142.环形链表_2

题目大意：

给定一个链表的头节点 head ，返回链表开始入环的第一个节点。 如果链表无环，则返回 null。

不允许修改 链表。

解题思路：

哈希表遍历（查找只需要o(1)时间）
发现重复的节点就说明是入环节点

复杂度分析：

时间复杂度：o(n)。需要访问链表中的每一个节点。
空间复杂度：o(n)。将链表中的每个节点都保存在哈希表当中。

完整代码：

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

def detectCycle(self, head:ListNode) -> ListNode:
    visited = set()  # 哈希集合：存储访问过的节点
    current = head  # 当前节点

    # 遍历链表
    while current:
        if current in visited:  # 如果当前节点已经被访问过，说明为环入口
            return current
        visited.add(current)
        current = current.next
    return None
```

21.合并两个有序链表

题目大意：

将两个升序链表合并为一个新的 升序 链表并返回。新链表是通过拼接给定的两个链表的所有节点组成的。

解题思路：

递归
两个链表头部值较小的一个节点与剩下元素的 merge 操作结果合并：

$$\begin{cases} list1[0] + merge(list1[1:], list2) & list1[0] < list2[0] \\ list2[0] + merge(list1, list2[1:]) & otherwise \end{cases}$$

时间复杂度分析：

时间复杂度：o(n+m)，其中 n 和 m 分别为两个链表的长度。因为每次调用递归都会去掉 l1 或者 l2 的头节点（直到至少有一个链表为空），函数 mergeTwoList 至多只会递归调用每个节点一次。因此，时间复杂度取决于合并后的链表长度，即 O(n+m)。
空间复杂度：O(n+m)，其中 n 和 m 分别为两个链表的长度。递归调用 mergeTwoLists 函数时需要消耗栈空间，栈空间的大小取决于递归调用的深度。结束递归调用时 mergeTwoLists 函数最多调用 n+m 次，因此空间复杂度为 O(n+m)

完整代码：

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

def mergeTwoLists(self, l1:ListNode, l2:ListNode) -> ListNode:
    # 如果有链表为空，直接返回另一个链表
    if not l1:
        return l2
    if not l2:
        return l1

    # 比较当前节点的值，递归合并剩余部分
    if l1.val < l2.val:
        l1.next = mergeTwoLists(l1.next, l2)
        return l1
    else:
        l2.next = mergeTwoLists(l1, l2.next)
        return l2
```

2.两数相加

题目大意：

给你两个 非空 的链表，表示两个非负的整数。它们每位数字都是按照 逆序 的方式存储的，并且每个节点只能存储 一位 数字。
请你将两个数相加，并以相同形式返回一个表示和的链表。
你可以假设除了数字 0 之外，这两个数都不会以 0 开头。

解题思路：

模拟法
相同位置的数字直接相加
carry存储进位值

时间复杂度分析：

时间复杂度：O(max(m,n))，其中 m 和 n 分别为两个链表的长度。我们要遍历两个链表的全部位置，而处理每个位置只需要 O(1) 的时间。
空间复杂度：O(1)。注意返回值不计入空间复杂度。

完整代码：

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

def addTwoNumbers(self, l1:ListNode, l2:ListNode) -> ListNode:
    # 初始化虚拟头节点和当前指针
    dummy = ListNode(0)
    current = dummy
    carry = 0  # 存储进位值

    # 遍历两个链表，直到两个链表都为空
    while l1 or l2:
        # 取当前节点的值，如果链表为空则取 0
        n1 = l1.val if l1 else 0
        n2 = l2.val if l2 else 0

        # 计算当前位的和以及进位
        sum = n1 + n2 + carry
        carry = sum // 10  # 计算进位（每增加一个10就进1位）
        current.next = ListNode(sum % 10)  # 创建新节点存储当前位的值
        current = current.next  # 指针指向下一个位置

        # 移动链表指针
        if l1:
            l1 = l1.next
        if l2:
            l2 = l2.next

    # 如果最后还有进位，那么还要创建一个新节点
    if carry > 0:
        current.next = ListNode(carry)

    return dummy.next
```

19.删除链表的倒数第N个结点

题目大意：

给你一个链表，删除链表的倒数第 n 个结点，并且返回链表的头结点。

解题思路：

首先从头节点开始对链表进行一次遍历，得到链表的长度 L。
再从头节点开始对链表进行一次遍历，当遍历到第 L-n+1 个节点时，它就是我们需要删除的节点。

时间复杂度分析：

时间复杂度：O(L)，其中 L 是链表的长度。
空间复杂度：O(1)

完整代码：

```
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def removeNthFromEnd(self, head:ListNode, n:int) -> ListNode:
    dummy = ListNode(0, head)  # 定义哑节点，其下一个节点指向链表头节点head
    L = 0  # 记录链表长度
    # 计算链表长度
    while head:
        L += 1
        head = head.next

    current = dummy  # 当前指针

    # 从哑节点第二个节点（链表头节点head）开始向后遍历，至倒数第L-n个（要删除节点的前一个）
    for i in range(1, L - n + 1):
        current = current.next
    current.next = current.next.next  # 将下一个节点设置为越过待删除节点的下一个

    return dummy.next
```

24.两两交换链表中的节点

题目大意：

给你一个链表，两两交换其中相邻的节点，并返回交换后链表的头节点。你必须在不修改节点内部的值的情况下完成本题（即，只能进行节点交换）。

解题思路：

递归
终止条件：是链表中没有节点，或者链表中只有一个节点，此时无法进行交换。

时间复杂度分析：

时间复杂度：O(n)，其中 n 是链表的节点数量。需要对每个节点进行更新指针的操作。
空间复杂度：O(n)，其中 n 是链表的节点数量。空间复杂度主要取决于递归调用的栈空间。

完整代码：

```
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def swapPairs(self, head:ListNode) -> ListNode:
    # 如果节点不存在或者当前只有一个节点，那么无法交换，直接返回
    if not head or not head.next:
        return head

    newHead = head.next  # 新的链表头节点（原始链表的第二个节点）
    head.next = self.swapPairs(newHead.next)  # 而第二个节点的下一个节点变为从原始链表第三个节点开始往后的节点（交换好的）
    newHead.next = head  # 新的链表头节点下一个为原始链表的头节点（已更新好其next节点）

    return newHead
```