238.除自身以外数组的乘积

题目大意：

给你一个整数数组 nums，返回 数组 answer ，其中 answer[i] 等于 nums 中除 nums[i] 之外其余各元素的乘积。

解题思路：

1.定义左右两侧的乘积列表：L 和 R
    L[i] 和 R[i] 分别表示nums[i]左右两侧的乘积
2.遍历，计算answer：
    answer[i] = L[i] * R[i]

时间复杂度分析：

时间复杂度：O(N)，其中 N 指的是数组 nums 的大小。预处理 L 和 R 数组以及最后的遍历计算都是 O(N) 的时间复杂度。
空间复杂度：O(N)，其中 N 指的是数组 nums 的大小。使用了 L 和 R 数组去构造答案，L 和 R 数组的长度为数组 nums 的大小。

完整代码：

```
def productExceptSelf(self, nums:List[int]) -> List[int]:
    n = len(nums)

    # L 和 R 分别表示左右两侧的乘积列表
    L, R = [0] * n, [0] * n
    answer = [0] * n  # 记录最终答案

    # L[i] 为索引 i 左侧所有元素的乘积
    # 对于索引为 '0' 的元素，因为左侧没有元素，所以 L[0] = 1
    L[0] = 1
    for i in range(1, n):
        L[i] = nums[i - 1] * L[i - 1]

    # R[i] 为索引 i 右侧所有元素的乘积
    # 对于索引为 'n-1' 的元素，因为右侧没有元素，所以 R[n-1] = 1
    R[n - 1] = 1
    for i in range(n - 2, -1, -1):
        R[i] = nums[i + 1] * R[i + 1]

    # 对于索引 i，除 nums[i] 之外其余各元素的乘积就是左侧所有元素的乘积乘以右侧所有元素的乘积
    for i in range(n):
        answer[i] = L[i] * R[i]

    return answer
```

41.缺失的第一个正数

题目大意：

给你一个未排序的整数数组 `nums` ，请你找出其中没有出现的最小的正整数。

请你实现时间复杂度为 `O(n)` 并且只使用常数级别额外空间的解决方案。

解题思路：

Steps:
1.遍历数组，确保每个数字都在它应该出现的位置。如果数字 `x` 在 `[1, n]` 范围内且它没有在索引 `x-1` 的位置上，就交换它到该位置。
2.交换过程中，如果交换的位置上的数字与目标位置上的数字相同，则跳过该位置。
3.遍历结束后，检查从左到右第一个不在正确位置上的索引 `i`，那么 `i + 1` 就是我们要找的最小正整数。如果所有位置都正确，则说明数组中包含了 `[1, n]` 之间的所有整数，因此最小缺失值是 `n+1`。

时间复杂度分析：

时间复杂度：o(n)
空间复杂度：o(1)

完整代码：

```
def firstMissingPositive(self, nums:List[int]) -> int:
    n = len(nums)

    # 将所有小于等于0的数或者大于n的数变为n+1，因为这些数不会影响我们寻找的目标
    for i in range(n):
        if nums[i] <= 0 or nums[i] > n:
            nums[i] = n + 1

    # 在数组中放置每个数到它的正确位置上
    for i in range(n):
        num = abs(nums[i])  # 获取当前数的绝对值
        if 1 <= num <= n:  # 如果 num 在有效范围内
            # 将num放到正确的位置 nums[num-1]
            if nums[num - 1] > 0:  # 如果 nums[num-1] 是正数，标记为负数
                nums[num - 1] = -nums[num - 1]

     # 找到第一个没有被标记的索引
    for i in range(n):
        if nums[i] > 0:
            return i + 1  # 如果 nums[i] 是正数，说明 i+1 没有出现

    # 如果没有找到，说明数组包含了所有 1 到 n 的数，返回 n+1
    return n + 1
```

73.矩阵置零

题目大意：

给定一个 m x n 的矩阵，如果一个元素为 0 ，则将其所在行和列的所有元素都设为 0 。请使用 原地 算法。

解题思路：

用两个标记数组分别记录每一行和每一列是否有零出现。
Steps:
1.遍历该数组一次，如果某个元素为 0，那么就将该元素所在的行和列所对应标记数组的位置置为 true
2.再次遍历该数组，用标记数组更新原数组即可。

时间复杂度分析：

时间复杂度：O(mn)，其中 m 是矩阵的行数，n 是矩阵的列数。我们至多只需要遍历该矩阵两次。
空间复杂度：O(m+n)，其中 m 是矩阵的行数，n 是矩阵的列数。我们需要分别记录每一行或每一列是否有零出现。

完整代码：

```
def setZeroes(self, matrix:List[List[int]]) -> None:
    # 矩阵行，列大小：m, n
    m, n = len(matrix), len(matrix[0])
    row, col = [False] * m, [False] * n  # 初始化标记行数组、标记列数组

    # 遍历行、列
    for i in range(m):
        for j in range(n):
            # 如果当前元素为0，将该元素所在的行、列所对应标记数组的位置置为
true
            if matrix[i][j] == 0:
                row[i] = col[j] = True

    # 再次遍历，用标记数组更新原数组
    for i in range(m):
        for j in range(n):
            if row[i] or col[j]:
                matrix[i][j] = 0
```

54.螺旋矩阵

题目大意：

给你一个 m 行 n 列的矩阵 matrix ，请按照 顺时针螺旋顺序 ，返回矩阵中的所有元素。

解题思路：

1.模拟螺旋矩阵的路径：
    初始位置是矩阵的左上角，初始方向是向右，当路径超出界限或者进入之前访问过的位置时，顺时针旋转，进入下一个方向。
2.判断路径是否进入之前访问过的位置：
    使用一个与输入矩阵大小相同的辅助矩阵 visited：其中的每个元素表示该位置是否被访问过。
3.判断路径是否结束：
    当路径的长度达到矩阵中的元素数量时即为完整路径

时间复杂度分析：

时间复杂度：O(mn)，其中 m 和 n 分别是输入矩阵的行数和列数。矩阵中的每个元素都要被访问一次。
空间复杂度：O(mn)。需要创建一个大小为 m×n 的矩阵 visited 记录每个位置是否被访问过。

完整代码：

```
def spiralOrder(self, matrix:List[List[int]]) -> List[int]:

    # 如果输入矩阵为空或者第一行为空，那么返回空列表
    if not matrix or not matrix[0]:
        return list()

    # 初始化参数
    rows, columns = len(matrix), len(matrix[0])
    visited = [[False] * columns for _ in range(rows)]  # 标记访问状态（大小与输入矩阵一致）
    total = rows * columns  # 矩阵中的元素总数
    order = [0] * total  # 用来存储螺旋顺序的结果

    # 方向数组（里面各个列表的各个元素对应row和column增加的步数）
    directions = [[0, 1], [1, 0], [0, -1], [-1, 0]]  # 表示四个方向的数组，依次为向右、向下、向左、向上。

    # 初始化起始位置
    row, column = 0, 0
    directionIndex = 0  # 初始方向为“向右”

    # 遍历矩阵并填充结果
    for i in range(total):
        order[i] = matrix[row][column]  # 将当前元素添加到结果中
        visited[row][column] = True  # 标记当前位置已经访问过

        # 为下一轮移动准备
        nextRow, nextColumn = row + directions[directionIndex][0], column + directions[directionIndex][1]

        # 检查是否需要改变方向
```

```
        # 如果接下来的位置超出了矩阵的边界，或者该位置已经访问过，就需要改变方向。
        if not (0 <= nextRow < rows and 0 <= nextColumn < columns and not visited[nextRow][nextColumn]):
            directionIndex = (directionIndex + 1) % 4  # 旋转方向(通过对 4 取余来循环变换方向，依次是右、下、左、上。)

        # 移动到下一个位置
        row += directions[directionIndex][0]
        column += directions[directionIndex][1]

    return order
```

48.旋转图像

题目大意：

给定一个 n × n 的二维矩阵 matrix 表示一个图像。请你将图像顺时针旋转 90 度。

你必须在 原地 旋转图像，这意味着你需要直接修改输入的二维矩阵。请不要使用另一个矩阵来旋转图像。

解题思路：

关键思想：对于矩阵中第 i 行的第 j 个元素，在旋转后，它出现在倒数第 i 列的第 j 个位置。
因此对于矩阵中的元素 matrix[row][col]，在旋转后，它的新位置为 matrix_new[col][n−row−1]。

而题目要求原地旋转，直接使用matrix[col][n−row−1] = matrix[row][col]会使得后面无法访问原matrix_new[col][n−row−1]处的值。
因此，考虑使用临时变量temp暂存matrix[col][n−row−1]值，这样在其被覆盖后还是能通过temp访问原来的值。

而原本matrix[col][n−row−1]处的值经过旋转后，同理变为matrix[n−row−1][n−col−1]

matrix[n−row−1][n−col−1]处的值经过旋转后，同理变为matrix[n−col−1][row]

matrix[n−col−1][row]处的值经过旋转后，同理变为matrix[row][col]，回到了最初的起点。

综上可见，这四项处于一个循环中，每一项旋转后的位置就是下一项所在的位置。因此可以使用一个临时变量 temp 完成这四项的原地交换：

```
    temp = matrix[row][col]
    matrix[row][col] = matrix[n-col-1][row]
    matrix[n-col-1][row] = matrix[n-row-1][n-col-1]
    matrix[n-row-1][n-col-1] = matrix[col][n-row-1]
    matrix[col][n-row-1] = temp
```

应该枚举哪些位置(row, col)进行上述原地交换操作？：
    当 n 为偶数时，需要枚举(n^2)/4=(n/2)×(n/2) 个位置,可以将该图形分为四块.

当 n 为奇数时，由于中心的位置经过旋转后位置不变，我们需要枚举 (n^2−1)/4=((n−1)/2)×((n+1)/2) 个位置，需要换一种划分的方式.

时间复杂度分析：

时间复杂度：O(N^2)，其中 N 是 matrix 的边长。我们需要枚举的子矩阵大小为 O(⌊n/2⌋×⌊(n+1)/2⌋)=O(N^2)。
空间复杂度：O(1)。为原地旋转。

完整代码：

```
def rotate(self, matrix:List[List[int]]) -> None:
    n = len(matrix)
    # 按层次遍历（从外到内）
    for i in range(n//2):
        for j in range((n+1)//2):  # 表示在当前层内要交换的元素数量。（(n + 1) // 2适应了奇数和偶数大小矩阵的情况）
            # 依次交换四个角的位置
            matrix[i][j], matrix[n - j - 1][i], matrix[n - i - 1][n - j - 1], matrix[j][n - i - 1] \
                    = matrix[n - j - 1][i], matrix[n - i - 1][n - j - 1], matrix[j][n - i - 1], matrix[i][j]
```