240.搜索二维矩阵_2.0

题目大意：

编写一个高效的算法来搜索 m x n 矩阵 matrix 中的一个目标值 target 。该矩阵具有以下特性：
1.每行的元素从左到右升序排列。
2.每列的元素从上到下升序排列。

解题思路：

法一：直接遍历

法二：z字形查找
从矩阵 matrix 的右上角 (0,n-1) 进行搜索。
在每一步的搜索过程中，如果位于位置 (x,y)，则以 matrix 的左下角为左下角、以 (x,y) 为右上角的矩阵中进行搜索，即行的范围为 [x,m-1]，列的范围为 [0,y]：
1.如果matrix[x,y] == target，说明搜索完成
2.如果matrix[x,y] > target，说明目标值可能在当前元素的左边，y-1
3.如果matrix[x,y] < target，说明目标值可能在当前元素的左边，x+1
如果超出矩阵边界说明目标元素不存在。

复杂度分析：

法一：直接遍历
时间复杂度：o(mn)
空间复杂度：o(1)

法二：z字形查找
时间复杂度：o(m+n)
空间复杂度：o(1)

完整代码：

```
# 法一：直接遍历
def searchMatrix(self, matrix:List[List[int]], target:int) -> bool:
    for row in matrix:
        for element in row:
            if element == target:
                return True
    return False
```

```
# 法二：z字形查找
def searchMatrix(self, matrix:List[List[int]], target:int) -> bool:
    m, n = len(matrix), len(matrix[0])  # 行数、列数
    x, y = 0, n-1   # 从矩阵的最右上方开始
```

```
    # 在矩阵边界范围内
    while x < m and y >= 0:
        current = matrix[x][y]

        if current == target:
            return True
        elif current > target:
            y -= 1  # 向左走
        else:
            x += 1  # 向下走

    return False
```

160.相交链表

题目大意：

给你两个单链表的头节点 `headA` 和 `headB` ，请你找出并返回两个单链表相交的起始节点。
如果两个链表不存在相交节点，返回 `null` 。

解题思路：

双指针法：模拟两个指针在两个链表上来回“走”
Steps:
    1.让两个指针分别指向 `headA` 和 `headB`
    2.同时移动这两个指针。每当一个指针遍历完一个链表后，就将它指向另一个链表的头节点。这样，两个指针最终会在相交点相遇，或者都为 `null`。

时间复杂度分析：

时间复杂度：O(m+n)，其中 `m` 和 `n` 是分别是链表 `headA` 和 `headB` 的长度。两个指针同时遍历两个链表，每个指针遍历两个链表各一次。
空间复杂度：O(1)

完整代码：

```
# Definition for singly-linked list.
class ListNode:
    def __init__(self, x):
        self.val = x
        self.next = None

def getIntersectionNode(self, headA: ListNode, headB: ListNode) ->
ListNode:
    # 边界条件：如果任意一个链表为空，直接返回 None
    if not headA or not headB:
        return None

    # 初始化两个指针
```

```
    pointerA, pointerB = headA, headB

    # 当两个指针相遇时，说明找到了交点，或者两者都为 None
    while pointerA != pointerB:
        # 当 pointerA 到达链表 A 的末尾时，将其重新指向链表 B 的头节点
        pointerA = pointerA.next if pointerA else headB
        # 当 pointerB 到达链表 B 的末尾时，将其重新指向链表 A 的头节点
        pointerB = pointerB.next if pointerB else headA

    # 当 pointerA == pointerB 时，返回交点或 None
    return pointerA
```

206.反转链表

题目大意：

```
给你单链表的头节点 head ，请你反转链表，并返回反转后的链表。
```

解题思路：

```
迭代：
遍历链表，将当前节点的next指针改为指向前一个节点
```

时间复杂度分析：

```
时间复杂度：O(n)，其中 n 是链表的长度。需要遍历链表一次。
空间复杂度：O(1)
```

完整代码：

```
class ListNode:
    def __init__(self, x):
        self.val = x
        self.next = None

def reverseList(self, head:ListNode) -> ListNode:
    prev = None
    current = head

    # 遍历链表
    while current:
        next_node = current.next  # 下一个节点
        current.next = prev  # 当前节点指向前一个节点
        prev = current  # 更新prev
        current = next_node  # 更新current

    return prev  # 返回最终头节点
```

234.回文链表

题目大意：

> 给你一个单链表的头节点 `head` ，请你判断该链表是否为回文链表。如果是，返回 `true` ；否则，返回 `false` 。

解题思路：

> 将值复制到数组中后用双指针法
> （因为链表访问一个节点需要o(n)的时间，数组访问一个元素只需要o(1)时间）
> （将链表复制到数组中需要o(n)时间）

时间复杂度分析：

> 时间复杂度：O(n)，其中 n 指的是链表的元素个数。
> 空间复杂度：O(n)，其中 n 指的是链表的元素个数，使用了一个数组列表存放链表的元素值。

完整代码：

```
class ListNode:
    def __init__(self, x):
        self.val = x
        self.next = None

def isPalindrome(self, head:ListNode) -> bool:
    vals = []  # 存储数组
    current_node = head

    # 将链表复制到数组
    while current_node is not None:
        vals.append(current_node.val)
        current_node = current_node.next

    return vals == vals[::-1]
```

141.环形链表

题目大意：

> 给你一个链表的头节点 `head` ，判断链表中是否有环。
>
> 如果链表中有某个节点，可以通过连续跟踪 `next` 指针再次到达，则链表中存在环。
>
> 如果链表中存在环 ，则返回 `true` 。 否则，返回 `false` 。

解题思路：

哈希表
遍历所有节点，每次遍历到一个节点时，判断该节点此前是否被访问过。
使用哈希表来存储所有已经访问过的节点。
每次到达一个节点，如果该节点已经存在于哈希表中，则说明该链表是环形链表，否则就将该节点加入哈希表中。
重复直至遍历完整个链表

注：
Q：为什么使用哈希表不使用数组？
A：查找是否存在某个元素的时间复杂度是常数级别 O(1)，而使用数组时，查找需要 O(n) 的时间

时间复杂度分析：

时间复杂度：O(N)，其中 N 是链表中的节点数。最坏情况下我们需要遍历每个节点一次。
空间复杂度：O(N)，其中 N 是链表中的节点数。主要为哈希表的开销，最坏情况下我们需要将每个节点插入到哈希表中一次。

完整代码：

```
def hasCycle(self, head:ListNode) -> bool:
    seen = set()  # 创建哈希集合

    # 遍历链表
    while head:
        if head in seen:  # 如果当前节点已经在哈希表里了，说明存在环
            return True
        seen.add(head)  # 添加节点到哈希表
        head = head.next
    return False
```