88、

def merge(nums1, m, nums2, n):

    # 初始化指针 p1 和 p2，分别指向 nums1 和 nums2 的末尾

    p1 = m - 1

    p2 = n - 1

    # 初始化指针 p，指向 nums1 的末尾

    p = m + n - 1

    # 从后往前遍历，比较 nums1[p1] 和 nums2[p2] 的大小

    while p1 >= 0 and p2 >= 0:

        if nums1[p1] > nums2[p2]:

            nums1[p] = nums1[p1]

            p1 -= 1

        else:

            nums1[p] = nums2[p2]

            p2 -= 1

        p -= 1

    # 如果 nums2 还有剩余元素，则将其复制到 nums1 的前面

    while p2 >= 0:

        nums1[p] = nums2[p2]

        p2 -= 1

        p -= 1

21、

class ListNode:

    def \_\_init\_\_(self, val=0, next=None):

        self.val = val

        self.next = next

def mergeTwoLists(l1, l2):

    # 创建一个哑节点作为新链表的头节点

    dummy = ListNode(0)

    # 创建一个指针指向新链表的当前节点

    curr = dummy

    # 比较 l1 和 l2 的节点值，按升序将节点连接到新链表

    while l1 and l2:

        if l1.val <= l2.val:

            curr.next = l1

            l1 = l1.next

        else:

            curr.next = l2

            l2 = l2.next

        curr = curr.next

    # 将剩余的节点连接到新链表

    if l1:

        curr.next = l1

    if l2:

        curr.next = l2

    # 返回新链表的头节点

    return dummy.next

23、

class ListNode:

    def \_\_init\_\_(self, val=0, next=None):

        self.val = val

        self.next = next

def mergeKLists(lists):

    # 辅助函数，用于合并两个已排序链表

    def mergeTwoLists(l1, l2):

        dummy = ListNode(0)

        curr = dummy

        while l1 and l2:

            if l1.val <= l2.val:

                curr.next = l1

                l1 = l1.next

            else:

                curr.next = l2

                l2 = l2.next

            curr = curr.next

        if l1:

            curr.next = l1

        if l2:

            curr.next = l2

        return dummy.next

    # 递归终止条件

    if len(lists) == 0:

        return None

    if len(lists) == 1:

        return lists[0]

    # 递归分治，将链表数组一分为二，分别合并左右两部分

    mid = len(lists) // 2

    left = mergeKLists(lists[:mid])

    right = mergeKLists(lists[mid:])

    # 合并左右两部分链表

    return mergeTwoLists(left, right)

# 创建链表数组

lists = []

lists.append(ListNode(1))

lists.append(ListNode(2))

lists.append(ListNode(4))

# 合并链表数组

merged = mergeKLists(lists)

# 打印合并后的链表

while merged:

    print(merged.val, end=" ")

    merged = merged.next

147、

class ListNode:

    def \_\_init\_\_(self, val=0, next=None):

        self.val = val

        self.next = next

def insertionSortList(head):

    # 如果链表为空或只有一个节点，直接返回

    if not head or not head.next:

        return head

    # 创建哑节点作为已排序部分的头节点

    dummy = ListNode(0)

    dummy.next = head

    sorted\_tail = head  # 已排序部分的尾节点

    # 遍历链表，将每个节点插入到已排序部分的适当位置

    curr = head.next

    while curr:

        # 如果当前节点的值大于等于已排序部分的尾节点的值，直接将当前节点连接到已排序部分的尾部

        if curr.val >= sorted\_tail.val:

            sorted\_tail = sorted\_tail.next

        else:

            # 从已排序部分的头节点开始寻找插入位置

            prev = dummy

            while prev.next.val < curr.val:

                prev = prev.next

            # 将当前节点插入到已排序部分的适当位置

            temp = curr.next

            curr.next = prev.next

            prev.next = curr

            curr = temp

        # 更新已排序部分的尾节点

        if not curr:

            break

        sorted\_tail.next = curr

    return dummy.next

# 创建链表: 4->2->1->3

head = ListNode(4)

head.next = ListNode(2)

head.next.next = ListNode(1)

head.next.next.next = ListNode(3)

# 对链表进行插入排序

sorted\_head = insertionSortList(head)

# 打印排序后的链表

while sorted\_head:

    print(sorted\_head.val, end=" ")

    sorted\_head = sorted\_head.next

148、

class ListNode:

    def \_\_init\_\_(self, val=0, next=None):

        self.val = val

        self.next = next

def mergeTwoLists(l1, l2):

    if not l1:

        return l2

    if not l2:

        return l1

    if l1.val <= l2.val:

        l1.next = mergeTwoLists(l1.next, l2)

        return l1

    else:

        l2.next = mergeTwoLists(l1, l2.next)

        return l2

def sortList(head):

    if not head or not head.next:

        return head

    # 使用快慢指针找到链表的中点

    slow = head

    fast = head.next

    while fast and fast.next:

        slow = slow.next

        fast = fast.next.next

    # 切分链表为两部分

    mid = slow.next

    slow.next = None

    # 递归排序两部分链表

    left = sortList(head)

    right = sortList(mid)

    # 合并排序后的两部分链表

    return mergeTwoLists(left, right)

# 创建链表: 4->2->1->3

head = ListNode(4)

head.next = ListNode(2)

head.next.next = ListNode(1)

head.next.next.next = ListNode(3)

# 对链表进行排序

sorted\_head = sortList(head)

# 打印排序后的链表

while sorted\_head:

    print(sorted\_head.val, end=" ")

    sorted\_head = sorted\_head.next

41、

def findMissingPositive(nums):

    n = len(nums)

    # 第一次遍历，将每个正整数放到索引为 x-1 的位置上

    for i in range(n):

        while 1 <= nums[i] <= n and nums[i] != nums[nums[i] - 1]:

            # 交换元素，将 nums[i] 放到正确的位置上

            nums[nums[i] - 1], nums[i] = nums[i], nums[nums[i] - 1]

    # 第二次遍历，找到第一个位置上数值不等于索引加一的元素

    for i in range(n):

        if nums[i] != i + 1:

            return i + 1

    # 如果数组中所有位置上的数值都等于索引加一，则返回数组长度加一

    return n + 1

# 测试数组

nums = [3, 4, -1, 1]

# 找到未出现的最小正整数

result = findMissingPositive(nums)

print(result)

75、

def sortColors(nums):

    left = 0

    right = len(nums) - 1

    curr = 0

    while curr <= right:

        if nums[curr] == 0:

            nums[curr], nums[left] = nums[left], nums[curr]

            left += 1

            curr += 1

        elif nums[curr] == 2:

            nums[curr], nums[right] = nums[right], nums[curr]

            right -= 1

        else:

            curr += 1

# 测试数组

nums = [2, 0, 2, 1, 1, 0]

# 对数组进行排序

sortColors(nums)

# 打印排序后的数组

print(nums)

34、

def searchRange(nums, target):

    def findLeft(nums, target):

        left, right = 0, len(nums) - 1

        while left <= right:

            mid = (left + right) // 2

            if nums[mid] < target:

                left = mid + 1

            else:

                right = mid - 1

        return left

    left = findLeft(nums, target)

    right = findLeft(nums, target + 1) - 1

    if left <= right:

        return [left, right]

    else:

        return [-1, -1]

# 测试数组和目标值

nums = [5, 7, 7, 8, 8, 10]

target = 8

# 查找目标值的开始位置和结束位置

result = searchRange(nums, target)

print(result)

74、

def searchMatrix(matrix, target):

    if not matrix or not matrix[0]:

        return False

    rows, cols = len(matrix), len(matrix[0])

    row, col = 0, cols - 1

    while row < rows and col >= 0:

        if matrix[row][col] == target:

            return True

        elif matrix[row][col] > target:

            col -= 1

        else:

            row += 1

    return False

# 测试矩阵和目标值

matrix = [

    [1, 4, 7, 11, 15],

    [2, 5, 8, 12, 19],

    [3, 6, 9, 16, 22],

    [10, 13, 14, 17, 24],

    [18, 21, 23, 26, 30]

]

target = 9

# 判断目标值是否在矩阵中

result = searchMatrix(matrix, target)

print(result)