res
```

5. 组合总和II
   - 输入: candidates = [10,1,2,7,6,1,5], target = 8,
     输出:
     [
     [1,1,6],
     [1,2,5],
     [1,7],
     [2,6]
     ]
   - 有重复元素+不能重复选择
   - ```
     class Solution:
         def combinationSum2(self, candidates: List[int], target: int) -> List[List[int]]:
             res = []
             cur = []
             candidates.sort()
             def recur(start, diff):
                 if diff < 0:
                     return
                 if diff == 0:
                     res.append(cur[:])
                     return
                 for i in range(start, len(candidates)):
                     if i > start and candidates[i] == candidates[i-1]:
                         continue
                     cur.append(candidates[i])
                     recur(i+1, diff-candidates[i])
                     cur.pop()
             recur(0, target)
             return res
     ```

6. 组合总和III

   找出所有相加之和为 n 的 k 个数的组合，且满足下列条件：

   - 只使用数字1到9
   - 每个数字 **最多使用一次**

   返回 *所有可能的有效组合的列表*。该列表不能包含相同的组合两次，组合可以以任何顺序返回。

   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。

   - 无重复元素+不可重复选择+固定组合数目
   - ```
     class Solution:
         def combinationSum3(self, k: int, n: int) -> List[List[int]]:
             res = []
             cur = []
             def recur(start, diff, k):
                 if len(cur) == k and diff == 0:
                     res.append(cur[:])
                     return
                 if len(cur) == k or diff == 0:
                     return
                 for i in range(start, 10):
                     cur.append(i)
                     recur(i+1, diff-i, k)
                     cur.pop()
             recur(1, n, k)
             return res
     ```

7. 组合总和IV

   给你一个由 **不同** 整数组成的数组 nums ，和一个目标整数 target 。请你从 nums 中找出并返回总和为 target 的元素组合的个数。

   题目数据保证答案符合 32 位整数范围。

   示例 1:

   输入: nums = [1,2,3], target = 4
   输出: 7
   解释:
   所有可能的组合为:
   (1, 1, 1, 1)
   (1, 1, 2)
   (1, 2, 1)
   (1, 3)
   (2, 1, 1)
   (2, 2)
   (3, 1)
   请注意，顺序不同的序列被视作不同的组合。

   - 无重复元素+元素可重复使用
   - 回溯法超时
   - ```
     class Solution:
         def combinationSum4(self, nums: List[int], target: int) -> int:
             self.res = 0
             cur = []
             def recur(diff):
                 if diff < 0:
                     return
     ```
     ```

```
                if diff == 0:
                    self.res += 1
                    return
                for i in range(len(nums)):
                    recur(diff-nums[i])
            recur(target)
            return self.res
```

- 动态规划法+零钱兑换问题

```
class Solution:
    def combinationSum4(self, nums: List[int], target: int) -> int:
        dp = [0] * (target+1)
        dp[0] = 1
        for i in range(target+1):
            for item in nums:
                if i - item < 0:
                    continue
                dp[i] += dp[i-item]
        return dp[-1]
```

8. 全排列

```
输入：nums = [1,2,3]
输出：[[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]
```

- 无重复元素

```
class Solution:
    def permute(self, nums: List[int]) -> List[List[int]]:
        res = []
        cur = []
        vis = [0] * len(nums)
        def recur(x, l):
            if x == l:
                res.append(cur[:])
                return
            for i in range(len(nums)):
                if vis[i] == 1:
                    continue
                cur.append(nums[i])
                vis[i] = 1
                recur(x+1, l)
                vis[i] = 0
                cur.pop()
        recur(0, len(nums))
        return res
```

9. 全排列II

给定一个可包含重复数字的序列 `nums` ，***按任意顺序*** 返回所有不重复的全排列。

示例 1：

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

- 有重复元素

```
class Solution:
    def permuteUnique(self, nums: List[int]) -> List[List[int]]:
        if not nums:
            return []
        res = []
        cur = []
        nums.sort()
        vis = [0] * len(nums)
        def recur(x, l):
            if x == l:
                res.append(cur[:])
                return
            for i in range(len(nums)):
                if vis[i] == 1 or i > 0 and nums[i] == nums[i-1] and vis[i-1] == 0:
                    continue
                cur.append(nums[i])
                vis[i] = 1
                recur(x+1, l)
                vis[i] = 0
                cur.pop()
        recur(0, len(nums))
        return res
```

10. 岛屿数量

给你一个由 `'1'` （陆地）和 `'0'` （水）组成的的二维网格，请你计算网格中岛屿的数量。

岛屿总是被水包围，并且每座岛屿只能由水平方向和/或竖直方向上相邻的陆地连接形成。

此外，你可以假设该网格的四条边均被水包围。

示例 1：

```
输入：grid = [
  ["1","1","1","1","0"],
  ["1","1","0","1","0"],
  ["1","1","0","0","0"],
  ["0","0","0","0","0"]
]
输出：1
```

```
class Solution(object):
    def numIslands(self, grid):
        """
        :type grid: List[List[str]]
        :rtype: int
```

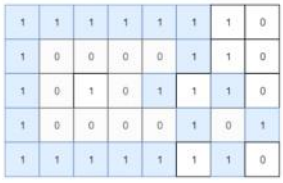
```
        """
        res = 0    # 记录总岛屿数
        for i in range(len(grid)):
            for j in range(len(grid[0])):
                if grid[i][j] == '1':
                    res += 1
                    self.fill_0(grid, i, j)
        return res
    # 将陆地（1）淹没为海水（0）
    def fill_0(self, grid, row, col):
        # 索引超出边界
        if row < 0 or col < 0 or row >= len(grid) or col >= len(grid[0]):
            return
        # grid[row][col]已经是海水
        if grid[row][col] == '0':
            return
        grid[row][col] = '0'
        self.fill_0(grid, row-1, col)
        self.fill_0(grid, row+1, col)
        self.fill_0(grid, row, col-1)
        self.fill_0(grid, row, col+1)
```

11. 统计封闭岛屿的数量

二维矩阵 grid 由 0 （土地）和 1 （水）组成。岛是由最大的4个方向连通的 0 组成的群，封闭岛是一个 完全 由1包围（左、上、右、下）的岛。

请返回 *封闭岛屿* 的数目。

示例 1：

- 

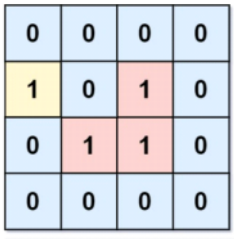
输入： grid = [[1,1,1,1,1,1,1,0],[1,0,0,0,0,1,1,0],[1,0,1,0,1,1,1,0],[1,0,0,0,0,1,0,1],[1,1,1,1,1,1,1,0]]
输出： 2
解释：
灰色区域的岛屿是封闭岛屿，因为这座岛屿完全被水域包围（即被 1 区域包

- 先将处于边界的岛屿淹没+统计岛屿数量
- ```
  class Solution(object):
      def closedIsland(self, grid):
          """
          :type grid: List[List[int]]
          :rtype: int
          """
          height = len(grid)
          width = len(grid[0])
          for i in range(height):
              self.fill_1(grid, i, 0)
              self.fill_1(grid, i, width-1)
          for j in range(width):
              self.fill_1(grid, 0, j)
              self.fill_1(grid, height-1, j)
          res = 0
          for m in range(1, height-1):
              for n in range(1, width-1):
                  if grid[m][n] == 0:
                      res += 1
                      self.fill_1(grid, m, n)
          return res

      def fill_1(self, grid, row, col):
          if row < 0 or col < 0 or row >= len(grid) or col >= len(grid[0]):
              return
          if grid[row][col] == 1:
              return
          grid[row][col] = 1
          self.fill_1(grid, row-1, col)
          self.fill_1(grid, row+1, col)
          self.fill_1(grid, row, col-1)
          self.fill_1(grid, row, col+1)
  ```

12. 飞地的数量

示例 1：

- 

输入： grid = [[0,0,0,0],[1,0,1,0],[0,1,1,0],[0,0,0,0]]
输出： 3
解释： 有三个 1 被 0 包围。一个 1 没有被包围，因为它在边界上。

- 相当于求封闭岛屿的面积+淹没边界岛屿+数封闭岛屿的格子数量
- ```
  class Solution:
      def numEnclaves(self, grid: List[List[int]]) -> int:
          for i in range(len(grid)):
              self.fill_0(i, 0, grid)
              self.fill_0(i, len(grid[0])-1, grid)
  ```

```
        for i in range(len(grid[0])):
            self.fill_0(0, i, grid)
            self.fill_0(len(grid)-1, i, grid)
        res = 0
        for i in range(len(grid)):
            for j in range(len(grid[0])):
                if grid[i][j] == 1:
                    res += 1
        return res
    def fill_0(self, row, col, grid):
        if row < 0 or col < 0 or row >= len(grid) or col >= len(grid[0]):
            return
        if grid[row][col] == 0:
            return
        grid[row][col] = 0
        self.fill_0(row-1, col, grid)
        self.fill_0(row+1, col, grid)
        self.fill_0(row, col-1, grid)
        self.fill_0(row, col+1, grid)
```

13. 岛屿最大的面积

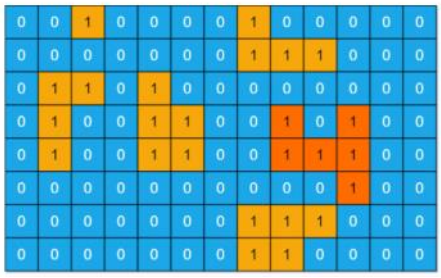
给你一个大小为 m x n 的二进制矩阵 grid 。

岛屿 是由一些相邻的 1 (代表土地) 构成的组合，这里的「相邻」要求两个 1 必须在 水平或者竖直的四个方向 上 相邻。你可以假设 grid 的四个边缘都被 0 (代表水) 包围着。

岛屿的面积是岛上值为 1 的单元格的数目。

计算并返回 grid 中最大的岛屿面积。如果没有岛屿，则返回面积为 0 。

示例 1：



- 在每一次淹没时记录当前岛屿的面积

```
class Solution:
    def maxAreaOfIsland(self, grid: List[List[int]]) -> int:
        res = 0
        for i in range(len(grid)):
            for j in range(len(grid[0])):
                if grid[i][j] == 1:
                    self.cur = 0
                    self.fill_0(grid, i, j)
                    res = max(res, self.cur)
        return res
    def fill_0(self, grid, row, col):
        if row < 0 or col < 0 or row >= len(grid) or col >= len(grid[0]):
            return
        if grid[row][col] == 0:
            return
        grid[row][col] = 0
        self.cur += 1
        self.fill_0(grid, row-1, col)
        self.fill_0(grid, row+1, col)
        self.fill_0(grid, row, col-1)
        self.fill_0(grid, row, col+1)
```

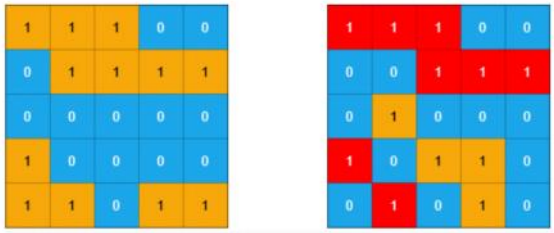
14. 统计子岛屿

给你两个 m x n 的二进制矩阵 grid1 和 grid2 ，它们只包含 0 （表示水域）和 1 （表示陆地）。一个 **岛屿** 是由 **四个方向** （水平或者竖直）上相邻的 1 组成的区域。任何矩阵以外的区域都视为水域。

如果 grid2 的一个岛屿，被 grid1 的一个岛屿 **完全** 包含，也就是说 grid2 中该岛屿的每一个格子都被 grid1 中同一个岛屿完全包含，那么我们称 grid2 中的这个岛屿为 **子岛屿** 。

请你返回 grid2 中 **子岛屿** 的 **数目** 。

示例 1：

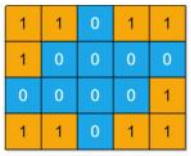


- 先将B中为陆地A中为海水的区域淹没+剩下的就是子岛+统计岛屿数量

```
class Solution:
    def countSubIslands(self, grid1: List[List[int]], grid2: List[List[int]]) -> int:
        # 如果岛屿 B中存在一片陆地，在岛屿 A的对应位置是海水，那么岛屿 B 就不是岛屿 A 的子岛
        for i in range(len(grid1)):
            for j in range(len(grid1[0])):
                if grid1[i][j] == 0 and grid2[i][j] == 1:
                    self.fill_0(grid2, i, j)
        res = 0
        for i in range(len(grid2)):
            for j in range(len(grid2[0])):
                if grid2[i][j] == 1:
                    res += 1
                    self.fill_0(grid2, i, j)
        return res
```

```
    def fill_0(self, grid, row, col):
        if (row < 0 or col < 0 or row >= len(grid) or col >= len(grid[0])):
            return
        if grid[row][col] == 0:
            return
        grid[row][col] = 0
        self.fill_0(grid, row - 1, col)
        self.fill_0(grid, row + 1, col)
        self.fill_0(grid, row, col - 1)
        self.fill_0(grid, row, col + 1)
```

15. 不同岛屿的数量



- 在淹没的时候记录每次的淹没方向形成一个字符串
- ```
  class Solition(object):
      def numDistinctIslands(self, grid):
          self.route_s = []
          for i in range(len(grid)):
              for j in range(len(grid[0])):
                  if grid[i][j] == 1:
                      self.route = []
                      self.fill_0(grid, i, j, 0)
                      self.route_s.append(str(self.route))
          return len(set(self.route_s))
      # 上下左右: 1,2,3,4
      # dir用于记录方向
      def fill_0(self, grid, row, col, dir):
          if row<0 or col<0 or row>=len(grid) or col>= len(grid[0]):
              return
          if grid[row][col] == 0:
              return
          # 前序位置,刚进入节点
          grid[row][col] = 0
          self.route.append(dir)
          self.fill_0(grid, row-1, col, 1)
          self.fill_0(grid, row+1, col, 2)
          self.fill_0(grid, row, col-1, 3)
          self.fill_0(grid, row, col+1, 4)
          # 后序位置,逆向回传
          self.route.append(-dir)
  ```

16. 打家劫舍

你是一个专业的小偷，计划偷窃沿街的房屋。每间房内都藏有一定的现金，影响你偷窃的唯一制约因素就是相邻的房屋装有相互连通的防盗系统，**如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警。**

给定一个代表每个房屋存放金额的非负整数数组，计算你 **不触动警报装置的情况下**，一夜之内能够偷窃到的最高金额。

示例 1：

```
输入：[1,2,3,1]
输出：4
解释：偷窃 1 号房屋 (金额 = 1) ，然后偷窃 3 号房屋 (金额 = 3)。
     偷窃到的最高金额 = 1 + 3 = 4 。
```

- ```
  class Solution:
      def rob(self, nums: List[int]) -> int:
          if not nums:
              return 0
          dp_00 = 0
          dp_01 = nums[0]
          for i in range(1, len(nums)):
              temp = dp_00
              dp_00 = max(dp_00, dp_01)
              dp_01 = temp + nums[i]
          return max(dp_00, dp_01)
  ```

17. 打家劫舍II

你是一个专业的小偷，计划偷窃沿街的房屋，每间房内都藏有一定的现金。这个地方所有的房屋都 **围成一圈** ，这意味着第一个房屋和最后一个房屋是紧挨着的。同时，相邻的房屋装有相互连通的防盗系统，**如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警** 。

给定一个代表每个房屋存放金额的非负整数数组，计算你 **在不触动警报装置的情况下** ，今晚能够偷窃到的最高金额。

示例 1：

```
输入：nums = [2,3,2]
输出：3
解释：你不能先偷窃 1 号房屋（金额 = 2），然后偷窃 3 号房屋（金额 = 2），因为他们是相邻的。
```

- 房子围成圈+从当前房子开始偷可以获得的最高金额
- ```
  class Solution:
      def rob(self, nums: List[int]) -> int:
  ```
  ```

```
        if not nums:
            return 0
        if len(nums) == 1:
            return nums[0]
        return max(self.money(nums, 0, len(nums)-2), self.money(nums, 1, len(nums)-1))
    def money(self, nums, start, end):
        dp = [0] * (len(nums)+2)
        dp[-1] = 0
        dp[-2] = 0
        if end < len(nums) - 1:
            dp[-3] = 0
        for i in range(end, start-1, -1):
            dp[i] = max(dp[i+1], dp[i+2]+nums[i])
        return dp[start]
```

18. 打家劫舍III

小偷又发现了一个新的可行窃的地区。这个地区只有一个入口，我们称之为 `root` 。

除了 `root` 之外，每栋房子有且只有一个"父"房子与之相连。一番侦察之后，聪明的小偷意识到"这个地方的所有房屋的排列类似于一棵二叉树"。 如果 **两个直接相连的房子在同一天晚上被打劫** ，房屋将自动报警。

给定二叉树的 `root` 。返回 ***在不触动警报的情况下*** *，小偷能够盗取的最高金额* 。

- 每个节点有抢和不抢两种选择+（抢root+不抢root.left+不抢root.right）+（不抢root+抢root.left+抢root.right）

```
class Solution:
    def rob(self, root: TreeNode) -> int:
        if not root:
            return 0
        Yes_res, No_res = self.money(root)
        return max(Yes_res, No_res)
    def money(self, root):
        if not root:
            return 0, 0
        yes_l, no_l = self.money(root.left)
        yes_r, no_r = self.money(root.right)
        yes_root = root.val + no_l + no_r
        no_root = max(yes_l, no_l) + max(yes_r, no_r)
        return yes_root, no_root
```

19. 股票买卖的最佳时机

给定一个数组 `prices` ，它的第 `i` 个元素 `prices[i]` 表示一支给定股票第 `i` 天的价格。

你只能选择 **某一天** 买入这只股票，并选择在 **未来的某一个不同的日子** 卖出该股票。设计一个算法来计算你所能获取的最大利润。

返回你可以从这笔交易中获取的最大利润。如果你不能获取任何利润，返回 `0` 。

示例 1：

```
输入：[7,1,5,3,6,4]
输出：5
解释：在第 2 天（股票价格 = 1）的时候买入，在第 5 天（股票价格 = 6）的时候卖出，最大利润 = 6-1 = 5 。
     注意利润不能是 7-1 = 6，因为卖出价格需要大于买入价格；同时，你不能在买入前卖出股票。
```

- 只能交易一次

```
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        if not prices:
            return 0
        dp_00 = 0
        dp_01 = -prices[0]
        for i in range(1, len(prices)):
            dp_00 = max(dp_00, dp_01+prices[i])
            dp_01 = max(dp_01, -prices[i])
        return dp_00
```

20. 股票买卖的最佳时机II

给定一个数组，它的第 `i` 个元素是一支给定的股票在第 `i` 天的价格。

设计一个算法来计算你所能获取的最大利润。你最多可以完成 **两笔** 交易。

注意： 你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。

示例 1：

```
输入：prices = [3,3,5,0,0,3,1,4]
输出：6
解释：在第 4 天（股票价格 = 0）的时候买入，在第 6 天（股票价格 = 3）的时候卖出，这笔交易所能获得利润 = 3-0 = 3 。
     随后，在第 7 天（股票价格 = 1）的时候买入，在第 8 天 （股票价格 = 4）的时候卖出，这笔交易所能获得利润 = 4-1 = 3 。
```

- 两次交易

```
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        if not prices:
            return 0
        dp_00 = 0
        dp_01 = -prices[0]
        dp_10 = 0
        dp_11 = -prices[0]
        for i in range(1, len(prices)):
            temp = dp_00
            dp_00 = max(dp_00, dp_01+prices[i])
            dp_01 = max(dp_01, -prices[i])
            dp_10 = max(dp_10, dp_11+prices[i])
```

```
            dp_11 = max(dp_11, temp-prices[i])
        return dp_10
```

21. 买卖股票最佳时机III

    给你一个整数数组 `prices` ，其中 `prices[i]` 表示某支股票第 `i` 天的价格。

    在每一天，你可以决定是否购买和/或出售股票。你在任何时候 **最多** 只能持有 **一股** 股票。你也可以先购买，然后在 **同一天** 出售。

    返回 *你能获得的* ***最大*** *利润* 。

    - **示例 1:**

      ```
      输入：prices = [7,1,5,3,6,4]
      输出：7
      解释：在第 2 天（股票价格 = 1）的时候买入，在第 3 天（股票价格 = 5）的时候卖出，这笔交易所能获得利润 = 5 - 1 = 4 。
           随后，在第 4 天（股票价格 = 3）的时候买入，在第 5 天（股票价格 = 6）的时候卖出，这笔交易所能获得利润 = 6 - 3 = 3 。
           总利润为 4 + 3 = 7 。
      ```

    - 无限次交易

    - ```
      class Solution:
          def maxProfit(self, prices: List[int]) -> int:
              if not prices:
                  return 0
              dp_0 = 0
              dp_1 = -prices[0]
              for i in range(1, len(prices)):
                  temp = dp_0
                  dp_0 = max(dp_0, dp_1+prices[i])
                  dp_1 = max(dp_1, temp-prices[i])
              return dp_0
      ```

22. 买卖股票的最佳时机IV

    给定一个整数数组 `prices` ，它的第 `i` 个元素 `prices[i]` 是一支给定的股票在第 `i` 天的价格。

    设计一个算法来计算你所能获取的最大利润。你最多可以完成 **k** 笔交易。

    注意：你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。

    - **示例 1:**

      ```
      输入：k = 2, prices = [2,4,1]
      输出：2
      解释：在第 1 天 (股票价格 = 2) 的时候买入，在第 2 天 (股票价格 = 4) 的时候卖出，这笔交易所能获得利润 = 4-2 = 2 。
      ```

    - 交易次数为k

    - ```
      class Solution:
          def maxProfit(self, k: int, prices: List[int]) -> int:
              if not prices:
                  return 0
              dp = [[[0, 0] for _ in range(k+1)] for _ in range(len(prices))]
              for i in range(len(prices)):
                  for j in range(1, k+1):
                      if i == 0:
                          dp[i][j][0] = 0
                          dp[i][j][1] = -prices[i]
                          continue
                      dp[i][j][0] = max(dp[i-1][j][0], dp[i-1][j][1]+prices[i])
                      dp[i][j][1] = max(dp[i-1][j][1], dp[i-1][j-1][0]-prices[i])
              return dp[-1][-1][0]
      ```

23. 买卖股票的最佳时机-含冷冻期

    给定一个整数数组`prices`，其中第 `prices[i]` 表示第 `i` 天的股票价格。

    设计一个算法计算出最大利润。在满足以下约束条件下，你可以尽可能地完成更多的交易（多次买卖一支股票）：

    - 卖出股票后，你无法在第二天买入股票 (即冷冻期为 1 天)。

    注意：你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。

    - **示例 1:**

      ```
      输入：prices = [1,2,3,0,2]
      输出：3
      解释：对应的交易状态为：[买入，卖出，冷冻期，买入，卖出]
      ```

    - 含冷冻期+第一天需要特殊考虑

    - ```
      class Solution:
          def maxProfit(self, prices: List[int]) -> int:
              if not prices:
                  return 0
              dp = [[0, 0]for _ in range(len(prices))]
              for i in range(len(prices)):
                  if i == 0:
                      dp[i][0] = 0
                      dp[i][1] = -prices[i]
                      continue
                  if i == 1:
                      dp[i][0] = max(dp[i-1][0], dp[i-1][1]+prices[i])
                      dp[i][1] = max(dp[i-1][1], -prices[i])
                      continue
                  dp[i][0] = max(dp[i-1][0], dp[i-1][1]+prices[i])
                  dp[i][1] = max(dp[i-1][1], dp[i-2][0]-prices[i])
      ```
      ```

```
        return dp[-1][0]
```

24. 买卖股票的最佳时机-含手续费

给定一个整数数组 `prices`，其中 `prices[i]` 表示第 `i` 天的股票价格；整数 `fee` 代表了交易股票的手续费用。

你可以无限次地完成交易，但是你每笔交易都需要付手续费。如果你已经购买了一个股票，在卖出它之前你就不能再继续购买股票了。

返回获得利润的最大值。

注意：这里的一笔交易指买入持有并卖出股票的整个过程，每笔交易你只需要为支付一次手续费。

- **示例 1：**

```
输入：prices = [1, 3, 2, 8, 4, 9], fee = 2
输出：8
解释：能够达到的最大利润:
在此处买入 prices[0] = 1
在此处卖出 prices[3] = 8
在此处买入 prices[4] = 4
在此处卖出 prices[5] = 9
总利润: ((8 - 1) - 2) + ((9 - 4) - 2) = 8
```

- 每次买入时交手续费
- ```
  class Solution:
      def maxProfit(self, prices: List[int], fee: int) -> int:
          if not prices:
              return 0
          dp_0 = 0
          dp_1 = -prices[0]-fee
          for i in range(1, len(prices)):
              temp = dp_0
              dp_0 = max(dp_0, dp_1+prices[i])
              dp_1 = max(dp_1, temp-prices[i]-fee)
          return dp_0
  ```

25. 找到字符串中所有字母异位词

给定两个字符串 `s` 和 `p`，找到 `s` 中所有 `p` 的**异位词**的子串，返回这些子串的起始索引。不考虑答案输出的顺序。

异位词 指由相同字母重排列形成的字符串（包括相同的字符串）。

- **示例 1：**

```
输入：s = "cbaebabacd", p = "abc"
输出：[0,6]
解释：
起始索引等于 0 的子串是 "cba", 它是 "abc" 的异位词。
起始索引等于 6 的子串是 "bac", 它是 "abc" 的异位词。
```

- 滑动窗口法+找到可行解+判断可行解的长度不能有多余的字母
- ```
  class Solution:
      def findAnagrams(self, s: str, p: str) -> List[int]:
          if not s:
              return []
          left = 0
          right = 0
          need = {}
          for item in p:
              if item in need:
                  need[item] += 1
              else:
                  need[item] = 1
          valid = 0
          window = {}
          res = []
          while right < len(s):
              cur = s[right]
              right += 1
              if cur in need:
                  if cur in window:
                      window[cur] += 1
                  else:
                      window[cur] = 1
                  if window[cur] == need[cur]:
                      valid += 1
              while valid == len(need):
                  if right-left == len(p):
                      res.append(left)
                  cur_pop = s[left]
                  left += 1
                  if cur_pop in need:
                      window[cur_pop] -= 1
                      if window[cur_pop] < need[cur_pop]:
                          valid -= 1
                      if window[cur_pop] == 0:
                          del window[cur_pop]
          return res
  ```

26. 字符串的排列

给你两个字符串 `s1` 和 `s2` ，写一个函数来判断 `s2` 是否包含 `s1` 的排列。如果是，返回 `true` ；否则，返回 `false` 。

换句话说， `s1` 的排列之一是 `s2` 的 **子串** 。

- **示例 1：**

```
输入：s1 = "ab" s2 = "eidbaooo"
输出：true
解释：s2 包含 s1 的排列之一 ("ba").
```

- 滑动窗口+找异位词
- ```
  class Solution:
      def checkInclusion(self, s1: str, s2: str) -> bool:
          if not s2:
              return False
          need = {}
          for item in s1:
              if item in need:
                  need[item] += 1
              else:
                  need[item] = 1
          window = {}
          valid = 0
          left = 0
          right = 0
          while right < len(s2):
              cur = s2[right]
              right += 1
              if cur in need:
                  if cur in window:
                      window[cur] += 1
                  else:
                      window[cur] = 1
                  if window[cur] == need[cur]:
                      valid += 1
              while left < right and valid == len(need):
                  if right - left == len(s1):
                      return True
                  cur_pop = s2[left]
                  left += 1
                  if cur_pop in need:
                      window[cur_pop] -= 1
                      if window[cur_pop] < need[cur_pop]:
                          valid -= 1
                      if window[cur_pop] == 0:
                          del window[cur_pop]
          return False
  ```

27. 最长公共前缀

编写一个函数来查找字符串数组中的最长公共前缀。

如果不存在公共前缀，返回空字符串 `""`。

- **示例 1：**

```
输入：strs = ["flower","flow","flight"]
输出："fl"
```

- 用最短字符串的前缀与其他字符串作比较+flag做标记
- ```
  class Solution:
      def longestCommonPrefix(self, strs: List[str]) -> str:
          if not strs:
              return ''
          min_str = strs[0]
          for item in strs:
              if len(item) < len(min_str):
                  min_str = item
          res = ''
          flag = True
          for i in range(1, len(min_str)+1):
              cur = min_str[:i]
              for item in strs:
                  if item[:i] == cur:
                      continue
                  else:
                      flag = False
                      break
              if flag:
                  if len(cur) > len(res):
                      res = cur
              else:
                  break
          return res
  ```

28. 最长公共子序列

给定两个字符串 `text1` 和 `text2`，返回这两个字符串的最长 **公共子序列** 的长度。如果不存在 **公共子序列** ，返回 `0` 。

一个字符串的 **子序列** 是指这样一个新的字符串：它是由原字符串在不改变字符的相对顺序的情况下删除某些字符（也可以不删除任何字符）后组成的新字符串。

- 例如，"ace" 是 "abcde" 的子序列，但 "aec" 不是 "abcde" 的子序列。

两个字符串的 **公共子序列** 是这两个字符串所共同拥有的子序列。

示例 1：

```
输入：text1 = "abcde", text2 = "ace"
输出：3
解释：最长公共子序列是 "ace" ，它的长度为 3 。
```

二维dp

```
class Solution:
    def longestCommonSubsequence(self, text1: str, text2: str) -> int:
        dp = [[0 for _ in range(len(text2)+1)] for _ in range(len(text1)+1)]
        for i in range(1, len(text1)+1):
            for j in range(1, len(text2)+1):
                if text1[i-1] == text2[j-1]:
                    dp[i][j] = dp[i-1][j-1] + 1
                else:
                    dp[i][j] = max(dp[i-1][j], dp[i][j-1])
        return dp[-1][-1]
```

29. 最长回文串

给定一个包含大写字母和小写字母的字符串 `s` ，返回 *通过这些字母构造成的* ***最长的回文串*** 。

在构造过程中，请注意 **区分大小写** 。比如 "Aa" 不能当做一个回文字符串。

示例 1：

```
输入:s = "abccccdd"
输出:7
解释:
我们可以构造的最长的回文串是"dccaccd", 它的长度是 7。
```

统计字母个数+偶数全部用+奇数的只能用一个，其他转成偶数

```
class Solution:
    def longestPalindrome(self, s: str) -> int:
        if len(s) < 2:
            return len(s)
        num_dict = {}
        for item in s:
            if item in num_dict:
                num_dict[item] += 1
            else:
                num_dict[item] = 1
        res = 0
        flag = True
        for k, v in num_dict.items():
            if v % 2 == 0:
                res += v
            else:
                if flag:
                    res += v
                    flag = False
                else:
                    res += v - 1
        return res
```

30. 最长回文子序列

给你一个字符串 `s` ，找出其中最长的回文子序列，并返回该序列的长度。

子序列定义为：不改变剩余字符顺序的情况下，删除某些字符或者不删除任何字符形成的一个序列。

示例 1：

```
输入：s = "bbbab"
输出：4
解释：一个可能的最长回文子序列为 "bbbb" 。
```

相当于求s与它的反转字符串的最长公共子序列

```
class Solution(object):
    def longestPalindromeSubseq(self, s):
        """
        :type s: str
        :rtype: int
        """
        dp = [[0 for _ in range(len(s)+1)] for _ in range(len(s)+1)]
        reverse_s = s[::-1]
        for i in range(1, len(s)+1):
            for j in range(1, len(s)+1):
                if s[i-1] == reverse_s[j-1]:
                    dp[i][j] = dp[i-1][j-1] + 1
                else:
                    dp[i][j] = max(dp[i-1][j], dp[i][j-1])
        return dp[-1][-1]
```

31. 最长回文子串

5. 最长回文子串 labuladong 题解 思路

难度 中等 5229

给你一个字符串 `s`，找到 `s` 中最长的回文子串。

示例 1:

```
输入：s = "babad"
输出："bab"
解释："aba" 同样是符合题意的答案。
```

- 双指针+由中间向两边扩展
- ```
  class Solution:
      def longestPalindrome(self, s: str) -> str:
          if not s:
              return 0
          res = ''
          for i in range(len(s)):
              s1 = self.find_rome(s, i, i)
              s2 = self.find_rome(s, i, i+1)
              if len(s1) > len(res):
                  res = s1
              if len(s2) > len(res):
                  res = s2
          return res
      def find_rome(self, s, i, j):
          while i >= 0 and j < len(s) and s[i] == s[j]:
              i -= 1
              j += 1
          return s[i+1:j]
  ```

32. 最长公共子串
    - ```
      class Solution:
          def LCS1(self, str1, str2):
              # write code here
              left = 0
              res = ''
              for i in range(len(str1)+1):
                  if str1[left:i+1] in str2:
                      res = str1[left:i+1]
                  else:
                      left += 1
              return res
      # 最长公共子串的长度
          def LCS2(self, str1, str2):
              len_1 = len(str1)
              len_2 = len(str2)
              dp = [[0 for _ in range(len_2+1)] for _ in range(len_1+1)]
              res = 0
              for i in range(1, len_1+1):
                  for j in range(1, len_2+1):
                      if str1[i-1] == str2[j-1]:
                          dp[i][j] = dp[i-1][j-1] + 1
                          res = max(res, dp[i][j])
                      else:
                          dp[i][j] = 0
              return res
      ```
33. 二进制求和

给你两个二进制字符串，返回它们的和（用二进制表示）。

输入为 **非空** 字符串且只包含数字 `1` 和 `0`。

示例 1:

```
输入：a = "11", b = "1"
输出："100"
```

示例 2:

```
输入：a = "1010", b = "1011"
输出："10101"
```

- 类似于十进制求和
- ```
  class Solution:
      def addBinary(self, a: str, b: str) -> str:
          max_len = max(len(a), len(b))
          a = a.zfill(max_len)
          b = b.zfill(max_len)
          res = ''
          i = max_len-1
          carry = 0
          while i >= 0:
              temp = int(a[i]) + int(b[i]) + carry
              if temp > 1:
                  carry = 1
                  temp = temp - 2
              else:
                  carry = 0
              res = str(temp) + res
              i -= 1
          if carry == 1:
              res = str(carry) + res
          return res
  ```

34. 验证回文串
  ```

给定一个字符串，验证它是否是回文串，只考虑字母和数字字符，可以忽略字母的大小写。

说明：本题中，我们将空字符串定义为有效的回文串。

- **示例 1:**

```
输入: "A man, a plan, a canal: Panama"
输出: true
解释: "amanaplanacanalpanama" 是回文串
```

- 去除除了字母和数字以外的元素+双指针判断
- ```
  class Solution:
      def isPalindrome(self, s: str) -> bool:
          if not s:
              return True
          temp = ''
          for i in range(len(s)):
              if s[i].isalnum():
                  temp += s[i].lower()
          left = 0
          right = len(temp) - 1
          while left <= right:
              if temp[left] == temp[right]:
                  left += 1
                  right -= 1
              else:
                  return False
          return True
  ```

35. 同构字符串

205. 同构字符串

难度 简单 460

给定两个字符串 `s` 和 `t` ，判断它们是否是同构的。

如果 `s` 中的字符可以按某种映射关系替换得到 `t` ，那么这两个字符串是同构的。

- 每个出现的字符都应当映射到另一个字符，同时不改变字符的顺序。不同字符不能映射到同一个字符上，相同字符只能映射到同一个字符上，字符可以映射到自己本身。

示例 1:

```
输入: s = "egg", t = "add"
输出: true
```

- 哈希表记录映射关系+保证key不同（每个字母只出现一遍）+value不同（每个字母只出现一遍）
- ```
  class Solution:
      def isIsomorphic(self, s: str, t: str) -> bool:
          hash_dict = {}
          if len(s) != len(t):
              return False
          for i in range(len(s)):
              if s[i] in hash_dict.keys():
                  if hash_dict[s[i]] == t[i]:
                      continue
                  else:
                      return False
              else:
                  if t[i] in hash_dict.values():
                      return False
                  else:
                      hash_dict[s[i]] = t[i]
          return True
  ```

36. 四数之和

- 排序+三数之和+两数之和
- ```
  class Solution:
      def fourSum(self, nums: List[int], target: int) -> List[List[int]]:
          if not nums:
              return []
          nums.sort()
          res = []
          for i in range(len(nums)):
              if i > 0 and nums[i] == nums[i-1]:
                  continue
              temp = self.threeSum(nums[i+1:], target-nums[i])
              if temp:
                  for item in temp:
                      item.append(nums[i])
                      res.append(item)
          return res

      def threeSum(self, nums, target):
          res = []
          for i in range(len(nums)):
              if i > 0 and nums[i] == nums[i-1]:
                  continue
              temp = self.twoSum(nums[i+1:], target-nums[i])
              if temp:
                  for item in temp:
                      item.append(nums[i])
                      res.append(item)
          return res

      def twoSum(self, nums, target):
          res = []
          left = 0
          right = len(nums) - 1
          while left < right:
              cur_left = nums[left]
  ```
  ```

```
                cur_right = nums[right]
                if cur_left + cur_right == target:
                    res.append([cur_left, cur_right])
                    while left < right and nums[left] == cur_left:
                        left += 1
                    while left < right and nums[right] == cur_right:
                        right -= 1
                elif cur_left + cur_right > target:
                    while left < right and nums[right] == cur_right:
                        right -= 1
                elif cur_left + cur_right < target:
                    while left < right and nums[left] == cur_left:
                        left += 1
            return res
```

37. 删除有序数组中的重复项

示例 1：

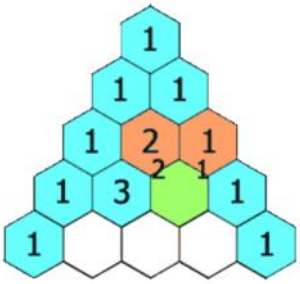
```
输入：nums = [1,1,2]
输出：2, nums = [1,2,_]
解释：函数应该返回新的长度 2 ，并且原数组 nums 的前两个元素被修改为 1, 2 。不需要考虑数组中超出新长度后面的元素。
```

- **示例 2：**

```
输入：nums = [0,0,1,1,1,2,2,3,3,4]
输出：5, nums = [0,1,2,3,4]
解释：函数应该返回新的长度 5 ， 并且原数组 nums 的前五个元素被修改为 0, 1, 2, 3, 4 。不需要考虑数组中超出新长度后面的元素。
```

- 快慢指针+循换
- ```
  class Solution:
      def removeDuplicates(self, nums: List[int]) -> int:
          if len(nums) <= 1:
              return len(nums)
          slow = 0
          for fast in range(1, len(nums)):
              if nums[fast] == nums[slow]:
                  continue
              else:
                  slow += 1
                  nums[slow] = nums[fast]
          return slow+1
  ```

38. 杨辉三角

- 

示例 1：

```
输入：numRows = 5
输出：[[1],[1,1],[1,2,1],[1,3,3,1],[1,4,6,4,1]]
```

- Cur.append(1) +生成中间的数+cur.append(1)
- ```
  class Solution:
      def generate(self, numRows: int) -> List[List[int]]:
          if numRows == 1:
              return [[1]]
          res = [[1]]
          for i in range(2, numRows+1):
              cur = [1]
              for j in range(len(res[-1])-1):
                  cur.append(res[-1][j]+res[-1][j+1])
              cur.append(1)
              res.append(cur)
          return res
  ```

39. 回文数

给你一个整数 `x` ，如果 `x` 是一个回文整数，返回 `true` ；否则，返回 `false` 。

回文数是指正序（从左向右）和倒序（从右向左）读都是一样的整数。

- 例如，`121` 是回文，而 `123` 不是。

- **示例 1：**

```
输入：x = 121
输出：true
```

示例 2：

```
输入：x = -121
输出：false
```

- 全部反转
- ```
  class Solution:
      def isPalindrome(self, x: int) -> bool:
          if x < 0 :
              return False
          temp = 0
          cur = x
          while cur:
              temp = temp * 10 + cur % 10
              cur = cur // 10
          if x == temp:
              return True
          else:
              return False
  ```

40. 2的幂

    给你一个整数 `n`，请你判断该整数是否是 2 的幂次方。如果是，返回 `true` ；否则，返回 `false` 。

    如果存在一个整数 `x` 使得 `n == 2^x` ，则认为 `n` 是 2 的幂次方。

    示例 1：

    ```
    输入：n = 1
    输出：true
    解释：2^0 = 1
    ```

    示例 2：

    ```
    输入：n = 16
    输出：true
    解释：2^4 = 16
    ```

    - 递归法
    - ```
      class Solution:
          def isPowerOfTwo(self, n: int) -> bool:
              if n == 0:
                  return False
              if n == 1:
                  return True
              if n % 2 == 1:
                  return False
              else:
                  return self.isPowerOfTwo(n // 2)
      ```

41. 丑数
    - ```
      class Solution:
          def isUgly(self, n: int) -> bool:
              if n == 0:
                  return False
              if n == 1:
                  return True
              if n % 2 == 0:
                  return self.isUgly(n // 2)
              elif n % 3 == 0:
                  return self.isUgly(n // 3)
              elif n % 5 == 0:
                  return self.isUgly(n // 5)
              else:
                  return False
      ```

42. 丑数II
    - 求第n个丑数+三指针
    - ```
      class Solution:
          def nthUglyNumber(self, n: int) -> int:
              if n == 1:
                  return 1
              a = 0
              b = 0
              c = 0
              temp = [1]
              for i in range(2, n+1):
                  temp.append(min(temp[a]*2, temp[b]*3, temp[c]*5))
                  if temp[-1] == temp[a] * 2:
                      a += 1
                  if temp[-1] == temp[b] * 3:
                      b += 1
                  if temp[-1] == temp[c] * 5:
                      c += 1
              return temp[-1]
      ```

43. 存在重复元素
    -

    给你一个整数数组 `nums` 。如果任一值在数组中出现 **至少两次** ，返回 `true` ；如果数组中每个元素互不相同，返回 `false` 。

    示例 1：

    ```
    输入：nums = [1,2,3,1]
    输出：true
    ```

    示例 2：

    ```
    输入：nums = [1,2,3,4]
    输出：false
    ```

    - ```
      class Solution(object):
          def containsDuplicate(self, nums):
              """
              :type nums: List[int]
              :rtype: bool
              """
      ```
      ```

```
        set_list = list(set(nums))
        if len(nums) == len(set_list):
            return False
        else:
            return True
```

44. 存在重复元素II

给你一个整数数组 `nums` 和一个整数 `k` ，判断数组中是否存在两个 **不同的索引** `i` 和 `j` ，满足 `nums[i] == nums[j]` 且 `abs(i - j) <= k` 。如果存在，返回 `true` ；否则，返回 `false` 。

示例 1：

```
输入：nums = [1,2,3,1], k = 3
输出：true
```

示例 2：

```
输入：nums = [1,0,1,1], k = 1
输出：true
```

- 哈希表记录元素最后一次出现时的下标

```
class Solution:
    def containsNearbyDuplicate(self, nums: List[int], k: int) -> bool:
        if not nums:
            return False
        nums_dict = {}
        for i in range(len(nums)):
            if nums[i] not in nums_dict:
                nums_dict[nums[i]] = i
            elif nums[i] in nums_dict:
                if abs(nums_dict[nums[i]]-i) <= k:
                    return True
                else:
                    nums_dict[nums[i]] = i
        return False
```

45. 判断子序列

字符串的一个子序列是原始字符串删除一些（也可以不删除）字符而不改变剩余字符相对位置形成的新字符串。（例如，`"ace"` 是 `"abcde"` 的一个子序列，而 `"aec"` 不是）。

进阶：

如果有大量输入的 S，称作 S1, S2, ... , Sk 其中 k >= 10亿，你需要依次检查它们是否为 T 的子序列。在这种情况下，你会怎样改变代码？

致谢：

特别感谢 @pbrother 添加此问题并且创建所有测试用例。

示例 1：

```
输入：s = "abc", t = "ahbgdc"
输出：true
```

- 动态规划

```
class Solution:
    def isSubsequence(self, s: str, t: str) -> bool:
        dp = [[False for _ in range(len(t)+1)] for _ in range(len(s)+1)]
        for i in range(len(s)+1):
            for j in range(0, len(t)+1):
                if i == 0:
                    dp[i][j] = True
                    continue
                if j > 0 and s[i-1] == t[j-1]:
                    dp[i][j] = dp[i-1][j-1]
                elif j > 0 and s[i-1] != t[j-1]:
                    dp[i][j] = dp[i][j-1]
        return dp[-1][-1]
```

46. 使用最小花费爬楼梯

给你一个整数数组 `cost` ，其中 `cost[i]` 是从楼梯第 `i` 个台阶向上爬需要支付的费用。一旦你支付此费用，即可选择向上爬一个或者两个台阶。

你可以选择从下标为 `0` 或下标为 `1` 的台阶开始爬楼梯。

请你计算并返回达到楼梯顶部的最低花费。

示例 1：

```
输入：cost = [10,15,20]
输出：15
解释：你将从下标为 1 的台阶开始。
- 支付 15 ，向上爬两个台阶，到达楼梯顶部。
总花费为 15 。
```

- 有条件的爬楼梯+考虑费用+取最小值

```
class Solution:
    def minCostClimbingStairs(self, cost: List[int]) -> int:
        dp = [0] * (len(cost)+1)
        for i in range(2, len(cost)+1):
            dp[i] = min(dp[i-1]+cost[i-1], dp[i-2]+cost[i-2])
        return dp[-1]
```

47. 二叉树的直径

给定一棵二叉树，你需要计算它的直径长度。一棵二叉树的直径长度是任意两个结点路径长度中的最大值。这条路径可能穿过也可能不穿过根结点。

示例：
给定二叉树

```
          1
         / \
        2   3
       / \
      4   5
```

返回 **3**, 它的长度是路径 [4,2,1,3] 或者 [5,2,1,3]。

- 二叉树的直径=过根节点的直径，左子树的最大直径，右子树的最大直径 的最大值

```
class Solution:
    def diameterOfBinaryTree(self, root: Optional[TreeNode]) -> int:
        if not root:
            return 0
        l_d = self.depth(root.left)
        r_d = self.depth(root.right)
        root_dia = max(l_d+r_d, self.diameterOfBinaryTree(root.left), self.diameterOfBinaryTree(root.right))
        return root_dia
    def depth(self, root):
        if not root:
            return 0
        l_d = self.depth(root.left)
        r_d = self.depth(root.right)
        root_d = max(l_d, r_d) + 1
        return root_d
```

48. 打开转盘锁

你有一个带有四个圆形拨轮的转盘锁。每个拨轮都有10个数字： `'0'`, `'1'`, `'2'`, `'3'`, `'4'`, `'5'`, `'6'`, `'7'`, `'8'`, `'9'` 。每个拨轮可以自由旋转：例如把 `'9'` 变为 `'0'`，`'0'` 变为 `'9'` 。每次旋转都只能旋转一个拨轮的一位数字。

锁的初始数字为 `'0000'` ，一个代表四个拨轮的数字的字符串。

列表 `deadends` 包含了一组死亡数字，一旦拨轮的数字和列表里的任何一个元素相同，这个锁将会被永久锁定，无法再被旋转。

字符串 `target` 代表可以解锁的数字，你需要给出解锁需要的最小旋转次数，如果无论如何不能解锁，返回 `-1` 。

示例 1:

```
输入：deadends = ["0201","0101","0102","1212","2002"],
target = "0202"
输出：6
解释：
可能的移动序列为 "0000" -> "1000" -> "1100" -> "1200" ->
"1201" -> "1202" -> "0202"。
```

- BFS+每次生成一层的节点（判断）+目标节点的深度就是最少次数+上旋+下旋

```
class Solution(object):
    def openLock(self, deadends, target):
        queue = ['0000']  # 初始状态加入队列
        visited = ['0000'] + deadends # 访问过的节点，即已经生成过的节点
        step = 0
        while queue:
            l = len(queue)
            for _ in range(l):
                cur = queue.pop(0)
                if cur in deadends:
                    continue
                if cur == target:
                    return step
                for j in range(4):
                    U_rel = self.U_r(cur, j)
                    D_rel = self.D_r(cur, j)
                    # if U_rel not in deadends and U_rel not in visited:
                    if U_rel not in visited:
                        queue.append(U_rel)
                        visited.append(U_rel)
                    # if D_rel not in deadends and D_rel not in visited:
                    if D_rel not in visited:
                        queue.append(D_rel)
                        visited.append(D_rel)
            step += 1
        return -1
    # 向上旋转：数字变小
    def U_r(self, state, num):
        if int(state[num]) == 0:
            state = state[:num]+'9'+state[num+1:]
        else:
            state = state[:num]+str(int(state[num])-1)+state[num+1:]
        return state
    # 向下旋转：数字变大
    def D_r(self, state, num):
        if int(state[num]) == 9:
            state = state[:num]+'0'+state[num+1:]
        else:
            state = state[:num]+str(int(state[num])+1)+state[num+1:]
        return state
```

49. 0-1背包问题

- 给你一个可装载重量为 W 的背包和 N 个物品，每个物品有重量和价值两个属性。其中第 i 个物品的重量为 wt[i]，价值为 val[i]，现在让你用这个背包装物品，最多能装的价值是多少？
- 每个背包有 选择 和 不选择 两种状态 + 比较选最大值

- ```
  class Solution(object):
      def knapsack(self, W, w, val):
          # dp[i][j]表示在前i个物品中选择物品装入容量为j的背包，可以获取的最大价值
          # 每一个物品有 装入 和 不装入  两种状态
          dp = [[0 for _ in range(W+1)] for _ in range(len(w)+1)]
          for i in range(1, len(w)+1):
              for j in range(W+1):
                  # 不能装入
                  if j - w[i-1] < 0:
                      dp[i][j] = dp[i-1][j]
                  else:
                      dp[i][j] = max(dp[i-1][j], dp[i-1][j-w[i-1]]+val[i-1])
          return dp[-1][-1]
  ```

50. 分割等和子集

    给你一个 **只包含正整数** 的 **非空** 数组 `nums` 。请你判断是否可以将这个数组分割成两个子集，使得两个子集的元素和相等。

    - **示例 1：**

      ```
      输入：nums = [1,5,11,5]
      输出：true
      解释：数组可以分割成 [1, 5, 5] 和 [11] 。
      ```

    - 转换为背包问题：数组中是否存在元素组合可以装满容量为summ//2的背包
    - ```
      class Solution:
          def canPartition(self, nums):
              # 可以看给容量为summ//2的背包中装nums中的数值，nums中是否存在一个组合可以满足这个要求
              if len(nums) <= 1:
                  return False
              summ = 0
              for item in nums:
                  summ += item
              if summ % 2 == 1:
                  return False
              summ = summ // 2
              # dp[i][j]表示用nums中的前i个元素组成的子数组装容量为j的背包，能否装满
              dp = [[False for _ in range(summ+1)] for _ in range(len(nums)+1)]
              # base case
              for i in range(len(nums)+1):
                  dp[i][0] = True
              for i in range(1, len(nums)+1):
                  for j in range(1, summ+1):
                      if j - nums[i-1] < 0:
                          dp[i][j] = dp[i-1][j]
                      else:
                          dp[i][j] = dp[i-1][j] | dp[i-1][j-nums[i-1]]
              return dp[-1][-1]
      ```

51. 零钱兑换II

    给你一个整数数组 `coins` 表示不同面额的硬币，另给一个整数 `amount` 表示总金额。

    请你计算并返回可以凑成总金额的硬币组合数。如果任何硬币组合都无法凑出总金额，返回 `0` 。

    假设每一种面额的硬币有无限个。

    - 题目数据保证结果符合 32 位带符号整数。

    示例 1：

    ```
    输入：amount = 5, coins = [1, 2, 5]
    输出：4
    ```

    - 完全背包问题：容量为amount的背包装入coins中的物品，最多有几种装法
    - ```
      class Solution:
          def change(self, amount, coins):
              # dp[i][j]表示用前i个物品装满容量为j的背包，最多有多少种装法
              dp = [[0 for _ in range(amount+1)] for _ in range(len(coins)+1)]
              # base case
              for i in range(len(coins)+1):
                  dp[i][0] = 1
              for i in range(1, len(coins)+1):
                  for j in range(1, amount+1):
                      if j - coins[i-1] < 0:
                          dp[i][j] = dp[i-1][j]
                      else:
                          # 选 + 不选
                          dp[i][j] = dp[i-1][j] + dp[i][j-coins[i-1]]
              return dp[-1][-1]
      ```

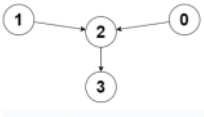
52. 求均方根

    - ```
      class Solution:
          def sqrt(self , x ):
              # write code here
              if x <= 0:
                  return 0
              left = 0
              right = x
              while left <= right:
                  mid = (left + right) // 2
                  if mid * mid < x:
                      left = mid + 1
                  elif mid * mid > x:
                      right = mid - 1
                  else:
                      return mid
              return right
      ```

53. 整数除法

    - 位运算+判断target中有多少个num
    - ```
      class Solution:
          def divide(self, a: int, b: int) -> int:
      ```
      ```

```
        def recur(target, num):
            if target < num:
                return 0
            count = 0
            temp = num << 1
            while temp < target:
                temp <<= 1
                count += 1
            temp >>= 1
            res = 2 ** count + recur(target-temp, num)
            MAX_INT = 2 ** 31 - 1
            MIN_INT = -2 ** 31
            if flag:
                return min(res, MAX_INT)
            else:
                return max(-res, MIN_INT)
        flag = True
        if a < 0 and b > 0 or a > 0 and b < 0:
            flag = False
        a, b = abs(a), abs(b)
        return recur(a, b)
```

54. 只出现一次的数
    - 32位保存+取余+判断负数
    - ```
      class Solution:
          def singleNumber(self, nums):
              dp = [0] * 32
              for item in nums:
                  for i in range(len(dp)):
                      dp[i] += item & 1
                      item >>= 1
              res = 0
              for i in range(32):
                  res <<= 1
                  res |= dp[31-i] % 3
              # 只出现一次的数是正数
              if dp[31] % 3 == 0:
                  return res
              # 只出现一次的数是负数
              else:
                  return ~(res ^ 0xffffffff)
      ```
55. 单词长度的最大乘积
    - 将单词表示成数字+与运算
    - ```
      class Solution:
          def maxProduct(self, words: List[str]) -> int:
              cur = []
              for i in range(len(words)):
                  temp = 0
                  for item in words[i]:
                      temp |= 1 << ord(item)-ord('a')
                  cur.append(temp)
              res = 0
              for i in range(len(words)-1):
                  for j in range(i+1, len(words)):
                      if cur[i] & cur[j] == 0:
                          res = max(res, len(words[i]*len(words[j])))
              return res
      ```
56. 乘积<k的子数组的个数
    - 滑动窗口
    - ```
      class Solution:
          def numSubarrayProductLessThanK(self, nums: List[int], k: int) -> int:
              if k <= 1:
                  return 0
              left = 0
              right = 0
              res = 0
              mul = 1
              while right < len(nums):
                  mul *= nums[right]
                  while left < right and mul >= k:
                      mul = mul / nums[left]
                      left += 1
                  # nums[left:right]中,以nums[right-1]元素结尾的,子数组的个数
                  res += right - left + 1
                  right += 1
              return res
      ```
57. 数据流中的中位数
    - ```
      class MedianFinder:
          import heapq
          def __init__(self):
              """
              initialize your data structure here.
              """
              self.max_stack = []
              self.min_stack = []
          def addNum(self, num: int) -> None:
              if len(self.max_stack) == len(self.min_stack):
                  heapq.heappush(self.max_stack, -heapq.heappushpop(self.min_stack, num))
              else:
                  heapq.heappush(self.min_stack, -heapq.heappushpop(self.max_stack, -num))
          def findMedian(self) -> float:
              if len(self.max_stack) > len(self.min_stack):
                  return -self.max_stack[0]
              else:
                  return float((-self.max_stack[0]+self.min_stack[0])/2)
      ```
58. 找到离给定的两个节点最近的节点
    - 

    - 
      输入：edges = [2,2,3,-1], node1 = 0, node2 = 1
      输出：2
      解释：从节点 0 到节点 2 的距离为 1 ，从节点 1 到节点 2 的距离为 1 。
      两个距离的较大值为 1 。我们无法找到一个比 1 更小的较大值，所以我们返回节点 2 。
    - 记录给定节点到其他节点的距离+依次比较距离+先max在min
    - ```
      class Solution:
      ```
      ```

```
    def closestMeetingNode(self, edges: List[int], node1: int, node2: int) -> int:
        N = len(edges)
        # 当前节点cur到每一个节点i的距离,不能到达则为无穷大值
        def recur(cur):
            dis = [float('inf')] * N
            d = 0
            # 当cur! =-1也就意味着还可以往下走
            # 当dis[cur] == float('inf')表示当前点还没有更新距离
            while cur >= 0 and dis[cur] == float('inf'):
                dis[cur] = d
                d += 1
                cur = edges[cur]
            return dis
        dis1 = recur(node1)
        dis2 = recur(node2)
        # 寻找node1和node2都可以到达的节点
        # 如果有一方不能到达，则对应位置为float('-inf')
        idx = -1
        min_dis = float('inf')
        for i in range(N):
            d = max(dis1[i], dis2[i])
            if d < min_dis:
                min_dis = d
                idx = i
        return idx
```

59. 丢棋子问题
    - 一座大楼有 n+1 层，地面算作第0层，最高的一层为第 n 层。已知棋子从第0层掉落肯定不会摔碎，从第 i 层掉落可能会摔碎，也可能不会摔碎。

      给定整数 n 作为楼层数，再给定整数 k 作为棋子数，返回如果想找到棋子不会摔碎的最高层数，即使在最差的情况下扔的最小次数。一次只能扔一个棋子。

      数据范围：$1 \le i \le n, k \le 10^6$
      要求：空间复杂度 $O(k)$，时间复杂度 $O(kn)$(n是最终返回的结果)

      示例1

      输入：10,1
      返回值：10
      说明：因为只有1根棋子，所以不得不从第1层开始一直试到第10层，在最差的情况下，即第10层是不会摔坏的最高层，最少也要扔10次
    - 看左边和左上角+dp[i][j]表示i个棋子扔j次，最多测多少层楼
    - ```
      class Solution:
          def solve(self , n: int, k: int) -> int:
              # write code here
              if n == 1:
                  return 1
              # dp[i][j]表示i次操作，j个棋子可以验证的最高楼层数
              dp = [[0 for _ in range(k+1)] for _ in range(n+1)]
              for j in range(1, k+1):
                  dp[1][j] = 1
              res = -1
              for i in range(2, n+1):
                  for j in range(1, k+1):
                      dp[i][j] = dp[i-1][j-1] + dp[i-1][j] + 1
                  if dp[i][j] >= n:
                      res = i
                      break
              return res
      ```

60. 递增子序列
    - 回溯法
    - ```
      输入：nums = [4,6,7,7]
      输出：[[4,6],[4,6,7],[4,6,7,7],[4,7],[4,7,7],[6,7],[6,7,7],[7,7]]
      ```
    - ```
      class Solution:
          def findSubsequences(self, nums: List[int]) -> List[List[int]]:
              if not nums:
                  return []
              res = []
              cur = []
              def recur(start, pre):
                  if len(cur) >= 2 and cur not in res:
                      res.append(cur[:])
                  for i in range(start, len(nums)):
                      if i > 0 and nums[i] < pre:
                          continue
                      cur.append(nums[i])
                      recur(i+1, nums[i])
                      cur.pop()
              recur(0, -200)
              return res
      ```

61. 最小未被占据椅子的编号
    - 两个堆
    - ```
      from heapq import heappop, heappush
      class Solution:
          def smallestChair(self, times: List[List[int]], targetFriend: int) -> int:
              n = len(times)
              arrival = []    # 到达操作的时刻和对应的人
              leaving = []    # 离开操作的时刻和对应的人
              for i in range(n):
                  arrival.append((times[i][0], i))
                  leaving.append((times[i][1], i))
              # 将到达与离开操作按照时间升序排序以方便模拟
              arrival.sort()
              leaving.sort()
              available = list(range(n))    # 未被占据的椅子
              seats = {}    # 每个人占据的椅子
              # 双指针模拟
              j = 0
              for time, person in arrival:
                  # 处理到达时间与之前的离开操作
                  # 将释放的椅子加入未被占据椅子中
                  while j < n and leaving[j][0] <= time:
                      heappush(available, seats[leaving[j][1]])
      ```

```
                j += 1
            # 处理到达操作
            # 占据当前编号最小且未被占据的椅子
            seat = heappop(available)
            seats[person] = seat
            if person == targetFriend:
                # 如果当前人为目标，则返回对应的椅子
                return seat
        return -1
```

62. 删除字符串中的所有相邻重复项
    - 栈+弹出相同的
    - ```
      class Solution:
          def removeDuplicates(self, s: str) -> str:
              stk = []
              for ch in s:
                  if stk and stk[-1] == ch:
                      stk.pop()
                  else:
                      stk.append(ch)
              return "".join(stk)
      ```
63. 长度为K的子数组的最大平均值
    - ```
      class Solution:
          def findMaxAverage(self, nums: List[int], k: int) -> float:
              maxTotal = total = sum(nums[:k])
              n = len(nums)
              for i in range(k, n):
                  total = total - nums[i - k] + nums[i]
                  maxTotal = max(maxTotal, total)
              return maxTotal / k
      ```
64. 二叉树的最大宽度
    - BFS+给每个节点编号
    - ```
      class Solution:
          def widthOfBinaryTree(self, root: Optional[TreeNode]) -> int:
              res = 1
              arr = [[root, 1]]
              while arr:
                  tmp = []
                  for node, index in arr:
                      if node.left:
                          tmp.append([node.left, index * 2])
                      if node.right:
                          tmp.append([node.right, index * 2 + 1])
                  res = max(res, arr[-1][1] - arr[0][1] + 1)
                  arr = tmp
              return res
      ```
65. 反转没对括号间的子串
    - 栈+遇到字母及左括号压入+遇到右括号弹出
    - ```
      class Solution:
          def reverseParentheses(self, s: str) -> str:
              stack = []
              for c in s:
                  if c != ')':
                      stack.append(c)
                  elif c == ')':
                      tmp = []
                      # 注意stack不为空才可以读取栈顶
                      while stack and stack[-1] != '(':
                          tmp.append(stack.pop())
                      if stack:
                          stack.pop() # 将左括号抛出
                      stack += tmp
              return "".join(stack)
      ```
66. 水仙花数
    - 一个三位数的个位、十位、百位数字的立方和等于本身
    - ```
      for n in range(100,1000):
          i = n // 100            #取百位
          j = n //10 %10          #取十位
          k = n % 10              #取个位
          if n == (i**3+j**3+k**3):
              print(n)
      ```
67. 求解方程
    - 把 “-” 替换成 “+-”， 然后按 “+” 切割， 就可以分离出等号左右两边的式子的每一项，然后就是左右是x系数/常数的相加走遍流程
    - ```
      class Solution:
          def solveEquation(self, equation: str) -> str:
              left, right = equation.split('=')
              factor = const = 0

              for part, group in enumerate([left, right]):
                  for tmp in group.replace('-', '+-').split('+'):
                      if 'x' in tmp:
                          if (tmp := tmp.replace('x', '')) in ('-', '', '+'):
                              tmp = f"{tmp}1"
                          factor += -int(tmp) if part else int(tmp)
                      elif tmp:
                          const += int(tmp) if part else -int(tmp)

              if factor == 0:
                  return "Infinite solutions" if const == 0 else "No solution"

              return f"x={const // factor}"
      ```
68. 插入排序
    - 在已排序数组中从后往前寻找当前待插入元素的位置进行插入，直到整个数组有序
    - ```
      def insertionSort(arr):
          for i in range(1, len(arr)):
              key = arr[i]
              j = i-1
              while j >=0 and key < arr[j] :
                      arr[j+1] = arr[j]
                      j -= 1
              arr[j+1] = key
          return arr
      ```
69. 冒泡排序
      ```

```python
def bubbleSort(arr):
    n = len(arr)
    # 遍历所有数组元素
    for i in range(n):
        # Last i elements are already in place
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1] :
                arr[j], arr[j+1] = arr[j+1], arr[j]
    return arr
```