

1.不同的子序列
 2.最长公共子序列
 3.编辑距离
 4.爬楼梯
 5.零钱兑换
 6.零钱兑换2
 7.打家劫舍
 8.打家劫舍2
 9.交错字符串
 10.最长递增子序列
 11.最长有效括号
 12.最长重复子数组
 13.最长回文子串
 14.最长回文子序列
 15.最大子序和
 16.最大矩形
 17.最大正方形
 18.回文子串
 19.分割回文串2
 20.分割回文串3
 21.区域和检索-数组不可变
 22.区域和检索-矩阵不可变
 23.不同路径
 24.不同路径2
 25.三角形最小路径和
 26.最小路径和
 27.完全平方数
 28.跳跃游戏
 29.跳跃游戏2
 30.买卖股票的最佳时机
 31.买卖股票的最佳时机2
 32.买卖股票的最佳时机3
 33.买卖股票的最佳时机4
 34.买卖股票的最佳时机含冷冻期
 35.买卖股票的最佳时机含手续费
 36.乘积最大子数组
 37.任务调度器
 38.解码方法
 39.赛车
 40.戳气球
 41.最小覆盖子串
 42.学生出勤记录2
 43.分割数组的最大值
 44.青蛙过河
 45.矩形区域不超过K的最大数值和
 46.除数博弈
 47.鸡蛋掉落

115. 不同的子序列

难度 困难 山 569 ☆ 收藏 分享 切换为英文 接收动态 反馈

给定一个字符串 s 和一个字符串 t ，计算在 s 的子序列中 t 出现的个数。

字符串的一个 **子序列** 是指，通过删除一些（也可以不删除）字符且不干扰剩余字符相对位置所形成的一个新字符串。（例如，“ACE”是“ABCDE”的一个子序列，而“AEC”不是）

题目数据保证答案符合 32 位带符号整数范围。

示例 1：

输入: $s = "rabbbit"$, $t = "rabbit"$

输出: 3

解释:

如下图所示，有 3 种可以从 s 中得到 “rabbit”的方案。

rabbbit

rabbbit

rabbbit

```
class Solution:
    def numDistinct(self, s: str, t: str) -> int:
        n1 = len(t)
        n2 = len(s)
        dp = [[0] * (n2 + 1) for _ in range(n1 + 1)]
        for j in range(n2 + 1):
            dp[0][j] = 1
        for i in range(1, n1 + 1):
            for j in range(1, n2 + 1):
                if t[i - 1] == s[j - 1]:
                    dp[i][j] = dp[i - 1][j - 1] + dp[i][j - 1]
                else:
                    dp[i][j] = dp[i][j - 1]
        return dp[-1][-1]
```

1143. 最长公共子序列

难度 中等 636 收藏 分享 切换为英文 接收动态 反馈

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。如果不包含子序列，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。

示例 2：

输入: `text1 = "abc"`, `text2 = "abc"`

输出: 3

解释: 最长公共子序列是 "abc"，它的长度为 3。

```
def longestCommonSubsequence1(self, text1: str, text2: str) ->
    m, n = len(text1), len(text2)
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if text1[i - 1] == text2[j - 1]:
                dp[i][j] = 1 + dp[i - 1][j - 1]
            else:
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])

    res = ''
    i = m
    j = n
    while i and j and dp[i][j] >= 1:
        if text1[i - 1] == text2[j - 1]:
            res += text1[i - 1]
            i -= 1
            j -= 1
        elif dp[i - 1][j] >= dp[i][j - 1]:
            i -= 1
        else:
            j -= 1

    res = res[::-1]
    return dp[-1][-1]
```

72. 编辑距离

难度 困难 1743 收藏 分享 切换为英文 接收动态

给你两个单词 `word1` 和 `word2`，请你计算出将 `word1` 转换成 `word2` 所使用的最少操

你可以对一个单词进行如下三种操作：

- 插入一个字符
- 删一个字符
- 替换一个字符

示例 1：

输入: `word1 = "horse", word2 = "ros"`

输出: 3

解释:

```
horse -> rorse (将 'h' 替换为 'r')
rorse -> rose (删除 'r')
rose -> ros (删除 'e')
```

```
class Solution:
    def minDistance(self, word1: str, word2: str) -> int:
        n1 = len(word1)
        n2 = len(word2)
        dp = [[0] * (n2 + 1) for _ in range(n1 + 1)]
        for j in range(1, n2 + 1):
            dp[0][j] = dp[0][j - 1] + 1
        for i in range(1, n1 + 1):
            dp[i][0] = dp[i - 1][0] + 1
        for i in range(1, n1 + 1):
            for j in range(1, n2 + 1):
                if word1[i - 1] == word2[j - 1]:
                    dp[i][j] = dp[i - 1][j - 1]
                else:
                    dp[i][j] = min(dp[i - 1][j], dp[i][j - 1], dp[i - 1][j - 1]) + 1
        return dp[-1][-1]
```

70. 爬楼梯

难度 简单 1792 收藏 分享 切换为英文 接收动态 反馈

假设你正在爬楼梯。需要 n 阶你才能到达楼顶。

每次你可以爬 1 或 2 个台阶。你有多少种不同的方法可以爬到楼顶呢？

注意: 给定 n 是一个正整数。

示例 1:

输入: 2

输出: 2

解释: 有两种方法可以爬到楼顶。

1. 1 阶 + 1 阶

2. 2 阶

```
1 class Solution:
2     def climbStairs(self, n: int) -> int:
3         a, b = 1, 1
4         for i in range(n):
5             a, b = b, a + b
6         return a
```

322. 零钱兑换

难度 中等 1414 收藏 分享 切换为英文 接收动态 反馈

给你一个整数数组 `coins`，表示不同面额的硬币；以及一个整数 `amount`，表示总金额。

计算并返回可以凑成总金额所需的 **最少的硬币个数**。如果没有任何一种硬币组合能组成总金额，返回 `-1`。

你可以认为每种硬币的数量是无限的。

示例 1:

```
输入: coins = [1, 2, 5], amount = 11
输出: 3
解释: 11 = 5 + 5 + 1
```

示例 2:

```
输入: coins = [2], amount = 3
输出: -1
```

```
1 class Solution:
2     def coinChange(self, coins: List[int], amount: int) -> int:
3         dp = [float('inf')] * (amount + 1)
4         dp[0] = 0
5         for coin in coins:
6             for x in range(coin, amount + 1):
7                 dp[x] = min(dp[x], dp[x - coin] + 1)
8         return dp[-1] if dp[-1] != float('inf') else -1
```

518. 零钱兑换 II

难度 中等 574 收藏 分享 切换为英文 接收动态 反馈

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
解释: 有四种方式可以凑成总金额:
5=5
5=2+2+1
5=2+1+1+1
5=1+1+1+1+1
```

示例 2:

```
输入: amount = 3, coins = [2]
输出: 0
解释: 只用面额 2 的硬币不能凑成总金额 3。
```

```
class Solution:
    def change(self, amount: int, coins: List[int]) -> int:
        dp = [0] * (amount + 1)
        dp[0] = 1
        for coin in coins:
            for x in range(coin, amount + 1):
                dp[x] += dp[x - coin]
        return dp[amount]
```

198. 打家劫舍

难度 中等 1585 收藏 分享 切换为英文 接收动态 反馈

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**在不触动警报装置的情况下**，一夜之间能偷窃到的最高金额。

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。

示例 2：

输入: [2,7,9,3,1]

输出: 12

解释: 偷窃 1 号房屋 (金额 = 2)，偷窃 3 号房屋 (金额 = 9)，接着偷窃 5 号房屋 (金额 = 1)。
偷窃到的最高金额 = 2 + 9 + 1 = 12。

```
class Solution:
```

```
    def rob(self, nums: List[int]) -> int:
        pre, cur = 0, 0
        for i in nums:
            pre, cur = cur, max(cur, pre + i)
        return cur
```

213. 打家劫舍 II

难度 中等 744 收藏 分享 切换为英文 接收动态 反馈

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)，因为它们是相邻的。

示例 2：

输入: nums = [1,2,3,1]

输出: 4

解释: 你可以先偷窃 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:
        def my_rob(nums):
            prev, curr = 0, 0
            for i in nums:
                prev, curr = curr, max(curr, prev + i)
            return curr

        return max(my_rob(nums[:-1]), my_rob(nums[1:])) if len(nums) > 1 else nums[0]
```

97. 交错字符串

难度 中等 山 512 ☆ 收藏 ⚡ 分享 切换为英文 接收动态 反馈

给定三个字符串 s_1 、 s_2 、 s_3 ，请你帮忙验证 s_3 是否是由 s_1 和 s_2 交错 组成的。

两个字符串 s 和 t 交错 的定义与过程如下，其中每个字符串都会被分割成若干 非空 子字符串：

- $s = s_1 + s_2 + \dots + s_n$
- $t = t_1 + t_2 + \dots + t_m$
- $|n - m| \leq 1$
- 交错 是 $s_1 + t_1 + s_2 + t_2 + s_3 + t_3 + \dots$ 或者 $t_1 + s_1 + t_2 + s_2 + \dots$
 \dots

提示： $a + b$ 意味着字符串 a 和 b 连接。

示例 1：

```
输入: s1 = "aabcc", s2 = "dbbca", s3 = "aadbbcbcac"
输出: true
```

示例 2：

```
输入: s1 = "aabcc", s2 = "dbbca", s3 = "aadbbbaccc"
输出: false
```

class Solution:

```
def isInterleave(self, s1: str, s2: str, s3: str) -> bool:
    len1 = len(s1)
    len2 = len(s2)
    len3 = len(s3)
    if len1 + len2 != len3:
        return False
    dp = [[False] * (len2 + 1) for _ in range(len1 + 1)]
    dp[0][0] = True

    for i in range(1, 1 + len1):
        dp[i][0] = dp[i - 1][0] and s1[i - 1] == s3[i - 1]

    for j in range(1, len2 + 1):
        dp[0][j] = dp[0][j - 1] and s2[j - 1] == s3[j - 1]

    for i in range(1, len1 + 1):
        for j in range(1, len2 + 1):
            dp[i][j] = (dp[i - 1][j] and s1[i - 1] == s3[i + j - 1]) or (dp[i][j - 1] and s2[j - 1] == s3[i + j - 1])
    return dp[-1][-1]
```

最长递增子序列

300. 最长递增子序列

难度 中等 1774 收藏 分享 切换为英文 接收动态 反馈

给你一个整数数组 `nums`，找到其中最长严格递增子序列的长度。

子序列是由数组派生而来的序列，删除（或不删除）数组中的元素而不改变其余元素的顺序。

`[3,6,2,7]` 是数组 `[0,3,1,6,2,2,7]` 的子序列。

示例 1：

输入: `nums = [10,9,2,5,3,7,101,18]`

输出: 4

解释: 最长递增子序列是 `[2,3,7,101]`，因此长度为 4。

示例 2：

输入: `nums = [0,1,0,3,2,3]`

输出: 4

示例 3：

输入: `nums = [7,7,7,7,7,7,7]`

输出: 1

```
class Solution:
```

```
    def LIS(self, nums):
        n = len(nums)
        tail = [nums[0]]
        dp = [1] * n
        for i in range(1, n):
            if nums[i] > tail[-1]:
                tail.append(nums[i])
                dp[i] = len(tail)
            continue
```

```
left, right = 0, len(tail) - 1
```

```
while left < right:
```

```
    mid = (left + right) // 1
```

```
    if tail[mid] < nums[i]:
```

```
        left = mid + 1
```

```
    else:
```

```
        right = mid
```

```
tail[left] = nums[i]
```

```
dp[i] = left + 1
```

```
res = []
```

```
j = len(tail)
```

```
i = n - 1
```

```
while j >= 1 and i >= 0:
```

```
    if dp[i] == j:
```

```
        res.append(nums[i])
```

```
        j -= 1
```

```
i -= 1
```

```
return res[::-1]
```

32. 最长有效括号

难度 困难 1400 收藏 分享 切换为英文 接收动态 反馈

给你一个只包含 '(' 和 ')' 的字符串，找出最长有效（格式正确且连续）括号子串的长度

示例 1:

输入: s = "(()"

输出: 2

解释: 最长有效括号子串是 "()"

示例 2:

输入: s = ")()())"

输出: 4

解释: 最长有效括号子串是 "()()"

示例 3:

输入: s = ""

输出: 0

class Solution:

```
def longestValidParentheses(self, s: str) -> int:
```

```
    n = len(s)
```

```
    if n == 0:
```

```
        return 0
```

```
    dp = [0] * n
```

```
    res = 0
```

```
    for i in range(n):
```

```
        if i > 0 and s[i] == ")":
```

```
            if s[i - 1] == "(":
```

```
                dp[i] = dp[i - 2] + 2
```

```
            elif s[i - 1] == ")" and i - dp[i - 1] - 1 >= 0 and
```

```
                dp[i] = dp[i - 1] + 2 + dp[i - dp[i - 1] - 2]
```

```
            if dp[i] > res:
```

```
                res = dp[i]
```

```
    return res
```

718. 最长重复子数组

难度 中等 505 收藏 分享 切换为英文 接收动态 反馈

给两个整数数组 A 和 B，返回两个数组中公共的、长度最长的子数组的长度。

示例:

输入:

A: [1,2,3,2,1]

B: [3,2,1,4,7]

输出: 3

解释:

长度最长的公共子数组是 [3, 2, 1]。

```

class Solution:
    def findLength(self, nums1: List[int], nums2: List[int]) -> int:
        n = len(nums1)
        m = len(nums2)
        ans = 0
        dp = [[0] * (m + 1) for _ in range(n + 1)]
        for i in range(1, n + 1):
            for j in range(1, m + 1):
                if nums1[i - 1] == nums2[j - 1]:
                    dp[i][j] = dp[i - 1][j - 1] + 1
                    ans = max(ans, dp[i][j])
        return ans

```

5. 最长回文子串

难度 中等 3928 收藏 分享 切换为英文 接收动态

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。

示例 2：

输入: `s = "cbbd"`
输出: "bb"

示例 3：

输入: `s = "a"`
输出: "a"

```

class Solution:
    def longestPalindrome(self, s: str) -> str:
        size = len(s)
        if size < 2:
            return s
        dp = [[False] * size for _ in range(size)]
        max_len = 1
        start = 0
        for i in range(size):
            dp[i][i] = True
        for j in range(1, size):
            for i in range(0, j):
                if s[i] == s[j] and (j - i < 3 or dp[i + 1][j - 1]):
                    if dp[i][j] and j - i + 1 > max_len:
                        max_len = j - i + 1
                        start = i
        return s[start:start + max_len]

```

516. 最长回文子序列

难度 中等 492 收藏 分享 切换为英文 接收动态 反馈

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

示例 1：

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

示例 2：

输入: `s = "cbbd"`
输出: 2
解释: 一个可能的最长回文子序列为 "bb" 。

```
class Solution:
    def longestPalindromeSubseq(self, s: str) -> int:
        n = len(s)
        dp = [[0] * n for _ in range(n)]
        for i in range(n):
            dp[i][i] = 1
        for j in range(1, n):
            for i in range(j - 1, -1, -1):
                dp[i][j] = dp[i + 1][j - 1] + 2 if s[i] == s[j] else dp[i + 1][j - 1]
        return dp[0][-1]
```

剑指 Offer 42. 连续子数组的最大和

难度 简单 362 收藏 分享 切换为英文 接收动态 反馈

输入一个整型数组，数组中的一个或连续多个整数组成一个子数组。求所有子数组的和的最大值，要求时间复杂度为O(n)。

示例1：

输入: nums = [-2,1,-3,4,-1,2,1,-5,4]

输出: 6

解释: 连续子数组 [4,-1,2,1] 的和最大，为 6。

```
class Solution:
    def maxSubArray(self, nums: List[int]) -> int:
        n = len(nums)
        if n == 0:
            return 0
        pre = nums[0]
        res = pre
        for i in range(1, n):
            pre = max(nums[i], pre + nums[i])
            res = max(res, pre)
        return res
```

85. 最大矩形

难度 困难 984 收藏 分享 切换为英文 接收动态 反馈

给定一个仅包含 0 和 1、大小为 rows x cols 的二维二进制矩阵，找出只包含 1 的最大矩形，并返回其面积。

示例 1：

1	0	1	0	0
1	0	1	1	1
1	1	1	1	1
1	0	0	1	0

输入: matrix = [[1,0,1,0,0],[1,0,1,1,1],[1,1,1,1,1],[1,0,0,1,0]]

输出: 6

解释: 最大矩形如上图所示。

```

class Solution:
    def maximalRectangle(self, matrix: List[List[str]]) -> int:
        maxarea = 0

        dp = [[0] * len(matrix[0]) for _ in range(len(matrix))]
        for i in range(len(matrix)):
            for j in range(len(matrix[0])):
                if matrix[i][j] == '0':
                    continue

                width = dp[i][j] = dp[i][j - 1] + 1 if j else 1

                for k in range(i, -1, -1):
                    width = min(width, dp[k][j])
                    maxarea = max(maxarea, width * (i - k + 1))
        return maxarea

```

221. 最大正方形

难度 中等 832 收藏 分享 切换为英文 接收动态 反馈

在一个由 '0' 和 '1' 组成的二维矩阵内，找到只包含 '1' 的最大正方形，并返回其面积。

示例 1：

1	0	1	0	0
1	0	1	1	1
1	1	1	1	1
1	0	0	1	0

输入: matrix = [["1","0","1","0","0"], ["1","0","1","1","1"], ["1","1","1","1","1"], ["1","0","0","1","0"]]

输出: 4

```

class Solution:
    def maximalSquare(self, matrix: List[List[str]]) -> int:
        if not matrix or len(matrix) == 0 or len(matrix[0]) == 0:
            return 0
        rows, cols = len(matrix), len(matrix[0])
        dp = [[0 for i in range(cols + 1)] for j in range(rows + 1)]

        maxSide = 0
        for i in range(rows):
            for j in range(cols):
                if matrix[i][j] == "1":
                    dp[i + 1][j + 1] = min(dp[i][j], dp[i + 1][j], dp[i][j + 1]) + 1
                    maxSide = max(maxSide, dp[i + 1][j + 1])
        return maxSide * maxSide

```

647. 回文子串

难度 中等 647 收藏 分享 切换为英文 接收动态 反馈

给定一个字符串，你的任务是计算这个字符串中有多少个回文子串。

具有不同开始位置或结束位置的子串，即使是由相同的字符组成，也会被视作不同的子串。

示例 1:

输入: "abc"
输出: 3
解释: 三个回文子串: "a", "b", "c"

示例 2:

输入: "aaa"
输出: 6
解释: 6个回文子串: "a", "a", "a", "aa", "aa", "aaa"

```

class Solution:
    def countSubstrings(self, s: str) -> int:
        n = len(s)
        dp = [[False] * n for _ in range(n)]
        ans = 0
        for i in range(n):
            dp[i][i] = True
            ans += 1
        for j in range(1, n):
            for i in range(j):
                if dp[i][j] = s[i] == s[j] and (j - i < 3 or dp[i + 1][j - 1]):
                    ans += (1 if dp[i][j] else 0)
        return ans

```

132. 分割回文串 II

难度 困难 455 收藏 分享 切换为英文 接收动态 反馈

给你一个字符串 s ，请你将 s 分割成一些子串，使每个子串都是回文。

返回符合要求的 **最少分割次数**。

示例 1:

输入: $s = "aab"$
输出: 1
解释: 只需一次分割就可将 s 分割成 $["aa", "b"]$ 这样两个回文子串。

示例 2:

输入: $s = "a"$
输出: 0

示例 3:

输入: $s = "ab"$
输出: 1

```

class Solution:
    def minCut(self, s: str) -> int:
        n = len(s)
        if n < 2:
            return 0
        dp = [i for i in range(n)]
        check = [[False for _ in range(n)] for _ in range(n)]
        for i in range(n):
            check[i][i] = True

        for j in range(1, n):
            for i in range(j):
                check[i][j] = s[i] == s[j] and (j - i < 3 or check[i+1][j-1])

        for i in range(1, n):
            if check[0][i]:
                dp[i] = 0
            else:
                dp[i] = min([dp[j] + 1 for j in range(i) if check[j+1][i]])
        return dp[-1]

```

1278. 分割回文串 III

难度 困难 82 收藏 分享 切换为英文 接收动态 反馈

给你一个由小写字母组成的字符串 s ，和一个整数 k 。

请你按下面的要求分割字符串：

- 首先，你可以将 s 中的部分字符修改为其他的小写英文字母。
- 接着，你需要把 s 分割成 k 个非空且不相交的子串，并且每个子串都是回文串。

请返回以这种方式分割字符串所需修改的最少字符数。

示例 1：

输入: $s = "abc"$, $k = 2$

输出: 1

解释: 你可以把字符串分割成 "ab" 和 "c"，并修改 "ab" 中的 1 个字符，将它变成

示例 2：

输入: $s = "aabbc"$, $k = 3$

输出: 0

解释: 你可以把字符串分割成 "aa"、"bb" 和 "c"，它们都是回文串。

```

class Solution:
    def palindromePartition(self, s: str, k: int) -> int:
        n = len(s)
        cost = [[0] * n for _ in range(n)]
        for j in range(1, n):
            for i in range(j):
                cost[i][j] = cost[i + 1][j - 1] + (0 if s[i] == s[j] else 1)

        f = [[10 ** 9] * (k + 1) for _ in range(n + 1)]
        f[0][0] = 0
        for i in range(1, n + 1):
            for j in range(1, min(i, k) + 1):
                if j == 1:
                    f[i][j] = cost[0][i - 1]
                else:
                    for i0 in range(j - 1, i):
                        f[i][j] = min(f[i][j], f[i0][j - 1] + cost[i0][i])
        return f[n][k]

```

303. 区域和检索 - 数组不可变

难度 简单 353 收藏 分享 切换为英文 接收动态 反馈

给定一个整数数组 `nums`，求出数组从索引 `i` 到 `j` (`i ≤ j`) 范围内元素的总和，包含点。

实现 `NumArray` 类：

- `NumArray(int[] nums)` 使用数组 `nums` 初始化对象
- `int sumRange(int i, int j)` 返回数组 `nums` 从索引 `i` 到 `j` (`i ≤ j`) 范围内元素的总和，包含 `i`、`j` 两点（也就是 `sum(nums[i], nums[i + 1], ..., nums[j])`）

示例：

输入：

```

["NumArray", "sumRange", "sumRange", "sumRange"]
[[[-2, 0, 3, -5, 2, -1]], [0, 2], [2, 5], [0, 5]]

```

输出：

```
[null, 1, -1, -3]
```

解释：

```

NumArray numArray = new NumArray([-2, 0, 3, -5, 2, -1]);
numArray.sumRange(0, 2); // return 1((-2) + 0 + 3)
numArray.sumRange(2, 5); // return -1(3 + (-5) + 2 + (-1))
numArray.sumRange(0, 5); // return -3((-2) + 0 + 3 + (-5) + 2 + (-1))

```

```

1 class NumArray:
2
3     def __init__(self, nums: List[int]):
4         self.dp = [0]
5         for num in nums:
6             self.dp.append(self.dp[-1] + num)
7
8     def sumRange(self, i: int, j: int) -> int:
9         return self.dp[j + 1] - self.dp[i]
10
11
12 # Your NumArray object will be instantiated and called as such:
13 # obj = NumArray(nums)
14 # param_1 = obj.sumRange(i,j)

```

304. 二维区域和检索 - 矩阵不可变

难度 中等 山 285 收藏 分享 切换为英文 接收动态 反馈

给定一个二维矩阵 `matrix`，以下类型的多个请求：

- 计算其子矩形范围内元素的总和，该子矩阵的左上角为 `(row1, col1)`，右下角为 `(row2, col2)`。

实现 `NumMatrix` 类：

- `NumMatrix(int[][] matrix)` 给定整数矩阵 `matrix` 进行初始化
- `int sumRegion(int row1, int col1, int row2, int col2)` 返回左上角 `(row1, col1)`、右下角 `(row2, col2)` 的子矩阵的元素总和。

示例 1：

3	0	1	4	2
5	6	3	2	1
1	2	0	1	5
4	1	0	1	7
1	0	3	0	5

```

class NumMatrix:

    def __init__(self, matrix: List[List[int]]):
        if not matrix or not matrix[0]:
            pass
        row = len(matrix)
        col = len(matrix[0])
        self.dp = [[0] * (col + 1) for _ in range(row + 1)]
        for i in range(row + 1):
            for j in range(col + 1):
                self.dp[i][j] = self.dp[i][j - 1] + self.dp[i - 1][
                    [i - 1][j - 1]

    def sumRegion(self, row1: int, col1: int, row2: int, col2: int)
        return self.dp[row2 + 1][col2 + 1] - self.dp[row2 + 1][col1]
        dp[row1][col1]

# Your NumMatrix object will be instantiated and called as such:
# obj = NumMatrix(matrix)
# param_1 = obj.sumRegion(row1,col1,row2,col2)

```

62. 不同路径

难度 中等 **1074** 收藏 分享 切换为英文 接收动态 反馈

一个机器人位于一个 $m \times n$ 网格的左上角（起始点在下图中标记为 "Start"）。

机器人每次只能向下或者向右移动一步。机器人试图达到网格的右下角（在下图中标记为 "Finish"）。问总共有多少条不同的路径？

示例 1:



输入: $m = 3$, $n = 7$

输出: 28

```

class Solution:
    def uniquePaths(self, m: int, n: int) -> int:
        cur = [1] * n
        for i in range(1, m):
            for j in range(1, n):
                cur[j] += cur[j - 1]
        return cur[-1]

```

63. 不同路径 II

难度 中等 难度 600 收藏 分享 切换为英文 接收动态 反馈

一个机器人位于一个 $m \times n$ 网格的左上角（起始点在下图中标记为“Start”）。

机器人每次只能向下或者向右移动一步。机器人试图达到网格的右下角（在下图中标记为“Finish”）。

现在考虑网格中有障碍物。那么从左上角到右下角将会有多少条不同的路径？



网格中的障碍物和空位置分别用 1 和 0 来表示。

```
class Solution:
    def uniquePathsWithObstacles(self, obstacleGrid: List[List[int]]):
        m = len(obstacleGrid)
        n = len(obstacleGrid[0])
        dp = [1] + [0] * n
        for i in range(0, m):
            for j in range(0, n):
                dp[j] = 0 if obstacleGrid[i][j] else dp[j] + dp[i-1][j]
        return dp[-2]
```

120. 三角形最小路径和

难度 中等 难度 809 收藏 分享 切换为英文 接收动态 反馈

给定一个三角形 triangle，找出自顶向下的最小路径和。

每一步只能移动到下一行中相邻的结点上。**相邻的结点** 在这里指的是 **下标** 与 **上一层结点下标** 等于 **上一层结点下标 + 1** 的两个结点。也就是说，如果正位于当前行的下标 i ，那么下一步到下一行的下标 i 或 $i + 1$ 。

示例 1:

输入: triangle = [[2],[3,4],[6,5,7],[4,1,8,3]]

输出: 11

解释: 如下面简图所示:

```
2
3 4
6 5 7
4 1 8 3
```

自顶向下的最小路径和为 11 (即, $2 + 3 + 5 + 1 = 11$)。

示例 2:

输入: triangle = [[-10]]

输出: -10

```
class Solution:
    def minimumTotal(self, triangle: List[List[int]]) -> int:
        if not triangle:
            return
        res = triangle[-1]
        for row in range(len(triangle) - 2, -1, -1):
            for col in range(len(triangle[row])):
                res[col] = min(res[col], res[col + 1]) + triangle[row][col]
        return res[0]
```

64. 最小路径和

难度 中等 1 收藏 分享 切换为英文 接收动态 反馈

给定一个包含非负整数的 $m \times n$ 网格 `grid`，请找出一条从左上角到右下角的路径，使得总和为最小。

说明：每次只能向下或者向右移动一步。

示例 1：

1	3	1
1	5	1
4	2	1

输入: `grid = [[1,3,1],[1,5,1],[4,2,1]]`

输出: 7

解释: 因为路径 1→3→1→1→1 的总和最小。

Solution:

```
def minPathSum(self, grid: List[List[int]]) -> int:
    for i in range(len(grid)):
        for j in range(len(grid[0])):
            if i == j == 0:
                continue
            elif i == 0:
                grid[i][j] += grid[i][j-1]
            elif j == 0:
                grid[i][j] += grid[i-1][j]
            else:
                grid[i][j] += min(grid[i-1][j], grid[i][j-1])
    return grid[-1][-1]
```

279. 完全平方数

难度 中等 1 收藏 分享 切换为英文 接收动态 反馈

给定正整数 n ，找到若干个完全平方数（比如 `1, 4, 9, 16, ...`）使得它们的和等于 n 。组成和的完全平方数的个数最少。

给你一个整数 `n`，返回和为 `n` 的完全平方数的 **最少数量**。

完全平方数 是一个整数，其值等于另一个整数的平方；换句话说，其值等于一个整数自乘的结果。`1`、`4`、`9` 和 `16` 都是完全平方数，而 `3` 和 `11` 不是。

示例 1：

输入: `n = 12`

输出: 3

解释: $12 = 4 + 4 + 4$

示例 2：

输入: `n = 13`

输出: 2

解释: $13 = 4 + 9$

```
class Solution:
    def numSquares(self, n: int) -> int:
        dp = [i for i in range(n + 1)]
        for i in range(1, n + 1):
            for j in range(1, n + 1):
                if i - j * j < 0:
                    break
                else:
                    dp[i] = min(dp[i], dp[i - j * j] + 1)
        return dp[-1]
```

55. 跳跃游戏

难度 中等 1294 收藏 分享 切换为英文 接收动态 反馈

给定一个非负整数数组 `nums`，你最初位于数组的 **第一个下标**。

数组中的每个元素代表你在该位置可以跳跃的最大长度。

判断你是否能够到达最后一个下标。

示例 1:

输入: `nums = [2,3,1,1,4]`

输出: `true`

解释: 可以先跳 1 步，从下标 0 到达下标 1，然后再从下标 1 跳 3 步到达最后一个

示例 2:

输入: `nums = [3,2,1,0,4]`

输出: `false`

解释: 无论怎样，总会到达下标为 3 的位置。但该下标的最大跳跃长度是 0，所以永远到达最后一个下标。

```
class Solution:
    def canJump(self, nums: List[int]) -> bool:
        if nums == [0]:
            return True

        maxDist = 0
        end_index = len(nums) - 1
        for i, jump in enumerate(nums):
            if maxDist >= i and i + jump > maxDist:
                maxDist = i + jump
            if maxDist >= end_index:
                return True

        return False
```

45. 跳跃游戏 II

难度 中等 1095 收藏 分享 切换为英文 接收动态 反馈

给你一个非负整数数组 `nums`，你最初位于数组的第一个位置。

数组中的每个元素代表你在该位置可以跳跃的最大长度。

你的目标是使用最少的跳跃次数到达数组的最后一个位置。

假设你总是可以到达数组的最后一个位置。

示例 1:

输入: `nums = [2,3,1,1,4]`

输出: `2`

解释: 跳到最后一个位置的最小跳跃数是 2。

从下标为 0 跳到下标为 1 的位置，跳 1 步，然后跳 3 步到达数组的最后一个位

示例 2:

输入: `nums = [2,3,0,1,4]`

输出: `2`

```
class Solution:
    def jump(self, nums: List[int]) -> int:
        end = 0
        maxpos = 0
        steps = 0
        n = len(nums)
        for i in range(n - 1):
            maxpos = max(maxpos, i + nums[i])
            if i == end:
                end = maxpos
                steps += 1
        return steps
```

121. 买卖股票的最佳时机

难度 简单 1766 收藏 分享 切换为英文 接收动态

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。你可以尽可能地完成更多的交易（多次买卖一支股票）。

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，这样就能得到利润 = $6 - 1 = 5$ 。

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，你不能在同一天买入和卖出股票。

示例 2:

输入: `prices = [7,6,4,3,1]`

输出: `0`

解释: 在这种情况下，没有交易完成，所以最大利润为 `0`。

```
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        minprice = 10 ** 9
        maxprofit = 0
        for price in prices:
            minprice = min(price, minprice)
            maxprofit = max(maxprofit, price - minprice)
        return maxprofit
```

122. 买卖股票的最佳时机 II

难度 简单 1307 收藏 分享 切换为英文 接收动态 反馈

给定一个数组 `prices`，其中 `prices[i]` 是一支给定股票第 `i` 天的价格。

设计一个算法来计算你所能获取的最大利润。你可以尽可能地完成更多的交易（多次买卖一支股票）。

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 = [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，这样就能得到利润 = $5 - 1 = 4$ 。

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，这样就能得到利润 = $6 - 3 = 3$ 。

示例 2:

输入: `prices = [1,2,3,4,5]`

输出: `4`

解释: 在第 1 天 (股票价格 = 1) 的时候买入，在第 5 天 (股票价格 = 5) 的时候卖出，这样就能得到利润 = $5 - 1 = 4$ 。

注意你不能在第 1 天和第 2 天接连购买股票，之后再将它们卖出。因为这样属于多笔交易，你必须在再次购买前出售掉之前的股票。

```
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        return sum(b - a for a, b in zip(prices, prices[1:])) if
```

123. 买卖股票的最佳时机 III

难度 困难 837 收藏 分享 切换为英文 接收动态 反馈

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) 卖出，这笔交易所能获得利润 = $4 - 1 = 3$ 。

示例 2：

输入: prices = [1,2,3,4,5]

输出: 4

解释: 在第 1 天 (股票价格 = 1) 的时候买入，在第 5 天 (股票价格 = 5) 的时候卖出所能获得利润 = $5 - 1 = 4$ 。

注意你不能在第 1 天和第 2 天接连购买股票，之后再将它们卖出。

因为这样属于同时参与了多笔交易，你必须在再次购买前出售掉之前的股票。

```
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        n = len(prices)
        if n <= 1:
            return 0
        k = 2
        dp = [[[0, 0] for _ in range(k + 1)] for _ in range(n)]
        for i in range(n):
            for j in range(k, -1, -1):
                if i == 0:
                    dp[i][j][0] = 0
                    dp[i][j][1] = -prices[0]
                elif j == 0:
                    dp[i][j][0] = 0
                    dp[i][j][1] = float('-inf')
                else:
                    dp[i][j][0] = max(dp[i - 1][j][0], dp[i - 1][j][1])
                    dp[i][j][1] = max(dp[i - 1][j][1], dp[i - 1][j - 1][0])
        return dp[-1][k][0]
```

```
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        n = len(prices)
        if n <= 1:
            return 0
        k = 2
        dp = [[0 if i == 0 else -prices[0] for i in range(2)] for _ in range(n)]
        for i in range(1, n):
            for j in range(k, 0, -1):
                dp[j][0] = max(dp[j][0], dp[j][1] + prices[i])
                dp[j][1] = max(dp[j][1], dp[j - 1][0] - prices[i])
        return dp[k][0]
```

188. 买卖股票的最佳时机 IV

难度 困难 557 收藏 分享 切换为英文 接收动态 反馈

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 = $4 - 2 = 2$ 。

示例 2：

输入: $k = 2$, `prices` = [3,2,6,5,0,3]

输出: 7

解释: 在第 2 天 (股票价格 = 2) 的时候买入，在第 3 天 (股票价格 = 6) 的时候卖出。这笔交易所能获得利润 = $6 - 2 = 4$ 。

随后，在第 5 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 = $3 - 0 = 3$ 。

class Solution:

```
def maxProfit(self, k: int, prices: List[int]) -> int:
    n = len(prices)

    if k >= n / 2:
        return sum(b - a for a, b in zip(prices, prices[1:]))

    dp = [[0 if i == 0 else -prices[0] for i in range(2)] for _ in range(k + 1)]
    for i in range(1, n):
        for j in range(k, 0, -1):
            dp[j][0] = max(dp[j][0], dp[j][1] + prices[i])
            dp[j][1] = max(dp[j][1], dp[j - 1][0] - prices[i])
    return dp[k][0]
```

309. 最佳买卖股票时机含冷冻期

难度 中等 852 收藏 分享 切换为英文 接收动态 反馈

给定一个整数数组，其中第 i 个元素代表了第 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（多次买卖股票）：

- 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）。
- 卖出股票后，你无法在第二天买入股票 (即冷冻期为 1 天)。

示例：

输入: [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]

```
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        if not prices:
            return 0
        n = len(prices)
        dp_i0, dp_i1, dp_pre0 = 0, float('-inf'), 0
        for i in range(n):
            tmp = dp_i0
            dp_i0 = max(dp_i0, dp_i1 + prices[i])
            dp_i1 = max(dp_i1, dp_pre0 - prices[i])
            dp_pre0 = tmp
        return dp_i0
```

714. 买卖股票的最佳时机含手续费

难度 中等 527 收藏 分享 切换为英文 接收动态 反馈

给定一个整数数组 `prices`，其中第 `i` 个元素代表了第 `i` 天的股票价格；整数 `fee` 代表股票的手续费。

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
```

```
class Solution:
    def maxProfit(self, prices: List[int], fee: int) -> int:
        if not prices:
            return 0
        n = len(prices)
        dp_i0, dp_i1 = 0, float('-inf')
        for i in range(n):
            tmp = dp_i0
            dp_i0 = max(dp_i0, dp_i1 + prices[i])
            dp_i1 = max(dp_i1, tmp - prices[i] - fee)
        return dp_i0
```

乘积最大子数组

152. 乘积最大子数组

难度 中等 1224 收藏 分享 切换为英文 接收动态 反馈

给你一个整数数组 `nums`，请你找出数组中乘积最大的连续子数组（该子数组中至少包含一个元素）并返回该子数组所对应的乘积。

示例 1：

```
输入: [2,3,-2,4]
输出: 6
解释: 子数组 [2,3] 有最大乘积 6。
```

示例 2：

```
输入: [-2,0,-1]
输出: 0
解释: 结果不能为 2，因为 [-2,-1] 不是子数组。
```

```

class Solution:
    def maxProduct(self, nums: List[int]) -> int:
        Max, imax, imin = float("-inf"), 1, 1
        for i in range(len(nums)):
            if nums[i] < 0:
                imax, imin = imin, imax
            imax = max(imax * nums[i], nums[i])
            imin = min(imin * nums[i], nums[i])
            Max = max(Max, imax)
        return Max

```

621. 任务调度器

难度 中等 695 收藏 分享 切换为英文 接收动态 反馈

给你一个用字符数组 `tasks` 表示的 CPU 需要执行的任务列表。其中每个字母表示一种不同任务。任务可以以任意顺序执行，并且每个任务都可以在 1 个单位时间内执行完。在任何一个单位时间内，CPU 可以完成一个任务，或者处于待命状态。

然而，两个 **相同种类** 的任务之间必须有长度为整数 `n` 的冷却时间，因此至少有连续 `n` 个单位时间内，CPU 在执行不同的任务，或者在待命状态。

你需要计算完成所有任务所需要的 **最短时间**。

示例 1:

输入: `tasks = ["A","A","A","B","B","B"]`, `n = 2`

输出: 8

解释: A → B → (待命) → A → B → (待命) → A → B

在本示例中，两个相同类型任务之间必须间隔长度为 `n = 2` 的冷却时间，而执行只需要一个单位时间，所以中间出现了 (待命) 状态。

示例 2:

输入: `tasks = ["A","A","A","B","B","B"]`, `n = 0`

输出: 6

解释: 在这种情况下，任何大小为 6 的排列都可以满足要求，因为 `n = 0`

`["A","A","A","B","B","B"]`

`["A","B","A","B","A","B"]`

`["B","B","B","A","A","A"]`

...

class Solution:

```

def leastInterval(self, tasks: List[str], n: int) -> int:
    from collections import Counter
    ct = Counter(tasks)
    nbucket = ct.most_common(1)[0][1]
    last_bucket_size = list(ct.values()).count(nbucket)
    res = (nbucket - 1) * (n + 1) + last_bucket_size
    return max(res, len(tasks))

```

91. 解码方法

难度 中等 904 收藏 分享 切换为英文 接收动态 反馈

一条包含字母 A-Z 的消息通过以下映射进行了 编码：

```
'A' -> 1  
'B' -> 2  
...  
'Z' -> 26
```

要解码已编码的消息，所有数字必须基于上述映射的方法，反向映射回字母（可能有多种方案，"11106" 可以映射为：

- "AAJF"，将消息分组为 (1 1 10 6)
- "KJF"，将消息分组为 (11 10 6)

注意，消息不能分组为 (1 11 06)，因为 "06" 不能映射为 "F"，这是由于 "6" 和 "0" 在映射中并不等价。

给你一个只含数字的 非空 字符串 s，请计算并返回 解码 方法的 总数。

题目数据保证答案肯定是一个 32 位 的整数。

示例 1：

输入: s = "12"

输出: 2

解释：它可以解码为 "AB" (1 2) 或者 "L" (12)。

Solution:

```
class Solution:  
    def numDecodings(self, s: str) -> int:  
        n = len(s)  
        if not s or s[0] == '0':  
            return 0  
        dp = [0] * (n + 1)  
        dp[0] = 1  
        dp[1] = 1  
        for i in range(1, n):  
            if s[i] == '0':  
                if s[i - 1] == '1' or s[i - 1] == '2':  
                    dp[i + 1] = dp[i - 1]  
                else:  
                    return 0  
            else:  
                if s[i - 1] == '1' or (s[i - 1] == '2' and '1' < s[i] <='6'): # 11-19  
                    dp[i + 1] = dp[i] + dp[i - 1]  
                else:  
                    dp[i + 1] = dp[i]  
        return dp[-1]
```

818. 赛车

难度 困难 106 收藏 分享 切换为英文 接收动态 反馈

你的赛车起始停留在位置 0，速度为 +1，正行驶在一个无限长的数轴上。（车也可以向负数驶。）

你的车会根据一系列由 A (加速) 和 R (倒车) 组成的指令进行自动驾驶。

当车得到指令 "A" 时，将会做出以下操作：`position += speed, speed *= 2`。

当车得到指令 "R" 时，将会做出以下操作：如果当前速度是正数，则将车速调整为 `speed = -speed`；否则将车速调整为 `speed = 1`。（当前所处位置不变。）

例如，当得到一系列指令 "AAR" 后，你的车将会走过位置 0->1->3->3，并且速度变化为 1->-1。

现在给定一个目标位置，请给出能够到达目标位置的最短指令列表的**长度**。

示例 1：

输入：

```
target = 3
```

输出：

2

解释：

最短指令列表为 "AA"

位置变化为 0->1->3

Solution:

```
class Solution:
    def racecar(self, target: int) -> int:
        def race(t):
            if t not in dp:
                n = t.bit_length()
                if (1 << n) - 1 == t:
                    dp[t] = n
                else:
                    dp[t] = n + 1 + race((1 << n) - 1 - t)
            for m in range(n - 1):
                dp[t] = min(dp[t], n + m + 1 + race(t - (1 << m) + m))
            return dp[t]

        dp = {0: 0}
        return race(target)
```

312. 截气球

难度 困难 768 收藏 分享 切换为英文 接收动态 反馈

有 n 个气球，编号为 0 到 $n - 1$ ，每个气球上都标有一个数字，这些数字存在数组 num

现在要求你截破所有的气球。截破第 i 个气球，你可以获得 $\text{nums}[i - 1] * \text{nums}[i] + 1$ 枚硬币。这里的 $i - 1$ 和 $i + 1$ 代表和 i 相邻的两个气球的序号。如果 $i - 1$ 超出了数组的边界，那么就当它是一个数字为 1 的气球。

求所能获得硬币的最大数量。

示例 1：

输入: $\text{nums} = [3,1,5,8]$

输出: 167

解释:

$\text{nums} = [3,1,5,8] \rightarrow [3,5,8] \rightarrow [3,8] \rightarrow [8] \rightarrow []$

$\text{coins} = 3*1*5 + 3*5*8 + 1*3*8 + 1*8*1 = 167$

示例 2：

输入: $\text{nums} = [1,5]$

输出: 10

class Solution:

```
def maxCoins(self, nums: List[int]) -> int:
    nums.insert(0, 1)
    nums.append(1)
    store = [[0] * len(nums) for _ in range(len(nums))]
    def range_best(i, j):
        m = 0
        for k in range(i + 1, j):
            left = store[i][k]
            right = store[k][j]
            a = left + nums[i] * nums[k] * nums[j] + right
            if a > m:
                m = a
        store[i][j] = m
    for n in range(2, len(nums)):
        for i in range(0, len(nums) - n):
            range_best(i, i + n)
    return store[0][len(nums) - 1]
```

76. 最小覆盖子串

难度 困难 1280 收藏 分享 切换为英文 接收动态 反馈

给你一个字符串 s 、一个字符串 t 。返回 s 中涵盖 t 所有字符的最小子串。如果 s 不涵盖 t 所有字符的子串，则返回空字符串 ""。

注意：

- 对于 t 中重复字符，我们寻找的子字符串中该字符数量必须不少于 t 中该字符数量
- 如果 s 中存在这样的子串，我们保证它是唯一的答案。

示例 1：

输入: $s = "ADOBECODEBANC"$, $t = "ABC"$
输出: "BANC"

示例 2：

输入: $s = "a"$, $t = "a"$
输出: "a"

示例 3：

输入: $s = "a"$, $t = "aa"$
输出: ""
解释: t 中两个字符 'a' 均应包含在 s 的子串中，因此没有符合条件的子字符串，返回空字符串。

class Solution:

```
def minWindow(self, s: str, t: str) -> str:
    start, left, right, match, minLen = 0, 0, 0, 0, float('inf')
    window = {}
    needs = dict((i, t.count(i)) for i in t)

    while right < len(s):
        c1 = s[right]
        if c1 in needs.keys():
            window[c1] = window.get(c1, 0) + 1
            if window[c1] == needs[c1]:
                match += 1
        right += 1

    while match == len(needs):
        if right - left < minLen:
            start = left
            minLen = right - left
        c2 = s[left]
        if c2 in needs.keys():
            window[c2] -= 1
            if window[c2] < needs[c2]:
                match -= 1
        left += 1

    return '' if minLen == float('inf') else s[start:start + minLen]
```

552. 学生出勤记录 II

难度 困难 143 收藏 分享 切换为英文 接收动态 反馈

给定一个正整数 n ，返回长度为 n 的所有可被视为可奖励的出勤记录的数量。答案可能非常大，返回结果 $\text{mod } 10^9 + 7$ 的值。

学生出勤记录是只包含以下三个字符的字符串：

1. 'A' : Absent, 缺勤
2. 'L' : Late, 迟到
3. 'P' : Present, 到场

如果记录不包含**多于一个'A' (缺勤) 或超过两个连续的'L' (迟到)**，则该记录被视为可奖励。

示例 1:

输入: $n = 2$

输出: 8

解释:

有8个长度为2的记录将被视为可奖励：

"PP" , "AP", "PA", "LP", "PL", "AL", "LA", "LL"

只有"AA"不会被视为可奖励，因为缺勤次数超过一次。

Solution:

```
class Solution:
    def checkRecord(self, n: int) -> int:
        _Mod = 1000000007
        dp00 = dp01 = dp10 = 1
        dp11 = dp02 = dp12 = 0
        for i in range(2, n + 1):
            t00, t01, t02, t10, t11, t12 = dp00, dp01, dp02, dp10, dp11, dp12
            dp00 = (t00 + t01 + t02) % _Mod
            dp10 = (t10 + t11 + t12) % _Mod
            dp01 = t00
            dp02 = t01
            dp11 = t10
            dp12 = t11
            dp10 += (t00 + t01 + t02) % _Mod
        return (dp00 + dp01 + dp02 + dp10 + dp11 + dp12) % _Mod
```

410. 分割数组的最大值

难度 困难 530 收藏 分享 切换为英文 接收动态 反馈

给定一个非负整数数组 `nums` 和一个整数 `m`，你需要将这个数组分成 `m` 个非空的连续子数组。设计一个算法使得这 `m` 个子数组各自和的最大值最小。

示例 1：

输入: `nums = [7,2,5,10,8]`, `m = 2`

输出: 18

解释:

一共有四种方法将 `nums` 分割为 2 个子数组。其中最好的方式是将其分为 `[7,2,5]` 和 `[10,8]`。

因为此时这两个子数组各自的和的最大值为18，在所有情况中最小。

示例 2：

输入: `nums = [1,2,3,4,5]`, `m = 2`

输出: 9

示例 3：

输入: `nums = [1,4,4]`, `m = 3`

输出: 4

class Solution:

```
def splitArray(self, nums: List[int], m: int) -> int:
    left, right = max(nums), sum(nums)

    def test_mid(mid):
        num = 1
        s = 0
        for i in nums:
            if s + i > mid:
                s = i
                num += 1
            else:
                s += i
        return num > m

    while left < right:
        mid = (left + right) // 2
        if_right = test_mid(mid)
        if if_right:
            left = mid + 1
        else:
            right = mid
    return left
```

403. 青蛙过河

难度 困难 357 收藏 分享 切换为英文 接收动态 反馈

一只青蛙想要过河。假定河流被等分为若干个单元格，并且在每一个单元格内都有可能放有一块石子（也有可能没有）。青蛙可以跳上石子，但是不可以跳入水中。

给你石子的位置列表 `stones`（用单元格序号 升序 表示），请判定青蛙能否成功过河（即能否一步跳至最后一块石子上）。

开始时，青蛙默认已站在第一块石子上，并可以假定它第一步只能跳跃一个单位（即只能从单元格 1 跳至单元格 2）。

如果青蛙上一步跳跃了 `k` 个单位，那么它接下来的跳跃距离只能选择为 `k - 1`、`k` 或 `k + 1` 个单位。另请注意，青蛙只能向前进（终点的方向）跳跃。

示例 1：

输入: `stones = [0,1,3,5,6,8,12,17]`

输出: `true`

解释：青蛙可以成功过河，按照如下方案跳跃：跳 1 个单位到第 2 块石子，然后跳 2 个单位到第 3 块石子，接着跳 2 个单位到第 4 块石子，然后跳 3 个单位到第 6 块石子，最后跳 5 个单位到第 7 块石子，最后，跳 5 个单位到第 8 块石子（即最后一块石子）。

示例 2：

输入: `stones = [0,1,2,3,4,8,9,11]`

输出: `false`

解释：这是因为第 5 和第 6 个石子之间的间距太大，没有可选的方案供青蛙跳跃过去。

Solution:

```
class Solution:
    def canCross(self, stones: List[int]) -> bool:
        from functools import lru_cache
        end = stones[-1]
        s = set(stones)
        @lru_cache(None)
        def helper(start, jump):
            if start == end:
                return True
            for j in [jump - 1, jump, jump + 1]:
                if j <= 0:
                    continue
                if start + j in s and helper(start + j, j):
                    return True
            return False
        return helper(0, 0)
```

363. 矩形区域不超过 k 的最大数值和

难度 困难 400 收藏 分享 切换为英文 接收动态 反馈

给你一个 $m \times n$ 的矩阵 `matrix` 和一个整数 `k`，找出并返回矩阵内部矩形区域的不超过 `k` 数值和。

题目数据保证总会存在一个数值和不超过 `k` 的矩形区域。

示例 1：

1	0	1
0	-2	3

输入: `matrix = [[1,0,1],[0,-2,3]]`, `k = 2`

输出: 2

解释: 蓝色边框圈出来的矩形区域 $[[0, 1], [-2, 3]]$ 的数值和是 2，且 2 是不超过最大数字 ($k = 2$)。

示例 2：

输入: `matrix = [[2,2,-1]]`, `k = 3`

输出: 3

`class Solution:`

```
def maxSumSubmatrix(self, matrix: List[List[int]], k: int) -> int:
    rows, cols, _max = len(matrix), len(matrix[0]), float('-inf')
    for l in range(cols):
        rowSum = [0] * rows
        for r in range(l, cols):
            for i in range(rows):
                rowSum[i] += matrix[i][r]
            _max = max(_max, self.dpmax(rowSum, k))
            if _max == k:
                return k
    return _max
```

`def dpmax(self, arr, k):`

```
rollSum = arr[0]
rollMax = rollSum
for i in range(1, len(arr)):
    if rollSum > 0:
        rollSum += arr[i]
    else:
        rollSum = arr[i]
    if rollSum > rollMax:
        rollMax = rollSum
if rollMax <= k:
    return rollMax
```

`_max = float('-inf')`

```
for l in range(len(arr)):
    _sum = 0
    for r in range(l, len(arr)):
        _sum += arr[r]
        if _sum > _max and _sum <= k:
            _max = _sum
        if _max == k:
            return k
```

`return _max`

1025. 除数博弈

难度 简单 316 收藏 分享 切换为英文 接收动态 反馈

爱丽丝和鲍勃一起玩游戏，他们轮流行动。爱丽丝先手开局。

最初，黑板上有一个数字 N 。在每个玩家的回合，玩家需要执行以下操作：

- 选出任一 x ，满足 $0 < x < N$ 且 $N \% x == 0$ 。
- 用 $N - x$ 替换黑板上的数字 N 。

如果玩家无法执行这些操作，就会输掉游戏。

只有在爱丽丝在游戏中取得胜利时才返回 `True`，否则返回 `False`。假设两个玩家都以最佳方式玩游戏。

示例 1：

```
输入: 2
输出: true
解释: 爱丽丝选择 1, 鲍勃无法进行操作。
```

示例 2：

```
输入: 3
输出: false
解释: 爱丽丝选择 1, 鲍勃也选择 1, 然后爱丽丝无法进行操作。
```

```
class Solution:
    def divisorGame(self, N: int) -> bool:
        dp = {}
        dp[1] = False
        dp[2] = True
        for i in range(3, N + 1):
            dp[i] = False
            for j in range(1, i):
                if i % j == 0 and dp[i - j] == False:
                    dp[i] = True
                    break
        return dp[N]
```