==========
文件夹: class006_LinkedListReversalAndRelatedAlgorithms
==========

[Markdown 文件]
==========
文件: README.md
==========
Class009 - 链表反转专题

概述
本模块专注于链表反转相关的算法问题,涵盖了从基础的单链表反转到复杂的环形链表检测等多个方面。所有题目均提供 Java、C++、Python 三种语言的实现,并包含详细的注释和复杂度分析。

核心算法思想

1. 迭代法反转链表
- **核心思想**: 使用三个指针(pre, current, next)逐个反转节点指向
- **时间复杂度**: O(n)
- **空间复杂度**: O(1)
- **适用场景**: 完整反转链表或反转链表的某个区间

2. 递归法反转链表
- **核心思想**: 递归到链表末尾,在回溯过程中反转指针
- **时间复杂度**: O(n)
- **空间复杂度**: O(n) - 递归栈深度
- **适用场景**: 代码简洁,但需要注意栈溢出风险

3. 快慢指针
- **核心思想**: 快指针每次走两步,慢指针每次走一步
- **适用场景**:
  - 寻找链表中点
  - 检测环形链表
  - 判断回文链表

题目列表

基础题目

1. 反转链表
题目来源:
- LeetCode 206 - https://leetcode.cn/problems/reverse-linked-list/
- 牛客网 - https://www.nowcoder.com/practice/75e878df47f24fdc9dc3e400ec6058ca

- 剑指 Offer 24
- LintCode 35
- HackerRank

难度: 简单

解法: 迭代法、递归法

关键点:
- 使用三个指针: pre, current, next
- 注意空链表和单节点链表的边界处理

---

2. 反转链表 II
题目来源: LeetCode 92 - https://leetcode.cn/problems/reverse-linked-list-ii/

难度: 中等

题目描述: 反转链表从位置 left 到位置 right 的部分

解法: 头插法

关键点:
- 使用虚拟头节点简化边界处理
- 头插法可以在 O(1)空间内完成区间反转

---

3. K 个一组翻转链表
题目来源: LeetCode 25 - https://leetcode.cn/problems/reverse-nodes-in-k-group/

难度: 困难

题目描述: 每 k 个节点一组进行翻转,不足 k 个的保持原顺序

解法: 分组反转

关键点:
- 先计算链表长度
- 对每组进行 k-1 次头插操作
- 处理剩余不足 k 个的节点

进阶题目

4. 回文链表
题目来源:
- LeetCode 234 - https://leetcode.cn/problems/palindrome-linked-list/
- LintCode 223
- 牛客网 NC78

难度: 简单

解法: 快慢指针 + 反转后半部分链表

关键点:
- 使用快慢指针找到中点
- 反转后半部分链表
- 比较前半部分和反转后的后半部分
- 可选: 恢复链表原结构(良好的工程实践)

---

5. 旋转链表
题目来源:
- LeetCode 61 - https://leetcode.cn/problems/rotate-list/
- LintCode 170

难度: 中等

解法: 首尾相连成环,然后在适当位置断开

关键点:
- k 对链表长度取模,避免多余旋转
- 将链表首尾相连形成环
- 找到新的尾节点位置并断开

---

6. 合并两个有序链表
题目来源:
- LeetCode 21 - https://leetcode.cn/problems/merge-two-sorted-lists/
- LintCode 165
- 剑指 Offer 25

- 牛客网 NC33

难度: 简单

解法: 迭代法或递归法

关键点:
- 使用虚拟头节点简化代码
- 逐个比较两个链表的节点值
- 处理剩余部分

---

7. 两两交换链表中的节点
题目来源:
- LeetCode 24 - https://leetcode.cn/problems/swap-nodes-in-pairs/
- LintCode 451

难度: 中等

解法: 迭代法

关键点:
- 使用虚拟头节点
- 每次交换相邻的两个节点
- 注意指针的正确移动

---

8. 重排链表
题目来源:
- LeetCode 143 - https://leetcode.cn/problems/reorder-list/
- LintCode 99

难度: 中等

题目描述: 按照 L0 → Ln → L1 → Ln-1 → L2 → Ln-2 重新排列

解法: 快慢指针 + 反转 + 合并

关键点:
- 找到中点
- 反转后半部分

- 交替合并两个链表

---

9. 删除链表的倒数第 N 个节点
题目来源:
- LeetCode 19 - https://leetcode.cn/problems/remove-nth-node-from-end-of-list/
- LintCode 174
- 牛客网 NC53
- 剑指 Offer 22

难度: 中等

解法: 快慢指针(双指针法)

关键点:
- 快指针先走 n+1 步
- 快慢指针同时移动
- 慢指针最终指向待删除节点的前一个节点

---

10. 奇偶链表
题目来源:
- LeetCode 328 - https://leetcode.cn/problems/odd-even-linked-list/
- LintCode 1292
- 牛客网 NC142

难度: 中等

题目描述: 将所有奇数位置的节点排在偶数位置的节点之前

解法: 双指针法

关键点:
- odd 指针连接所有奇数位置节点
- even 指针连接所有偶数位置节点
- 最后连接两个链表

---

11. 分隔链表
题目来源:

- LeetCode 86 - https://leetcode.cn/problems/partition-list/
- LintCode 96
- 牛客网 NC188

难度: 中等

题目描述: 将所有小于 x 的节点排在大于等于 x 的节点之前

解法: 双链表法

关键点:
- before 链表存储小于 x 的节点
- after 链表存储大于等于 x 的节点
- 连接两个链表

---

12. 链表求和
题目来源:
- LeetCode 2 - https://leetcode.cn/problems/add-two-numbers/
- LeetCode 445 - https://leetcode.cn/problems/add-two-numbers-ii/
- LintCode 167
- 牛客网 NC40

难度: 中等

题目描述: 两个链表表示的数字相加

解法: 模拟加法过程

关键点:
- 逐位相加
- 处理进位
- 处理不同长度的链表

高级题目

13. 环形链表
题目来源:
- LeetCode 141 - https://leetcode.cn/problems/linked-list-cycle/
- LeetCode 142 - https://leetcode.cn/problems/linked-list-cycle-ii/

- LintCode 102
- 牛客网 NC4
- 剑指 Offer 23

难度: 简单(141) / 中等(142)

题目描述:
- 141: 判断链表是否有环
- 142: 找到环的入口节点

解法: Floyd 判圈算法(快慢指针)

关键点:
- 快指针每次走两步,慢指针每次走一步
- 有环则必然相遇
- 相遇后找入口: 一个指针从头开始,另一个从相遇点开始,相遇点即为入口

数学证明:
设链表头到环入口距离为 a,环入口到相遇点距离为 b,相遇点到环入口距离为 c
- 慢指针走过: a + b
- 快指针走过: a + b + n(b + c)
- 因为快指针速度是慢指针 2 倍: 2(a + b) = a + b + n(b + c)
- 化简: a = (n-1)(b + c) + c
- 这意味着从头到入口的距离 = 从相遇点到入口的距离(加上若干圈环)

---

14. 相交链表
题目来源:
- LeetCode 160 - https://leetcode.cn/problems/intersection-of-two-linked-lists/
- LintCode 380
- 牛客网 NC66
- 剑指 Offer 52

难度: 简单

题目描述: 找到两个单链表相交的起始节点

解法: 双指针法

关键点:
- 两个指针分别从两个链表头开始
- 到达末尾后跳转到另一个链表头部

- 相交则会在相交点相遇,不相交则都会到达 null

巧妙之处:
- pA 走的路径: A 链表 + B 链表 = a + c + b
- pB 走的路径: B 链表 + A 链表 = b + c + a
- 两者路径长度相同,必然在相交点相遇(或都为 null)

---

15. 排序链表
题目来源:
- LeetCode 148 - https://leetcode.cn/problems/sort-list/
- LintCode 98

难度: 中等

题目描述: 在 O(n log n)时间复杂度和常数级空间复杂度下对链表排序

解法: 归并排序

关键点:
- 使用快慢指针找到中点
- 递归排序两半
- 合并两个有序链表
- 自顶向下的归并排序空间复杂度为 O(log n)
- 自底向上的归并排序空间复杂度为 O(1),但实现较复杂

技巧总结

1. 虚拟头节点(Dummy Head)
适用场景:
- 需要修改头节点的情况
- 简化边界条件处理

示例题目:
- 反转链表 II
- 两两交换链表中的节点
- 删除链表的倒数第 N 个节点

代码模板:
```java
```

```
ListNode dummy = new ListNode(0);
dummy.next = head;
// ... 操作
return dummy.next;
```

2. 快慢指针
适用场景:
- 寻找链表中点
- 检测环形链表
- 判断回文链表
- 找到倒数第 k 个节点

代码模板:
```java
ListNode slow = head;
ListNode fast = head;
while (fast != null && fast.next != null) {
    slow = slow.next;
    fast = fast.next.next;
}
// slow 指向中点(或后半部分第一个节点)
```

3. 头插法
适用场景:
- 反转链表区间
- K 个一组翻转链表

代码模板:
```java
ListNode pre = dummy;
ListNode start = pre.next;
ListNode then = start.next;

for (int i = 0; i < k - 1; i++) {
    start.next = then.next;
    then.next = pre.next;
    pre.next = then;
    then = start.next;
}
```

4. 双指针法
适用场景:
- 相交链表
- 删除倒数第 N 个节点
- 奇偶链表
- 分隔链表

核心思想: 使用两个指针分别维护不同的状态或位置

复杂度分析

时间复杂度对比

算法	时间复杂度	备注
迭代反转	O(n)	最优
递归反转	O(n)	递归调用栈
快慢指针找中点	O(n)	最优
归并排序链表	O(n log n)	最优
检测环	O(n)	最优
双指针相交	O(m+n)	最优

空间复杂度对比

算法	空间复杂度	备注
迭代反转	O(1)	最优
递归反转	O(n)	递归栈深度
归并排序(自顶向下)	O(log n)	递归栈
归并排序(自底向上)	O(1)	最优但实现复杂
其他大部分算法	O(1)	只使用常数个指针

边界场景处理

1. 空链表
```java
if (head == null) {
    return null;
}
```

```

### 2. 单节点链表
```java
if (head == null || head.next == null) {
    return head;
}
```

3. 两节点链表
需要特别注意快慢指针的初始化和终止条件

4. 环形链表
注意防止无限循环

语言特性差异

Java vs C++ vs Python

1. 内存管理
- **Java**: 自动垃圾回收
- **C++**: 需要手动 delete 释放内存
- **Python**: 自动垃圾回收

C++示例:
```cpp
ListNode* toDelete = slow->next;
slow->next = slow->next->next;
delete toDelete;  // 必须手动释放内存
```

2. 空指针表示
- **Java**: null
- **C++**: nullptr (C++11) 或 NULL
- **Python**: None

3. 类型系统
- **Java**: 强类型,编译时检查
- **C++**: 强类型,编译时检查
- **Python**: 动态类型,运行时检查
```

工程化考量

1. 异常处理
```java
public static ListNode reverseListSafe(ListNode head) throws IllegalArgumentException {
    if (head == null) {
        return null;
    }

    // 检测环
    if (hasCycle(head)) {
        throw new IllegalArgumentException("链表中存在环,无法进行反转");
    }

    return reverseListIterative(head);
}
```

2. 单元测试
- 正常情况测试
- 边界情况测试(空链表、单节点、两节点)
- 极端情况测试(超长链表)
- 异常情况测试

3. 性能优化
- 减少不必要的遍历
- 避免重复计算
- 考虑缓存策略

4. 线程安全
链表操作通常不是线程安全的,在多线程环境下需要:
- 使用同步机制(synchronized, lock)
- 考虑使用并发容器
- 或者使用不可变数据结构

常见陷阱与调试技巧

1. 断链问题
问题: 修改指针前没有保存下一个节点

```java
// 错误
head.next = pre;
head = head.next;  // head 已经指向 pre,无法前进

// 正确
next = head.next;
head.next = pre;
head = next;
```

2. 边界条件遗漏
- 忘记处理空链表
- 忘记处理单节点链表
- 快慢指针的终止条件写错

3. 调试技巧
打印中间状态:
```java
System.out.println("当前节点: " + current.val);
System.out.println("pre: " + (pre != null ? pre.val : "null"));
```

断言验证:
```java
assert slow != null : "slow 指针不应为 null";
```

面试技巧

1. 思路表达
1. 先说明使用什么算法(迭代/递归/快慢指针等)
2. 说明时间和空间复杂度
3. 提及边界情况的处理
4. 如有更优解,主动说明

2. 代码规范
- 变量命名清晰(pre, current, next, slow, fast)
- 添加关键注释
- 代码结构清晰,易读

3. 主动测试
写完代码后主动说:
- "让我用几个测试用例验证一下"
- "空链表的情况是..."
- "单节点的情况是..."

4. 举一反三
- "这个题目可以扩展到双链表"
- "如果要求空间复杂度 O(1),可以..."
- "这个算法在实际工程中的应用场景是..."

实际应用场景

1. 文本编辑器
- 撤销/重做功能(双向链表)
- 光标移动

2. 浏览器历史记录
- 前进/后退功能

3. LRU 缓存
- 结合哈希表和双向链表实现

4. 音乐播放器
- 播放列表管理
- 循环播放(环形链表)

学习建议

1. 掌握基础
先掌握基础的迭代反转和递归反转

2. 理解原理
深入理解快慢指针的数学原理

3. 多练习
- 手写代码,不要只看
- 画图理解指针的变化
- 尝试用不同方法解决同一问题

4. 总结规律
- 什么时候用虚拟头节点
- 什么时候用快慢指针
- 不同场景的最优解是什么

5. 关注细节
- 边界条件
- 内存泄漏(C++)
- 空指针异常

补充题目 16-20

16. 链表随机节点
题目来源:
- LeetCode 382 - https://leetcode.cn/problems/linked-list-random-node/
- 蓄水池抽样算法应用

难度: 中等

题目描述: 从链表中随机选择一个节点，每个节点被选中的概率相等

解法: 蓄水池抽样算法

关键点:
- 遍历链表时，以 1/i 的概率选择当前节点
- 保证每个节点被选中的概率都是 1/n
- 只需要遍历一次，空间复杂度 O(1)

时间复杂度: O(n)
空间复杂度: O(1)

应用场景:
- 大数据流中的随机抽样
- 在线算法设计
- 推荐系统中的随机推荐

---

17. 复制带随机指针的链表
题目来源:

- LeetCode 138 - https://leetcode.cn/problems/copy-list-with-random-pointer/
- 剑指 Offer 35

难度: 中等

题目描述: 深度复制一个包含随机指针的链表

解法: 三次遍历法

关键点:
1. 第一次遍历：在每个节点后插入复制节点
2. 第二次遍历：设置复制节点的随机指针
3. 第三次遍历：分离原链表和复制链表

时间复杂度: O(n)
空间复杂度: O(1) - 不计算结果链表

工程意义:
- 深拷贝复杂数据结构
- 对象序列化和反序列化
- 内存管理中的对象复制

---

18. 链表组件
题目来源:
- LeetCode 817 - https://leetcode.cn/problems/linked-list-components/

难度: 中等

题目描述: 计算链表中在给定数组中的连续节点段的数量

解法: 哈希集合 + 遍历

关键点:
- 使用哈希集合存储给定数字，实现 O(1)查找
- 遍历链表，统计连续段的开始位置
- 注意边界条件和空链表处理

时间复杂度: O(n + m) - n 为链表长度，m 为数组长度
空间复杂度: O(m) - 哈希集合存储数组元素

实际应用:

- 网络连接中的连通组件检测
- 图像处理中的连通区域标记
- 社交网络中的社区发现

---

19. 链表中的下一个更大节点
题目来源:
- LeetCode 1019 - https://leetcode.cn/problems/next-greater-node-in-linked-list/

难度: 中等

题目描述: 对于链表中的每个节点，找到它后面第一个比它大的节点

解法: 单调栈

关键点:
1. 将链表转换为数组
2. 从右向左遍历，使用单调递减栈
3. 栈中存储的是尚未找到下一个更大节点的元素

时间复杂度: O(n)
空间复杂度: O(n) - 栈和结果数组

算法思想:
- 单调栈的经典应用
- 处理"下一个更大元素"问题的通用模式
- 可以扩展到二维和三维情况

---

20. 链表最大孪生和
题目来源:
- LeetCode 2130 - https://leetcode.cn/problems/maximum-twin-sum-of-a-linked-list/

难度: 中等

题目描述: 找到链表中对称位置节点值的最大和

解法: 快慢指针 + 反转

关键点:
1. 使用快慢指针找到中点

2. 反转后半部分链表
3. 同时遍历前半部分和反转后的后半部分，计算和的最大值

时间复杂度: O(n)
空间复杂度: O(1)

数学原理:
- 链表长度的奇偶性处理
- 对称位置的数学关系：第 i 个节点和第 n-1-i 个节点
- 双指针技术的灵活应用

---

21. 合并 K 个升序链表
题目来源:
- LeetCode 23 - https://leetcode.cn/problems/merge-k-sorted-lists/
- LintCode 104 - https://www.lintcode.com/problem/merge-k-sorted-lists/
- 牛客网 NC127 - 合并 k 个已排序的链表

难度: 困难

题目描述: 给你一个链表数组，每个链表都已经按升序排列。请你将所有链表合并到一个升序链表中，返回合并后的链表。

解法: 优先队列(最小堆)

关键点:
- 使用优先队列维护当前所有链表的最小节点
- 将所有非空链表的头节点加入最小堆
- 每次从堆中取出最小节点，加入结果链表
- 将取出节点的下一个节点加入堆中(如果不为空)

时间复杂度: O(N log K) - N 是所有节点总数，K 是链表数量
空间复杂度: O(K) - 堆的大小
是否为最优解: 是

---

22. 删除链表中的节点
题目来源:
- LeetCode 237 - https://leetcode.cn/problems/delete-node-in-a-linked-list/
- LintCode 37 - https://www.lintcode.com/problem/delete-node-in-a-linked-list/
- 牛客网 NC138 - 删除链表的节点

难度: 简单

题目描述: 有一个单链表的 head，我们想删除它其中的一个节点 node。给你一个需要删除的节点 node 。你将 无法访问 第一个节点 head。

解法: 值替换法

关键点:
- 将下一个节点的值复制到当前节点
- 删除下一个节点
- 注意题目保证不会删除最后一个节点

时间复杂度: O(1) - 只需要常数时间
空间复杂度: O(1) - 只使用常数级别的额外空间
是否为最优解: 是

---

23. 删除排序链表中的重复元素
题目来源:
- LeetCode 83 - https://leetcode.cn/problems/remove-duplicates-from-sorted-list/
- LintCode 112 - https://www.lintcode.com/problem/remove-duplicates-from-sorted-list/
- 牛客网 NC141 - 判断一个链表是否为回文结构

难度: 简单

题目描述: 给定一个已排序的链表的头 head ，删除所有重复的元素，使每个元素只出现一次 。返回已排序的链表。

解法: 单指针遍历

关键点:
- 利用链表已排序的特性
- 当当前节点值等于下一个节点值时，跳过下一个节点

时间复杂度: O(n) - 需要遍历链表一次
空间复杂度: O(1) - 只使用常数级别的额外空间
是否为最优解: 是

---

24. 删除排序链表中的重复元素 II

题目来源:
- LeetCode 82 - https://leetcode.cn/problems/remove-duplicates-from-sorted-list-ii/
- LintCode 113 - https://www.lintcode.com/problem/remove-duplicates-from-sorted-list-ii/
- 牛客网 NC140

难度: 中等

题目描述: 给定一个已排序的链表的头 head ，删除原始链表中所有重复数字的节点，只留下不同的数字 。返回已排序的链表。

解法: 虚拟头节点 + 双指针

关键点:
- 与题目 23 不同，需要删除所有重复节点
- 使用虚拟头节点简化边界处理
- 需要记录重复值并在遇到时跳过

时间复杂度: O(n) - 需要遍历链表一次
空间复杂度: O(1) - 只使用常数级别的额外空间
是否为最优解: 是

---

25. 移除链表元素
题目来源:
- LeetCode 203 - https://leetcode.cn/problems/remove-linked-list-elements/
- 牛客网相关题目

难度: 简单

题目描述: 给你一个链表的头节点 head 和一个整数 val ，请你删除链表中所有满足 Node.val == val 的节点，并返回新的头节点。

解法: 虚拟头节点法

关键点:
- 使用虚拟头节点简化删除头节点的情况
- 遍历链表，遇到值等于 val 的节点就删除

时间复杂度: O(n) - 需要遍历链表一次
空间复杂度: O(1) - 只使用常数级别的额外空间
是否为最优解: 是

扩展阅读与进阶题目

相关数据结构深入理解
- **双向链表**: 支持双向遍历，适用于需要前后移动的场景
- **循环链表**: 尾节点指向头节点，适用于环形缓冲区
- **跳表(Skip List)**: 多级索引结构，支持快速查找
- **XOR 链表**: 使用异或操作存储前后节点指针，节省空间

相关算法扩展
- **LRU 缓存算法**: 结合哈希表和双向链表实现 O(1)操作
- **约瑟夫环问题**: 使用循环链表模拟淘汰过程
- **链表排序算法**: 归并排序、快速排序在链表上的应用
- **多级链表扁平化**: 处理嵌套链表结构

Floyd 判圈算法的其他应用
- **检测图中的环**: 应用于有向图和无向图的环检测
- **伪随机数生成器的周期检测**: 判断随机数序列的周期性
- **寻找重复数字**: 在数组中找到重复元素

更多算法平台题目补充

HackerRank 相关题目
1. **Reverse a linked list** - https://www.hackerrank.com/challenges/reverse-a-linked-list
2. **Insert a node at a specific position** - https://www.hackerrank.com/challenges/insert-a-node-at-a-specific-position-in-a-linked-list
3. **Delete a node** - https://www.hackerrank.com/challenges/delete-a-node-from-a-linked-list

Codeforces 相关题目
1. **A. Reverse a Substring** - https://codeforces.com/problemset/problem/1155/A
2. **B. Reverse Binary Strings** - https://codeforces.com/problemset/problem/1437/B

AtCoder 相关题目
1. **Reverse and Compare** - https://atcoder.jp/contests/agc019/tasks/agc019_b
2. **Reverse LCS** - https://atcoder.jp/contests/abc147/tasks/abc147_d

USACO 相关题目
1. **Reverse Engineering** - USACO 训练题目，涉及字符串反转操作
2. **Linked List Problems** - USACO 竞赛中的链表相关题目

洛谷相关题目
1. **P1996 约瑟夫问题** - https://www.luogu.com.cn/problem/P1996
2. **P1160 队列安排** - https://www.luogu.com.cn/problem/P1160

各大高校OJ题目
1. **杭电OJ 2066** - 链表操作题目
2. **北大OJ 1007** - 链表反转相关
3. **浙大OJ 1005** - 链表排序题目

工程化实践与性能优化

1. 内存管理最佳实践
- **Java**: 利用垃圾回收机制，注意避免内存泄漏
- **C++**: 手动管理内存，确保正确释放
- **Python**: 引用计数和垃圾回收结合

2. 线程安全考虑
- 使用同步机制保护共享链表
- 考虑使用不可变数据结构
- 读写锁优化并发访问

3. 性能监控与调优
- 监控链表操作的时间复杂度
- 分析内存使用情况
- 优化缓存命中率

4. 测试策略
- 单元测试覆盖所有边界情况
- 性能测试验证大规模数据处理能力
- 集成测试确保系统稳定性

机器学习与链表算法的联系

1. 图神经网络(GNN)中的链表思想
- 消息传递机制类似于链表遍历
- 邻居聚合操作借鉴了链表连接思想

2. 序列模型中的链表应用
- RNN、LSTM处理序列数据时类似链表操作

- 注意力机制中的位置编码与链表顺序相关

3. 强化学习中的状态转移
- 马尔可夫决策过程的状态转移类似链表连接
- 经验回放缓冲区使用链表-like 结构

反直觉但关键的设计要点

1. 虚拟头节点的必要性
- 简化边界条件处理
- 统一操作逻辑，减少特殊判断

2. 快慢指针的数学原理
- Floyd 判圈算法的数学证明
- 中点寻找的精确性分析

3. 递归与迭代的选择依据
- 空间复杂度考虑
- 代码可读性平衡
- 栈深度限制

极端场景与边界处理

1. 超大规模数据处理
- 分块处理策略
- 流式处理避免内存溢出

2. 并发访问场景
- 读写锁优化
- 无锁数据结构考虑

3. 异常输入处理
- 空指针检查
- 非法参数验证
- 循环链表检测

语言特性差异深度分析

Java vs C++ vs Python 关键差异

内存管理差异
```java
// Java - 自动垃圾回收
ListNode node = new ListNode(1);
// 不需要手动释放
```

```cpp
// C++ - 手动内存管理
ListNode* node = new ListNode(1);
// 必须手动释放
delete node;
```

```python
# Python - 引用计数 + 垃圾回收
node = ListNode(1)
# 自动管理，但有循环引用风险
```

性能特征对比
- **Java**: JIT 优化，运行时性能优秀
- **C++**: 编译时优化，运行效率最高
- **Python**: 解释执行，开发效率高但运行慢

总结与学习路径

掌握链表反转的四个层次

第一层：基础操作
- 理解指针操作原理
- 掌握迭代和递归两种方法
- 处理简单边界情况

第二层：进阶技巧
- 使用虚拟头节点简化代码
- 掌握快慢指针等高级技巧
- 处理复杂边界场景

第三层：工程实践
- 考虑线程安全和性能优化
- 实现完整的错误处理机制
- 编写高质量的单元测试

第四层：创新应用
- 将链表思想应用到其他领域
- 设计新的链表算法和数据结构
- 解决实际工程问题

补充题目 21-30

21. 合并 K 个升序链表
题目来源:
- LeetCode 23 - https://leetcode.cn/problems/merge-k-sorted-lists/
- LintCode 104
- 牛客网 NC127

难度: 困难

题目描述: 给你一个链表数组，每个链表都已经按升序排列。请你将所有链表合并到一个升序链表中，返回合并后的链表。

解法: 分治法、优先队列(最小堆)

关键点:
- 使用分治思想将 K 个链表两两合并
- 或使用优先队列维护当前所有链表的最小节点
- 时间复杂度 O(N log K)，其中 N 是所有节点总数，K 是链表数量

---

22. 删除链表中的节点
题目来源:
- LeetCode 237 - https://leetcode.cn/problems/delete-node-in-a-linked-list/
- LintCode 37
- 牛客网 NC138

难度: 简单

题目描述: 有一个单链表的 head，我们想删除它其中的一个节点 node。给你一个需要删除的节点 node 。你将 无法访问 第一个节点 head。

解法: 值替换法

关键点:
- 将下一个节点的值复制到当前节点
- 删除下一个节点
- 注意题目保证不会删除最后一个节点

---

23. 删除排序链表中的重复元素
题目来源:
- LeetCode 83 - https://leetcode.cn/problems/remove-duplicates-from-sorted-list/
- LintCode 112
- 牛客网 NC141

难度: 简单

题目描述: 给定一个已排序的链表的头 head ，删除所有重复的元素，使每个元素只出现一次 。返回已排序的链表。

解法: 单指针遍历

关键点:
- 利用链表已排序的特性
- 当当前节点值等于下一个节点值时，跳过下一个节点

---

24. 删除排序链表中的重复元素 II
题目来源:
- LeetCode 82 - https://leetcode.cn/problems/remove-duplicates-from-sorted-list-ii/
- LintCode 113
- 牛客网 NC140

难度: 中等

题目描述: 给定一个已排序的链表的头 head ，删除原始链表中所有重复数字的节点，只留下不同的数字 。返回已排序的链表。

解法: 虚拟头节点 + 双指针

关键点:

- 与题目 23 不同，需要删除所有重复节点
- 使用虚拟头节点简化边界处理
- 需要记录重复值并在遇到时跳过

---

25. 设计链表
题目来源:
- LeetCode 707 - https://leetcode.cn/problems/design-linked-list/
- LintCode 1843

难度: 中等

题目描述: 设计链表的实现。您可以选择使用单链表或双链表。

解法: 单链表或双链表实现

关键点:
- 实现 get、addAtHead、addAtTail、addAtIndex、deleteAtIndex 等操作
- 注意边界条件和索引有效性检查
- 双链表可以优化某些操作的时间复杂度

---

26. LRU 缓存
题目来源:
- LeetCode 146 - https://leetcode.cn/problems/lru-cache/
- LintCode 134
- 牛客网 NC143

难度: 中等

题目描述: 设计和构建一个"最近最少使用"缓存，该缓存会删除最近最少使用的项目。

解法: 哈希表 + 双向链表

关键点:
- 使用双向链表维护访问顺序
- 使用哈希表实现 O(1)的查找
- 访问或插入节点时将其移到链表头部
- 容量满时删除链表尾部节点

---

27. 复制带随机指针的链表
题目来源:
- LeetCode 138 - https://leetcode.cn/problems/copy-list-with-random-pointer/
- LintCode 105
- 剑指 Offer 35

难度: 中等

题目描述: 给你一个长度为 n 的链表，每个节点包含一个额外增加的随机指针 random ，该指针可以指向链表中的任何节点或空节点。

解法: 三次遍历法、哈希表法

关键点:
- 方法 1：三次遍历，在每个节点后插入复制节点
- 方法 2：使用哈希表存储原节点到新节点的映射
- 注意处理 random 指针的正确指向

---

28. 有序链表转换二叉搜索树
题目来源:
- LeetCode 109 - https://leetcode.cn/problems/convert-sorted-list-to-binary-search-tree/
- LintCode 106
- 牛客网 NC144

难度: 中等

题目描述: 给定一个单链表的头节点 head ，其中的元素按升序排序，将其转换为高度平衡的二叉搜索树。

解法: 快慢指针 + 递归

关键点:
- 使用快慢指针找到链表中点作为根节点
- 递归构建左右子树
- 注意断开链表以避免重复访问

---

29. 设计跳表
题目来源:

- LeetCode 1206 - https://leetcode.cn/problems/design-skiplist/
- LintCode 1844

难度: 困难

题目描述: 不使用任何库函数，设计一个跳表。

解法: 多层链表结构

关键点:
- 跳表是一种概率性数据结构
- 通过多层索引实现 O(log n)的查找、插入、删除
- 每个节点可能出现在多个层级
- 需要随机化算法决定节点层级

---

30. 链表相加
题目来源:
- LeetCode 2 - https://leetcode.cn/problems/add-two-numbers/
- LeetCode 445 - https://leetcode.cn/problems/add-two-numbers-ii/
- LintCode 167
- 牛客网 NC40

难度: 中等

题目描述: 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方式存储的，并且每个节点只能存储一位数字。

解法: 模拟加法

关键点:
- 从低位到高位逐位相加
- 处理进位
- 处理两个链表长度不同的情况
- 注意最高位可能有进位

学习建议
1. **循序渐进**: 从简单题目开始，逐步增加难度
2. **多语言实现**: 用不同语言实现同一算法，理解语言特性
3. **实际应用**: 将学到的技巧应用到实际项目中
4. **持续练习**: 定期复习和练习，保持熟练度

各大算法平台题目汇总

LeetCode 链表相关题目
- [LeetCode 206. 反转链表](https://leetcode.cn/problems/reverse-linked-list/)
- [LeetCode 92. 反转链表 II](https://leetcode.cn/problems/reverse-linked-list-ii/)
- [LeetCode 25. K 个一组翻转链表](https://leetcode.cn/problems/reverse-nodes-in-k-group/)
- [LeetCode 234. 回文链表](https://leetcode.cn/problems/palindrome-linked-list/)
- [LeetCode 61. 旋转链表](https://leetcode.cn/problems/rotate-list/)
- [LeetCode 21. 合并两个有序链表](https://leetcode.cn/problems/merge-two-sorted-lists/)
- [LeetCode 23. 合并 K 个升序链表](https://leetcode.cn/problems/merge-k-sorted-lists/)
- [LeetCode 24. 两两交换链表中的节点](https://leetcode.cn/problems/swap-nodes-in-pairs/)
- [LeetCode 143. 重排链表](https://leetcode.cn/problems/reorder-list/)
- [LeetCode 19. 删除链表的倒数第 N 个节点](https://leetcode.cn/problems/remove-nth-node-from-end-of-list/)
- [LeetCode 328. 奇偶链表](https://leetcode.cn/problems/odd-even-linked-list/)
- [LeetCode 86. 分隔链表](https://leetcode.cn/problems/partition-list/)
- [LeetCode 2. 两数相加](https://leetcode.cn/problems/add-two-numbers/)
- [LeetCode 445. 两数相加 II](https://leetcode.cn/problems/add-two-numbers-ii/)
- [LeetCode 141. 环形链表](https://leetcode.cn/problems/linked-list-cycle/)
- [LeetCode 142. 环形链表 II](https://leetcode.cn/problems/linked-list-cycle-ii/)
- [LeetCode 160. 相交链表](https://leetcode.cn/problems/intersection-of-two-linked-lists/)
- [LeetCode 148. 排序链表](https://leetcode.cn/problems/sort-list/)
- [LeetCode 382. 链表随机节点](https://leetcode.cn/problems/linked-list-random-node/)
- [LeetCode 138. 复制带随机指针的链表](https://leetcode.cn/problems/copy-list-with-random-pointer/)
- [LeetCode 817. 链表组件](https://leetcode.cn/problems/linked-list-components/)
- [LeetCode 1019. 链表中的下一个更大节点](https://leetcode.cn/problems/next-greater-node-in-linked-list/)
- [LeetCode 2130. 链表最大孪生和](https://leetcode.cn/problems/maximum-twin-sum-of-a-linked-list/)
- [LeetCode 237. 删除链表中的节点](https://leetcode.cn/problems/delete-node-in-a-linked-list/)
- [LeetCode 83. 删除排序链表中的重复元素](https://leetcode.cn/problems/remove-duplicates-from-sorted-list/)
- [LeetCode 82. 删除排序链表中的重复元素 II](https://leetcode.cn/problems/remove-duplicates-from-sorted-list-ii/)
- [LeetCode 707. 设计链表](https://leetcode.cn/problems/design-linked-list/)
- [LeetCode 146. LRU 缓存](https://leetcode.cn/problems/lru-cache/)
- [LeetCode 109. 有序链表转换二叉搜索树](https://leetcode.cn/problems/convert-sorted-list-to-binary-search-tree/)
- [LeetCode 1206. 设计跳表](https://leetcode.cn/problems/design-skiplist/)

牛客网链表相关题目
- [牛客网 反转链表](https://www.nowcoder.com/practice/75e878df47f24fdc9dc3e400ec6058ca)

- [牛客网 NC78 链表中倒数最后 k 个结点](https://www.nowcoder.com/practice/529d3ae5a407492994ad2a246518148a)
- [牛客网 NC33 合并两个排序的链表](https://www.nowcoder.com/practice/d8b6b4358f774294a89de2a6ac4d9337)
- [牛客网 NC53 删除链表的倒数第 n 个节点](https://www.nowcoder.com/practice/f95dcdafbde44b22a6d741baf716510f)
- [牛客网 NC142 链表的奇偶重排](https://www.nowcoder.com/practice/02bf49ea45cd486caca48f74d007dc97)
- [牛客网 NC188 分隔链表](https://www.nowcoder.com/practice/02bf49ea45cd486caca48f74d007dc97)
- [牛客网 NC40 链表相加（二）](https://www.nowcoder.com/practice/c56f6c70fb3f4849bc56e33ff2a50b6b)
- [牛客网 NC4 判断链表中是否有环](https://www.nowcoder.com/practice/650474f313294468a4ded3ce0f7898b5)
- [牛客网 NC66 两个链表的第一个公共结点](https://www.nowcoder.com/practice/6ab1d9a29e88450685099d45c9e31e46)
- [牛客网 NC12 重建二叉树](https://www.nowcoder.com/practice/8a19cbe657394eeaac2f6ea9b0f6fcf6)
- [牛客网 NC138 删除链表的节点](https://www.nowcoder.com/practice/fc533c45b73a41b0b44ccba763f866ef)
- [牛客网 NC141 判断一个链表是否为回文结构](https://www.nowcoder.com/practice/3fed228444e740c8be66232ce8b87c2f)
- [牛客网 NC143 LRU 缓存结构设计](https://www.nowcoder.com/practice/3fed228444e740c8be66232ce8b87c2f)
- [牛客网 NC127 合并 k 个已排序的链表](https://www.nowcoder.com/practice/65cfde9e5b9b4cf2b6bafa5f3ef33fa6)

CodeChef 链表相关题目
- [CodeChef LKDNGOLF](https://www.codechef.com/problems/LKDNGOLF) (链表相关思维题)
- [CodeChef LSTGRPH](https://www.codechef.com/problems/LSTGRPH) (链表图论结合)

SPOJ 链表相关题目
- [SPOJ ETF](https://www.spoj.com/problems/ETF/) (链表与数论结合)
- [SPOJ DQUERY](https://www.spoj.com/problems/DQUERY/) (链表与数据结构结合)

Project Euler 链表相关题目
- [Project Euler Problem 19](https://projecteuler.net/problem=19) (链表与日期计算)
- [Project Euler Problem 67](https://projecteuler.net/problem=67) (链表与动态规划)

HackerEarth 链表相关题目
- [HackerEarth Monk and the Magical Candy Bags](https://www.hackerearth.com/practice/algorithms/greedy/basics-of-greedy-algorithms/practice-problems/algorithm/monk-and-the-magical-candy-bags/) (链表与贪心算法)
- [HackerEarth Remove Friends](https://www.hackerearth.com/practice/algorithms/greedy/basics-of-greedy-algorithms/practice-problems/algorithm/remove-friends-5/) (链表操作)

计蒜客链表相关题目
- [计蒜客 T1001 链表基本操作](https://nanti.jisuanke.com/t/T1001)
- [计蒜客 T1002 链表反转](https://nanti.jisuanke.com/t/T1002)

各大高校 OJ 链表相关题目
- [杭电 OJ 2066](http://acm.hdu.edu.cn/showproblem.php?pid=2066) (链表应用)
- [北大 OJ 1007](http://poj.org/problem?id=1007) (DNA 排序与链表)
- [浙大 OJ 1005](http://acm.zju.edu.cn/onlinejudge/showProblem.do?problemCode=1005) (链表模拟)
- [华科 OJ 1001](http://acm.hust.edu.cn/problem/show/1001) (链表基础)

USACO 链表相关题目
- [USACO January 2015 Gold - Moovie Mooving](http://www.usaco.org/index.php?page=viewproblem2&cpid=530) (链表与动态规划)
- [USACO December 2014 Silver - Marathon](http://www.usaco.org/index.php?page=viewproblem2&cpid=491) (链表与最短路径)

Codeforces 链表相关题目
- [Codeforces 1155A Reverse a Substring](https://codeforces.com/problemset/problem/1155/A)
- [Codeforces 1437B Reverse Binary Strings](https://codeforces.com/problemset/problem/1437/B)

AtCoder 链表相关题目
- [AtCoder AGC019B Reverse and Compare](https://atcoder.jp/contests/agc019/tasks/agc019_b)
- [AtCoder ABC147D Xor Sum 4](https://atcoder.jp/contests/abc147/tasks/abc147_d)

洛谷 链表相关题目
- [洛谷 P1996 约瑟夫问题](https://www.luogu.com.cn/problem/P1996)
- [洛谷 P1160 队列安排](https://www.luogu.com.cn/problem/P1160)

MarsCode 链表相关题目
- [MarsCode P1001 链表反转](https://www.marscode.cn/problem/P1001)
- [MarsCode P1002 链表合并](https://www.marscode.cn/problem/P1002)

UVa OJ 链表相关题目
- [UVa 11988 Broken Keyboard](https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=3139) (链表模拟)

TimusOJ 链表相关题目
- [Timus 1601 AntiCAPS](https://acm.timus.ru/problem.aspx?space=1&num=1601) (链表字符串处理)

AizuOJ 链表相关题目
- [Aizu ALDS1_1_A Insertion Sort](http://judge.u-

aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_1_A) (链表排序)

Comet OJ 链表相关题目
- [Comet OJ C1001 链表操作](https://www.cometoj.com/problem/C1001)

LOJ 链表相关题目
- [LOJ #10199. 「一本通 1.3 练习 4」Addition Chains](https://loj.ac/problem/10199) (链表与搜索)

LintCode 链表相关题目
- [LintCode 35. 翻转链表](https://www.lintcode.com/problem/reverse-linked-list)
- [LintCode 36. 翻转链表 II](https://www.lintcode.com/problem/reverse-linked-list-ii)
- [LintCode 451. 两两交换链表中的节点](https://www.lintcode.com/problem/swap-nodes-in-pairs)
- [LintCode 99. 重排链表](https://www.lintcode.com/problem/reorder-list)
- [LintCode 165. 合并两个排序链表](https://www.lintcode.com/problem/merge-two-sorted-lists)
- [LintCode 174. 删除链表中倒数第 n 个节点](https://www.lintcode.com/problem/remove-nth-node-from-end-of-list)
- [LintCode 170. 旋转链表](https://www.lintcode.com/problem/rotate-list)
- [LintCode 1292. 奇偶链表](https://www.lintcode.com/problem/odd-even-linked-list)
- [LintCode 96. 分隔链表](https://www.lintcode.com/problem/partition-list)
- [LintCode 167. 链表求和](https://www.lintcode.com/problem/add-two-numbers)
- [LintCode 102. 环形链表](https://www.lintcode.com/problem/linked-list-cycle)
- [LintCode 380. 相交链表](https://www.lintcode.com/problem/intersection-of-two-linked-lists)
- [LintCode 98. 排序链表](https://www.lintcode.com/problem/sort-list)
- [LintCode 223. 回文链表](https://www.lintcode.com/problem/palindrome-linked-list)
- [LintCode 105. 复制带随机指针的链表](https://www.lintcode.com/problem/copy-list-with-random-pointer)
- [LintCode 104. 合并 k 个排序链表](https://www.lintcode.com/problem/merge-k-sorted-lists)
- [LintCode 37. 删除链表中的节点](https://www.lintcode.com/problem/delete-node-in-a-linked-list)
- [LintCode 112. 删除排序链表中的重复元素](https://www.lintcode.com/problem/remove-duplicates-from-sorted-list)
- [LintCode 113. 删除排序链表中的重复元素 II](https://www.lintcode.com/problem/remove-duplicates-from-sorted-list-ii)
- [LintCode 106. 有序链表转换二叉搜索树](https://www.lintcode.com/problem/convert-sorted-list-to-binary-search-tree)
- [LintCode 134. LRU 缓存](https://www.lintcode.com/problem/lru-cache)

HackerRank 链表相关题目
- [HackerRank Reverse a linked list](https://www.hackerrank.com/challenges/reverse-a-linked-list)
- [HackerRank Insert a node at a specific position in a linked list](https://www.hackerrank.com/challenges/insert-a-node-at-a-specific-position-in-a-linked-list)
- [HackerRank Delete a node from a linked list](https://www.hackerrank.com/challenges/delete-a-node-from-a-linked-list)

- [HackerRank Print in Reverse](https://www.hackerrank.com/challenges/print-the-elements-of-a-linked-list-in-reverse)
- [HackerRank Reverse a doubly linked list](https://www.hackerrank.com/challenges/reverse-a-doubly-linked-list)
- [HackerRank Find Merge Point of Two Lists](https://www.hackerrank.com/challenges/find-the-merge-point-of-two-joined-linked-lists)
- [HackerRank Inserting a Node Into a Sorted Doubly Linked List](https://www.hackerrank.com/challenges/insert-a-node-into-a-sorted-doubly-linked-list)

剑指 Offer 链表相关题目
- [剑指 Offer 24. 反转链表](https://leetcode.cn/problems/fan-zhuan-lian-biao-lcof/)
- [剑指 Offer 22. 链表中倒数第 k 个节点](https://leetcode.cn/problems/lian-biao-zhong-dao-shu-di-kge-jie-dian-lcof/)
- [剑指 Offer 25. 合并两个排序的链表](https://leetcode.cn/problems/he-bing-liang-ge-pai-xu-de-lian-biao-lcof/)
- [剑指 Offer 06. 从尾到头打印链表](https://leetcode.cn/problems/cong-wei-dao-tou-da-yin-lian-biao-lcof/)
- [剑指 Offer 52. 两个链表的第一个公共节点](https://leetcode.cn/problems/liang-ge-lian-biao-de-di-yi-ge-gong-gong-jie-dian-lcof/)
- [剑指 Offer 23. 链表中环的入口节点](https://leetcode.cn/problems/lian-biao-zhong-huan-de-ru-kou-jie-dian-lcof/)
- [剑指 Offer 35. 复杂链表的复制](https://leetcode.cn/problems/fu-za-lian-biao-de-fu-zhi-lcof/)

Codeforces 链表相关题目
- [Codeforces 1155A Reverse a Substring](https://codeforces.com/problemset/problem/1155/A)
- [Codeforces 1437B Reverse Binary Strings](https://codeforces.com/problemset/problem/1437/B)

AtCoder 链表相关题目
- [AtCoder AGC019B Reverse and Compare](https://atcoder.jp/contests/agc019/tasks/agc019_b)
- [AtCoder ABC147D Xor Sum 4](https://atcoder.jp/contests/abc147/tasks/abc147_d)

洛谷 链表相关题目
- [洛谷 P1996 约瑟夫问题](https://www.luogu.com.cn/problem/P1996)
- [洛谷 P1160 队列安排](https://www.luogu.com.cn/problem/P1160)

各大高校 OJ 链表相关题目
- [杭电 OJ 2066](http://acm.hdu.edu.cn/showproblem.php?pid=2066)
- [北大 OJ 1007](http://poj.org/problem?id=1007)
- [浙大 OJ 1005](http://acm.zju.edu.cn/onlinejudge/showProblem.do?problemCode=1005)

代码文件说明与使用指南

文件结构
- `ListReverse.java`: Java 实现，包含 15 个核心题目和完整测试
- `ListReverse.cpp`: C++实现，包含内存管理和性能优化
- `ListReverse.py`: Python 实现，注重代码简洁和可读性

编译运行指南

Java 版本
```bash
javac ListReverse.java
java ListReverse
```

C++版本
```bash
g++ -std=c++11 -o ListReverse_test ListReverse.cpp
./ListReverse_test
```

Python 版本
```bash
python ListReverse.py
```

测试覆盖
- 每个算法都有对应的测试函数
- 覆盖正常情况、边界情况和异常情况
- 包含性能测试和正确性验证

未来学习方向

算法进阶
- 学习更复杂的数据结构（树、图等）
- 掌握动态规划、贪心算法等高级技巧
- 研究算法复杂度分析和优化方法

系统设计
- 学习大规模系统设计原则
- 掌握分布式系统基础知识
- 了解高并发和高可用设计

工程实践
- 参与开源项目贡献
- 学习软件工程最佳实践
- 掌握代码重构和优化技巧

最后更新: 2025-10-28
版本: 2.3
更新内容:
- 补充更多算法平台题目
- 增加工程化实践内容
- 添加机器学习联系
- 完善测试和文档
- 添加题目 21-30
- 补充各大算法平台题目链接
- 添加 CodeChef、SPOJ、Project Euler、HackerEarth、计蒜客等平台题目
- 补充各大高校 OJ、USACO、Codeforces 等平台题目
- 添加合并 K 个升序链表、删除链表中的节点、删除排序链表中的重复元素、删除排序链表中的重复元素 II、移除链表元素等 5 个新题目

==========

[代码文件]
==========
文件: ListReverse.cpp
==========
```
#include <iostream>
#include <vector>
#include <stack>
#include <unordered_set>
#include <chrono>
#include <stdexcept>
#include <cstdlib>
#include <ctime>

using namespace std;

/**
 * 链表反转相关算法题目集合
 * 包含 LeetCode、牛客网、Codeforces、LintCode、HackerRank 等平台的相关题目
 * 每个题目都提供详细的解题思路、复杂度分析和多种解法
```

```
 *
 * 题目列表：
 * 1. 反转链表 (LeetCode 206, 牛客网, HackerRank, LintCode 35, 剑指 Offer 24)
 * 2. 反转链表 II (LeetCode 92)
 * 3. K 个一组翻转链表 (LeetCode 25)
 * 4. 回文链表 (LeetCode 234, LintCode 223, 牛客网 NC78)
 * 5. 旋转链表 (LeetCode 61, LintCode 170)
 * 6. 合并两个有序链表 (LeetCode 21, LintCode 165, 剑指 Offer 25, 牛客网 NC33)
 * 7. 两两交换链表中的节点 (LeetCode 24, LintCode 451)
 * 8. 重排链表 (LeetCode 143, LintCode 99)
 * 9. 删除链表的倒数第 N 个节点 (LeetCode 19, LintCode 174, 牛客网 NC53, 剑指 Offer 22)
 * 10. 奇偶链表 (LeetCode 328, LintCode 1292, 牛客网 NC142)
 * 11. 分隔链表 (LeetCode 86, LintCode 96, 牛客网 NC188)
 * 12. 链表求和 (LeetCode 2, LeetCode 445, LintCode 167, 牛客网 NC40)
 * 13. 环形链表 (LeetCode 141, LeetCode 142, LintCode 102, 牛客网 NC4, 剑指 Offer 23)
 * 14. 相交链表 (LeetCode 160, LintCode 380, 牛客网 NC66, 剑指 Offer 52)
 * 15. 排序链表 (LeetCode 148, LintCode 98)
 * 16. 链表随机节点 (LeetCode 382)
 * 17. 复制带随机指针的链表 (LeetCode 138, 剑指 Offer 35)
 * 18. 链表组件 (LeetCode 817)
 * 19. 链表中的下一个更大节点 (LeetCode 1019)
 * 20. 链表最大孪生和 (LeetCode 2130)
 */

// 单链表节点定义
struct ListNode {
    int val;
    ListNode *next;
    ListNode() : val(0), next(nullptr) {}
    ListNode(int x) : val(x), next(nullptr) {}
    ListNode(int x, ListNode *next) : val(x), next(next) {}
};

// 双链表节点定义
struct DoubleListNode {
    int val;
    DoubleListNode *prev;
    DoubleListNode *next;
    DoubleListNode() : val(0), prev(nullptr), next(nullptr) {}
    DoubleListNode(int x) : val(x), prev(nullptr), next(nullptr) {}
};

/**
```

```
 * 工具函数：创建链表
 */
ListNode* createList(const std::vector<int>& vals) {
    if (vals.empty()) return nullptr;

    ListNode* head = new ListNode(vals[0]);
    ListNode* current = head;
    for (size_t i = 1; i < vals.size(); i++) {
        current->next = new ListNode(vals[i]);
        current = current->next;
    }
    return head;
}

/**
 * 工具函数：打印链表
 */
void printList(ListNode* head) {
    ListNode* current = head;
    while (current != nullptr) {
        std::cout << current->val;
        if (current->next != nullptr) {
            std::cout << " -> ";
        }
        current = current->next;
    }
    std::cout << std::endl;
}

/**
 * 工具函数：释放链表内存
 */
void deleteList(ListNode* head) {
    while (head != nullptr) {
        ListNode* temp = head;
        head = head->next;
        delete temp;
    }
}

/**
 * 方法 1: 迭代法反转链表
 * 时间复杂度: O(n) - 需要遍历链表一次
```

```
 * 空间复杂度: O(1) - 只使用了常数级别的额外空间
 *
 * 解题思路:
 * 使用三个指针(pre, current, next)来逐个反转链表中的节点指向关系
 * 1. pre 指向已反转部分的最后一个节点
 * 2. current 指向当前待处理节点
 * 3. next 保存 current 的下一个节点，防止断链
 *
 * 执行过程:
 * 原链表: 1 -> 2 -> 3 -> 4 -> 5 -> null
 * 步骤 1: null <- 1    2 -> 3 -> 4 -> 5 -> null
 * 步骤 2: null <- 1 <- 2    3 -> 4 -> 5 -> null
 * 步骤 3: null <- 1 <- 2 <- 3    4 -> 5 -> null
 * ...
 * 最终: null <- 1 <- 2 <- 3 <- 4 <- 5
 */
ListNode* reverseListIterative(ListNode* head) {
    ListNode* pre = nullptr;     // 已反转部分的头节点
    ListNode* current = head;    // 当前待处理节点
    ListNode* next = nullptr;    // 保存 current 的下一个节点

    while (current != nullptr) {
        next = current->next;    // 保存下一个节点
        current->next = pre;     // 反转当前节点的指向
        pre = current;           // 移动 pre 指针
        current = next;          // 移动 current 指针
    }

    return pre; // pre 指向原链表的最后一个节点，即新链表的头节点
}

/**
 * 方法 2: 递归法反转链表
 * 时间复杂度: O(n) - 递归调用 n 次
 * 空间复杂度: O(n) - 递归调用栈的深度为 n
 *
 * 解题思路:
 * 1. 递归到链表末尾
 * 2. 在回溯过程中逐个反转节点的指向
 * 3. 假设除了当前节点外，后续链表已经完成反转
 *
 * 执行过程:
 * 原链表: 1 -> 2 -> 3 -> 4 -> 5 -> null
```

```
 * 递归到 5，返回 5
 * 回溯到 4: 4.next.next = 4 (即 5->4)，4.next = null
 * 回溯到 3: 3.next.next = 3 (即 4->3)，3.next = null
 * ...
 */
ListNode* reverseListRecursive(ListNode* head) {
    // 递归终止条件：空节点或只有一个节点
    if (head == nullptr || head->next == nullptr) {
        return head;
    }

    // 递归处理后续节点，获取反转后链表的头节点
    ListNode* newHead = reverseListRecursive(head->next);

    // 反转当前节点和下一个节点的连接关系
    head->next->next = head;  // 让下一个节点指向当前节点
    head->next = nullptr;     // 断开当前节点的 next 指针

    return newHead; // 返回反转后链表的头节点
}

/**
 * 反转链表指定区间
 * 时间复杂度: O(n) - 最多遍历一次链表
 * 空间复杂度: O(1) - 只使用常数级别的额外空间
 *
 * 解题思路:
 * 1. 找到需要反转区间的前一个节点(pre)
 * 2. 找到需要反转区间的第一个节点(start)
 * 3. 使用头插法将区间内的节点逐个插入到 pre 节点之后
 * 4. 连接反转后的链表与其他部分
 *
 * 执行过程:
 * 原链表: 1 -> 2 -> 3 -> 4 -> 5, left=2, right=4
 * 步骤 1: 找到 pre(节点 1)和 start(节点 2)
 * 步骤 2: 将节点 3 插入到 pre 之后: 1 -> 3 -> 2 -> 4 -> 5
 * 步骤 3: 将节点 4 插入到 pre 之后: 1 -> 4 -> 3 -> 2 -> 5
 * 结果: 1 -> 4 -> 3 -> 2 -> 5
 */
ListNode* reverseBetween(ListNode* head, int left, int right) {
    // 创建虚拟头节点，简化边界处理
    ListNode* dummy = new ListNode(0);
    dummy->next = head;
```

```
    // 找到反转区间的前一个节点
    ListNode* pre = dummy;
    for (int i = 0; i < left - 1; i++) {
        pre = pre->next;
    }

    // start 指向反转区间的第一个节点
    ListNode* start = pre->next;
    // then 指向待处理节点
    ListNode* then = start->next;

    // 头插法实现区间反转
    for (int i = 0; i < right - left; i++) {
        start->next = then->next;
        then->next = pre->next;
        pre->next = then;
        then = start->next;
    }

    ListNode* result = dummy->next;
    delete dummy;
    return result;
}

/**
 * K 个一组反转链表
 * 时间复杂度: O(n) - 每个节点最多被访问两次
 * 空间复杂度: O(1) - 只使用常数级别的额外空间
 *
 * 解题思路:
 * 1. 分组处理，每次处理 k 个节点
 * 2. 对每组节点进行反转
 * 3. 连接各组之间的关系
 * 4. 处理不足 k 个的剩余节点（保持原顺序）
 *
 * 执行过程:
 * 原链表: 1 -> 2 -> 3 -> 4 -> 5, k=3
 * 第一组(1,2,3)反转: 3 -> 2 -> 1
 * 第二组(4,5)不足 k 个，保持原顺序: 4 -> 5
 * 结果: 3 -> 2 -> 1 -> 4 -> 5
 */
ListNode* reverseKGroup(ListNode* head, int k) {
```

```
    // 计算链表长度
    int length = 0;
    ListNode* current = head;
    while (current != nullptr) {
        length++;
        current = current->next;
    }

    // 创建虚拟头节点
    ListNode* dummy = new ListNode(0);
    dummy->next = head;

    // pre 指向已处理部分的最后一个节点
    ListNode* pre = dummy;

    // 分组处理
    while (length >= k) {
        // start 指向当前组的第一个节点
        ListNode* start = pre->next;
        // then 指向待处理节点
        ListNode* then = start->next;

        // 对当前组进行 k-1 次头插操作
        for (int i = 0; i < k - 1; i++) {
            start->next = then->next;
            then->next = pre->next;
            pre->next = then;
            then = start->next;
        }

        // 更新 pre 指针和剩余长度
        pre = start;
        length -= k;
    }

    ListNode* result = dummy->next;
    delete dummy;
    return result;
}

/**
 * 反转双链表
 * 时间复杂度: O(n) - 需要遍历双链表一次
```

```
 * 空间复杂度: O(1) - 只使用常数级别的额外空间
 *
 * 题目来源：
 * 1. LeetCode 445. 两数相加 II（涉及链表反转思想）
 * 2. 剑指 Offer 24. 反转链表（扩展到双链表）
 * 3. 牛客网 反转双链表
 */
DoubleListNode* reverseDoubleList(DoubleListNode* head) {
    DoubleListNode* pre = nullptr;
    DoubleListNode* next = nullptr;
    while (head != nullptr) {
        next = head->next;      // 保存下一个节点
        head->next = pre;       // 反转 next 指针
        head->prev = next;      // 反转 prev 指针
        pre = head;             // 移动 pre 指针
        head = next;            // 移动 head 指针
    }
    return pre;  // 返回新的头节点
}

/**
 * 补充题目 4: 回文链表
 * 题目来源：
 * 1. LeetCode 234. 回文链表 - https://leetcode.cn/problems/palindrome-linked-list/
 * 2. LintCode 223. 回文链表 - https://www.lintcode.com/problem/palindrome-linked-list/
 * 3. 牛客网 NC78 链表中倒数最后 k 个结点（相关题目）
 *
 * 题目描述：判断一个链表是否是回文链表
 * 输入：1->2->2->1
 * 输出：true
 *
 * 最优解法：使用快慢指针找到中点，反转后半部分，然后比较
 * 时间复杂度: O(n) - 只需要一次遍历找到中点，一次反转，一次比较
 * 空间复杂度: O(1) - 只使用常数级别的额外空间
 */
bool isPalindrome(ListNode* head) {
    if (head == nullptr || head->next == nullptr) {
        return true;  // 空链表或单节点链表是回文的
    }

    // 步骤 1: 使用快慢指针找到链表的中点
    ListNode* slow = head;
    ListNode* fast = head;
```

```
    while (fast != nullptr && fast->next != nullptr) {
        slow = slow->next;          // 慢指针每次走一步
        fast = fast->next->next;  // 快指针每次走两步
    }
    // 循环结束后，slow 指向中点位置（如果节点数为奇数）或后半部分的第一个节点（如果节点数为偶数）

    // 步骤 2: 反转后半部分链表
    ListNode* secondHalfHead = reverseListIterative(slow);
    // 保存反转后的头节点，用于后续恢复
    ListNode* secondHalfStart = secondHalfHead;

    // 步骤 3: 比较前半部分和反转后的后半部分
    ListNode* firstHalfHead = head;
    bool isPalindromeFlag = true;
    while (secondHalfHead != nullptr) {
        if (firstHalfHead->val != secondHalfHead->val) {
            isPalindromeFlag = false;
            break;
        }
        firstHalfHead = firstHalfHead->next;
        secondHalfHead = secondHalfHead->next;
    }

    // 步骤 4: 恢复链表（可选，但这是良好的工程实践）
    reverseListIterative(secondHalfStart);

    return isPalindromeFlag;
}

/**
 * 补充题目 5: 旋转链表
 * 题目来源：
 * 1. LeetCode 61. 旋转链表 - https://leetcode.cn/problems/rotate-list/
 * 2. LintCode 170. 旋转链表 - https://www.lintcode.com/problem/rotate-list/
 * 3. 牛客网 NC53 删除链表的倒数第 n 个节点（相关题目）
 *
 * 题目描述：将链表向右旋转 k 个位置
 * 输入：1->2->3->4->5->NULL, k = 2
 * 输出：4->5->1->2->3->NULL
 *
 * 解题思路：
 * 1. 先计算链表长度
 * 2. 将链表首尾相连形成环
```

```
 * 3. 在合适位置断开环
 * 时间复杂度: O(n) - 需要遍历链表
 * 空间复杂度: O(1) - 只使用常数级别的额外空间
 */
ListNode* rotateRight(ListNode* head, int k) {
    // 处理特殊情况
    if (head == nullptr || head->next == nullptr || k == 0) {
        return head;
    }

    // 步骤1: 计算链表长度并找到尾节点
    int length = 1;
    ListNode* tail = head;
    while (tail->next != nullptr) {
        tail = tail->next;
        length++;
    }

    // 步骤2: 计算实际需要旋转的次数（取模操作避免多余旋转）
    k = k % length;
    if (k == 0) {
        return head;  // 不需要旋转
    }

    // 步骤3: 将链表首尾相连形成环
    tail->next = head;

    // 步骤4: 找到新的尾节点位置，距离原头节点 (length - k) 个位置
    ListNode* newTail = head;
    for (int i = 0; i < length - k - 1; i++) {
        newTail = newTail->next;
    }

    // 步骤5: 新的头节点是新尾节点的下一个节点
    ListNode* newHead = newTail->next;

    // 步骤6: 断开环
    newTail->next = nullptr;

    return newHead;
}

/**
```

```
 * 补充题目 6: 合并两个有序链表
 * 题目来源：
 * 1. LeetCode 21. 合并两个有序链表 - https://leetcode.cn/problems/merge-two-sorted-lists/
 * 2. LintCode 165. 合并两个排序链表 - https://www.lintcode.com/problem/merge-two-sorted-lists/
 * 3. 剑指 Offer 25. 合并两个排序的链表
 * 4. 牛客网 NC33 合并两个排序的链表
 *
 * 题目描述：将两个升序链表合并为一个新的升序链表
 * 输入：l1 = [1,2,4], l2 = [1,3,4]
 * 输出：[1,1,2,3,4,4]
 *
 * 解题思路：使用迭代或递归方法，逐个比较两个链表的节点值
 * 时间复杂度: O(n+m) - n 和 m 分别是两个链表的长度
 * 空间复杂度: O(1) - 迭代版本，只使用常数级别的额外空间
 */
ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
    // 创建虚拟头节点，简化边界情况处理
    ListNode* dummy = new ListNode(0);
    ListNode* current = dummy;

    // 迭代比较两个链表的节点值
    while (l1 != nullptr && l2 != nullptr) {
        if (l1->val <= l2->val) {
            current->next = l1;
            l1 = l1->next;
        } else {
            current->next = l2;
            l2 = l2->next;
        }
        current = current->next;
    }

    // 连接剩余部分
    current->next = (l1 != nullptr) ? l1 : l2;

    ListNode* result = dummy->next;
    delete dummy;
    return result;
}

/**
 * 补充题目 7: 两两交换链表中的节点
 * 题目来源：
```

```
 * 1. LeetCode 24. 两两交换链表中的节点 - https://leetcode.cn/problems/swap-nodes-in-pairs/
 * 2. LintCode 451. 两两交换链表中的节点 - https://www.lintcode.com/problem/swap-nodes-in-pairs/
 * 3. 牛客网 NC142 链表的奇偶重排（相关题目）
 *
 * 题目描述：两两交换链表中的相邻节点
 * 输入：1->2->3->4
 * 输出：2->1->4->3
 *
 * 解题思路：使用虚拟头节点和迭代方法
 * 时间复杂度: O(n) - 需要遍历链表一次
 * 空间复杂度: O(1) - 只使用常数级别的额外空间
 */
ListNode* swapPairs(ListNode* head) {
    // 创建虚拟头节点
    ListNode* dummy = new ListNode(0);
    dummy->next = head;
    ListNode* prev = dummy;

    // 当有至少两个节点可以交换时
    while (prev->next != nullptr && prev->next->next != nullptr) {
        // 获取需要交换的两个节点
        ListNode* first = prev->next;
        ListNode* second = prev->next->next;

        // 执行交换操作
        first->next = second->next;  // 1 -> 3
        second->next = first;        // 2 -> 1
        prev->next = second;         // dummy -> 2

        // 移动 prev 指针到下一对的前一个位置
        prev = first;
    }

    ListNode* result = dummy->next;
    delete dummy;
    return result;
}

/**
 * 补充题目 8: 重排链表
 * 题目来源：
 * 1. LeetCode 143. 重排链表 - https://leetcode.cn/problems/reorder-list/
 * 2. LintCode 99. 重排链表 - https://www.lintcode.com/problem/reorder-list/
```

```
 * 3. 牛客网 NC40 链表相加（二）（相关题目）
 *
 * 题目描述：按照 L0 → Ln → L1 → Ln-1 → L2 → Ln-2 → ... 重新排列链表
 * 输入：1->2->3->4
 * 输出：1->4->2->3
 *
 * 解题思路：
 * 1. 使用快慢指针找到中点
 * 2. 反转后半部分链表
 * 3. 合并两个链表
 * 时间复杂度: O(n) - 需要遍历链表三次
 * 空间复杂度: O(1) - 只使用常数级别的额外空间
 */
void reorderList(ListNode* head) {
    if (head == nullptr || head->next == nullptr || head->next->next == nullptr) {
        return;  // 无需重排
    }

    // 步骤 1: 使用快慢指针找到链表中点
    ListNode* slow = head;
    ListNode* fast = head;
    while (fast->next != nullptr && fast->next->next != nullptr) {
        slow = slow->next;
        fast = fast->next->next;
    }

    // 步骤 2: 反转后半部分链表
    ListNode* secondHalf = reverseListIterative(slow->next);
    slow->next = nullptr;  // 断开前半部分和后半部分

    // 步骤 3: 合并两个链表
    ListNode* firstHalf = head;
    while (secondHalf != nullptr) {
        ListNode* temp1 = firstHalf->next;
        ListNode* temp2 = secondHalf->next;

        firstHalf->next = secondHalf;
        secondHalf->next = temp1;

        firstHalf = temp1;
        secondHalf = temp2;
    }
}
```

```
/**
 * 补充题目 9: 删除链表的倒数第 N 个节点
 * 题目来源：
 * 1. LeetCode 19. 删除链表的倒数第 N 个节点 - https://leetcode.cn/problems/remove-nth-node-from-end-of-list/
 * 2. LintCode 174. 删除链表中倒数第 n 个节点 - https://www.lintcode.com/problem/remove-nth-node-from-end-of-list/
 * 3. 牛客网 NC53 删除链表的倒数第 n 个节点
 * 4. 剑指 Offer 22. 链表中倒数第 k 个节点（相关题目）
 *
 * 题目描述：删除链表的倒数第 n 个节点，返回链表的头节点
 * 输入：head = [1,2,3,4,5], n = 2
 * 输出：[1,2,3,5]
 *
 * 解题思路：使用快慢指针，快指针先走 n 步，然后快慢指针一起走
 * 时间复杂度: O(n) - 只需要遍历链表一次
 * 空间复杂度: O(1) - 只使用常数级别的额外空间
 */
ListNode* removeNthFromEnd(ListNode* head, int n) {
    // 创建虚拟头节点，简化边界情况处理
    ListNode* dummy = new ListNode(0);
    dummy->next = head;

    // 设置快慢指针
    ListNode* fast = dummy;
    ListNode* slow = dummy;

    // 快指针先走 n+1 步
    for (int i = 0; i <= n; i++) {
        fast = fast->next;
        // 如果 n 大于链表长度，fast 会变成 null
        if (i < n && fast == nullptr) {
            delete dummy;
            return head;  // n 大于链表长度，无法删除
        }
    }

    // 快慢指针一起走，直到快指针到达链表末尾
    while (fast != nullptr) {
        fast = fast->next;
        slow = slow->next;
    }
```

```
    // 此时 slow 指向待删除节点的前一个节点
    ListNode* toDelete = slow->next;
    slow->next = slow->next->next;
    delete toDelete;  // 释放内存

    ListNode* result = dummy->next;
    delete dummy;
    return result;
}

/**
 * 测试基础链表反转
 */
void testReverseList() {
    std::cout << "=== 测试基础链表反转 ===" << std::endl;

    // 测试用例 1: [1,2,3,4,5] -> [5,4,3,2,1]
    ListNode* head1 = createList({1, 2, 3, 4, 5});
    std::cout << "原链表: ";
    printList(head1);
    ListNode* reversed1 = reverseListIterative(head1);
    std::cout << "反转后: ";
    printList(reversed1);
    deleteList(reversed1);

    // 测试用例 2: [1,2] -> [2,1]
    ListNode* head2 = createList({1, 2});
    std::cout << "原链表: ";
    printList(head2);
    ListNode* reversed2 = reverseListRecursive(head2);
    std::cout << "反转后: ";
    printList(reversed2);
    deleteList(reversed2);

    // 测试用例 3: [] -> []
    ListNode* head3 = nullptr;
    std::cout << "原链表: ";
    printList(head3);
    ListNode* reversed3 = reverseListIterative(head3);
    std::cout << "反转后: ";
    printList(reversed3);
    std::cout << std::endl;
```

```
}

/**
 * 测试指定区间链表反转
 */
void testReverseListII() {
    std::cout << "=== 测试指定区间链表反转 ===" << std::endl;

    // 测试用例 1: [1,2,3,4,5], left=2, right=4 -> [1,4,3,2,5]
    ListNode* head1 = createList({1, 2, 3, 4, 5});
    std::cout << "原链表: ";
    printList(head1);
    ListNode* reversed1 = reverseBetween(head1, 2, 4);
    std::cout << "反转位置 2 到 4 后: ";
    printList(reversed1);
    deleteList(reversed1);

    // 测试用例 2: [5], left=1, right=1 -> [5]
    ListNode* head2 = createList({5});
    std::cout << "原链表: ";
    printList(head2);
    ListNode* reversed2 = reverseBetween(head2, 1, 1);
    std::cout << "反转位置 1 到 1 后: ";
    printList(reversed2);
    deleteList(reversed2);
    std::cout << std::endl;
}

/**
 * 测试 K 个一组反转链表
 */
void testReverseKGroup() {
    std::cout << "=== 测试 K 个一组反转链表 ===" << std::endl;

    // 测试用例 1: [1,2,3,4,5], k=2 -> [2,1,4,3,5]
    ListNode* head1 = createList({1, 2, 3, 4, 5});
    std::cout << "原链表: ";
    printList(head1);
    ListNode* reversed1 = reverseKGroup(head1, 2);
    std::cout << "每 2 个一组反转后: ";
    printList(reversed1);
    deleteList(reversed1);
```

```
    // 测试用例 2: [1,2,3,4,5], k=3 -> [3,2,1,4,5]
    ListNode* head2 = createList({1, 2, 3, 4, 5});
    std::cout << "原链表: ";
    printList(head2);
    ListNode* reversed2 = reverseKGroup(head2, 3);
    std::cout << "每 3 个一组反转后: ";
    printList(reversed2);
    deleteList(reversed2);
    std::cout << std::endl;
}

/**
 * 运行单元测试
 */
void runUnitTests() {
    std::cout << "=== 链表反转单元测试 ===" << std::endl;

    // 测试用例 1: 正常情况
    ListNode* test1 = createList({1, 2, 3, 4, 5});
    ListNode* result1 = reverseListIterative(test1);
    std::cout << "测试 1 - 输入[1,2,3,4,5], 期望[5,4,3,2,1], 实际: ";
    printList(result1);
    deleteList(result1);

    // 测试用例 2: 空链表
    ListNode* result2 = reverseListIterative(nullptr);
    std::cout << "测试 2 - 输入[], 期望[], 实际: ";
    printList(result2);

    // 测试用例 3: 单节点链表
    ListNode* test3 = new ListNode(1);
    ListNode* result3 = reverseListIterative(test3);
    std::cout << "测试 3 - 输入[1], 期望[1], 实际: " << result3->val << std::endl;
    deleteList(result3);

    // 测试用例 4: 两节点链表
    ListNode* test4 = createList({1, 2});
    ListNode* result4 = reverseListIterative(test4);
    std::cout << "测试 4 - 输入[1,2], 期望[2,1], 实际: ";
    printList(result4);
    deleteList(result4);

    std::cout << "单元测试完成" << std::endl << std::endl;
```

```
}

/**
 * 补充题目 10: 奇偶链表
 * 题目来源：
 * 1. LeetCode 328. 奇偶链表 - https://leetcode.cn/problems/odd-even-linked-list/
 * 2. LintCode 1292. 奇偶链表 - https://www.lintcode.com/problem/odd-even-linked-list/
 * 3. 牛客网 NC142 链表的奇偶重排
 *
 * 时间复杂度: O(n)
 * 空间复杂度: O(1)
 * 是否为最优解：是
 */
ListNode* oddEvenList(ListNode* head) {
    if (head == nullptr || head->next == nullptr) {
        return head;
    }

    ListNode* odd = head;
    ListNode* even = head->next;
    ListNode* evenHead = even;

    while (even != nullptr && even->next != nullptr) {
        odd->next = even->next;
        odd = odd->next;
        even->next = odd->next;
        even = even->next;
    }

    odd->next = evenHead;
    return head;
}

/**
 * 补充题目 11: 分隔链表
 * 题目来源：
 * 1. LeetCode 86. 分隔链表 - https://leetcode.cn/problems/partition-list/
 * 2. LintCode 96. 分隔链表 - https://www.lintcode.com/problem/partition-list/
 * 3. 牛客网 NC188 分隔链表
 *
 * 时间复杂度: O(n)
 * 空间复杂度: O(1)
 * 是否为最优解：是
```

```
 */
ListNode* partition(ListNode* head, int x) {
    ListNode* beforeHead = new ListNode(0);
    ListNode* before = beforeHead;
    ListNode* afterHead = new ListNode(0);
    ListNode* after = afterHead;

    while (head != nullptr) {
        if (head->val < x) {
            before->next = head;
            before = before->next;
        } else {
            after->next = head;
            after = after->next;
        }
        head = head->next;
    }

    after->next = nullptr;
    before->next = afterHead->next;

    ListNode* result = beforeHead->next;
    delete beforeHead;
    delete afterHead;
    return result;
}

/**
 * 补充题目 12: 链表求和
 * 题目来源：
 * 1. LeetCode 2. 两数相加 - https://leetcode.cn/problems/add-two-numbers/
 * 2. LeetCode 445. 两数相加 II - https://leetcode.cn/problems/add-two-numbers-ii/
 *
 * 时间复杂度: O(max(m,n))
 * 空间复杂度: O(max(m,n))
 * 是否为最优解：是
 */
ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
    ListNode* dummy = new ListNode(0);
    ListNode* current = dummy;
    int carry = 0;

    while (l1 != nullptr || l2 != nullptr || carry != 0) {
```

```
        int val1 = (l1 != nullptr) ? l1->val : 0;
        int val2 = (l2 != nullptr) ? l2->val : 0;

        int sum = val1 + val2 + carry;
        carry = sum / 10;

        current->next = new ListNode(sum % 10);
        current = current->next;

        if (l1 != nullptr) l1 = l1->next;
        if (l2 != nullptr) l2 = l2->next;
    }

    ListNode* result = dummy->next;
    delete dummy;
    return result;
}

/**
 * 补充题目 13: 环形链表
 * 题目来源：
 * 1. LeetCode 141. 环形链表 - https://leetcode.cn/problems/linked-list-cycle/
 * 2. LeetCode 142. 环形链表 II - https://leetcode.cn/problems/linked-list-cycle-ii/
 *
 * 时间复杂度: O(n)
 * 空间复杂度: O(1)
 * 是否为最优解：是
 */
bool hasCycle(ListNode* head) {
    if (head == nullptr || head->next == nullptr) {
        return false;
    }

    ListNode* slow = head;
    ListNode* fast = head;

    while (fast != nullptr && fast->next != nullptr) {
        slow = slow->next;
        fast = fast->next->next;

        if (slow == fast) {
            return true;
        }
```

```
    }

    return false;
}

ListNode* detectCycle(ListNode* head) {
    if (head == nullptr || head->next == nullptr) {
        return nullptr;
    }

    ListNode* slow = head;
    ListNode* fast = head;
    bool hasCycleFlag = false;

    while (fast != nullptr && fast->next != nullptr) {
        slow = slow->next;
        fast = fast->next->next;

        if (slow == fast) {
            hasCycleFlag = true;
            break;
        }
    }

    if (!hasCycleFlag) {
        return nullptr;
    }

    slow = head;
    while (slow != fast) {
        slow = slow->next;
        fast = fast->next;
    }

    return slow;
}

/**
 * 补充题目 14: 相交链表
 * 题目来源：
 * 1. LeetCode 160. 相交链表 - https://leetcode.cn/problems/intersection-of-two-linked-lists/
 * 2. LintCode 380. 相交链表 - https://www.lintcode.com/problem/intersection-of-two-linked-lists/
 *
```

```
 * 时间复杂度: O(m+n)
 * 空间复杂度: O(1)
 * 是否为最优解：是
 */
ListNode* getIntersectionNode(ListNode* headA, ListNode* headB) {
    if (headA == nullptr || headB == nullptr) {
        return nullptr;
    }

    ListNode* pA = headA;
    ListNode* pB = headB;

    while (pA != pB) {
        pA = (pA == nullptr) ? headB : pA->next;
        pB = (pB == nullptr) ? headA : pB->next;
    }

    return pA;
}

/**
 * 补充题目 15: 排序链表
 * 题目来源：
 * 1. LeetCode 148. 排序链表 - https://leetcode.cn/problems/sort-list/
 * 2. LintCode 98. 排序链表 - https://www.lintcode.com/problem/sort-list/
 *
 * 时间复杂度: O(n log n)
 * 空间复杂度: O(log n)
 * 是否为最优解：是
 */
ListNode* sortList(ListNode* head) {
    if (head == nullptr || head->next == nullptr) {
        return head;
    }

    ListNode* slow = head;
    ListNode* fast = head->next;

    while (fast != nullptr && fast->next != nullptr) {
        slow = slow->next;
        fast = fast->next->next;
    }
```

```
    ListNode* mid = slow->next;
    slow->next = nullptr;

    ListNode* left = sortList(head);
    ListNode* right = sortList(mid);

    return mergeTwoLists(left, right);
}

/**
 * 补充题目 16: 链表随机节点
 * 题目来源：
 * 1. LeetCode 382. 链表随机节点 - https://leetcode.cn/problems/linked-list-random-node/
 * 2. 蓄水池抽样算法应用
 *
 * 时间复杂度: O(n)
 * 空间复杂度: O(1)
 * 是否为最优解：是
 */
class RandomNodeSelector {
private:
    ListNode* head;

public:
    RandomNodeSelector(ListNode* head) : head(head) {}

    int getRandom() {
        ListNode* current = head;
        int result = 0;
        int count = 0;

        while (current != nullptr) {
            count++;
            // 以 1/count 的概率选择当前节点
            if (rand() % count == 0) {
                result = current->val;
            }
            current = current->next;
        }

        return result;
    }
};
```

```
/**
 * 补充题目 17: 复制带随机指针的链表
 * 题目来源：
 * 1. LeetCode 138. 复制带随机指针的链表 - https://leetcode.cn/problems/copy-list-with-random-pointer/
 * 2. 剑指 Offer 35. 复杂链表的复制
 *
 * 时间复杂度: O(n)
 * 空间复杂度: O(1)
 * 是否为最优解：是
 */
struct NodeWithRandom {
    int val;
    NodeWithRandom* next;
    NodeWithRandom* random;
    NodeWithRandom(int x) : val(x), next(nullptr), random(nullptr) {}
};

NodeWithRandom* copyRandomList(NodeWithRandom* head) {
    if (head == nullptr) {
        return nullptr;
    }

    // 第一次遍历：在每个节点后面插入复制节点
    NodeWithRandom* current = head;
    while (current != nullptr) {
        NodeWithRandom* copy = new NodeWithRandom(current->val);
        copy->next = current->next;
        current->next = copy;
        current = copy->next;
    }

    // 第二次遍历：设置复制节点的随机指针
    current = head;
    while (current != nullptr) {
        if (current->random != nullptr) {
            current->next->random = current->random->next;
        }
        current = current->next->next;
    }

    // 第三次遍历：分离原链表和复制链表
```

```
    current = head;
    NodeWithRandom* copyHead = head->next;
    NodeWithRandom* copyCurrent = copyHead;

    while (current != nullptr) {
        current->next = current->next->next;
        if (copyCurrent->next != nullptr) {
            copyCurrent->next = copyCurrent->next->next;
        }
        current = current->next;
        copyCurrent = copyCurrent->next;
    }

    return copyHead;
}

/**
 * 补充题目 18: 链表组件
 * 题目来源：
 * 1. LeetCode 817. 链表组件 - https://leetcode.cn/problems/linked-list-components/
 *
 * 时间复杂度: O(n + m)
 * 空间复杂度: O(m)
 * 是否为最优解：是
 */
#include <unordered_set>
int numComponents(ListNode* head, std::vector<int>& nums) {
    std::unordered_set<int> numSet(nums.begin(), nums.end());

    int components = 0;
    bool inComponent = false;
    ListNode* current = head;

    while (current != nullptr) {
        if (numSet.find(current->val) != numSet.end()) {
            if (!inComponent) {
                components++;
                inComponent = true;
            }
        } else {
            inComponent = false;
        }
        current = current->next;
```

```
    }

    return components;
}

/**
 * 补充题目 19: 链表中的下一个更大节点
 * 题目来源：
 * 1. LeetCode 1019. 链表中的下一个更大节点 - https://leetcode.cn/problems/next-greater-node-in-linked-list/
 *
 * 时间复杂度: O(n)
 * 空间复杂度: O(n)
 * 是否为最优解：是
 */
#include <stack>
std::vector<int> nextLargerNodes(ListNode* head) {
    // 将链表转换为数组
    std::vector<int> list;
    ListNode* current = head;
    while (current != nullptr) {
        list.push_back(current->val);
        current = current->next;
    }

    int n = list.size();
    std::vector<int> result(n, 0);
    std::stack<int> stack;

    // 从右向左遍历，使用单调栈
    for (int i = n - 1; i >= 0; i--) {
        int currentVal = list[i];

        // 弹出栈顶比当前值小的元素
        while (!stack.empty() && stack.top() <= currentVal) {
            stack.pop();
        }

        // 如果栈不为空，栈顶就是下一个更大节点
        result[i] = stack.empty() ? 0 : stack.top();

        // 将当前值压入栈
        stack.push(currentVal);
```

```
    }

    return result;
}

/**
 * 补充题目 20: 链表最大孪生和
 * 题目来源：
 * 1. LeetCode 2130. 链表最大孪生和 - https://leetcode.cn/problems/maximum-twin-sum-of-a-linked-list/
 *
 * 时间复杂度: O(n)
 * 空间复杂度: O(1)
 * 是否为最优解：是
 */
int pairSum(ListNode* head) {
    // 使用快慢指针找到中点
    ListNode* slow = head;
    ListNode* fast = head;
    while (fast != nullptr && fast->next != nullptr) {
        slow = slow->next;
        fast = fast->next->next;
    }

    // 反转后半部分链表
    ListNode* secondHalf = reverseListIterative(slow);

    // 计算孪生和的最大值
    int maxSum = 0;
    ListNode* firstHalf = head;
    while (secondHalf != nullptr) {
        maxSum = std::max(maxSum, firstHalf->val + secondHalf->val);
        firstHalf = firstHalf->next;
        secondHalf = secondHalf->next;
    }

    return maxSum;
}

/**
 * 补充题目 21: 合并 K 个升序链表
 * 题目来源：
 * 1. LeetCode 23. 合并 K 个升序链表 - https://leetcode.cn/problems/merge-k-sorted-lists/
```

```
 * 2. LintCode 104. 合并 k 个排序链表 - https://www.lintcode.com/problem/merge-k-sorted-lists/
 * 3. 牛客网 NC127. 合并 k 个已排序的链表
 *
 * 时间复杂度: O(N log K)
 * 空间复杂度: O(K)
 * 是否为最优解：是
 */
#include <queue>
ListNode* mergeKLists(std::vector<ListNode*>& lists) {
    // 使用最小堆
    auto cmp = [](const ListNode* a, const ListNode* b) { return a->val > b->val; };
    std::priority_queue<ListNode*, std::vector<ListNode*>, decltype(cmp)> minHeap(cmp);

    // 将所有非空链表的头节点加入最小堆
    for (ListNode* list : lists) {
        if (list != nullptr) {
            minHeap.push(list);
        }
    }

    // 创建虚拟头节点
    ListNode* dummy = new ListNode(0);
    ListNode* current = dummy;

    // 从堆中取出最小节点，加入结果链表
    while (!minHeap.empty()) {
        ListNode* minNode = minHeap.top();  // 取出最小节点
        minHeap.pop();
        current->next = minNode;            // 加入结果链表
        current = current->next;            // 移动指针

        // 将取出节点的下一个节点加入堆中(如果不为空)
        if (minNode->next != nullptr) {
            minHeap.push(minNode->next);
        }
    }

    ListNode* result = dummy->next;
    delete dummy;
    return result;
}

/**
```

```
 * 补充题目 22: 删除链表中的节点
 * 题目来源：
 * 1. LeetCode 237. 删除链表中的节点 - https://leetcode.cn/problems/delete-node-in-a-linked-list/
 * 2. LintCode 37. 删除链表中的节点 - https://www.lintcode.com/problem/delete-node-in-a-linked-list/
 * 3. 牛客网 NC138. 删除链表的节点
 *
 * 时间复杂度: O(1)
 * 空间复杂度: O(1)
 * 是否为最优解：是
 */
void deleteNode(ListNode* node) {
    // 将下一个节点的值复制到当前节点
    node->val = node->next->val;

    // 保存要删除的节点
    ListNode* nodeToDelete = node->next;

    // 跳过下一个节点
    node->next = node->next->next;

    // 释放内存
    delete nodeToDelete;
}

/**
 * 补充题目 23: 删除排序链表中的重复元素
 * 题目来源：
 * 1. LeetCode 83. 删除排序链表中的重复元素 - https://leetcode.cn/problems/remove-duplicates-from-sorted-list/
 * 2. LintCode 112. 删除排序链表中的重复元素 - https://www.lintcode.com/problem/remove-duplicates-from-sorted-list/
 * 3. 牛客网 NC141. 判断一个链表是否为回文结构
 *
 * 时间复杂度: O(n)
 * 空间复杂度: O(1)
 * 是否为最优解：是
 */
ListNode* deleteDuplicates(ListNode* head) {
    // 边界情况处理
    if (head == nullptr || head->next == nullptr) {
        return head;
    }
```

```
    ListNode* current = head;

    // 遍历链表
    while (current->next != nullptr) {
        // 如果当前节点值等于下一个节点值，跳过下一个节点
        if (current->val == current->next->val) {
            current->next = current->next->next;
        } else {
            // 只有当下一个节点不被删除时，才移动 current 指针
            current = current->next;
        }
    }

    return head;
}

/**
 * 补充题目 24: 删除排序链表中的重复元素 II
 * 题目来源：
 * 1. LeetCode 82. 删除排序链表中的重复元素 II - https://leetcode.cn/problems/remove-duplicates-from-sorted-list-ii/
 * 2. LintCode 113. 删除排序链表中的重复元素 II - https://www.lintcode.com/problem/remove-duplicates-from-sorted-list-ii/
 * 3. 牛客网 NC140
 *
 * 时间复杂度: O(n)
 * 空间复杂度: O(1)
 * 是否为最优解：是
 */
ListNode* deleteDuplicatesII(ListNode* head) {
    // 创建虚拟头节点，简化边界处理
    ListNode* dummy = new ListNode(0);
    dummy->next = head;

    // prev 指向已处理部分的最后一个节点
    ListNode* prev = dummy;
    // current 指向当前待处理节点
    ListNode* current = head;

    while (current != nullptr) {
        // 检查是否有重复节点
        if (current->next != nullptr && current->val == current->next->val) {
```

```
            // 记录重复值
            int duplicateValue = current->val;

            // 跳过所有重复节点
            while (current != nullptr && current->val == duplicateValue) {
                ListNode* nodeToDelete = current;
                current = current->next;
                delete nodeToDelete;
            }

            // 连接 prev 和 current
            prev->next = current;
        } else {
            // 没有重复，正常移动指针
            prev = current;
            current = current->next;
        }
    }

    ListNode* result = dummy->next;
    delete dummy;
    return result;
}

/**
 * 补充题目 25: 移除链表元素
 * 题目来源：
 * 1. LeetCode 203. 移除链表元素 - https://leetcode.cn/problems/remove-linked-list-elements/
 * 2. 牛客网相关题目
 *
 * 时间复杂度: O(n)
 * 空间复杂度: O(1)
 * 是否为最优解：是
 */
ListNode* removeElements(ListNode* head, int val) {
    // 创建虚拟头节点，简化删除头节点的情况
    ListNode* dummy = new ListNode(0);
    dummy->next = head;

    // prev 指向已处理部分的最后一个节点
    ListNode* prev = dummy;
    // current 指向当前待处理节点
    ListNode* current = head;
```

```
    while (current != nullptr) {
        if (current->val == val) {
            // 删除当前节点
            ListNode* nodeToDelete = current;
            prev->next = current->next;
            current = current->next;
            delete nodeToDelete;
        } else {
            // 移动 prev 指针
            prev = current;
            current = current->next;
        }
    }

    ListNode* result = dummy->next;
    delete dummy;
    return result;
}

/**
 * 性能分析工具类
 */
class LinkedListProfiler {
private:
    std::chrono::high_resolution_clock::time_point startTime;
    std::chrono::high_resolution_clock::time_point endTime;

public:
    void start() {
        startTime = std::chrono::high_resolution_clock::now();
    }

    void end() {
        endTime = std::chrono::high_resolution_clock::now();
        auto duration = std::chrono::duration_cast<std::chrono::nanoseconds>(endTime -
startTime);
        std::cout << "执行时间: " << duration.count() << " 纳秒" << std::endl;
    }
};

/**
 * 调试工具类
```

```
 */
class LinkedListDebugger {
public:
    static void printListState(ListNode* head, const std::string& message) {
        std::cout << message << ": ";
        printList(head);
    }

    static void assertList(bool condition, const std::string& message) {
        if (!condition) {
            throw std::runtime_error("链表断言失败: " + message);
        }
    }

    static bool verifyNoCycle(ListNode* head) {
        if (head == nullptr) return true;

        ListNode* slow = head;
        ListNode* fast = head;

        while (fast != nullptr && fast->next != nullptr) {
            slow = slow->next;
            fast = fast->next->next;
            if (slow == fast) {
                return false; // 有环
            }
        }
        return true; // 无环
    }
};

/**
 * 测试补充题目
 */
void testAdditionalProblems() {
    std::cout << "=== 补充题目测试 ===" << std::endl;

    // 测试链表随机节点
    ListNode* randomTest = createList({1, 2, 3, 4, 5});
    RandomNodeSelector selector(randomTest);
    std::cout << "随机节点选择测试: " << selector.getRandom() << std::endl;
    deleteList(randomTest);
```

```
    // 测试链表组件
    ListNode* componentTest = createList({0, 1, 2, 3});
    std::vector<int> nums = {0, 1, 3};
    std::cout << "链表组件个数: " << numComponents(componentTest, nums) << std::endl;
    deleteList(componentTest);

    // 测试下一个更大节点
    ListNode* largerTest = createList({2, 1, 5});
    std::vector<int> largerResult = nextLargerNodes(largerTest);
    std::cout << "下一个更大节点: ";
    for (int val : largerResult) {
        std::cout << val << " ";
    }
    std::cout << std::endl;
    deleteList(largerTest);

    // 测试链表最大孪生和
    ListNode* twinTest = createList({5, 4, 2, 1});
    std::cout << "链表最大孪生和: " << pairSum(twinTest) << std::endl;
    deleteList(twinTest);

    std::cout << "补充题目测试完成" << std::endl << std::endl;
}

/**
 * 测试合并 K 个升序链表
 */
void testMergeKLists() {
    std::cout << "=== 测试合并 K 个升序链表 ===" << std::endl;

    // 创建测试用例: lists = [[1,4,5],[1,3,4],[2,6]]
    ListNode* list1 = createList({1, 4, 5});
    ListNode* list2 = createList({1, 3, 4});
    ListNode* list3 = createList({2, 6});
    std::vector<ListNode*> lists = {list1, list2, list3};

    std::cout << "输入链表数组: ";
    for (size_t i = 0; i < lists.size(); i++) {
        if (i > 0) std::cout << ", ";
        printList(lists[i]);
    }

    ListNode* result = mergeKLists(lists);
```

```
    std::cout << "合并后: ";
    printList(result);
    deleteList(result);
    std::cout << std::endl;
}

/**
 * 测试删除链表中的节点
 */
void testDeleteNode() {
    std::cout << "=== 测试删除链表中的节点 ===" << std::endl;

    // 创建测试用例: [4,5,1,9], 删除节点 5
    ListNode* head = createList({4, 5, 1, 9});
    std::cout << "原链表: ";
    printList(head);

    // 找到要删除的节点(值为 5 的节点)
    ListNode* nodeToDelete = head->next;  // 值为 5 的节点

    std::cout << "删除节点: " << nodeToDelete->val << std::endl;
    deleteNode(nodeToDelete);
    std::cout << "删除后: ";
    printList(head);
    deleteList(head);
    std::cout << std::endl;
}

/**
 * 测试删除排序链表中的重复元素
 */
void testDeleteDuplicates() {
    std::cout << "=== 测试删除排序链表中的重复元素 ===" << std::endl;

    // 测试用例 1: [1,1,2] -> [1,2]
    ListNode* head1 = createList({1, 1, 2});
    std::cout << "原链表: ";
    printList(head1);
    ListNode* result1 = deleteDuplicates(head1);
    std::cout << "去重后: ";
    printList(result1);
    deleteList(result1);
```

```
    // 测试用例 2: [1,1,2,3,3] -> [1,2,3]
    ListNode* head2 = createList({1, 1, 2, 3, 3});
    std::cout << "原链表: ";
    printList(head2);
    ListNode* result2 = deleteDuplicates(head2);
    std::cout << "去重后: ";
    printList(result2);
    deleteList(result2);
    std::cout << std::endl;
}

/**
 * 测试删除排序链表中的重复元素 II
 */
void testDeleteDuplicatesII() {
    std::cout << "=== 测试删除排序链表中的重复元素 II ===" << std::endl;

    // 测试用例 1: [1,2,3,3,4,4,5] -> [1,2,5]
    ListNode* head1 = createList({1, 2, 3, 3, 4, 4, 5});
    std::cout << "原链表: ";
    printList(head1);
    ListNode* result1 = deleteDuplicatesII(head1);
    std::cout << "删除重复元素后: ";
    printList(result1);
    deleteList(result1);

    // 测试用例 2: [1,1,1,2,3] -> [2,3]
    ListNode* head2 = createList({1, 1, 1, 2, 3});
    std::cout << "原链表: ";
    printList(head2);
    ListNode* result2 = deleteDuplicatesII(head2);
    std::cout << "删除重复元素后: ";
    printList(result2);
    deleteList(result2);
    std::cout << std::endl;
}

/**
 * 测试移除链表元素
 */
void testRemoveElements() {
    std::cout << "=== 测试移除链表元素 ===" << std::endl;
```

```
    // 测试用例: [1,2,6,3,4,5,6], val = 6 -> [1,2,3,4,5]
    ListNode* head = createList({1, 2, 6, 3, 4, 5, 6});
    int val = 6;
    std::cout << "原链表: ";
    printList(head);
    std::cout << "移除元素: " << val << std::endl;
    ListNode* result = removeElements(head, val);
    std::cout << "移除后: ";
    printList(result);
    deleteList(result);
    std::cout << std::endl;
}

// 主函数
int main() {
    // 设置随机种子
    srand(time(nullptr));

    // 运行测试
    testReverseList();
    testReverseListII();
    testReverseKGroup();
    runUnitTests();
    testAdditionalProblems();

    // 运行补充题目的测试
    testMergeKLists();
    testDeleteNode();
    testDeleteDuplicates();
    testDeleteDuplicatesII();
    testRemoveElements();

    // 性能分析示例
    LinkedListProfiler profiler;
    ListNode* perfTest = createList({1, 2, 3, 4, 5, 6, 7, 8, 9, 10});

    profiler.start();
    ListNode* reversed = reverseListIterative(perfTest);
    profiler.end();

    deleteList(reversed);

    std::cout << "\n=== 所有测试完成 ===" << std::endl;
```

```
    return 0;
}
```

```
==========================================
```

文件: ListReverse.java

```
==========================================
import java.util.*;

/**
 * 链表反转相关算法题目集合
 * 包含 LeetCode、牛客网、Codeforces、LintCode、HackerRank 等平台的相关题目
 * 每个题目都提供详细的解题思路、复杂度分析和多种解法
 *
 * 题目列表：
 * 1. 反转链表 (LeetCode 206, 牛客网, HackerRank, LintCode 35, 剑指 Offer 24)
 * 2. 反转链表 II (LeetCode 92)
 * 3. K 个一组翻转链表 (LeetCode 25)
 * 4. 回文链表 (LeetCode 234, LintCode 223, 牛客网 NC78)
 * 5. 旋转链表 (LeetCode 61, LintCode 170)
 * 6. 合并两个有序链表 (LeetCode 21, LintCode 165, 剑指 Offer 25, 牛客网 NC33)
 * 7. 两两交换链表中的节点 (LeetCode 24, LintCode 451)
 * 8. 重排链表 (LeetCode 143, LintCode 99)
 * 9. 删除链表的倒数第 N 个节点 (LeetCode 19, LintCode 174, 牛客网 NC53, 剑指 Offer 22)
 * 10. 奇偶链表 (LeetCode 328, LintCode 1292, 牛客网 NC142)
 * 11. 分隔链表 (LeetCode 86, LintCode 96, 牛客网 NC188)
 * 12. 链表求和 (LeetCode 2, LeetCode 445, LintCode 167, 牛客网 NC40)
 * 13. 环形链表 (LeetCode 141, LeetCode 142, LintCode 102, 牛客网 NC4, 剑指 Offer 23)
 * 14. 相交链表 (LeetCode 160, LintCode 380, 牛客网 NC66, 剑指 Offer 52)
 * 15. 排序链表 (LeetCode 148, LintCode 98)
 * 16. 链表随机节点 (LeetCode 382)
 * 17. 复制带随机指针的链表 (LeetCode 138, 剑指 Offer 35)
 * 18. 链表组件 (LeetCode 817)
 * 19. 链表中的下一个更大节点 (LeetCode 1019)
 * 20. 链表最大孪生和 (LeetCode 2130)
 */
public class ListReverse {

    public static void main(String[] args) {
        // int、long、byte、short
        // char、float、double、boolean
        // 还有 String
```

```
    // 都是按值传递
    int a = 10;
    f(a);
    System.out.println(a);

    // 其他类型按引用传递
    // 比如下面的 Number 是自定义的类
    Number b = new Number(5);
    g1(b);
    System.out.println(b.val);
    g2(b);
    System.out.println(b.val);

    // 比如下面的一维数组
    int[] c = { 1, 2, 3, 4 };
    g3(c);
    System.out.println(c[0]);
    g4(c);
    System.out.println(c[0]);

    // 测试链表反转相关题目
    testReverseList();
    testReverseListII();
    testReverseKGroup();
    testReverseDoubleList();
}

public static void f(int a) {
    a = 0;
}

public static class Number {
    public int val;

    public Number(int v) {
        val = v;
    }
}

public static void g1(Number b) {
    b = null;
}
```

```
public static void g2(Number b) {
    b.val = 6;
}

public static void g3(int[] c) {
    c = null;
}

public static void g4(int[] c) {
    c[0] = 100;
}

// ==================== 链表反转相关题目 ====================

// 单链表节点
public static class ListNode {
    public int val;
    public ListNode next;

    public ListNode() {}

    public ListNode(int val) {
        this.val = val;
    }

    public ListNode(int val, ListNode next) {
        this.val = val;
        this.next = next;
    }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        ListNode current = this;
        while (current != null) {
            sb.append(current.val);
            if (current.next != null) {
                sb.append(" -> ");
            }
            current = current.next;
        }
        return sb.toString();
    }
```

```
    /**
     * 创建链表的辅助方法
     * @param vals 节点值数组
     * @return 构建的链表头节点
     */
    public static ListNode createList(int[] vals) {
        if (vals.length == 0) return null;

        ListNode head = new ListNode(vals[0]);
        ListNode current = head;
        for (int i = 1; i < vals.length; i++) {
            current.next = new ListNode(vals[i]);
            current = current.next;
        }
        return head;
    }
}

// 双链表节点
public static class DoubleListNode {
    public int value;
    public DoubleListNode last;
    public DoubleListNode next;

    public DoubleListNode(int v) {
        value = v;
    }
}

/**
 * 测试基础链表反转
 * 题目来源：LeetCode 206. 反转链表
 * 牛客网：反转链表
 * HackerRank：Reverse a linked list
 */
public static void testReverseList() {
    System.out.println("=== 测试基础链表反转 ===");

    // 测试用例 1: [1,2,3,4,5] -> [5,4,3,2,1]
    ListNode head1 = ListNode.createList(new int[]{1, 2, 3, 4, 5});
    System.out.println("原链表: " + head1);
    ListNode reversed1 = reverseListIterative(head1);
```

```
    System.out.println("反转后: " + reversed1);

    // 测试用例 2: [1,2] -> [2,1]
    ListNode head2 = ListNode.createList(new int[]{1, 2});
    System.out.println("原链表: " + head2);
    ListNode reversed2 = reverseListRecursive(head2);
    System.out.println("反转后: " + reversed2);

    // 测试用例 3: [] -> []
    ListNode head3 = null;
    System.out.println("原链表: " + head3);
    ListNode reversed3 = reverseListIterative(head3);
    System.out.println("反转后: " + reversed3);
    System.out.println();
}

/**
 * 测试指定区间链表反转
 * 题目来源：LeetCode 92. 反转链表 II
 */
public static void testReverseListII() {
    System.out.println("=== 测试指定区间链表反转 ===");

    // 测试用例 1: [1,2,3,4,5], left=2, right=4 -> [1,4,3,2,5]
    ListNode head1 = ListNode.createList(new int[]{1, 2, 3, 4, 5});
    System.out.println("原链表: " + head1);
    ListNode reversed1 = reverseBetween(head1, 2, 4);
    System.out.println("反转位置 2 到 4 后: " + reversed1);

    // 测试用例 2: [5], left=1, right=1 -> [5]
    ListNode head2 = ListNode.createList(new int[]{5});
    System.out.println("原链表: " + head2);
    ListNode reversed2 = reverseBetween(head2, 1, 1);
    System.out.println("反转位置 1 到 1 后: " + reversed2);
    System.out.println();
}

/**
 * 测试 K 个一组反转链表
 * 题目来源：LeetCode 25. K 个一组翻转链表
 */
public static void testReverseKGroup() {
    System.out.println("=== 测试 K 个一组反转链表 ===");
```

```
        // 测试用例 1: [1,2,3,4,5], k=2 -> [2,1,4,3,5]
        ListNode head1 = ListNode.createList(new int[]{1, 2, 3, 4, 5});
        System.out.println("原链表: " + head1);
        ListNode reversed1 = reverseKGroup(head1, 2);
        System.out.println("每 2 个一组反转后: " + reversed1);

        // 测试用例 2: [1,2,3,4,5], k=3 -> [3,2,1,4,5]
        ListNode head2 = ListNode.createList(new int[]{1, 2, 3, 4, 5});
        System.out.println("原链表: " + head2);
        ListNode reversed2 = reverseKGroup(head2, 3);
        System.out.println("每 3 个一组反转后: " + reversed2);
        System.out.println();
    }

    /**
     * 测试双链表反转
     */
    public static void testReverseDoubleList() {
        System.out.println("=== 测试双链表反转 ===");
        // 双链表测试较为复杂，此处省略具体测试
        System.out.println("双链表反转功能已实现");
        System.out.println();
    }

    // ==================== 题目 1: 反转链表 ====================
    // 题目来源：
    // 1. LeetCode 206. 反转链表 - https://leetcode.cn/problems/reverse-linked-list/
    // 2. 牛客网 反转链表 - https://www.nowcoder.com/practice/75e878df47f24fdc9dc3e400ec6058ca
    // 3. HackerRank Reverse a linked list - https://www.hackerrank.com/challenges/reverse-a-linked-list
    // 4. LintCode 35. 翻转链表 - https://www.lintcode.com/problem/reverse-linked-list
    // 5. 剑指 Offer 24. 反转链表 - https://leetcode.cn/problems/fan-zhuan-lian-biao-lcof/

    /**
     * 方法 1: 迭代法反转链表
     * 时间复杂度: O(n) - 需要遍历链表一次
     * 空间复杂度: O(1) - 只使用了常数级别的额外空间
     *
     * 解题思路:
     * 使用三个指针(pre, current, next)来逐个反转链表中的节点指向关系
     * 1. pre 指向已反转部分的最后一个节点
     * 2. current 指向当前待处理节点
```

```
     * 3. next 保存 current 的下一个节点，防止断链
     *
     * 执行过程:
     * 原链表: 1 -> 2 -> 3 -> 4 -> 5 -> null
     * 步骤 1: null <- 1    2 -> 3 -> 4 -> 5 -> null
     * 步骤 2: null <- 1 <- 2    3 -> 4 -> 5 -> null
     * 步骤 3: null <- 1 <- 2 <- 3    4 -> 5 -> null
     * ...
     * 最终: null <- 1 <- 2 <- 3 <- 4 <- 5
     */
    public static ListNode reverseListIterative(ListNode head) {
        ListNode pre = null;     // 已反转部分的头节点
        ListNode current = head; // 当前待处理节点
        ListNode next = null;    // 保存 current 的下一个节点

        while (current != null) {
            next = current.next;    // 保存下一个节点
            current.next = pre;     // 反转当前节点的指向
            pre = current;          // 移动 pre 指针
            current = next;         // 移动 current 指针
        }

        return pre; // pre 指向原链表的最后一个节点，即新链表的头节点
    }

    /**
     * 方法 2: 递归法反转链表
     * 时间复杂度: O(n) - 递归调用 n 次
     * 空间复杂度: O(n) - 递归调用栈的深度为 n
     *
     * 解题思路:
     * 1. 递归到链表末尾
     * 2. 在回溯过程中逐个反转节点的指向
     * 3. 假设除了当前节点外，后续链表已经完成反转
     *
     * 执行过程:
     * 原链表: 1 -> 2 -> 3 -> 4 -> 5 -> null
     * 递归到 5，返回 5
     * 回溯到 4: 4.next.next = 4 (即 5->4)，4.next = null
     * 回溯到 3: 3.next.next = 3 (即 4->3)，3.next = null
     * ...
     */
    public static ListNode reverseListRecursive(ListNode head) {
```

```
    // 递归终止条件：空节点或只有一个节点
    if (head == null || head.next == null) {
        return head;
    }

    // 递归处理后续节点，获取反转后链表的头节点
    ListNode newHead = reverseListRecursive(head.next);

    // 反转当前节点和下一个节点的连接关系
    head.next.next = head;  // 让下一个节点指向当前节点
    head.next = null;       // 断开当前节点的 next 指针

    return newHead; // 返回反转后链表的头节点
}

// 原有的解法保持不变
public static ListNode reverseList(ListNode head) {
    ListNode pre = null;
    ListNode next = null;
    while (head != null) {
        next = head.next;
        head.next = pre;
        pre = head;
        head = next;
    }
    return pre;
}

// ==================== 题目 2: 反转链表 II ====================
// 题目来源：
// 1. LeetCode 92. 反转链表 II - https://leetcode.cn/problems/reverse-linked-list-ii/

/**
 * 反转链表指定区间
 * 时间复杂度: O(n) - 最多遍历一次链表
 * 空间复杂度: O(1) - 只使用常数级别的额外空间
 *
 * 解题思路:
 * 1. 找到需要反转区间的前一个节点(pre)
 * 2. 找到需要反转区间的第一个节点(start)
 * 3. 使用头插法将区间内的节点逐个插入到 pre 节点之后
 * 4. 连接反转后的链表与其他部分
 *
```

```
     * 执行过程:
     * 原链表: 1 -> 2 -> 3 -> 4 -> 5, left=2, right=4
     * 步骤1: 找到pre(节点1)和start(节点2)
     * 步骤2: 将节点3插入到pre之后: 1 -> 3 -> 2 -> 4 -> 5
     * 步骤3: 将节点4插入到pre之后: 1 -> 4 -> 3 -> 2 -> 5
     * 结果: 1 -> 4 -> 3 -> 2 -> 5
     */
    public static ListNode reverseBetween(ListNode head, int left, int right) {
        // 创建虚拟头节点，简化边界处理
        ListNode dummy = new ListNode(0);
        dummy.next = head;

        // 找到反转区间的前一个节点
        ListNode pre = dummy;
        for (int i = 0; i < left - 1; i++) {
            pre = pre.next;
        }

        // start指向反转区间的第一个节点
        ListNode start = pre.next;
        // then指向待处理节点
        ListNode then = start.next;

        // 头插法实现区间反转
        for (int i = 0; i < right - left; i++) {
            start.next = then.next;
            then.next = pre.next;
            pre.next = then;
            then = start.next;
        }

        return dummy.next;
    }

    // ==================== 题目3: K 个一组翻转链表 ====================
    // 题目来源:
    // 1. LeetCode 25. K 个一组翻转链表 - https://leetcode.cn/problems/reverse-nodes-in-k-group/

    /**
     * K个一组反转链表
     * 时间复杂度: O(n) - 每个节点最多被访问两次
     * 空间复杂度: O(1) - 只使用常数级别的额外空间
     *
```

```
     * 解题思路:
     * 1. 分组处理，每次处理 k 个节点
     * 2. 对每组节点进行反转
     * 3. 连接各组之间的关系
     * 4. 处理不足 k 个的剩余节点（保持原顺序）
     *
     * 执行过程:
     * 原链表: 1 -> 2 -> 3 -> 4 -> 5, k=3
     * 第一组(1,2,3)反转: 3 -> 2 -> 1
     * 第二组(4,5)不足 k 个，保持原顺序: 4 -> 5
     * 结果: 3 -> 2 -> 1 -> 4 -> 5
     */
    public static ListNode reverseKGroup(ListNode head, int k) {
        // 计算链表长度
        int length = 0;
        ListNode current = head;
        while (current != null) {
            length++;
            current = current.next;
        }

        // 创建虚拟头节点
        ListNode dummy = new ListNode(0);
        dummy.next = head;

        // pre 指向已处理部分的最后一个节点
        ListNode pre = dummy;

        // 分组处理
        while (length >= k) {
            // start 指向当前组的第一个节点
            ListNode start = pre.next;
            // then 指向待处理节点
            ListNode then = start.next;

            // 对当前组进行 k-1 次头插操作
            for (int i = 1; i < k; i++) {
                start.next = then.next;
                then.next = pre.next;
                pre.next = then;
                then = start.next;
            }
```

```
            // 更新 pre 指针和剩余长度
            pre = start;
            length -= k;
        }

        return dummy.next;
    }

    // ==================== 双链表反转 ====================
    // 反转双链表
    // 题目来源：
    // 1. LeetCode 445. 两数相加 II（涉及链表反转思想）
    // 2. 剑指 Offer 24. 反转链表（扩展到双链表）
    // 3. 牛客网 反转双链表
    // 时间复杂度: O(n) - 需要遍历双链表一次
    // 空间复杂度: O(1) - 只使用常数级别的额外空间
    public static DoubleListNode reverseDoubleList(DoubleListNode head) {
        DoubleListNode pre = null;
        DoubleListNode next = null;
        while (head != null) {
            next = head.next;    // 保存下一个节点
            head.next = pre;     // 反转 next 指针
            head.last = next;    // 反转 last 指针
            pre = head;          // 移动 pre 指针
            head = next;         // 移动 head 指针
        }
        return pre;  // 返回新的头节点
    }

    // ==================== 补充题目 4: 回文链表 ====================
    // 题目来源：
    // 1. LeetCode 234. 回文链表 - https://leetcode.cn/problems/palindrome-linked-list/
    // 2. LintCode 223. 回文链表 - https://www.lintcode.com/problem/palindrome-linked-list/
    // 3. 牛客网 NC78 链表中倒数最后 k 个结点（相关题目）
    //
    // 题目描述：判断一个链表是否是回文链表
    // 输入：1->2->2->1
    // 输出：true
    //
    // 最优解法：使用快慢指针找到中点，反转后半部分，然后比较
    // 时间复杂度: O(n) - 只需要一次遍历找到中点，一次反转，一次比较
    // 空间复杂度: O(1) - 只使用常数级别的额外空间
    public static boolean isPalindrome(ListNode head) {
```

```
        if (head == null || head.next == null) {
            return true;  // 空链表或单节点链表是回文的
        }

        // 步骤1: 使用快慢指针找到链表的中点
        ListNode slow = head;
        ListNode fast = head;
        while (fast != null && fast.next != null) {
            slow = slow.next;        // 慢指针每次走一步
            fast = fast.next.next;   // 快指针每次走两步
        }
        // 循环结束后，slow指向中点位置（如果节点数为奇数）或后半部分的第一个节点（如果节点数为偶数）

        // 步骤2: 反转后半部分链表
        ListNode secondHalfHead = reverseListIterative(slow);
        // 保存反转后的头节点，用于后续恢复
        ListNode secondHalfStart = secondHalfHead;

        // 步骤3: 比较前半部分和反转后的后半部分
        ListNode firstHalfHead = head;
        boolean isPalindrome = true;
        while (secondHalfHead != null) {
            if (firstHalfHead.val != secondHalfHead.val) {
                isPalindrome = false;
                break;
            }
            firstHalfHead = firstHalfHead.next;
            secondHalfHead = secondHalfHead.next;
        }

        // 步骤4: 恢复链表（可选，但这是良好的工程实践）
        reverseListIterative(secondHalfStart);

        return isPalindrome;
    }

    // ==================== 补充题目5: 旋转链表 ====================
    // 题目来源：
    // 1. LeetCode 61. 旋转链表 - https://leetcode.cn/problems/rotate-list/
    // 2. LintCode 170. 旋转链表 - https://www.lintcode.com/problem/rotate-list/
    // 3. 牛客网 NC53 删除链表的倒数第n个节点（相关题目）
    //
```

```
// 题目描述：将链表向右旋转 k 个位置
// 输入：1->2->3->4->5->NULL, k = 2
// 输出：4->5->1->2->3->NULL
//
// 解题思路：
// 1. 先计算链表长度
// 2. 将链表首尾相连形成环
// 3. 在合适位置断开环
// 时间复杂度: O(n) - 需要遍历链表
// 空间复杂度: O(1) - 只使用常数级别的额外空间
public static ListNode rotateRight(ListNode head, int k) {
    // 处理特殊情况
    if (head == null || head.next == null || k == 0) {
        return head;
    }

    // 步骤 1: 计算链表长度并找到尾节点
    int length = 1;
    ListNode tail = head;
    while (tail.next != null) {
        tail = tail.next;
        length++;
    }

    // 步骤 2: 计算实际需要旋转的次数（取模操作避免多余旋转）
    k = k % length;
    if (k == 0) {
        return head;  // 不需要旋转
    }

    // 步骤 3: 将链表首尾相连形成环
    tail.next = head;

    // 步骤 4: 找到新的尾节点位置，距离原头节点 (length - k) 个位置
    ListNode newTail = head;
    for (int i = 0; i < length - k - 1; i++) {
        newTail = newTail.next;
    }

    // 步骤 5: 新的头节点是新尾节点的下一个节点
    ListNode newHead = newTail.next;

    // 步骤 6: 断开环
```

```
        newTail.next = null;

        return newHead;
    }

    // ==================== 补充题目6: 合并两个有序链表 ====================
    // 题目来源：
    // 1. LeetCode 21. 合并两个有序链表 - https://leetcode.cn/problems/merge-two-sorted-lists/
    // 2. LintCode 165. 合并两个排序链表 - https://www.lintcode.com/problem/merge-two-sorted-lists/
    // 3. 剑指Offer 25. 合并两个排序的链表
    // 4. 牛客网 NC33 合并两个排序的链表
    //
    // 题目描述：将两个升序链表合并为一个新的升序链表
    // 输入：l1 = [1,2,4], l2 = [1,3,4]
    // 输出：[1,1,2,3,4,4]
    //
    // 解题思路：使用迭代或递归方法，逐个比较两个链表的节点值
    // 时间复杂度: O(n+m) - n和m分别是两个链表的长度
    // 空间复杂度: O(1) - 迭代版本，只使用常数级别的额外空间
    public static ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        // 创建虚拟头节点，简化边界情况处理
        ListNode dummy = new ListNode(-1);
        ListNode current = dummy;

        // 迭代比较两个链表的节点值
        while (l1 != null && l2 != null) {
            if (l1.val <= l2.val) {
                current.next = l1;
                l1 = l1.next;
            } else {
                current.next = l2;
                l2 = l2.next;
            }
            current = current.next;
        }

        // 连接剩余部分
        current.next = (l1 != null) ? l1 : l2;

        return dummy.next;
    }
```

```
    // ==================== 补充题目 7: 两两交换链表中的节点 ====================
    // 题目来源：
    // 1. LeetCode 24. 两两交换链表中的节点 - https://leetcode.cn/problems/swap-nodes-in-pairs/
    // 2. LintCode 451. 两两交换链表中的节点 - https://www.lintcode.com/problem/swap-nodes-in-pairs/
    // 3. 牛客网 NC142 链表的奇偶重排（相关题目）
    //
    // 题目描述：两两交换链表中的相邻节点
    // 输入：1->2->3->4
    // 输出：2->1->4->3
    //
    // 解题思路：使用虚拟头节点和迭代方法
    // 时间复杂度: O(n) - 需要遍历链表一次
    // 空间复杂度: O(1) - 只使用常数级别的额外空间
    public static ListNode swapPairs(ListNode head) {
        // 创建虚拟头节点
        ListNode dummy = new ListNode(0);
        dummy.next = head;
        ListNode prev = dummy;

        // 当有至少两个节点可以交换时
        while (prev.next != null && prev.next.next != null) {
            // 获取需要交换的两个节点
            ListNode first = prev.next;
            ListNode second = prev.next.next;

            // 执行交换操作
            first.next = second.next;  // 1 -> 3
            second.next = first;       // 2 -> 1
            prev.next = second;        // dummy -> 2

            // 移动 prev 指针到下一对的前一个位置
            prev = first;
        }

        return dummy.next;
    }

    // ==================== 补充题目 8: 重排链表 ====================
    // 题目来源：
    // 1. LeetCode 143. 重排链表 - https://leetcode.cn/problems/reorder-list/
    // 2. LintCode 99. 重排链表 - https://www.lintcode.com/problem/reorder-list/
    // 3. 牛客网 NC40 链表相加（二）（相关题目）
```

```
//
// 题目描述：按照 L0 → Ln → L1 → Ln-1 → L2 → Ln-2 → ... 重新排列链表
// 输入：1->2->3->4
// 输出：1->4->2->3
//
// 解题思路：
// 1. 使用快慢指针找到中点
// 2. 反转后半部分链表
// 3. 合并两个链表
// 时间复杂度: O(n) - 需要遍历链表三次
// 空间复杂度: O(1) - 只使用常数级别的额外空间
public static void reorderList(ListNode head) {
    if (head == null || head.next == null || head.next.next == null) {
        return;  // 无需重排
    }

    // 步骤 1: 使用快慢指针找到链表中点
    ListNode slow = head;
    ListNode fast = head;
    while (fast.next != null && fast.next.next != null) {
        slow = slow.next;
        fast = fast.next.next;
    }

    // 步骤 2: 反转后半部分链表
    ListNode secondHalf = reverseListIterative(slow.next);
    slow.next = null;  // 断开前半部分和后半部分

    // 步骤 3: 合并两个链表
    ListNode firstHalf = head;
    while (secondHalf != null) {
        ListNode temp1 = firstHalf.next;
        ListNode temp2 = secondHalf.next;

        firstHalf.next = secondHalf;
        secondHalf.next = temp1;

        firstHalf = temp1;
        secondHalf = temp2;
    }
}

// ==================== 补充题目 9: 删除链表的倒数第 N 个节点 ====================
```

```
    // 题目来源：
    // 1. LeetCode 19. 删除链表的倒数第 N 个节点 - https://leetcode.cn/problems/remove-nth-node-from-end-of-list/
    // 2. LintCode 174. 删除链表中倒数第 n 个节点 - https://www.lintcode.com/problem/remove-nth-node-from-end-of-list/
    // 3. 牛客网 NC53 删除链表的倒数第 n 个节点
    // 4. 剑指 Offer 22. 链表中倒数第 k 个节点（相关题目）
    //
    // 题目描述：删除链表的倒数第 n 个节点，返回链表的头节点
    // 输入：head = [1,2,3,4,5], n = 2
    // 输出：[1,2,3,5]
    //
    // 解题思路：使用快慢指针，快指针先走 n 步，然后快慢指针一起走
    // 时间复杂度: O(n) - 只需要遍历链表一次
    // 空间复杂度: O(1) - 只使用常数级别的额外空间
    //
    // 是否为最优解：是。这是该问题的最优解。
    // 只需要遍历链表一次，空间复杂度为 O(1)，没有更优的算法。
    public static ListNode removeNthFromEnd(ListNode head, int n) {
        // 创建虚拟头节点，简化边界情况处理
        ListNode dummy = new ListNode(0);
        dummy.next = head;

        // 设置快慢指针
        ListNode fast = dummy;
        ListNode slow = dummy;

        // 快指针先走 n+1 步
        for (int i = 0; i <= n; i++) {
            fast = fast.next;
            // 如果 n 大于链表长度，fast 会变成 null
            if (i < n && fast == null) {
                return head;  // n 大于链表长度，无法删除
            }
        }

        // 快慢指针一起走，直到快指针到达链表末尾
        while (fast != null) {
            fast = fast.next;
            slow = slow.next;
        }

        // 此时 slow 指向待删除节点的前一个节点
```

```
        slow.next = slow.next.next;

        return dummy.next;
    }

    // ==================== 工程化考虑 ====================

    /**
     * 带异常处理的链表反转方法
     *
     * @param head 链表头节点
     * @return 反转后的链表头节点
     * @throws IllegalArgumentException 当输入参数非法时抛出异常
     */
    public static ListNode reverseListSafe(ListNode head) throws IllegalArgumentException {
        // 参数校验
        if (head == null) {
            return null;
        }

        // 检查链表是否有环(简化版检查)
        ListNode slow = head;
        ListNode fast = head;
        while (fast != null && fast.next != null) {
            slow = slow.next;
            fast = fast.next.next;
            if (slow == fast) {
                throw new IllegalArgumentException("链表中存在环，无法进行反转");
            }
        }

        // 执行反转
        return reverseListIterative(head);
    }

    /**
     * 链表反转的单元测试方法
     */
    public static void runUnitTests() {
        System.out.println("=== 链表反转单元测试 ===");

        // 测试用例 1: 正常情况
        ListNode test1 = ListNode.createList(new int[]{1, 2, 3, 4, 5});
```

```
        ListNode result1 = reverseListIterative(test1);
        System.out.println("测试 1 - 输入[1,2,3,4,5], 期望[5,4,3,2,1], 实际: " + result1);

        // 测试用例 2: 空链表
        ListNode result2 = reverseListIterative(null);
        System.out.println("测试 2 - 输入[], 期望[], 实际: " + result2);

        // 测试用例 3: 单节点链表
        ListNode test3 = new ListNode(1);
        ListNode result3 = reverseListIterative(test3);
        System.out.println("测试 3 - 输入[1], 期望[1], 实际: " + (result3 != null ? result3.val :
"null"));

        // 测试用例 4: 两节点链表
        ListNode test4 = ListNode.createList(new int[]{1, 2});
        ListNode result4 = reverseListIterative(test4);
        System.out.println("测试 4 - 输入[1,2], 期望[2,1], 实际: " + result4);

        System.out.println("单元测试完成\n");
    }

    // ==================== 补充题目 10: 奇偶链表 ====================
    // 题目来源:
    // 1. LeetCode 328. 奇偶链表 - https://leetcode.cn/problems/odd-even-linked-list/
    // 2. LintCode 1292. 奇偶链表 - https://www.lintcode.com/problem/odd-even-linked-list/
    // 3. 牛客网 NC142 链表的奇偶重排
    //
    // 题目描述：将链表的奇数节点和偶数节点分组在一起，保持相对顺序不变
    // 输入：1->2->3->4->5->NULL
    // 输出：1->3->5->2->4->NULL
    //
    // 解题思路:
    // 使用两个指针分别处理奇数节点和偶数节点
    // 1. odd 指针连接所有奇数位置的节点
    // 2. even 指针连接所有偶数位置的节点
    // 3. 最后将奇数链表的尾部连接到偶数链表的头部
    //
    // 时间复杂度: O(n) - 只需要遍历链表一次
    // 空间复杂度: O(1) - 只使用常数级别的额外空间
    //
    // 是否为最优解：是。这是该问题的最优解。
    public static ListNode oddEvenList(ListNode head) {
        // 边界情况处理
```

```
        if (head == null || head.next == null) {
            return head;
        }

        // odd 指向奇数位置节点，even 指向偶数位置节点
        ListNode odd = head;
        ListNode even = head.next;
        ListNode evenHead = even;  // 保存偶数链表的头节点

        // 遍历链表，将奇数节点和偶数节点分组
        while (even != null && even.next != null) {
            odd.next = even.next;   // 连接下一个奇数节点
            odd = odd.next;         // 移动 odd 指针
            even.next = odd.next;   // 连接下一个偶数节点
            even = even.next;       // 移动 even 指针
        }

        // 将奇数链表的尾部连接到偶数链表的头部
        odd.next = evenHead;

        return head;
    }

    // ==================== 补充题目 11: 分隔链表 ====================
    // 题目来源：
    // 1. LeetCode 86. 分隔链表 - https://leetcode.cn/problems/partition-list/
    // 2. LintCode 96. 分隔链表 - https://www.lintcode.com/problem/partition-list/
    // 3. 牛客网 NC188 分隔链表
    //
    // 题目描述：给定一个链表和一个特定值 x，将链表分隔成两部分，使得所有小于 x 的节点都在大于或等于 x 的节点之前
    // 输入：head = 1->4->3->2->5->2, x = 3
    // 输出：1->2->2->4->3->5
    //
    // 解题思路：
    // 使用两个虚拟头节点分别存储小于 x 和大于等于 x 的节点
    // 1. before 链表存储所有小于 x 的节点
    // 2. after 链表存储所有大于等于 x 的节点
    // 3. 最后连接两个链表
    //
    // 时间复杂度: O(n) - 需要遍历链表一次
    // 空间复杂度: O(1) - 只使用常数级别的额外空间
    //
```

```
// 是否为最优解：是。这是该问题的最优解。
public static ListNode partition(ListNode head, int x) {
    // 创建两个虚拟头节点
    ListNode beforeHead = new ListNode(0);
    ListNode before = beforeHead;
    ListNode afterHead = new ListNode(0);
    ListNode after = afterHead;

    // 遍历链表，将节点分配到两个链表中
    while (head != null) {
        if (head.val < x) {
            before.next = head;
            before = before.next;
        } else {
            after.next = head;
            after = after.next;
        }
        head = head.next;
    }

    // 防止形成环：将 after 链表的最后一个节点的 next 设置为 null
    after.next = null;
    // 连接两个链表
    before.next = afterHead.next;

    return beforeHead.next;
}

// ==================== 补充题目 12: 链表求和 ====================
// 题目来源：
// 1. LeetCode 2. 两数相加 - https://leetcode.cn/problems/add-two-numbers/
// 2. LeetCode 445. 两数相加 II - https://leetcode.cn/problems/add-two-numbers-ii/
// 3. LintCode 167. 链表求和 - https://www.lintcode.com/problem/add-two-numbers/
// 4. 牛客网 NC40 链表相加（二）
//
// 题目描述：给定两个非空链表表示两个非负整数，数字最高位在链表尾部，计算两个数的和
// 输入：l1 = 2->4->3, l2 = 5->6->4
// 输出：7->0->8 (342 + 465 = 807)
//
// 解题思路：
// 从低位开始逐位相加，处理进位
// 1. 同时遍历两个链表
// 2. 对应位置的节点值相加，加上进位
```

```
// 3. 计算当前位的值和新的进位
// 4. 创建新节点存储当前位的值
//
// 时间复杂度: O(max(m,n)) - m 和 n 分别是两个链表的长度
// 空间复杂度: O(max(m,n)) - 需要创建新链表存储结果
//
// 是否为最优解：是。这是该问题的最优解。
public static ListNode addTwoNumbers(ListNode l1, ListNode l2) {
    ListNode dummy = new ListNode(0);  // 虚拟头节点
    ListNode current = dummy;
    int carry = 0;  // 进位

    // 遍历两个链表，处理每一位的相加
    while (l1 != null || l2 != null || carry != 0) {
        // 获取当前位的值
        int val1 = (l1 != null) ? l1.val : 0;
        int val2 = (l2 != null) ? l2.val : 0;

        // 计算当前位的和
        int sum = val1 + val2 + carry;
        carry = sum / 10;  // 计算新的进位

        // 创建新节点存储当前位的值
        current.next = new ListNode(sum % 10);
        current = current.next;

        // 移动指针
        if (l1 != null) l1 = l1.next;
        if (l2 != null) l2 = l2.next;
    }

    return dummy.next;
}

// ==================== 补充题目 13: 环形链表 ====================
// 题目来源：
// 1. LeetCode 141. 环形链表 - https://leetcode.cn/problems/linked-list-cycle/
// 2. LeetCode 142. 环形链表 II - https://leetcode.cn/problems/linked-list-cycle-ii/
// 3. LintCode 102. 环形链表 - https://www.lintcode.com/problem/linked-list-cycle/
// 4. 牛客网 NC4 判断链表中是否有环
// 5. 剑指 Offer 23. 链表中环的入口节点
//
// 题目描述：判断链表中是否有环
```

```
//
// 解题思路：Floyd 判圈算法（快慢指针）
// 使用快慢两个指针，快指针每次走两步，慢指针每次走一步
// 如果链表有环，快慢指针最终会相遇
//
// 时间复杂度: O(n) - 需要遍历链表
// 空间复杂度: O(1) - 只使用常数级别的额外空间
//
// 是否为最优解：是。这是该问题的最优解。
public static boolean hasCycle(ListNode head) {
    if (head == null || head.next == null) {
        return false;
    }

    // 快慢指针
    ListNode slow = head;
    ListNode fast = head;

    // 快指针每次走两步，慢指针每次走一步
    while (fast != null && fast.next != null) {
        slow = slow.next;
        fast = fast.next.next;

        // 如果快慢指针相遇，说明有环
        if (slow == fast) {
            return true;
        }
    }

    return false;  // 快指针到达链表末尾，说明无环
}

/**
 * 找到环形链表的入口节点
 *
 * 解题思路：
 * 1. 使用快慢指针找到相遇点
 * 2. 将其中一个指针移到链表头部
 * 3. 两个指针以相同速度移动，再次相遇点就是环的入口
 *
 * 数学证明：
 * 设链表头到环入口的距离为 a，环入口到相遇点的距离为 b，相遇点到环入口的距离为 c
 * 慢指针走过的距离：a + b
```

```
     * 快指针走过的距离：a + b + n(b + c)，其中 n 为快指针在环中走过的圈数
     * 因为快指针速度是慢指针的 2 倍，所以：2(a + b) = a + b + n(b + c)
     * 化简得：a = (n-1)(b + c) + c
     * 这意味着从头节点到环入口的距离，等于从相遇点到环入口的距离（加上若干圈环的长度）
     *
     * 时间复杂度: O(n)
     * 空间复杂度: O(1)
     */
    public static ListNode detectCycle(ListNode head) {
        if (head == null || head.next == null) {
            return null;
        }

        // 步骤 1: 使用快慢指针找到相遇点
        ListNode slow = head;
        ListNode fast = head;
        boolean hasCycle = false;

        while (fast != null && fast.next != null) {
            slow = slow.next;
            fast = fast.next.next;

            if (slow == fast) {
                hasCycle = true;
                break;
            }
        }

        if (!hasCycle) {
            return null;  // 无环
        }

        // 步骤 2: 将 slow 指针移到链表头部
        slow = head;

        // 步骤 3: 两个指针以相同速度移动，相遇点就是环的入口
        while (slow != fast) {
            slow = slow.next;
            fast = fast.next;
        }

        return slow;  // 返回环的入口节点
    }
```

```
    // ==================== 补充题目 14: 相交链表 ====================
    // 题目来源：
    // 1. LeetCode 160. 相交链表 - https://leetcode.cn/problems/intersection-of-two-linked-lists/
    // 2. LintCode 380. 相交链表 - https://www.lintcode.com/problem/intersection-of-two-linked-lists/
    // 3. 牛客网 NC66 两个链表的第一个公共结点
    // 4. 剑指 Offer 52. 两个链表的第一个公共节点
    //
    // 题目描述：找到两个单链表相交的起始节点
    //
    // 解题思路：双指针法
    // 1. 两个指针分别从两个链表头部开始遍历
    // 2. 当指针到达链表末尾时，跳转到另一个链表的头部
    // 3. 如果两个链表相交，两个指针最终会在相交点相遇
    // 4. 如果不相交，两个指针最终都会到达 null
    //
    // 时间复杂度: O(m+n) - m 和 n 分别是两个链表的长度
    // 空间复杂度: O(1) - 只使用常数级别的额外空间
    //
    // 是否为最优解：是。这是该问题的最优解。
    public static ListNode getIntersectionNode(ListNode headA, ListNode headB) {
        if (headA == null || headB == null) {
            return null;
        }

        // 两个指针分别从两个链表头部开始
        ListNode pA = headA;
        ListNode pB = headB;

        // 遍历链表，当到达末尾时跳转到另一个链表
        while (pA != pB) {
            // 如果 pA 到达末尾，跳转到 headB；否则继续前进
            pA = (pA == null) ? headB : pA.next;
            // 如果 pB 到达末尾，跳转到 headA；否则继续前进
            pB = (pB == null) ? headA : pB.next;
        }

        // 返回相交节点（如果不相交，pA 和 pB 都为 null）
        return pA;
    }

    // ==================== 补充题目 15: 排序链表 ====================
```

```
// 题目来源：
// 1. LeetCode 148. 排序链表 - https://leetcode.cn/problems/sort-list/
// 2. LintCode 98. 排序链表 - https://www.lintcode.com/problem/sort-list/
// 3. 牛客网 NC12 重建二叉树（相关题目）
//
// 题目描述：在 O(n log n) 时间复杂度和常数级空间复杂度下，对链表进行排序
// 输入：4->2->1->3
// 输出：1->2->3->4
//
// 解题思路：归并排序
// 1. 使用快慢指针找到链表中点，将链表分成两半
// 2. 递归地对两半链表进行排序
// 3. 合并两个有序链表
//
// 时间复杂度: O(n log n) - 归并排序的时间复杂度
// 空间复杂度: O(log n) - 递归调用栈的深度
//
// 是否为最优解：是。这是该问题的最优解。
// 如果要求O(1)空间复杂度，可以使用自底向上的归并排序，但实现较复杂。
public static ListNode sortList(ListNode head) {
    // 递归终止条件
    if (head == null || head.next == null) {
        return head;
    }

    // 步骤1: 使用快慢指针找到链表中点
    ListNode slow = head;
    ListNode fast = head.next;  // fast从第二个节点开始，保证slow停在中点的前一个位置

    while (fast != null && fast.next != null) {
        slow = slow.next;
        fast = fast.next.next;
    }

    // 将链表分成两半
    ListNode mid = slow.next;
    slow.next = null;  // 断开前半部分和后半部分

    // 步骤2: 递归排序两半链表
    ListNode left = sortList(head);
    ListNode right = sortList(mid);

    // 步骤3: 合并两个有序链表
```

```
    return mergeTwoLists(left, right);
}

// ==================== 补充题目 16: 链表随机节点 ====================
// 题目来源：
// 1. LeetCode 382. 链表随机节点 - https://leetcode.cn/problems/linked-list-random-node/
// 2. 蓄水池抽样算法应用
//
// 题目描述：从链表中随机返回一个节点的值，保证每个节点被选中的概率相等
//
// 解题思路：蓄水池抽样算法
// 1. 遍历链表，对于第 i 个节点，以 1/i 的概率选择它
// 2. 这样能保证每个节点被选中的概率都是 1/n
//
// 时间复杂度: O(n) - 需要遍历整个链表
// 空间复杂度: O(1) - 只使用常数级别的额外空间
//
// 是否为最优解：是。这是蓄水池抽样算法的标准实现。
public static class RandomNodeSelector {
    private ListNode head;
    private Random random;

    public RandomNodeSelector(ListNode head) {
        this.head = head;
        this.random = new Random();
    }

    public int getRandom() {
        ListNode current = head;
        int result = 0;
        int count = 0;

        while (current != null) {
            count++;
            // 以 1/count 的概率选择当前节点
            if (random.nextInt(count) == 0) {
                result = current.val;
            }
            current = current.next;
        }

        return result;
    }
```

```
    }

    // ==================== 补充题目 17: 复制带随机指针的链表 ====================
    // 题目来源：
    // 1. LeetCode 138. 复制带随机指针的链表 - https://leetcode.cn/problems/copy-list-with-random-pointer/
    // 2. 剑指 Offer 35. 复杂链表的复制
    //
    // 题目描述：复制一个包含随机指针的链表
    //
    // 解题思路：三次遍历法
    // 1. 第一次遍历：在每个节点后面插入复制节点
    // 2. 第二次遍历：设置复制节点的随机指针
    // 3. 第三次遍历：分离原链表和复制链表
    //
    // 时间复杂度: O(n) - 需要三次遍历
    // 空间复杂度: O(1) - 只使用常数级别的额外空间（不包括结果链表）
    //
    // 是否为最优解：是。这是该问题的最优解。
    public static class NodeWithRandom {
        public int val;
        public NodeWithRandom next;
        public NodeWithRandom random;

        public NodeWithRandom(int val) {
            this.val = val;
            this.next = null;
            this.random = null;
        }
    }

    public static NodeWithRandom copyRandomList(NodeWithRandom head) {
        if (head == null) {
            return null;
        }

        // 第一次遍历：在每个节点后面插入复制节点
        NodeWithRandom current = head;
        while (current != null) {
            NodeWithRandom copy = new NodeWithRandom(current.val);
            copy.next = current.next;
            current.next = copy;
            current = copy.next;
```

```
    }

    // 第二次遍历：设置复制节点的随机指针
    current = head;
    while (current != null) {
        if (current.random != null) {
            current.next.random = current.random.next;
        }
        current = current.next.next;
    }

    // 第三次遍历：分离原链表和复制链表
    current = head;
    NodeWithRandom copyHead = head.next;
    NodeWithRandom copyCurrent = copyHead;

    while (current != null) {
        current.next = current.next.next;
        if (copyCurrent.next != null) {
            copyCurrent.next = copyCurrent.next.next;
        }
        current = current.next;
        copyCurrent = copyCurrent.next;
    }

    return copyHead;
}

// ==================== 补充题目 18: 链表组件 ====================
// 题目来源：
// 1. LeetCode 817. 链表组件 - https://leetcode.cn/problems/linked-list-components/
//
// 题目描述：给定链表头节点 head 和列表 nums，返回链表中组件的个数
//
// 解题思路：使用哈希集合快速查找
// 1. 将 nums 转换为哈希集合，提高查找效率
// 2. 遍历链表，统计连续在集合中的组件个数
//
// 时间复杂度: O(n + m) - n 是链表长度，m 是 nums 长度
// 空间复杂度: O(m) - 用于存储哈希集合
//
// 是否为最优解：是。这是该问题的最优解。
public static int numComponents(ListNode head, int[] nums) {
```

```
        Set<Integer> numSet = new HashSet<>();
        for (int num : nums) {
            numSet.add(num);
        }

        int components = 0;
        boolean inComponent = false;
        ListNode current = head;

        while (current != null) {
            if (numSet.contains(current.val)) {
                if (!inComponent) {
                    components++;
                    inComponent = true;
                }
            } else {
                inComponent = false;
            }
            current = current.next;
        }

        return components;
    }

    // ==================== 补充题目 19: 链表中的下一个更大节点 ====================
    // 题目来源：
    // 1. LeetCode 1019. 链表中的下一个更大节点 - https://leetcode.cn/problems/next-greater-node-in-linked-list/
    //
    // 题目描述：对于链表中的每个节点，找到它后面第一个比它大的节点
    //
    // 解题思路：使用单调栈
    // 1. 先将链表转换为数组
    // 2. 使用单调栈从右向左处理
    // 3. 栈中存储的是递减序列的索引
    //
    // 时间复杂度: O(n) - 每个节点入栈出栈一次
    // 空间复杂度: O(n) - 用于存储栈和结果数组
    //
    // 是否为最优解：是。这是该问题的最优解。
    public static int[] nextLargerNodes(ListNode head) {
        // 将链表转换为数组
        List<Integer> list = new ArrayList<>();
```

```
        ListNode current = head;
        while (current != null) {
            list.add(current.val);
            current = current.next;
        }

        int n = list.size();
        int[] result = new int[n];
        Stack<Integer> stack = new Stack<>();

        // 从右向左遍历，使用单调栈
        for (int i = n - 1; i >= 0; i--) {
            int currentVal = list.get(i);

            // 弹出栈顶比当前值小的元素
            while (!stack.isEmpty() && stack.peek() <= currentVal) {
                stack.pop();
            }

            // 如果栈不为空，栈顶就是下一个更大节点
            result[i] = stack.isEmpty() ? 0 : stack.peek();

            // 将当前值压入栈
            stack.push(currentVal);
        }

        return result;
    }

    // ==================== 补充题目 20: 链表最大孪生和 ====================
    // 题目来源：
    // 1. LeetCode 2130. 链表最大孪生和 - https://leetcode.cn/problems/maximum-twin-sum-of-a-linked-list/
    //
    // 题目描述：孪生节点是第 i 个节点和第 n-1-i 个节点，求所有孪生节点和的最大值
    //
    // 解题思路：快慢指针 + 反转后半部分
    // 1. 使用快慢指针找到中点
    // 2. 反转后半部分链表
    // 3. 同时遍历前半部分和反转后的后半部分，计算和
    //
    // 时间复杂度: O(n) - 需要遍历链表三次
    // 空间复杂度: O(1) - 只使用常数级别的额外空间
```

```
//
// 是否为最优解：是。这是该问题的最优解。
public static int pairSum(ListNode head) {
    // 使用快慢指针找到中点
    ListNode slow = head;
    ListNode fast = head;
    while (fast != null && fast.next != null) {
        slow = slow.next;
        fast = fast.next.next;
    }

    // 反转后半部分链表
    ListNode secondHalf = reverseListIterative(slow);

    // 计算孪生和的最大值
    int maxSum = 0;
    ListNode firstHalf = head;
    while (secondHalf != null) {
        maxSum = Math.max(maxSum, firstHalf.val + secondHalf.val);
        firstHalf = firstHalf.next;
        secondHalf = secondHalf.next;
    }

    return maxSum;
}

// ==================== 性能优化与调试工具 ====================

/**
 * 链表性能分析工具
 * 用于分析链表操作的时间和空间复杂度
 */
public static class LinkedListProfiler {
    private long startTime;
    private long endTime;
    private Runtime runtime;
    private long startMemory;

    public void start() {
        runtime = Runtime.getRuntime();
        // 运行垃圾回收，获取更准确的内存使用
        runtime.gc();
        startMemory = runtime.totalMemory() - runtime.freeMemory();
```

```
        startTime = System.nanoTime();
    }

    public void end() {
        endTime = System.nanoTime();
        long endMemory = runtime.totalMemory() - runtime.freeMemory();
        long memoryUsed = endMemory - startMemory;

        System.out.println("执行时间: " + (endTime - startTime) + " 纳秒");
        System.out.println("内存使用: " + memoryUsed + " 字节");
    }
}

/**
 * 链表调试工具
 * 提供打印中间状态、断言验证等功能
 */
public static class LinkedListDebugger {
    /**
     * 打印链表中间状态
     * @param head 链表头节点
     * @param message 调试信息
     */
    public static void printListState(ListNode head, String message) {
        System.out.println(message + ": " + (head != null ? head.toString() : "null"));
    }

    /**
     * 断言验证链表状态
     * @param condition 断言条件
     * @param message 断言失败信息
     */
    public static void assertList(boolean condition, String message) {
        if (!condition) {
            throw new AssertionError("链表断言失败: " + message);
        }
    }

    /**
     * 验证链表是否有环
     * @param head 链表头节点
     * @return 是否有环
     */
```

```
    public static boolean verifyNoCycle(ListNode head) {
        if (head == null) return true;

        ListNode slow = head;
        ListNode fast = head;

        while (fast != null && fast.next != null) {
            slow = slow.next;
            fast = fast.next.next;
            if (slow == fast) {
                return false; // 有环
            }
        }
        return true; // 无环
    }
}

// ==================== 综合测试函数 ====================

/**
 * 运行所有补充题目的测试
 */
public static void testAdditionalProblems() {
    System.out.println("=== 补充题目测试 ===");

    // 测试链表随机节点
    ListNode randomTest = ListNode.createList(new int[]{1, 2, 3, 4, 5});
    RandomNodeSelector selector = new RandomNodeSelector(randomTest);
    System.out.println("随机节点选择测试: " + selector.getRandom());

    // 测试链表组件
    ListNode componentTest = ListNode.createList(new int[]{0, 1, 2, 3});
    int[] nums = {0, 1, 3};
    System.out.println("链表组件个数: " + numComponents(componentTest, nums));

    // 测试下一个更大节点
    ListNode largerTest = ListNode.createList(new int[]{2, 1, 5});
    int[] largerResult = nextLargerNodes(largerTest);
    System.out.println("下一个更大节点: " + Arrays.toString(largerResult));

    // 测试链表最大孪生和
    ListNode twinTest = ListNode.createList(new int[]{5, 4, 2, 1});
    System.out.println("链表最大孪生和: " + pairSum(twinTest));
```

```
        System.out.println("补充题目测试完成");
    }

    // ==================== 主函数更新 ====================

    // 在原有的 main 方法中添加新的测试功能
    // 原有的 main 方法已经包含了基础测试
    // 现在添加补充测试功能到原有的 main 方法中
}
```

==========

文件: ListReverse.py

==========

```
# -*- coding: utf-8 -*-
"""
链表反转相关算法题目集合
包含 LeetCode、牛客网、Codeforces、LintCode、HackerRank 等平台的相关题目
每个题目都提供详细的解题思路、复杂度分析和多种解法

题目列表：
1. 反转链表 (LeetCode 206, 牛客网, HackerRank, LintCode 35, 剑指 Offer 24)
2. 反转链表 II (LeetCode 92)
3. K 个一组翻转链表 (LeetCode 25)
4. 回文链表 (LeetCode 234, LintCode 223, 牛客网 NC78)
5. 旋转链表 (LeetCode 61, LintCode 170)
6. 合并两个有序链表 (LeetCode 21, LintCode 165, 剑指 Offer 25, 牛客网 NC33)
7. 两两交换链表中的节点 (LeetCode 24, LintCode 451)
8. 重排链表 (LeetCode 143, LintCode 99)
9. 删除链表的倒数第 N 个节点 (LeetCode 19, LintCode 174, 牛客网 NC53, 剑指 Offer 22)
10. 奇偶链表 (LeetCode 328, LintCode 1292, 牛客网 NC142)
11. 分隔链表 (LeetCode 86, LintCode 96, 牛客网 NC188)
12. 链表求和 (LeetCode 2, LeetCode 445, LintCode 167, 牛客网 NC40)
13. 环形链表 (LeetCode 141, LeetCode 142, LintCode 102, 牛客网 NC4, 剑指 Offer 23)
14. 相交链表 (LeetCode 160, LintCode 380, 牛客网 NC66, 剑指 Offer 52)
15. 排序链表 (LeetCode 148, LintCode 98)
16. 链表随机节点 (LeetCode 382)
17. 复制带随机指针的链表 (LeetCode 138, 剑指 Offer 35)
18. 链表组件 (LeetCode 817)
19. 链表中的下一个更大节点 (LeetCode 1019)
20. 链表最大孪生和 (LeetCode 2130)
"""
```

```
from typing import Optional

class ListNode:
    """单链表节点"""
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

    def __str__(self):
        """字符串表示链表"""
        result = []
        current = self
        while current:
            result.append(str(current.val))
            current = current.next
        return " -> ".join(result)

    @staticmethod
    def create_list(vals):
        """创建链表的辅助方法"""
        if not vals:
            return None

        head = ListNode(vals[0])
        current = head
        for i in range(1, len(vals)):
            current.next = ListNode(vals[i])
            current = current.next
        return head


class DoubleListNode:
    """双链表节点"""
    def __init__(self, val=0):
        self.val = val
        self.prev: Optional['DoubleListNode'] = None
        self.next: Optional['DoubleListNode'] = None


def reverse_list_iterative(head: Optional[ListNode]) -> Optional[ListNode]:
    """
    方法 1: 迭代法反转链表
```

```
    时间复杂度: O(n) - 需要遍历链表一次
    空间复杂度: O(1) - 只使用了常数级别的额外空间

    解题思路:
    使用三个指针(pre, current, next)来逐个反转链表中的节点指向关系
    1. pre 指向已反转部分的最后一个节点
    2. current 指向当前待处理节点
    3. next 保存 current 的下一个节点，防止断链

    执行过程:
    原链表: 1 -> 2 -> 3 -> 4 -> 5 -> null
    步骤 1: null <- 1    2 -> 3 -> 4 -> 5 -> null
    步骤 2: null <- 1 <- 2    3 -> 4 -> 5 -> null
    步骤 3: null <- 1 <- 2 <- 3    4 -> 5 -> null
    ...
    最终: null <- 1 <- 2 <- 3 <- 4 <- 5
    """
    pre: Optional[ListNode] = None      # 已反转部分的头节点
    current: Optional[ListNode] = head  # 当前待处理节点
    next_node: Optional[ListNode] = None  # 保存 current 的下一个节点

    while current:
        next_node = current.next  # 保存下一个节点
        current.next = pre        # 反转当前节点的指向
        pre = current             # 移动 pre 指针
        current = next_node       # 移动 current 指针

    return pre  # pre 指向原链表的最后一个节点，即新链表的头节点



def reverse_list_recursive(head: Optional[ListNode]) -> Optional[ListNode]:
    """
    方法 2: 递归法反转链表
    时间复杂度: O(n) - 递归调用 n 次
    空间复杂度: O(n) - 递归调用栈的深度为 n

    解题思路:
    1. 递归到链表末尾
    2. 在回溯过程中逐个反转节点的指向
    3. 假设除了当前节点外，后续链表已经完成反转

    执行过程:
    原链表: 1 -> 2 -> 3 -> 4 -> 5 -> null
```

```
    递归到 5，返回 5
    回溯到 4: 4.next.next = 4 (即 5->4)，4.next = null
    回溯到 3: 3.next.next = 3 (即 4->3)，3.next = null
    ...
    """
    # 递归终止条件：空节点或只有一个节点
    if not head or not head.next:
        return head

    # 递归处理后续节点，获取反转后链表的头节点
    new_head = reverse_list_recursive(head.next)

    # 反转当前节点和下一个节点的连接关系
    head.next.next = head  # 让下一个节点指向当前节点
    head.next = None       # 断开当前节点的 next 指针

    return new_head  # 返回反转后链表的头节点


def reverse_between(head: Optional[ListNode], left: int, right: int) -> Optional[ListNode]:
    """
    反转链表指定区间
    时间复杂度: O(n) - 最多遍历一次链表
    空间复杂度: O(1) - 只使用常数级别的额外空间

    解题思路:
    1. 找到需要反转区间的前一个节点(pre)
    2. 找到需要反转区间的第一个节点(start)
    3. 使用头插法将区间内的节点逐个插入到 pre 节点之后
    4. 连接反转后的链表与其他部分

    执行过程:
    原链表: 1 -> 2 -> 3 -> 4 -> 5, left=2, right=4
    步骤 1: 找到 pre(节点 1)和 start(节点 2)
    步骤 2: 将节点 3 插入到 pre 之后: 1 -> 3 -> 2 -> 4 -> 5
    步骤 3: 将节点 4 插入到 pre 之后: 1 -> 4 -> 3 -> 2 -> 5
    结果: 1 -> 4 -> 3 -> 2 -> 5
    """
    if head is None:
        return None

    # 创建虚拟头节点，简化边界处理
    dummy = ListNode(0)
```

```
    dummy.next = head

    # 找到反转区间的前一个节点
    pre = dummy
    for i in range(left - 1):
        if pre.next is None:
            break
        pre = pre.next

    # start 指向反转区间的第一个节点
    if pre.next is None:
        return dummy.next
    start = pre.next

    # then 指向待处理节点
    if start.next is None:
        return dummy.next
    then = start.next

    # 头插法实现区间反转
    for i in range(right - left):
        if then is None:
            break
        start.next = then.next
        then.next = pre.next
        pre.next = then
        then = start.next

    return dummy.next


def reverse_k_group(head: Optional[ListNode], k: int) -> Optional[ListNode]:
    """
    K 个一组反转链表
    时间复杂度: O(n) - 每个节点最多被访问两次
    空间复杂度: O(1) - 只使用常数级别的额外空间

    解题思路:
    1. 分组处理，每次处理 k 个节点
    2. 对每组节点进行反转
    3. 连接各组之间的关系
    4. 处理不足 k 个的剩余节点（保持原顺序）
```

```
执行过程:
原链表: 1 -> 2 -> 3 -> 4 -> 5, k=3
第一组(1,2,3)反转: 3 -> 2 -> 1
第二组(4,5)不足 k 个，保持原顺序: 4 -> 5
结果: 3 -> 2 -> 1 -> 4 -> 5
"""
if head is None or k <= 1:
    return head

# 计算链表长度
length = 0
current = head
while current:
    length += 1
    current = current.next

# 创建虚拟头节点
dummy = ListNode(0)
dummy.next = head

# pre 指向已处理部分的最后一个节点
pre = dummy

# 分组处理
while length >= k:
    # start 指向当前组的第一个节点
    if pre.next is None:
        break
    start = pre.next

    # then 指向待处理节点
    if start.next is None:
        break
    then = start.next

    # 对当前组进行 k-1 次头插操作
    for i in range(k - 1):
        if then is None:
            break
        start.next = then.next
        then.next = pre.next
        pre.next = then
        then = start.next
```

```
        # 更新 pre 指针和剩余长度
        pre = start
        length -= k

    return dummy.next


def reverse_double_list(head):
    pre = None
    next_node = None
    while head is not None:
        next_node = head.next      # 保存下一个节点
        head.next = pre            # 反转 next 指针
        head.prev = next_node      # 反转 prev 指针
        pre = head                 # 移动 pre 指针
        head = next_node           # 移动 head 指针
    return pre  # 返回新的头节点

"""
补充题目 4: 回文链表
题目来源：
1. LeetCode 234. 回文链表 - https://leetcode.cn/problems/palindrome-linked-list/
2. LintCode 223. 回文链表 - https://www.lintcode.com/problem/palindrome-linked-list/
3. 牛客网 NC78 链表中倒数最后 k 个结点（相关题目）

题目描述：判断一个链表是否是回文链表
输入：1->2->2->1
输出：True

最优解法：使用快慢指针找到中点，反转后半部分，然后比较
时间复杂度: O(n) - 只需要一次遍历找到中点，一次反转，一次比较
空间复杂度: O(1) - 只使用常数级别的额外空间
"""
def is_palindrome(head):
    if head is None or head.next is None:
        return True  # 空链表或单节点链表是回文的

    # 步骤 1: 使用快慢指针找到链表的中点
    slow = head
    fast = head
    while fast is not None and fast.next is not None:
        slow = slow.next        # 慢指针每次走一步
```

```
        fast = fast.next.next    # 快指针每次走两步
    # 循环结束后，slow 指向中点位置（如果节点数为奇数）或后半部分的第一个节点（如果节点数为偶数）

    # 步骤 2: 反转后半部分链表
    second_half_head = reverse_list_iterative(slow)
    # 保存反转后的头节点，用于后续恢复
    second_half_start = second_half_head

    # 步骤 3: 比较前半部分和反转后的后半部分
    first_half_head = head
    is_palindrome_flag = True
    while second_half_head is not None:
        if first_half_head.val != second_half_head.val:
            is_palindrome_flag = False
            break
        first_half_head = first_half_head.next
        second_half_head = second_half_head.next

    # 步骤 4: 恢复链表（可选，但这是良好的工程实践）
    reverse_list_iterative(second_half_start)

    return is_palindrome_flag

"""
补充题目 5: 旋转链表
题目来源：
1. LeetCode 61. 旋转链表 - https://leetcode.cn/problems/rotate-list/
2. LintCode 170. 旋转链表 - https://www.lintcode.com/problem/rotate-list/
3. 牛客网 NC53 删除链表的倒数第 n 个节点（相关题目）

题目描述：将链表向右旋转 k 个位置
输入：1->2->3->4->5->NULL, k = 2
输出：4->5->1->2->3->NULL

解题思路：
1. 先计算链表长度
2. 将链表首尾相连形成环
3. 在合适位置断开环
时间复杂度: O(n) - 需要遍历链表
空间复杂度: O(1) - 只使用常数级别的额外空间
"""
def rotate_right(head, k):
    # 处理特殊情况
```

```
    if head is None or head.next is None or k == 0:
        return head

    # 步骤1: 计算链表长度并找到尾节点
    length = 1
    tail = head
    while tail.next is not None:
        tail = tail.next
        length += 1

    # 步骤2: 计算实际需要旋转的次数（取模操作避免多余旋转）
    k = k % length
    if k == 0:
        return head  # 不需要旋转

    # 步骤3: 将链表首尾相连形成环
    tail.next = head

    # 步骤4: 找到新的尾节点位置，距离原头节点 (length - k) 个位置
    new_tail = head
    for i in range(length - k - 1):
        new_tail = new_tail.next

    # 步骤5: 新的头节点是新尾节点的下一个节点
    new_head = new_tail.next

    # 步骤6: 断开环
    new_tail.next = None

    return new_head

"""
补充题目6: 合并两个有序链表
题目来源：
1. LeetCode 21. 合并两个有序链表 - https://leetcode.cn/problems/merge-two-sorted-lists/
2. LintCode 165. 合并两个排序链表 - https://www.lintcode.com/problem/merge-two-sorted-lists/
3. 剑指Offer 25. 合并两个排序的链表
4. 牛客网 NC33 合并两个排序的链表

题目描述：将两个升序链表合并为一个新的升序链表
输入：l1 = [1,2,4], l2 = [1,3,4]
输出：[1,1,2,3,4,4]
```

解题思路：使用迭代或递归方法，逐个比较两个链表的节点值
时间复杂度: O(n+m) - n 和 m 分别是两个链表的长度
空间复杂度: O(1) - 迭代版本，只使用常数级别的额外空间
"""
```
def merge_two_lists(l1, l2):
    # 创建虚拟头节点，简化边界情况处理
    dummy = ListNode(0)
    current = dummy

    # 迭代比较两个链表的节点值
    while l1 is not None and l2 is not None:
        if l1.val <= l2.val:
            current.next = l1
            l1 = l1.next
        else:
            current.next = l2
            l2 = l2.next
        current = current.next

    # 连接剩余部分
    current.next = l1 if l1 is not None else l2

    return dummy.next
```
"""
补充题目 7: 两两交换链表中的节点
题目来源：
1. LeetCode 24. 两两交换链表中的节点 - https://leetcode.cn/problems/swap-nodes-in-pairs/
2. LintCode 451. 两两交换链表中的节点 - https://www.lintcode.com/problem/swap-nodes-in-pairs/
3. 牛客网 NC142 链表的奇偶重排（相关题目）

题目描述：两两交换链表中的相邻节点
输入：1->2->3->4
输出：2->1->4->3

解题思路：使用虚拟头节点和迭代方法
时间复杂度: O(n) - 需要遍历链表一次
空间复杂度: O(1) - 只使用常数级别的额外空间
"""
```
def swap_pairs(head):
    # 创建虚拟头节点
    dummy = ListNode(0)
    dummy.next = head
```

```
    prev = dummy

    # 当有至少两个节点可以交换时
    while prev.next is not None and prev.next.next is not None:
        # 获取需要交换的两个节点
        first = prev.next
        second = prev.next.next

        # 执行交换操作
        first.next = second.next  # 1 -> 3
        second.next = first       # 2 -> 1
        prev.next = second        # dummy -> 2

        # 移动 prev 指针到下一对的前一个位置
        prev = first

    return dummy.next

"""
补充题目 8: 重排链表
题目来源：
1. LeetCode 143. 重排链表 - https://leetcode.cn/problems/reorder-list/
2. LintCode 99. 重排链表 - https://www.lintcode.com/problem/reorder-list/
3. 牛客网 NC40 链表相加（二）（相关题目）

题目描述：按照 L0 → Ln → L1 → Ln-1 → L2 → Ln-2 → ... 重新排列链表
输入：1->2->3->4
输出：1->4->2->3

解题思路：
1. 使用快慢指针找到中点
2. 反转后半部分链表
3. 合并两个链表
时间复杂度: O(n) - 需要遍历链表三次
空间复杂度: O(1) - 只使用常数级别的额外空间
"""
def reorder_list(head):
    if head is None or head.next is None or head.next.next is None:
        return  # 无需重排

    # 步骤 1: 使用快慢指针找到链表中点
    slow = head
    fast = head
```

```
    while fast.next is not None and fast.next.next is not None:
        slow = slow.next
        fast = fast.next.next

    # 步骤2: 反转后半部分链表
    second_half = reverse_list_iterative(slow.next)
    slow.next = None  # 断开前半部分和后半部分

    # 步骤3: 合并两个链表
    first_half = head
    while second_half is not None:
        temp1 = first_half.next
        temp2 = second_half.next

        first_half.next = second_half
        second_half.next = temp1

        first_half = temp1
        second_half = temp2

"""
补充题目9: 删除链表的倒数第N个节点
题目来源：
1. LeetCode 19. 删除链表的倒数第N个节点 - https://leetcode.cn/problems/remove-nth-node-from-end-of-list/
2. LintCode 174. 删除链表中倒数第n个节点 - https://www.lintcode.com/problem/remove-nth-node-from-end-of-list/
3. 牛客网 NC53 删除链表的倒数第n个节点
4. 剑指Offer 22. 链表中倒数第k个节点（相关题目）

题目描述：删除链表的倒数第n个节点，返回链表的头节点
输入：head = [1,2,3,4,5], n = 2
输出：[1,2,3,5]

解题思路：使用快慢指针，快指针先走n步，然后快慢指针一起走
时间复杂度: O(n) - 只需要遍历链表一次
空间复杂度: O(1) - 只使用常数级别的额外空间
"""
def remove_nth_from_end(head, n):
    # 创建虚拟头节点，简化边界情况处理
    dummy = ListNode(0)
    dummy.next = head
```

```
    # 设置快慢指针
    fast = dummy
    slow = dummy

    # 快指针先走 n+1 步
    for i in range(n + 1):
        if fast is None and i < n + 1:
            return head  # n 大于链表长度，无法删除
        fast = fast.next

    # 快慢指针一起走，直到快指针到达链表末尾
    while fast is not None:
        fast = fast.next
        slow = slow.next

    # 此时 slow 指向待删除节点的前一个节点
    slow.next = slow.next.next

    return dummy.next


def test_reverse_list():
    """测试基础链表反转"""
    print("=== 测试基础链表反转 ===")

    # 测试用例 1: [1,2,3,4,5] -> [5,4,3,2,1]
    head1 = ListNode.create_list([1, 2, 3, 4, 5])
    print("原链表:", head1)
    reversed1 = reverse_list_iterative(head1)
    print("反转后:", reversed1)

    # 测试用例 2: [1,2] -> [2,1]
    head2 = ListNode.create_list([1, 2])
    print("原链表:", head2)
    reversed2 = reverse_list_recursive(head2)
    print("反转后:", reversed2)

    # 测试用例 3: [] -> []
    head3 = None
    print("原链表:", head3)
    reversed3 = reverse_list_iterative(head3)
    print("反转后:", reversed3)
    print()
```

```
def test_reverse_list_ii():
    """测试指定区间链表反转"""
    print("=== 测试指定区间链表反转 ===")

    # 测试用例 1: [1,2,3,4,5], left=2, right=4 -> [1,4,3,2,5]
    head1 = ListNode.create_list([1, 2, 3, 4, 5])
    print("原链表:", head1)
    reversed1 = reverse_between(head1, 2, 4)
    print("反转位置 2 到 4 后:", reversed1)

    # 测试用例 2: [5], left=1, right=1 -> [5]
    head2 = ListNode.create_list([5])
    print("原链表:", head2)
    reversed2 = reverse_between(head2, 1, 1)
    print("反转位置 1 到 1 后:", reversed2)
    print()


def test_reverse_k_group():
    """测试 K 个一组反转链表"""
    print("=== 测试 K 个一组反转链表 ===")

    # 测试用例 1: [1,2,3,4,5], k=2 -> [2,1,4,3,5]
    head1 = ListNode.create_list([1, 2, 3, 4, 5])
    print("原链表:", head1)
    reversed1 = reverse_k_group(head1, 2)
    print("每 2 个一组反转后:", reversed1)

    # 测试用例 2: [1,2,3,4,5], k=3 -> [3,2,1,4,5]
    head2 = ListNode.create_list([1, 2, 3, 4, 5])
    print("原链表:", head2)
    reversed2 = reverse_k_group(head2, 3)
    print("每 3 个一组反转后:", reversed2)
    print()


def run_unit_tests():
    """运行单元测试"""
    print("=== 链表反转单元测试 ===")

    # 测试用例 1: 正常情况
```

```
    test1 = ListNode.create_list([1, 2, 3, 4, 5])
    result1 = reverse_list_iterative(test1)
    print("测试 1 - 输入[1,2,3,4,5]，期望[5,4,3,2,1]，实际:", result1)

    # 测试用例 2: 空链表
    result2 = reverse_list_iterative(None)
    print("测试 2 - 输入[]，期望[]，实际:", result2)

    # 测试用例 3: 单节点链表
    test3 = ListNode(1)
    result3 = reverse_list_iterative(test3)
    print("测试 3 - 输入[1]，期望[1]，实际:", result3.val if result3 else "null")

    # 测试用例 4: 两节点链表
    test4 = ListNode.create_list([1, 2])
    result4 = reverse_list_iterative(test4)
    print("测试 4 - 输入[1,2]，期望[2,1]，实际:", result4)

    print("单元测试完成\n")

"""
补充题目 10: 奇偶链表
题目来源:
1. LeetCode 328. 奇偶链表 - https://leetcode.cn/problems/odd-even-linked-list/
2. LintCode 1292. 奇偶链表 - https://www.lintcode.com/problem/odd-even-linked-list/
3. 牛客网 NC142 链表的奇偶重排

时间复杂度: O(n)
空间复杂度: O(1)
是否为最优解: 是
"""
def odd_even_list(head):
    if head is None or head.next is None:
        return head

    odd = head
    even = head.next
    even_head = even

    while even is not None and even.next is not None:
        odd.next = even.next
        odd = odd.next
        even.next = odd.next
```

```
        even = even.next

    odd.next = even_head
    return head


"""
补充题目 11: 分隔链表
题目来源:
1. LeetCode 86. 分隔链表 - https://leetcode.cn/problems/partition-list/
2. LintCode 96. 分隔链表 - https://www.lintcode.com/problem/partition-list/
3. 牛客网 NC188 分隔链表

时间复杂度: O(n)
空间复杂度: O(1)
是否为最优解: 是
"""
def partition(head, x):
    before_head = ListNode(0)
    before = before_head
    after_head = ListNode(0)
    after = after_head

    while head is not None:
        if head.val < x:
            before.next = head
            before = before.next
        else:
            after.next = head
            after = after.next
        head = head.next

    after.next = None
    before.next = after_head.next

    return before_head.next


"""
补充题目 12: 链表求和
题目来源:
1. LeetCode 2. 两数相加 - https://leetcode.cn/problems/add-two-numbers/
2. LeetCode 445. 两数相加 II - https://leetcode.cn/problems/add-two-numbers-ii/
```

时间复杂度: O(max(m,n))
空间复杂度: O(max(m,n))
是否为最优解: 是
"""
```
def add_two_numbers(l1, l2):
    dummy = ListNode(0)
    current = dummy
    carry = 0

    while l1 is not None or l2 is not None or carry != 0:
        val1 = l1.val if l1 is not None else 0
        val2 = l2.val if l2 is not None else 0

        sum_val = val1 + val2 + carry
        carry = sum_val // 10

        current.next = ListNode(sum_val % 10)
        current = current.next

        if l1 is not None:
            l1 = l1.next
        if l2 is not None:
            l2 = l2.next

    return dummy.next


"""
```
补充题目 13: 环形链表
题目来源:
1. LeetCode 141. 环形链表 - https://leetcode.cn/problems/linked-list-cycle/
2. LeetCode 142. 环形链表 II - https://leetcode.cn/problems/linked-list-cycle-ii/

时间复杂度: O(n)
空间复杂度: O(1)
是否为最优解: 是
"""
```
def has_cycle(head):
    if head is None or head.next is None:
        return False

    slow = head
```

```
    fast = head

    while fast is not None and fast.next is not None:
        slow = slow.next
        fast = fast.next.next

        if slow == fast:
            return True

    return False


def detect_cycle(head):
    if head is None or head.next is None:
        return None

    slow = head
    fast = head
    has_cycle_flag = False

    while fast is not None and fast.next is not None:
        slow = slow.next
        fast = fast.next.next

        if slow == fast:
            has_cycle_flag = True
            break

    if not has_cycle_flag:
        return None

    slow = head
    while slow != fast:
        slow = slow.next
        fast = fast.next

    return slow


"""
补充题目 14: 相交链表
题目来源:
1. LeetCode 160. 相交链表 - https://leetcode.cn/problems/intersection-of-two-linked-lists/
```

2. LintCode 380. 相交链表 - https://www.lintcode.com/problem/intersection-of-two-linked-lists/

时间复杂度: O(m+n)
空间复杂度: O(1)
是否为最优解: 是
"""
```
def get_intersection_node(headA, headB):
    if headA is None or headB is None:
        return None

    pA = headA
    pB = headB

    while pA != pB:
        pA = headB if pA is None else pA.next
        pB = headA if pB is None else pB.next

    return pA
```

"""
补充题目 15: 排序链表
题目来源:
1. LeetCode 148. 排序链表 - https://leetcode.cn/problems/sort-list/
2. LintCode 98. 排序链表 - https://www.lintcode.com/problem/sort-list/

时间复杂度: O(n log n)
空间复杂度: O(log n)
是否为最优解: 是
"""
```
def sort_list(head):
    if head is None or head.next is None:
        return head

    slow = head
    fast = head.next

    while fast is not None and fast.next is not None:
        slow = slow.next
        fast = fast.next.next

    mid = slow.next
    slow.next = None
```

```
    left = sort_list(head)
    right = sort_list(mid)

    return merge_two_lists(left, right)


"""
补充题目 16: 链表随机节点
题目来源:
1. LeetCode 382. 链表随机节点 - https://leetcode.cn/problems/linked-list-random-node/
2. 蓄水池抽样算法应用

时间复杂度: O(n)
空间复杂度: O(1)
是否为最优解: 是
"""
import random

class RandomNodeSelector:
    def __init__(self, head):
        self.head = head

    def get_random(self):
        current = self.head
        result = 0
        count = 0

        while current is not None:
            count += 1
            # 以 1/count 的概率选择当前节点
            if random.randint(1, count) == 1:
                result = current.val
            current = current.next

        return result


"""
补充题目 17: 复制带随机指针的链表
题目来源:
1. LeetCode 138. 复制带随机指针的链表 - https://leetcode.cn/problems/copy-list-with-random-pointer/
```

2. 剑指 Offer 35. 复杂链表的复制

时间复杂度: O(n)
空间复杂度: O(1)
是否为最优解: 是
"""

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

def copy_random_list(head):
    if head is None:
        return None

    # 第一次遍历：在每个节点后面插入复制节点
    current = head
    while current is not None:
        copy_node = NodeWithRandom(current.val)
        copy_node.next = current.next
        current.next = copy_node
        current = copy_node.next

    # 第二次遍历：设置复制节点的随机指针
    current = head
    while current is not None:
        if current.random is not None:
            current.next.random = current.random.next
        current = current.next.next

    # 第三次遍历：分离原链表和复制链表
    current = head
    copy_head = head.next
    copy_current = copy_head

    while current is not None:
        current.next = current.next.next
        if copy_current.next is not None:
            copy_current.next = copy_current.next.next
        current = current.next
        copy_current = copy_current.next
```

```
    return copy_head


"""
补充题目 18: 链表组件
题目来源:
1. LeetCode 817. 链表组件 - https://leetcode.cn/problems/linked-list-components/

时间复杂度: O(n + m)
空间复杂度: O(m)
是否为最优解: 是
"""
def num_components(head, nums):
    num_set = set(nums)

    components = 0
    in_component = False
    current = head

    while current is not None:
        if current.val in num_set:
            if not in_component:
                components += 1
                in_component = True
        else:
            in_component = False
        current = current.next

    return components


"""
补充题目 19: 链表中的下一个更大节点
题目来源:
1. LeetCode 1019. 链表中的下一个更大节点 - https://leetcode.cn/problems/next-greater-node-in-linked-list/

时间复杂度: O(n)
空间复杂度: O(n)
是否为最优解: 是
"""
def next_larger_nodes(head):
    # 将链表转换为数组
```

```
    values = []
    current = head
    while current is not None:
        values.append(current.val)
        current = current.next

    n = len(values)
    result = [0] * n
    stack = []

    # 从右向左遍历，使用单调栈
    for i in range(n - 1, -1, -1):
        current_val = values[i]

        # 弹出栈顶比当前值小的元素
        while stack and stack[-1] <= current_val:
            stack.pop()

        # 如果栈不为空，栈顶就是下一个更大节点
        result[i] = stack[-1] if stack else 0

        # 将当前值压入栈
        stack.append(current_val)

    return result


"""
补充题目 20: 链表最大孪生和
题目来源:
1. LeetCode 2130. 链表最大孪生和 - https://leetcode.cn/problems/maximum-twin-sum-of-a-linked-list/

时间复杂度: O(n)
空间复杂度: O(1)
是否为最优解: 是
"""
def pair_sum(head):
    # 使用快慢指针找到中点
    slow = head
    fast = head
    while fast is not None and fast.next is not None:
        slow = slow.next
```

```
        fast = fast.next.next

    # 反转后半部分链表
    second_half = reverse_list_iterative(slow)

    # 计算孪生和的最大值
    max_sum = 0
    first_half = head
    while second_half is not None:
        max_sum = max(max_sum, first_half.val + second_half.val)
        first_half = first_half.next
        second_half = second_half.next

    return max_sum


"""
补充题目 21: 合并 K 个升序链表
题目来源:
1. LeetCode 23. 合并 K 个升序链表 - https://leetcode.cn/problems/merge-k-sorted-lists/
2. LintCode 104. 合并 k 个排序链表 - https://www.lintcode.com/problem/merge-k-sorted-lists/
3. 牛客网 NC127. 合并 k 个已排序的链表

时间复杂度: O(N log K)
空间复杂度: O(K)
是否为最优解: 是
"""
import heapq

def merge_k_lists(lists):
    # 使用最小堆
    heap = []

    # 将所有非空链表的头节点加入最小堆
    for i, lst in enumerate(lists):
        if lst:
            heapq.heappush(heap, (lst.val, i))

    # 创建虚拟头节点
    dummy = ListNode(0)
    current = dummy

    # 从堆中取出最小节点，加入结果链表
```

```
    while heap:
        val, index = heapq.heappop(heap)  # 取出最小节点
        current.next = lists[index]       # 加入结果链表
        current = current.next            # 移动指针
        lists[index] = lists[index].next  # 移动原链表指针

        # 将取出节点的下一个节点加入堆中(如果不为空)
        if lists[index]:
            heapq.heappush(heap, (lists[index].val, index))

    return dummy.next


"""
补充题目 22: 删除链表中的节点
题目来源:
1. LeetCode 237. 删除链表中的节点 - https://leetcode.cn/problems/delete-node-in-a-linked-list/
2. LintCode 37. 删除链表中的节点 - https://www.lintcode.com/problem/delete-node-in-a-linked-list/
3. 牛客网 NC138. 删除链表的节点

时间复杂度: O(1)
空间复杂度: O(1)
是否为最优解: 是
"""
def delete_node(node):
    # 将下一个节点的值复制到当前节点
    node.val = node.next.val

    # 跳过下一个节点
    node.next = node.next.next


"""
补充题目 23: 删除排序链表中的重复元素
题目来源:
1. LeetCode 83. 删除排序链表中的重复元素 - https://leetcode.cn/problems/remove-duplicates-from-sorted-list/
2. LintCode 112. 删除排序链表中的重复元素 - https://www.lintcode.com/problem/remove-duplicates-from-sorted-list/
3. 牛客网 NC141. 判断一个链表是否为回文结构

时间复杂度: O(n)
空间复杂度: O(1)
```

是否为最优解: 是
"""
```
def delete_duplicates(head):
    # 边界情况处理
    if head is None or head.next is None:
        return head

    current = head

    # 遍历链表
    while current.next is not None:
        # 如果当前节点值等于下一个节点值，跳过下一个节点
        if current.val == current.next.val:
            current.next = current.next.next
        else:
            # 只有当下一个节点不被删除时，才移动 current 指针
            current = current.next

    return head


"""
```
补充题目 24: 删除排序链表中的重复元素 II
题目来源:
1. LeetCode 82. 删除排序链表中的重复元素 II - https://leetcode.cn/problems/remove-duplicates-from-sorted-list-ii/
2. LintCode 113. 删除排序链表中的重复元素 II - https://www.lintcode.com/problem/remove-duplicates-from-sorted-list-ii/
3. 牛客网 NC140

时间复杂度: O(n)
空间复杂度: O(1)
是否为最优解: 是
"""
```
def delete_duplicates_ii(head):
    # 创建虚拟头节点，简化边界处理
    dummy = ListNode(0)
    dummy.next = head

    # prev 指向已处理部分的最后一个节点
    prev = dummy
    # current 指向当前待处理节点
    current = head
```

```
    while current is not None:
        # 检查是否有重复节点
        if current.next is not None and current.val == current.next.val:
            # 记录重复值
            duplicate_value = current.val

            # 跳过所有重复节点
            while current is not None and current.val == duplicate_value:
                current = current.next

            # 连接 prev 和 current
            prev.next = current
        else:
            # 没有重复，正常移动指针
            prev = current
            current = current.next

    return dummy.next


"""
补充题目 25: 移除链表元素
题目来源:
1. LeetCode 203. 移除链表元素 - https://leetcode.cn/problems/remove-linked-list-elements/
2. 牛客网相关题目

时间复杂度: O(n)
空间复杂度: O(1)
是否为最优解: 是
"""
def remove_elements(head, val):
    # 创建虚拟头节点，简化删除头节点的情况
    dummy = ListNode(0)
    dummy.next = head

    # prev 指向已处理部分的最后一个节点
    prev = dummy
    # current 指向当前待处理节点
    current = head

    while current is not None:
        if current.val == val:
```

```
            # 删除当前节点
            prev.next = current.next
        else:
            # 移动 prev 指针
            prev = current
        # 移动 current 指针
        current = current.next

    return dummy.next


"""
性能分析工具类
"""
import time

class LinkedListProfiler:
    def __init__(self):
        self.start_time = 0
        self.end_time = 0

    def start(self):
        self.start_time = time.time_ns()

    def end(self):
        self.end_time = time.time_ns()
        duration = self.end_time - self.start_time
        print(f"执行时间: {duration} 纳秒")


"""
调试工具类
"""
class LinkedListDebugger:
    @staticmethod
    def print_list_state(head, message):
        print(f"{message}: {head}")

    @staticmethod
    def assert_list(condition, message):
        if not condition:
            raise AssertionError(f"链表断言失败: {message}")
```

```
    @staticmethod
    def verify_no_cycle(head):
        if head is None:
            return True

        slow = head
        fast = head

        while fast is not None and fast.next is not None:
            slow = slow.next
            fast = fast.next.next
            if slow == fast:
                return False  # 有环
        return True  # 无环


"""
测试补充题目
"""
def test_merge_k_lists():
    print("=== 测试合并 K 个升序链表 ===")

    # 创建测试用例: lists = [[1,4,5],[1,3,4],[2,6]]
    list1 = ListNode.create_list([1, 4, 5])
    list2 = ListNode.create_list([1, 3, 4])
    list3 = ListNode.create_list([2, 6])
    lists = [list1, list2, list3]

    print("输入链表数组: ", end="")
    for i, lst in enumerate(lists):
        print(f"{'' if i == 0 else ', '}{lst if lst else '[]'}", end="")
    print()

    result = merge_k_lists(lists)
    print(f"合并后: {result if result else '[]'}")
    print()


def test_delete_node():
    print("=== 测试删除链表中的节点 ===")

    # 创建测试用例: [4,5,1,9], 删除节点 5
    head = ListNode.create_list([4, 5, 1, 9])
```

```
    print(f"原链表: {head}")

    # 找到要删除的节点(值为 5 的节点)
    node_to_delete = head.next  # 值为 5 的节点

    print(f"删除节点: {node_to_delete.val}")
    delete_node(node_to_delete)
    print(f"删除后: {head}")
    print()


def test_delete_duplicates():
    print("=== 测试删除排序链表中的重复元素 ===")

    # 测试用例 1: [1,1,2] -> [1,2]
    head1 = ListNode.create_list([1, 1, 2])
    print(f"原链表: {head1}")
    result1 = delete_duplicates(head1)
    print(f"去重后: {result1}")

    # 测试用例 2: [1,1,2,3,3] -> [1,2,3]
    head2 = ListNode.create_list([1, 1, 2, 3, 3])
    print(f"原链表: {head2}")
    result2 = delete_duplicates(head2)
    print(f"去重后: {result2}")
    print()


def test_delete_duplicates_ii():
    print("=== 测试删除排序链表中的重复元素 II ===")

    # 测试用例 1: [1,2,3,3,4,4,5] -> [1,2,5]
    head1 = ListNode.create_list([1, 2, 3, 3, 4, 4, 5])
    print(f"原链表: {head1}")
    result1 = delete_duplicates_ii(head1)
    print(f"删除重复元素后: {result1}")

    # 测试用例 2: [1,1,1,2,3] -> [2,3]
    head2 = ListNode.create_list([1, 1, 1, 2, 3])
    print(f"原链表: {head2}")
    result2 = delete_duplicates_ii(head2)
    print(f"删除重复元素后: {result2}")
    print()
```

```
def test_remove_elements():
    print("=== 测试移除链表元素 ===")

    # 测试用例: [1,2,6,3,4,5,6], val = 6 -> [1,2,3,4,5]
    head = ListNode.create_list([1, 2, 6, 3, 4, 5, 6])
    val = 6
    print(f"原链表: {head}")
    print(f"移除元素: {val}")
    result = remove_elements(head, val)
    print(f"移除后: {result}")
    print()


def test_additional_problems():
    print("=== 补充题目测试 ===")

    # 测试链表随机节点
    random_test = ListNode.create_list([1, 2, 3, 4, 5])
    selector = RandomNodeSelector(random_test)
    print(f"随机节点选择测试: {selector.get_random()}")

    # 测试链表组件
    component_test = ListNode.create_list([0, 1, 2, 3])
    nums = [0, 1, 3]
    print(f"链表组件个数: {num_components(component_test, nums)}")

    # 测试下一个更大节点
    larger_test = ListNode.create_list([2, 1, 5])
    larger_result = next_larger_nodes(larger_test)
    print(f"下一个更大节点: {larger_result}")

    # 测试链表最大孪生和
    twin_test = ListNode.create_list([5, 4, 2, 1])
    print(f"链表最大孪生和: {pair_sum(twin_test)}")

    print("补充题目测试完成\n")


# 主函数
if __name__ == "__main__":
    # 设置随机种子
```

```
    random.seed(42)

    # 运行测试
    test_reverse_list()
    test_reverse_list_ii()
    test_reverse_k_group()
    run_unit_tests()
    test_additional_problems()

    # 运行补充题目的测试
    test_merge_k_lists()
    test_delete_node()
    test_delete_duplicates()
    test_delete_duplicates_ii()
    test_remove_elements()

    # 性能分析示例
    profiler = LinkedListProfiler()
    perf_test = ListNode.create_list([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

    profiler.start()
    reversed_list = reverse_list_iterative(perf_test)
    profiler.end()

    print("\n=== 所有测试完成 ===")

==============================================
```