==========
文件夹: class018_MergeSort
==========

[Markdown 文件]
==========
文件: MERGE_SORT_PROBLEMS.md
==========
归并排序经典题目汇总

一、LeetCode 系列题目

基础排序题目
1. **LeetCode 912. 排序数组**
   - 题目链接: https://leetcode.cn/problems/sort-an-array/
   - 题目描述: 给定一个整数数组 nums，将该数组升序排列
   - 解法: 直接应用归并排序模板
   - 时间复杂度: O(n log n)
   - 空间复杂度: O(n)
   - 是否最优: 是 - 基于比较的排序算法下界

2. **LeetCode 148. 排序链表**
   - 题目链接: https://leetcode.cn/problems/sort-list/
   - 题目描述: 给你链表的头结点 head，请将其按升序排列并返回排序后的链表
   - 解法: 链表归并排序，使用快慢指针找中点
   - 时间复杂度: O(n log n)
   - 空间复杂度: O(log n) - 递归调用栈
   - 是否最优: 是 - 链表排序的最佳选择

3. **LeetCode 23. 合并 K 个升序链表**
   - 题目链接: https://leetcode.cn/problems/merge-k-sorted-lists/
   - 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
   - 解法: 分治合并，类似归并排序
   - 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(log k) - 递归调用栈
   - 是否最优: 是 - 与优先队列方法时间复杂度相同

4. **LeetCode 88. 合并两个有序数组**
   - 题目链接: https://leetcode.cn/problems/merge-sorted-array/
   - 题目描述: 合并 nums2 到 nums1 中，使合并后的数组有序
   - 解法: 从后往前合并，避免覆盖 nums1 的有效数据
   - 时间复杂度: O(m + n)
   - 空间复杂度: O(1) - 原地修改

- 是否最优: 是 - 时间、空间都最优

5. **LeetCode 21. 合并两个有序链表**
   - 题目链接: https://leetcode.cn/problems/merge-two-sorted-lists/
   - 题目描述: 将两个升序链表合并为一个新的升序链表并返回
   - 解法: 双指针合并
   - 时间复杂度: O(m + n)
   - 空间复杂度: O(1)
   - 是否最优: 是

逆序对相关题目
6. **LeetCode 315. 计算右侧小于当前元素的个数**
   - 题目链接: https://leetcode.cn/problems/count-of-smaller-numbers-after-self/
   - 题目描述: 统计每个元素右侧比它小的元素个数
   - 解法: 归并排序 + 索引数组，在合并过程中利用有序性高效统计
   - 时间复杂度: O(n log n)
   - 空间复杂度: O(n)
   - 是否最优: 是

7. **LeetCode 493. 翻转对**
   - 题目链接: https://leetcode.cn/problems/reverse-pairs/
   - 题目描述: 统计满足 nums[i] > 2*nums[j] 且 i < j 的对数
   - 解法: 归并排序，在合并前先统计翻转对
   - 时间复杂度: O(n log n)
   - 空间复杂度: O(n)
   - 是否最优: 是

8. **LeetCode 327. 区间和的个数**
   - 题目链接: https://leetcode.cn/problems/count-of-range-sum/
   - 题目描述: 统计区间和在[lower, upper]范围内的子数组个数
   - 解法: 前缀和数组 + 归并排序
   - 时间复杂度: O(n log n)
   - 空间复杂度: O(n)
   - 是否最优: 是

9. **LeetCode LCR 170. 交易逆序对的总数** (剑指 Offer 51)
   - 题目链接: https://leetcode.cn/problems/shu-zu-zhong-de-ni-xu-dui-lcof/
   - 题目描述: 计算数组中的逆序对总数
   - 解法: 归并排序过程中统计逆序对
   - 时间复杂度: O(n log n)
   - 空间复杂度: O(n)
   - 是否最优: 是

10. **LeetCode 2426. 满足不等式的数对数目**
    - 题目链接: https://leetcode.cn/problems/number-of-pairs-satisfying-inequality/
    - 题目描述: 统计满足不等式的数对数目
    - 解法: 翻转对变种，处理不等式条件
    - 时间复杂度: O(n log n)
    - 空间复杂度: O(n)
    - 是否最优: 是

11. **LeetCode 4. 寻找两个正序数组的中位数**
    - 题目链接: https://leetcode.cn/problems/median-of-two-sorted-arrays/
    - 题目描述: 寻找两个正序数组的中位数
    - 解法: 二分查找，不是归并排序但涉及有序数组合并思想
    - 时间复杂度: O(log(min(m, n)))
    - 空间复杂度: O(1)
    - 是否最优: 是

其他相关题目
12. **LeetCode 1508. 子数组和排序后的区间和**
    - 题目链接: https://leetcode.cn/problems/range-sum-of-sorted-subarray-sums/
    - 题目描述: 子数组和排序后的区间和
    - 解法: 结合前缀和与归并排序思想
    - 时间复杂度: O(n^2 log n)
    - 空间复杂度: O(n^2)
    - 是否最优: 是

13. **LeetCode 1649. 通过指令创建有序数组**
    - 题目链接: https://leetcode.cn/problems/create-sorted-array-through-instructions/
    - 题目描述: 通过指令创建有序数组
    - 解法: 动态统计逆序对，可使用树状数组、线段树或分治
    - 时间复杂度: O(n log n)
    - 空间复杂度: O(n)
    - 是否最优: 是

二、洛谷系列题目

14. **洛谷 P1177. 【模板】快速排序**
    - 题目链接: https://www.luogu.com.cn/problem/P1177
    - 题目描述: 快速排序模板题（虽然题目是快速排序，但可以用归并排序通过）
    - 解法: 归并排序
    - 时间复杂度: O(n log n)
    - 空间复杂度: O(n)
    - 是否最优: 是

15. **洛谷 P1908. 逆序对**
    - 题目链接: https://www.luogu.com.cn/problem/P1908
    - 题目描述: 计算数组中的逆序对数量
    - 解法: 归并排序统计逆序对
    - 时间复杂度: O(n log n)
    - 空间复杂度: O(n)
    - 是否最优: 是

16. **洛谷 P1774. 最接近神的人**
    - 题目链接: https://www.luogu.com.cn/problem/P1774
    - 题目描述: 逆序对问题的变种
    - 解法: 逆序对统计
    - 时间复杂度: O(n log n)
    - 空间复杂度: O(n)
    - 是否最优: 是

17. **洛谷 P1966. [NOIP2013 提高组] 火柴排队**
    - 题目链接: https://www.luogu.com.cn/problem/P1966
    - 题目描述: 逆序对应用
    - 解法: 逆序对、离散化
    - 时间复杂度: O(n log n)
    - 空间复杂度: O(n)
    - 是否最优: 是

三、牛客网系列题目

18. **牛客 NC119. 最小的K个数**
    - 题目链接: https://www.nowcoder.com/practice/6a296eb82cf844ca8539b57c23e6e9bf
    - 题目描述: 寻找最小的K个数
    - 解法: 虽不是直接归并排序，但涉及排序思想，可使用堆或分治
    - 时间复杂度: O(n log k)
    - 空间复杂度: O(k)
    - 是否最优: 是

19. **牛客 NC37. 合并区间**
    - 题目链接: https://www.nowcoder.com/practice/65cfde9e5b9b4cf2b6bafa5f3ef33fa6
    - 题目描述: 合并区间
    - 解法: 区间合并的归并思想
    - 时间复杂度: O(n log n)
    - 空间复杂度: O(1)
    - 是否最优: 是

20. **牛客 NC51. 合并k个已排序的链表**

- 题目链接: https://www.nowcoder.com/practice/65cfde9e5b9b4cf2b6bafa5f3ef33fa6
- 题目描述: 合并 k 个已排序的链表
- 解法: 链表归并排序应用
- 时间复杂度: O(N log k)
- 空间复杂度: O(log k)
- 是否最优: 是

四、AcWing 系列题目

21. **AcWing 787. 归并排序**
- 题目链接: https://www.acwing.com/problem/content/789/
- 题目描述: 基础归并排序模板
- 解法: 归并排序
- 时间复杂度: O(n log n)
- 空间复杂度: O(n)
- 是否最优: 是

22. **AcWing 788. 逆序对的数量**
- 题目链接: https://www.acwing.com/problem/content/790/
- 题目描述: 经典逆序对问题
- 解法: 归并排序
- 时间复杂度: O(n log n)
- 空间复杂度: O(n)
- 是否最优: 是

23. **AcWing 107. 超快速排序**
- 题目链接: https://www.acwing.com/problem/content/109/
- 题目描述: 逆序对问题变种
- 解法: 逆序对统计
- 时间复杂度: O(n log n)
- 空间复杂度: O(n)
- 是否最优: 是

五、国际知名平台系列

24. **POJ 2299. Ultra-QuickSort**
- 题目链接: http://poj.org/problem?id=2299
- 题目描述: 经典逆序对问题，国际知名
- 解法: 归并排序统计逆序对
- 时间复杂度: O(n log n)
- 空间复杂度: O(n)
- 是否最优: 是

25. **HDU 1394. Minimum Inversion Number**
    - 题目链接: http://acm.hdu.edu.cn/showproblem.php?pid=1394
    - 题目描述: 循环移位中的逆序对
    - 解法: 逆序对应用
    - 时间复杂度: O(n log n)
    - 空间复杂度: O(n)
    - 是否最优: 是

26. **Codeforces 558E. A Simple Task**
    - 题目链接: https://codeforces.com/problemset/problem/558/E
    - 题目描述: 字符串排序与归并思想
    - 解法: 线段树、排序
    - 时间复杂度: O(n log n)
    - 空间复杂度: O(n)
    - 是否最优: 是

27. **HackerRank Merge Sort: Counting Inversions**
    - 题目链接: https://www.hackerrank.com/challenges/ctci-merge-sort/problem
    - 题目描述: 企业面试常见题
    - 解法: 逆序对统计
    - 时间复杂度: O(n log n)
    - 空间复杂度: O(n)
    - 是否最优: 是

28. **SPOJ INVCNT. Inversion Count**
    - 题目链接: https://www.spoj.com/problems/INVCNT/
    - 题目描述: 专门测试逆序对
    - 解法: 归并排序
    - 时间复杂度: O(n log n)
    - 空间复杂度: O(n)
    - 是否最优: 是

六、其他国内平台

29. **杭电OJ 1394. Minimum Inversion Number**
    - 题目链接: http://acm.hdu.edu.cn/showproblem.php?pid=1394
    - 题目描述: 经典题目，多校赛常见
    - 解法: 逆序对统计
    - 时间复杂度: O(n log n)
    - 空间复杂度: O(n)
    - 是否最优: 是

30. **赛码网 归并排序相关题目**

- 平台：https://www.acmcoder.com/
- 题目描述：企业笔试常见
- 解法：归并排序及其变种
- 是否最优：是

31. **计蒜客 排序相关题目**
- 平台：https://www.jisuanke.com/
- 题目描述：算法竞赛培训
- 解法：归并排序及其变种
- 是否最优：是

32. **MarsCode 算法题库**
- 平台：https://www.marscode.cn/
- 题目描述：新兴算法平台
- 解法：归并排序及其变种
- 是否最优：是

七、特殊类型题目

33. **外部排序相关题目**
- 题目描述：处理超大规模数据
- 解法：多路归并
- 应用场景：数据库排序、大数据处理
- 是否最优：是

34. **多路归并题目**
- 题目描述：合并多个有序序列
- 解法：分治合并
- 应用场景：大数据处理
- 是否最优：是

35. **并行归并排序题目**
- 题目描述：多线程优化
- 解法：并行化处理
- 应用场景：高性能计算
- 是否最优：是

36. **稳定排序应用题目**
- 题目描述：需要保持相等元素相对顺序
- 解法：归并排序
- 应用场景：数据库查询、金融系统
- 是否最优：是

八、综合训练题目（按难度分级）

入门级（掌握基础）
37. **基础归并排序实现**
38. **逆序对统计基础版**

进阶级（理解应用）
39. **链表归并排序**
40. **合并 K 个有序序列**
41. **区间和统计**

高手级（深入掌握）
42. **翻转对统计**
43. **复杂条件逆序对**
44. **外部排序实现**

专家级（工程应用）
45. **并行归并排序优化**
46. **稳定排序系统设计**
47. **大规模数据处理**

九、题目分类总结

类别	题目数量	核心算法	应用场景
基础排序	15+	归并排序模板	算法学习、面试基础
逆序对统计	20+	归并排序变种	数学统计、金融分析
链表排序	5+	链表归并排序	数据结构应用
多路合并	8+	分治合并	大数据处理
区间统计	6+	前缀和+归并	数据分析
工程优化	10+	各种优化技巧	实际系统开发

十、训练建议

1. **基础阶段**: 先掌握归并排序的递归和非递归实现
2. **应用阶段**: 练习逆序对、链表排序等变种问题
3. **提高阶段**: 解决复杂条件的统计问题
4. **实战阶段**: 参与在线评测，检验掌握程度

十一、学习资源推荐

1. **书籍**: 《算法导论》、《编程珠玑》
2. **视频**: 各大 MOOC 平台的算法课程

3. **练习**: LeetCode、洛谷、AcWing 等平台
4. **社区**: Stack Overflow、GitHub 开源项目

通过系统学习以上题目，可以全面掌握归并排序及其应用，为算法竞赛和面试打下坚实基础。

==========

文件: README.md
==========
归并排序专题详解

归并排序是一种基于分治思想的稳定排序算法，采用分而治之的策略，将数组不断分割成更小的子数组，然后将这些子数组合并成有序数组。

算法原理

归并排序的基本思想是：
1. **分解**：将待排序数组递归地分成两个子数组，直到每个子数组只有一个元素
2. **解决**：递归地对子数组进行排序
3. **合并**：将两个已排序的子数组合并成一个有序数组

算法复杂度

- **时间复杂度**：O(n log n) - 在所有情况下（最好、最坏、平均）都是这个复杂度
- **空间复杂度**：O(n) - 需要额外的数组空间来存储临时数据

稳定性

归并排序是一种稳定的排序算法，相等元素的相对位置在排序后不会改变。

核心实现

Java 版本

```java
// 合并函数
public static void merge(int[] arr, int l, int m, int r) {
    int i = l;
    int a = l;
    int b = m + 1;
    while (a <= m && b <= r) {
        help[i++] = arr[a] <= arr[b] ? arr[a++] : arr[b++];
    }
```

```
    while (a <= m) {
        help[i++] = arr[a++];
    }
    while (b <= r) {
        help[i++] = arr[b++];
    }
    for (i = l; i <= r; i++) {
        arr[i] = help[i];
    }
}

// 递归版本
public static void mergeSort1(int[] arr, int l, int r) {
    if (l == r) {
        return;
    }
    int m = (l + r) / 2;
    mergeSort1(arr, l, m);
    mergeSort1(arr, m + 1, r);
    merge(arr, l, m, r);
}

// 非递归版本
public static void mergeSort2(int[] arr) {
    int n = arr.length;
    for (int l, m, r, step = 1; step < n; step <<= 1) {
        l = 0;
        while (l < n) {
            m = l + step - 1;
            if (m + 1 >= n) {
                break;
            }
            r = Math.min(l + (step << 1) - 1, n - 1);
            merge(arr, l, m, r);
            l = r + 1;
        }
    }
}
```

C++版本

```cpp
```

```
// 合并函数
void merge(int l, int m, int r) {
    int i = l;
    int a = l;
    int b = m + 1;
    while (a <= m && b <= r) {
        help[i++] = arr[a] <= arr[b] ? arr[a++] : arr[b++];
    }
    while (a <= m) {
        help[i++] = arr[a++];
    }
    while (b <= r) {
        help[i++] = arr[b++];
    }
    for (i = l; i <= r; i++) {
        arr[i] = help[i];
    }
}

// 递归版本
void mergeSort1(int l, int r) {
    if (l == r) {
        return;
    }
    int m = (l + r) / 2;
    mergeSort1(l, m);
    mergeSort1(m + 1, r);
    merge(l, m, r);
}

// 非递归版本
void mergeSort2() {
    for (int l, m, r, step = 1; step < n; step <<= 1) {
        l = 0;
        while (l < n) {
            m = l + step - 1;
            if (m + 1 >= n) {
                break;
            }
            r = min(l + (step << 1) - 1, n - 1);
            merge(l, m, r);
            l = r + 1;
        }
```

```
    }
}
```

Python 版本

```python
# 合并函数
def merge(l: int, m: int, r: int) -> None:
    i = l
    a = l
    b = m + 1
    while a <= m and b <= r:
        if arr[a] <= arr[b]:
            help_arr[i] = arr[a]
            a += 1
        else:
            help_arr[i] = arr[b]
            b += 1
        i += 1

    while a <= m:
        help_arr[i] = arr[a]
        a += 1
        i += 1

    while b <= r:
        help_arr[i] = arr[b]
        b += 1
        i += 1

    for i in range(l, r + 1):
        arr[i] = help_arr[i]

# 递归版本
def merge_sort1(l: int, r: int) -> None:
    if l == r:
        return
    m = (l + r) // 2
    merge_sort1(l, m)
    merge_sort1(m + 1, r)
    merge(l, m, r)
```

```
# 非递归版本
def merge_sort2() -> None:
    step = 1
    while step < n:
        l = 0
        while l < n:
            m = l + step - 1
            if m + 1 >= n:
                break
            r = min(l + (step << 1) - 1, n - 1)
            merge(l, m, r)
            l = r + 1
        step <<= 1
```

归并排序算法深度解析

算法核心思想与数学原理

归并排序基于分治策略，其数学原理可以表示为：
```
T(n) = 2T(n/2) + O(n)
```
根据主定理(Master Theorem)，该递归式的时间复杂度为O(n log n)。

关键数学特性：
- 递归深度：$\log_2 n$
- 每层合并操作：O(n)
- 总操作数：$n \times \log_2 n$

详细题目解析与多语言实现

1. LeetCode 912. 排序数组
题目链接：https://leetcode.cn/problems/sort-an-array/
题目描述：给定一个整数数组 nums，将该数组升序排列。

解题思路：
- 直接应用归并排序模板
- 处理边界情况：空数组、单元素数组
- 选择递归或非递归版本

复杂度分析：
- 时间复杂度：O(n log n) - 最优解

- 空间复杂度：O(n) - 辅助数组
- 稳定性：稳定排序

Java 实现关键点：
```java
// 递归版本：代码简洁但可能栈溢出
// 非递归版本：更安全，适合大数据量
```

C++实现关键点：
```cpp
// 使用全局数组避免栈溢出
// 使用 ios::sync_with_stdio(false)加速 IO
```

Python 实现关键点：
```python
// 使用切片操作简化代码
// 注意递归深度限制，大数据量使用非递归版本
```

2. LeetCode 148. 排序链表
题目链接：https://leetcode.cn/problems/sort-list/
题目描述：给你链表的头结点 head，请将其按升序排列并返回排序后的链表。

解题思路：
1. **快慢指针找中点**：龟兔赛跑算法
2. **递归分割**：将链表分为左右两部分
3. **合并有序链表**：双指针合并

复杂度分析：
- 时间复杂度：O(n log n)
- 空间复杂度：O(log n) - 递归调用栈

链表排序的特殊性：
- 归并排序是链表排序的最佳选择
- 快速排序需要随机访问，链表不支持
- 堆排序也需要随机访问

工程化考量：
- 哑节点(dummy node)简化边界处理
- 非递归版本可降低空间复杂度到 O(1)
- 注意链表断开和连接的正确性

3. LeetCode 23. 合并 K 个升序链表
题目链接：https://leetcode.cn/problems/merge-k-sorted-lists/
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。

解法对比分析：

方法	时间复杂度	空间复杂度	优缺点
顺序合并	O(kN)	O(1)	简单但效率低
优先队列	O(N log k)	O(k)	需要堆数据结构
分治合并	O(N log k)	O(log k)	**最优解**

分治合并的核心思想：
```
mergeKLists(lists) = merge(mergeKLists(left), mergeKLists(right))
```

4. LeetCode 88. 合并两个有序数组
题目链接：https://leetcode.cn/problems/merge-sorted-array/
题目描述：合并 nums2 到 nums1 中，使合并后的数组有序。

关键技巧：**从后往前合并**
- 避免覆盖 nums1 的有效数据
- 直接利用 nums1 尾部的空位

复杂度分析：
- 时间复杂度：O(m + n)
- 空间复杂度：O(1) - 原地修改

边界情况处理：
- nums2 为空：直接返回
- nums1 有效数据为 0：直接拷贝 nums2
- 重复元素处理：保持稳定性

5. LeetCode 315. 计算右侧小于当前元素的个数
题目链接：https://leetcode.cn/problems/count-of-smaller-numbers-after-self/
题目描述：统计每个元素右侧比它小的元素个数。

算法核心：归并排序 + 索引数组
1. **索引数组**：记录原始位置
2. **合并时统计**：利用有序性高效统计
3. **关键逻辑**：左侧元素出队时统计右侧已处理的较小元素

统计时机分析：
```
当 helper[p1] <= helper[p2] 时：
    counts[原始位置] += (p2 - (mid + 1))
```

6. LeetCode 493. 翻转对
题目链接：https://leetcode.cn/problems/reverse-pairs/
题目描述：统计满足 nums[i] > 2*nums[j] 且 i < j 的对数。

与 315 题的区别：
- 315 题：统计右侧比自己小的元素
- 493 题：统计满足特定倍数关系的翻转对

关键优化：
1. **先统计后合并**：与 315 题顺序不同
2. **双指针技巧**：利用有序性，j 指针不重置
3. **防溢出**：使用 long 类型处理大数

7. LeetCode 327. 区间和的个数
题目链接：https://leetcode.cn/problems/count-of-range-sum/
题目描述：统计区间和在[lower, upper]范围内的子数组个数。

算法转换：
1. **前缀和数组**：将问题转化为统计前缀和差值
2. **归并排序应用**：在有序前缀和数组中统计满足条件的区间

数学原理：
```
prefix[j] - prefix[i] ∈ [lower, upper]
⇒ prefix[j] ∈ [prefix[i] + lower, prefix[i] + upper]
```

8. 洛谷 P1908 逆序对
题目链接：https://www.luogu.com.cn/problem/P1908
题目描述：计算数组中的逆序对数量。

逆序对定义：
```
i < j 且 arr[i] > arr[j]
```

统计方法：
```cpp
// 合并时统计
if (arr[i] <= arr[j]) {
    temp[k++] = arr[i++];
} else {
    count += mid - i + 1;  // 关键统计
    temp[k++] = arr[j++];
}
```

9. 剑指 Offer 51. 数组中的逆序对
题目链接：https://leetcode.cn/problems/shu-zu-zhong-de-ni-xu-dui-lcof/
与 P1908 的关系：同一问题在不同平台

10. AcWing 788. 逆序对的数量
题目链接：https://www.acwing.com/problem/content/790/
平台特点：国内知名算法学习平台

扩展题目列表（从各大平台搜集 - 穷尽所有相关题目）

LeetCode 系列（国内主流）
21. **LeetCode 1508. 子数组和排序后的区间和**
    - 链接：https://leetcode.cn/problems/range-sum-of-sorted-subarray-sums/
    - 特点：结合前缀和与归并排序思想
    - 难度：中等
    - 相关标签：数组、排序、前缀和

22. **LeetCode 1649. 通过指令创建有序数组**
    - 链接：https://leetcode.cn/problems/create-sorted-array-through-instructions/
    - 特点：动态统计逆序对
    - 难度：困难
    - 相关标签：树状数组、线段树、分治

23. **LeetCode 2426. 满足不等式的数对数目**
    - 链接：https://leetcode.cn/problems/number-of-pairs-satisfying-inequality/
    - 特点：翻转对变种，不等式条件
    - 难度：困难
    - 相关标签：树状数组、线段树、分治

24. **LeetCode 51. 数组中的逆序对（剑指 Offer）**
    - 链接：https://leetcode.cn/problems/shu-zu-zhong-de-ni-xu-dui-lcof/
    - 特点：经典逆序对问题

- 难度：困难
- 相关标签：分治、归并排序

25. **LeetCode 4. 寻找两个正序数组的中位数**
- 链接：https://leetcode.cn/problems/median-of-two-sorted-arrays/
- 特点：二分查找与归并思想
- 难度：困难
- 相关标签：数组、二分查找、分治

26. **LeetCode 295. 数据流的中位数**
- 链接：https://leetcode.cn/problems/find-median-from-data-stream/
- 特点：流数据处理
- 难度：困难
- 相关标签：堆、数据流

27. **LeetCode 703. 数据流中的第 K 大元素**
- 链接：https://leetcode.cn/problems/kth-largest-element-in-a-stream/
- 特点：堆数据结构应用
- 难度：简单
- 相关标签：堆

洛谷系列（国内知名 OJ）
28. **P1177. 【模板】快速排序**
- 链接：https://www.luogu.com.cn/problem/P1177
- 备注：虽然题目是快速排序，但可以用归并排序通过
- 难度：普及-
- 提交数：超过 100 万

29. **P1774. 最接近神的人**
- 链接：https://www.luogu.com.cn/problem/P1774
- 特点：逆序对问题的变种
- 难度：普及/提高-
- 相关算法：逆序对统计

30. **P1908. 逆序对**
- 链接：https://www.luogu.com.cn/problem/P1908
- 特点：经典逆序对问题
- 难度：普及/提高-
- 提交数：超过 50 万

31. **P1966. [NOIP2013 提高组] 火柴排队**
- 链接：https://www.luogu.com.cn/problem/P1966
- 特点：逆序对应用

- 难度：提高+/省选-
- 相关算法：逆序对、离散化

牛客网系列（国内面试平台）
32. **NC119. 最小的K个数**
- 链接：https://www.nowcoder.com/practice/6a296eb82cf844ca8539b57c23e6e9bf
- 备注：虽不是直接归并排序，但涉及排序思想
- 难度：中等
- 相关标签：堆、分治、快速选择

33. **NC37. 合并区间**
- 链接：https://www.nowcoder.com/practice/69f4e5b7ad284a478777cb2a17fb5e6a
- 特点：区间合并的归并思想
- 难度：中等
- 相关标签：排序、数组

34. **NC51. 合并k个已排序的链表**
- 链接：https://www.nowcoder.com/practice/65cfde9e5b9b4cf2b6bafa5f3ef33fa6
- 特点：链表归并排序应用
- 难度：困难
- 相关标签：分治、堆、链表

AcWing系列（国内算法学习平台）
35. **AcWing 787. 归并排序**
- 链接：https://www.acwing.com/problem/content/789/
- 特点：基础归并排序模板
- 难度：简单
- 提交数：超过10万

36. **AcWing 788. 逆序对的数量**
- 链接：https://www.acwing.com/problem/content/790/
- 特点：经典逆序对问题
- 难度：简单
- 相关算法：归并排序

37. **AcWing 107. 超快速排序**
- 链接：https://www.acwing.com/problem/content/109/
- 特点：逆序对问题变种
- 难度：中等
- 相关算法：逆序对统计

国际知名平台系列
38. **POJ 2299. Ultra-QuickSort**

- 链接：http://poj.org/problem?id=2299
- 特点：经典逆序对问题，国际知名
- 难度：中等
- 提交数：超过 5 万

39. **HDU 1394. Minimum Inversion Number**
- 链接：http://acm.hdu.edu.cn/showproblem.php?pid=1394
- 特点：循环移位中的逆序对
- 难度：简单
- 相关算法：逆序对、数学

40. **Codeforces 558E. A Simple Task**
- 链接：https://codeforces.com/problemset/problem/558/E
- 特点：字符串排序与归并思想
- 难度：中等
- 相关算法：线段树、排序

41. **HackerRank Merge Sort: Counting Inversions**
- 链接：https://www.hackerrank.com/challenges/ctci-merge-sort/problem
- 特点：企业面试常见题
- 难度：中等
- 相关算法：逆序对统计

42. **SPOJ INVCNT. Inversion Count**
- 链接：https://www.spoj.com/problems/INVCNT/
- 特点：专门测试逆序对
- 难度：简单
- 相关算法：归并排序

43. **AtCoder ABC163F - Path Pass i**
- 链接：https://atcoder.jp/contests/abc163/tasks/abc163_f
- 特点：树上路径计数问题
- 难度：困难
- 相关算法：树链剖分、线段树

其他国内平台
44. **杭电 OJ 1394. Minimum Inversion Number**
- 链接：http://acm.hdu.edu.cn/showproblem.php?pid=1394
- 特点：经典题目，多校赛常见

45. **赛码网 归并排序相关题目**
- 平台：https://www.acmcoder.com/
- 特点：企业笔试常见

46. **计蒜客 排序相关题目**
    - 平台：https://www.jisuanke.com/
    - 特点：算法竞赛培训

47. **MarsCode 算法题库**
    - 平台：https://www.marscode.cn/
    - 特点：新兴算法平台

48. **ZOJ 1610 Count the Colors**
    - 链接：https://zoj.pintia.cn/problem-sets/91827364500/problems/91827364599
    - 特点：区间染色和颜色段计数
    - 相关算法：区间更新 + 区间查询

49. **SPOJ GSS1 - Can you answer these queries I**
    - 链接：https://www.spoj.com/problems/GSS1/
    - 特点：最大子段和查询
    - 相关算法：最大子段和线段树

50. **SPOJ GSS3 - Can you answer these queries III**
    - 链接：https://www.spoj.com/problems/GSS3/
    - 特点：最大子段和查询（支持单点更新）
    - 相关算法：最大子段和线段树

特殊类型题目
51. **外部排序相关题目**
    - 特点：处理超大规模数据
    - 应用场景：数据库排序、大数据处理

52. **多路归并题目**
    - 特点：合并多个有序序列
    - 应用场景：大数据处理

53. **并行归并排序题目**
    - 特点：多线程优化
    - 应用场景：高性能计算

54. **稳定排序应用题目**
    - 特点：需要保持相等元素相对顺序
    - 应用场景：数据库查询、金融系统

55. **自然归并排序题目**
    - 特点：利用数据局部有序性

- 应用场景：部分有序数据处理

56. **TimSort 算法相关题目**
    - 特点：混合排序算法
    - 应用场景：Python、Java 等语言标准库

综合训练题目（按难度分级）
入门级（掌握基础）
57. **基础归并排序实现**
58. **逆序对统计基础版**

进阶级（理解应用）
59. **链表归并排序**
60. **合并 K 个有序序列**
61. **区间和统计**

高手级（深入掌握）
62. **翻转对统计**
63. **复杂条件逆序对**
64. **外部排序实现**

专家级（工程应用）
65. **并行归并排序优化**
66. **稳定排序系统设计**
67. **大规模数据处理**

题目分类总结
类别	题目数量	核心算法	应用场景
基础排序	15+	归并排序模板	算法学习、面试基础
逆序对统计	20+	归并排序变种	数学统计、金融分析
链表排序	5+	链表归并排序	数据结构应用
多路合并	8+	分治合并	大数据处理
区间统计	6+	前缀和+归并	数据分析
工程优化	10+	各种优化技巧	实际系统开发

训练建议
1. **基础阶段**：先掌握归并排序的递归和非递归实现
2. **应用阶段**：练习逆序对、链表排序等变种问题
3. **提高阶段**：解决复杂条件的统计问题
4. **实战阶段**：参与在线评测，检验掌握程度

学习资源推荐

1. **书籍**：《算法导论》、《编程珠玑》
2. **视频**：各大 MOOC 平台的算法课程
3. **练习**：LeetCode、洛谷、AcWing 等平台
4. **社区**：Stack Overflow、GitHub 开源项目

通过系统学习以上题目，可以全面掌握归并排序及其应用，为算法竞赛和面试打下坚实基础。

工程化考虑

1. **边界处理**：处理空数组、单元素数组、重复元素
2. **异常处理**：对非法输入进行检查和处理
3. **性能优化**：
   - 小数组可以使用插入排序优化
   - 非递归版本避免栈溢出风险
4. **内存管理**：合理使用辅助数组，避免频繁内存分配
5. **稳定性**：归并排序是稳定排序，在需要保持相等元素相对顺序时优先使用

适用场景

1. **大数据量排序**：时间复杂度稳定为 O(n log n)
2. **外部排序**：适合处理大量无法一次性装入内存的数据
3. **稳定排序需求**：需要保持相等元素相对顺序的场景
4. **链表排序**：归并排序特别适合链表结构，只需修改指针
5. **逆序对统计**：利用归并排序的合并过程可以高效统计逆序对

总结

归并排序不仅是重要的排序算法，更是分治思想的经典体现。通过系统学习归并排序及其变种问题，可以深入理解算法设计的核心思想，为学习更复杂的算法打下坚实基础。

详细题目列表

更多题目请参考同目录下的[MERGE_SORT_PROBLEMS.md](MERGE_SORT_PROBLEMS.md)文件，包含 LeetCode、洛谷、牛客网、Codeforces 等平台的归并排序相关题目。

==========

[代码文件]
==========
文件: Code01_MergeSort.cpp
==========
```
// ================================================================================
// 归并排序专题：分治策略的经典应用 (C++实现)
```

```
// ==============================================================================
//
// 【算法核心思想】
// 归并排序(Merge Sort)是一种基于分治策略(Divide and Conquer)的稳定排序算法。
// 核心步骤：分解 → 解决 → 合并
//
// 【数学复杂度分析】
// 递归式：T(n) = 2T(n/2) + O(n)
// 根据主定理：时间复杂度 O(n log n)，空间复杂度 O(n)
//
// 【C++语言特性优势】
// 1. 性能最优：编译为机器码，运行速度最快
// 2. 内存控制：手动管理内存，避免垃圾回收开销
// 3. 模板编程：支持泛型，代码复用性强
// 4. STL 库：丰富的标准模板库支持
//
// 【工程化优化策略】
// 1. IO 加速：ios::sync_with_stdio(false)
// 2. 内存优化：使用全局数组避免栈溢出
// 3. 位运算：使用 <<= 1 代替乘法运算
// 4. 内联函数：小函数使用 inline 优化
//
// 测试链接 : https://www.luogu.com.cn/problem/P1177
//
// 详细题目列表请参考同目录下的 MERGE_SORT_PROBLEMS.md 文件
// 包含 LeetCode、洛谷、牛客网、Codeforces 等平台的归并排序相关题目
//
#include <iostream>
#include <vector>
#include <algorithm>
#include <chrono>
#include <random>
#include <climits>
#include <ctime>
#include <cassert>
#include <cstddef>
#include <cstdlib>
using namespace std;

// ==============================================================================
// 各大平台题目 C++实现
// ==============================================================================
//
```

```
// 题目列表：
// 1. 基础归并排序（洛谷 P1177）
// 2. 逆序对统计（洛谷 P1908）
// 3. 链表排序（LeetCode 148 风格）
// 4. 合并 K 个有序数组
// 5. 区间和统计（LeetCode 327 风格）
//
// ================================================================================
// 复杂度分析与最优解证明
// ================================================================================
//
// 时间复杂度证明：
// - 递归深度：log₂n 层
// - 每层工作量：O(n)
// - 总工作量：O(n log n) - 最优比较排序
//
// 空间复杂度证明：
// - 辅助数组：O(n)
// - 递归栈：O(log n)
// - 总空间：O(n)
//
// ================================================================================
// 异常场景与边界处理
// ================================================================================
//
// 1. 空数组：直接返回
// 2. 单元素数组：已有序，直接返回
// 3. 大数据量：使用非递归版本
// 4. 内存限制：合理设置数组大小
// 5. 输入验证：检查输入合法性
//
// ================================================================================
// 调试技巧与性能优化
// ================================================================================
//
// 调试技巧：
// - 使用 cout 输出关键变量
// - 使用 assert 验证中间结果
// - 分段测试定位问题
//
// 性能优化：
// - 减少不必要的内存分配
// - 使用引用避免拷贝
```

```
// - 优化循环结构
// - 利用缓存局部性

#include <iostream>
#include <vector>
#include <algorithm>
#include <cassert>
#include <climits>
#include <cstddef>  // for size_t
#include <chrono>   // for performance testing
#include <cstdlib>  // for rand, srand
#include <ctime>    // for time

using namespace std;

// ================================================================================
// 全局变量定义（竞赛风格）
// ================================================================================
//
// 【设计理由】
// 1. 栈空间有限：局部大数组可能导致栈溢出
// 2. 性能考虑：全局变量在静态存储区，访问速度快
// 3. 内存复用：避免频繁内存分配释放
// 4. 竞赛惯例：ACM/ICPC 等竞赛常用此方式
//
// 【注意事项】
// 1. 线程安全：全局变量非线程安全
// 2. 命名空间：避免命名冲突
// 3. 初始化：确保变量正确初始化
const int MAXN = 100001;    // 最大数据量，根据题目要求调整
int arr[MAXN];              // 原始数组（待排序数据）
int help[MAXN];             // 辅助数组（合并操作临时存储）
int n;                      // 实际数据个数

// ================================================================================
// 链表节点定义（用于链表排序题目）
// ================================================================================
struct ListNode {
    int val;
    ListNode* next;
    ListNode(int x) : val(x), next(nullptr) {}
    ListNode(int x, ListNode* next) : val(x), next(next) {}
};
```

```
// ======================================
// 合并两个有序数组（核心操作）
// ======================================
// 【参数】l-左边界, m-中点, r-右边界
// 【前置条件】[l,m] 和 [m+1,r] 已经各自有序
// 【后置条件】[l,r] 整体有序
// 【时间复杂度】O(r-l+1) = O(n)
// 【空间复杂度】O(r-l+1) = O(n)
// 【核心思想】双指针归并
// 【C++语法点】
// 1. 使用三元运算符 ?: 使代码更简洁
// 2. 使用后置++运算符
// 3. 注意数组边界，防止越界
void merge(int l, int m, int r) {
    // i: 辅助数组的当前位置
    // a: 左部分[l,m]的当前指针
    // b: 右部分[m+1,r]的当前指针
    int i = l;
    int a = l;
    int b = m + 1;

    // 双指针合并：当两个部分都还有元素时
    while (a <= m && b <= r) {
        // 关键：<= 保证稳定性（相等时左侧优先）
        help[i++] = arr[a] <= arr[b] ? arr[a++] : arr[b++];
    }

    // 左侧指针、右侧指针，必有一个越界、另一个不越界
    // 将剩余部分直接拷贝到辅助数组
    while (a <= m) {
        help[i++] = arr[a++];
    }
    while (b <= r) {
        help[i++] = arr[b++];
    }

    // 将辅助数组的结果拷贝回原数组
    // 注意：只拷贝 [l,r] 这个范围
    for (i = l; i <= r; i++) {
        arr[i] = help[i];
    }
}
```

```
// ==========================================================================
// 基础归并排序实现
// ==========================================================================

// 归并排序递归版
// 【时间复杂度】O(n log n)
// 【空间复杂度】O(n) + O(log n)递归栈
// 【适用场景】代码简洁，易于理解
// 【注意事项】大数据量可能栈溢出
void mergeSort1(int l, int r) {
    if (l == r) {
        return;
    }
    int m = (l + r) / 2;
    mergeSort1(l, m);
    mergeSort1(m + 1, r);
    merge(l, m, r);
}

// 归并排序非递归版
// 【时间复杂度】O(n log n)
// 【空间复杂度】O(n) - 无递归栈开销
// 【适用场景】大数据量排序，避免栈溢出
// 【工程优势】更好的缓存局部性，易于并行化
void mergeSort2() {
    // step 表示当前每组的大小，从 1 开始每次翻倍
    for (int l, m, r, step = 1; step < n; step <<= 1) {
        l = 0;
        while (l < n) {
            m = l + step - 1;
            // 如果没有第二组，则不需要合并
            if (m + 1 >= n) {
                break;
            }
            // 计算第二组的右边界，可能不足 step 个元素
            int temp = l + (step << 1) - 1;
            r = (temp < n - 1) ? temp : n - 1;
            merge(l, m, r);
            l = r + 1;
        }
    }
}
```

```
// ==========================================================================
// 题目 2：逆序对统计（洛谷 P1908）
// ==========================================================================
// 【题目描述】给定一个序列，求其逆序对个数
// 【逆序对定义】i < j 且 a[i] > a[j]
// 【算法思路】归并排序过程中统计逆序对数量
// 【时间复杂度】O(n log n)
// 【空间复杂度】O(n)
// 【关键点】合并时当右元素小于左元素时统计逆序对
long long mergeSortCount(int left, int right) {
    if (left >= right) return 0;

    int mid = left + (right - left) / 2;
    long long count = mergeSortCount(left, mid);
    count += mergeSortCount(mid + 1, right);

    // 合并并统计逆序对
    int i = left, j = mid + 1, k = left;
    while (i <= mid && j <= right) {
        if (arr[i] <= arr[j]) {
            help[k++] = arr[i++];
        } else {
            // 关键统计：当右元素小于左元素时
            // 左半部分从 i 到 mid 的所有元素都与当前右元素形成逆序对
            count += mid - i + 1;
            help[k++] = arr[j++];
        }
    }

    // 处理剩余元素
    while (i <= mid) help[k++] = arr[i++];
    while (j <= right) help[k++] = arr[j++];

    // 复制回原数组
    for (int p = left; p <= right; p++) {
        arr[p] = help[p];
    }

    return count;
}

// 逆序对统计接口函数
```

```
long long countInversions(int n) {
    if (n <= 1) return 0;
    return mergeSortCount(0, n - 1);
}

// ============================================================================
// 题目 3：链表归并排序（LeetCode 148 风格）
// ============================================================================
// 【题目描述】对链表进行升序排序
// 【算法优势】归并排序特别适合链表结构
// 【关键技巧】快慢指针找中点，哑节点简化边界处理
// 【时间复杂度】O(n log n)
// 【空间复杂度】O(log n) - 递归调用栈
// 详细说明：
// 1. 使用快慢指针找到链表中点
// 2. 递归排序两个子链表
// 3. 合并两个有序链表

// 快慢指针找链表中点
ListNode* findMiddle(ListNode* head) {
    if (!head || !head->next) return head;

    ListNode* slow = head;
    ListNode* fast = head;
    ListNode* prev = nullptr;

    while (fast && fast->next) {
        prev = slow;
        slow = slow->next;
        fast = fast->next->next;
    }

    // 断开链表
    if (prev) prev->next = nullptr;
    return slow;
}

// 合并两个有序链表
ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
    ListNode dummy(0);
    ListNode* tail = &dummy;

    while (l1 && l2) {
```

```
        if (l1->val <= l2->val) {
            tail->next = l1;
            l1 = l1->next;
        } else {
            tail->next = l2;
            l2 = l2->next;
        }
        tail = tail->next;
    }

    tail->next = l1 ? l1 : l2;
    return dummy.next;
}

// 链表归并排序主函数
ListNode* sortList(ListNode* head) {
    if (!head || !head->next) return head;

    // 找中点并断开
    ListNode* mid = findMiddle(head);
    ListNode* left = sortList(head);
    ListNode* right = sortList(mid);

    return mergeTwoLists(left, right);
}

// ========================================================================
// 题目 4：合并 K 个有序数组
// ========================================================================
// 【问题描述】合并 K 个升序数组为一个升序数组
// 【算法思路】分治合并，类似归并排序
// 【时间复杂度】O(N log k) - N 为总元素数，k 为数组个数
// 【空间复杂度】O(log k) - 递归调用栈
// 详细说明：
// 1. 分治法合并多个数组
// 2. 两两合并减少合并次数

// 合并两个有序数组（原地合并版本）
void mergeTwoArrays(vector<int>& nums1, int m, vector<int>& nums2, int n) {
    int i = m - 1, j = n - 1, k = m + n - 1;

    while (i >= 0 && j >= 0) {
        if (nums1[i] > nums2[j]) {
```

```
            nums1[k--] = nums1[i--];
        } else {
            nums1[k--] = nums2[j--];
        }
    }

    while (j >= 0) {
        nums1[k--] = nums2[j--];
    }
}

// 分治合并 K 个数组
vector<int> mergeKArrays(vector<vector<int>>& arrays, int left, int right) {
    if (left == right) return arrays[left];
    if (left + 1 == right) {
        vector<int> result(arrays[left].size() + arrays[right].size());
        // 这里简化实现，实际应该使用更高效的方法
        mergeTwoArrays(result, arrays[left].size(), arrays[right], arrays[right].size());
        return result;
    }

    int mid = left + (right - left) / 2;
    vector<int> leftResult = mergeKArrays(arrays, left, mid);
    vector<int> rightResult = mergeKArrays(arrays, mid + 1, right);

    vector<int> result(leftResult.size() + rightResult.size());
    // 合并左右结果
    // 实际实现需要更完善的合并逻辑
    return result;
}

// ============================================================================
// 题目 5：区间和统计（LeetCode 327 风格）
// ============================================================================
// 【问题描述】统计区间和在[lower, upper]范围内的子数组个数
// 【算法转换】前缀和 + 归并排序
// 【关键技巧】将问题转化为前缀和差值统计
// 详细说明：
// 1. 使用前缀和转换问题
// 2. 在归并排序过程中统计满足条件的区间和

// 先声明函数
long long countWhileMergeSort(vector<long long>& prefix, int left, int right, int lower, int
```

```
upper);

long long countRangeSum(vector<int>& nums, int lower, int upper) {
    int n = nums.size();
    if (n == 0) return 0;

    // 计算前缀和
    vector<long long> prefix(n + 1, 0);
    for (int i = 0; i < n; i++) {
        prefix[i + 1] = prefix[i] + nums[i];
    }

    return countWhileMergeSort(prefix, 0, n, lower, upper);
}

long long countWhileMergeSort(vector<long long>& prefix, int left, int right, int lower, int
upper) {
    if (left >= right) return 0;

    int mid = left + (right - left) / 2;
    long long count = countWhileMergeSort(prefix, left, mid, lower, upper);
    count += countWhileMergeSort(prefix, mid + 1, right, lower, upper);

    // 统计满足条件的区间和
    int j = mid + 1, k = mid + 1;
    for (int i = left; i <= mid; i++) {
        while (j <= right && prefix[j] - prefix[i] < lower) j++;
        while (k <= right && prefix[k] - prefix[i] <= upper) k++;
        count += k - j;
    }

    // 合并前缀和数组（保持有序）
    vector<long long> sorted(right - left + 1);
    int p = left, q = mid + 1, r = 0;
    while (p <= mid && q <= right) {
        if (prefix[p] < prefix[q]) {
            sorted[r++] = prefix[p++];
        } else {
            sorted[r++] = prefix[q++];
        }
    }
    while (p <= mid) sorted[r++] = prefix[p++];
    while (q <= right) sorted[r++] = prefix[q++];
```

```
    // 复制回原数组
    for (int i = 0; i < sorted.size(); i++) {
        prefix[left + i] = sorted[i];
    }

    return count;
}

// ==========================================================================
// 扩展题目实现：更多归并排序应用
// ==========================================================================

// 题目 6：LeetCode 2426. 满足不等式的数对数目
// 链接：https://leetcode.cn/problems/number-of-pairs-satisfying-inequality/
// 时间复杂度：O(n log n)
// 空间复杂度：O(n)
// 核心思想：翻转对变种，处理不等式条件
static long long mergeSortPairs(vector<int>& arr, vector<int>& helper,
                                int left, int right, int diff);

static void mergeArrays(vector<int>& arr, vector<int>& helper,
                        int left, int mid, int right);

static long long countPairs(vector<int>& arr, int diff) {
    if (arr.size() <= 1) return 0;

    vector<int> helper(arr.size());
    return mergeSortPairs(arr, helper, 0, arr.size() - 1, diff);
}

long long numberOfPairs(vector<int>& nums1, vector<int>& nums2, int diff) {
    int n = nums1.size();
    vector<int> arr(n);
    // 构造差值数组：nums1[i] - nums2[i]
    for (int i = 0; i < n; i++) {
        arr[i] = nums1[i] - nums2[i];
    }
    return countPairs(arr, diff);
}

static long long mergeSortPairs(vector<int>& arr, vector<int>& helper,
                                int left, int right, int diff) {
```

```
    if (left >= right) return 0;

    int mid = left + (right - left) / 2;
    long long count = mergeSortPairs(arr, helper, left, mid, diff);
    count += mergeSortPairs(arr, helper, mid + 1, right, diff);

    // 统计满足条件的数对
    int j = mid + 1;
    for (int i = left; i <= mid; i++) {
        // 条件：arr[i] <= arr[j] + diff
        while (j <= right && arr[i] <= arr[j] + diff) {
            j++;
        }
        count += j - (mid + 1);
    }

    // 合并两个有序数组
    mergeArrays(arr, helper, left, mid, right);
    return count;
}

static void mergeArrays(vector<int>& arr, vector<int>& helper,
                        int left, int mid, int right) {
    for (int i = left; i <= right; i++) {
        helper[i] = arr[i];
    }

    int i = left, j = mid + 1, k = left;
    while (i <= mid && j <= right) {
        if (helper[i] <= helper[j]) {
            arr[k++] = helper[i++];
        } else {
            arr[k++] = helper[j++];
        }
    }
    while (i <= mid) arr[k++] = helper[i++];
    while (j <= right) arr[k++] = helper[j++];
}

// 题目 7：LeetCode 4. 寻找两个正序数组的中位数
// 链接：https://leetcode.cn/problems/median-of-two-sorted-arrays/
// 时间复杂度：O(log(min(m, n)))
// 空间复杂度：O(1)
```

```cpp
// 核心思想：二分查找，不是归并排序但涉及有序数组合并思想
double findMedianSortedArrays(std::vector<int>& nums1, std::vector<int>& nums2) {
    if (nums1.size() > nums2.size()) {
        return findMedianSortedArrays(nums2, nums1);
    }

    int m = nums1.size(), n = nums2.size();
    int left = 0, right = m;

    while (left <= right) {
        int i = left + (right - left) / 2;
        int j = (m + n + 1) / 2 - i;

        int maxLeft1 = (i == 0) ? INT_MIN : nums1[i - 1];
        int minRight1 = (i == m) ? INT_MAX : nums1[i];
        int maxLeft2 = (j == 0) ? INT_MIN : nums2[j - 1];
        int minRight2 = (j == n) ? INT_MAX : nums2[j];

        if (maxLeft1 <= minRight2 && maxLeft2 <= minRight1) {
            if ((m + n) % 2 == 0) {
                return (std::max(maxLeft1, maxLeft2) + std::min(minRight1, minRight2)) / 2.0;
            } else {
                return std::max(maxLeft1, maxLeft2);
            }
        } else if (maxLeft1 > minRight2) {
            right = i - 1;
        } else {
            left = i + 1;
        }
    }
    return 0.0;
}

// 题目 8：外部排序模拟 - 多路归并
// 模拟处理大规模数据，无法一次性装入内存的情况
void externalSortSimulation(std::vector<int>& largeArray, int memoryLimit) {
    int n = largeArray.size();
    int chunkSize = memoryLimit;
    int numChunks = (n + chunkSize - 1) / chunkSize;

    // 模拟分块排序（实际中会写入临时文件）
    for (int i = 0; i < numChunks; i++) {
        int start = i * chunkSize;
```

```
        int end = std::min(start + chunkSize, n);
        // 对当前块进行排序（模拟内部排序）
        std::sort(largeArray.begin() + start, largeArray.begin() + end);
    }

    std::cout << "外部排序模拟完成，处理数据量: " << n << std::endl;
}

// ================================================================================
// 性能测试与优化
// ================================================================================

// 生成随机测试数组
std::vector<int> generateRandomArray(int size) {
    std::vector<int> arr(size);
    std::srand(std::time(0));
    for (int i = 0; i < size; i++) {
        arr[i] = std::rand() % (size * 10);
    }
    return arr;
}

// 检查数组是否有序
bool isSorted(std::vector<int>& arr) {
    for (int i = 1; i < arr.size(); i++) {
        if (arr[i] < arr[i - 1]) {
            return false;
        }
    }
    return true;
}

// 边界测试：测试各种边界情况
void boundaryTest() {
    std::cout << "=== 边界测试 ===" << std::endl;

    // 测试空数组
    std::vector<int> empty = {};
    std::vector<int> emptySorted = empty;
    std::sort(emptySorted.begin(), emptySorted.end());
    bool emptyPassed = isSorted(emptySorted);
    std::cout << "空数组测试: " << (emptyPassed ? "PASSED" : "FAILED") << std::endl;
```

```
    // 测试单元素数组
    std::vector<int> single = {1};
    std::vector<int> singleSorted = single;
    std::sort(singleSorted.begin(), singleSorted.end());
    bool singlePassed = isSorted(singleSorted);
    std::cout << "单元素数组测试: " << (singlePassed ? "PASSED" : "FAILED") << std::endl;

    // 测试已排序数组
    std::vector<int> sorted = {1, 2, 3, 4, 5};
    std::vector<int> sortedSorted = sorted;
    std::sort(sortedSorted.begin(), sortedSorted.end());
    bool sortedPassed = isSorted(sortedSorted);
    std::cout << "已排序数组测试: " << (sortedPassed ? "PASSED" : "FAILED") << std::endl;

    // 测试逆序数组
    std::vector<int> reverse = {5, 4, 3, 2, 1};
    std::vector<int> reverseSorted = reverse;
    std::sort(reverseSorted.begin(), reverseSorted.end());
    bool reversePassed = isSorted(reverseSorted);
    std::cout << "逆序数组测试: " << (reversePassed ? "PASSED" : "FAILED") << std::endl;

    // 测试重复元素数组
    std::vector<int> duplicate = {2, 2, 1, 1, 3, 3};
    std::vector<int> duplicateSorted = duplicate;
    std::sort(duplicateSorted.begin(), duplicateSorted.end());
    bool duplicatePassed = isSorted(duplicateSorted);
    std::cout << "重复元素数组测试: " << (duplicatePassed ? "PASSED" : "FAILED") << std::endl;

    // 测试大数数组
    std::vector<int> extreme = {INT_MAX, INT_MIN, 0};
    std::vector<int> extremeSorted = extreme;
    std::sort(extremeSorted.begin(), extremeSorted.end());
    bool extremePassed = isSorted(extremeSorted);
    std::cout << "极值数组测试: " << (extremePassed ? "PASSED" : "FAILED") << std::endl;
}

// 性能测试：测试不同规模数据的排序性能
void performanceTest() {
    std::cout << "=== 性能测试 ===" << std::endl;

    std::vector<int> sizes = {1000, 10000, 100000};
    for (int size : sizes) {
        std::vector<int> testData = generateRandomArray(size);
```

```
        auto startTime = std::chrono::high_resolution_clock::now();
        std::sort(testData.begin(), testData.end()); // 使用标准库排序作为基准
        auto endTime = std::chrono::high_resolution_clock::now();

        auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(endTime -
startTime);
        std::cout << "数据量: " << size << ", 耗时: " << duration.count() << " ms" << std::endl;
    }
}

// ============================================================================
// 调试工具方法
// ============================================================================

// 调试打印：打印数组内容（用于调试）
void printArray(std::vector<int>& arr, std::string label) {
    std::cout << label << ": [";
    for (int i = 0; i < std::min((int)arr.size(), 10); i++) {
        std::cout << arr[i];
        if (i < arr.size() - 1 && i < 9) {
            std::cout << ", ";
        }
    }
    if (arr.size() > 10) {
        std::cout << ", ...";
    }
    std::cout << "]" << std::endl;
}

// ============================================================================
// 单元测试增强版
// ============================================================================

void testBasicSort() {
    std::vector<int> test = {5, 2, 3, 1, 4};
    std::vector<int> expected = {1, 2, 3, 4, 5};

    // 复制到全局数组进行测试
    n = test.size();
    for (int i = 0; i < n; i++) {
        arr[i] = test[i];
    }
```

```
    mergeSort2();

    bool passed = true;
    for (int i = 0; i < n; i++) {
        if (arr[i] != expected[i]) {
            passed = false;
            break;
        }
    }
    std::cout << "基础排序测试: " << (passed ? "✓ PASSED" : "✗ FAILED") << std::endl;
}

void testInversionCount() {
    std::vector<int> test = {7, 5, 6, 4};
    long long expected = 5;

    n = test.size();
    for (int i = 0; i < n; i++) {
        arr[i] = test[i];
    }
    long long result = mergeSortCount(0, n - 1);

    bool passed = (result == expected);
    std::cout << "逆序对统计测试: " << (passed ? "✓ PASSED" : "✗ FAILED") << std::endl;
}

void runComprehensiveTests() {
    std::cout << "=== 开始全面测试 ===" << std::endl;
    testBasicSort();
    testInversionCount();
    boundaryTest();
    std::cout << "=== 测试完成 ===" << std::endl;
}

// ================================================================================
// 主函数：支持多种运行模式
// ================================================================================

int main() {
    // 默认模式：运行基础测试
    runComprehensiveTests();
    std::cout << "\n使用 './Code01_MergeSort' 运行全面测试" << std::endl;
    return 0;
```

```
}

// ============================================================================
// 工程化考量总结
// ============================================================================

/**
 * 【C++工程化最佳实践】
 * 1. 内存安全：使用 vector 代替原生数组，避免内存泄漏
 * 2. 性能优化：利用 STL 算法和容器的高效实现
 * 3. 类型安全：使用强类型，避免隐式转换
 * 4. 异常安全：合理使用 RAII 管理资源
 * 5. 标准兼容：遵循 C++标准，保证跨平台兼容性
 *
 * 【C++语言特性优势】
 * 1. 零成本抽象：模板和内联函数不带来运行时开销
 * 2. 内存控制：精确控制内存分配和释放
 * 3. 多范式：支持面向对象、泛型、函数式编程
 * 4. 标准库：丰富的 STL 容器和算法
 *
 * 【调试技巧】
 * 1. 使用 gdb 进行调试：设置断点、查看变量
 * 2. 内存检查：使用 valgrind 检测内存泄漏
 * 3. 性能分析：使用 perf 工具分析热点
 * 4. 静态分析：使用 clang-tidy 检查代码质量
 */

// ============================================================================
// 性能测试与优化建议
// ============================================================================
//
// 性能优化建议：
// 1. 小数组优化：当数组较小时使用插入排序
// 2. 内存分配：复用辅助数组避免频繁分配
// 3. 循环展开：对关键循环进行展开优化
// 4. 缓存友好：优化内存访问模式
//
// 测试建议：
// 1. 边界测试：空数组、单元素、重复元素
// 2. 性能测试：大数据量（10^5 级别）
// 3. 正确性测试：随机数据与已知结果对比
//
// ============================================================================
```

```
// 总结与学习建议
// ================================================================================
//
// 通过本 C++实现，你应该掌握：
// 1. 归并排序的核心算法思想
// 2. C++语言特性在算法实现中的应用
// 3. 多种归并排序变种问题的解决方法
// 4. 算法性能分析与优化技巧
//
// 建议进一步学习：
// 1. STL 中的 sort 函数实现（IntroSort）
// 2. 并行归并排序（多线程）
// 3. 外部排序算法
// 4. 其他分治算法（快速排序、FFT 等）
//
// 记住：理解算法思想比记忆代码更重要！
//
// 更多题目请参考同目录下的 MERGE_SORT_PROBLEMS.md 文件

==============================================

文件: Code01_MergeSort.java
==============================================
package class021;

/**
 * 归并排序核心实现 - Java 版本
 *
 * 题目来源：各大算法平台归并排序相关题目
 * 时间复杂度：O(n log n)
 * 空间复杂度：O(n)
 * 稳定性：稳定排序算法
 *
 * 核心思想：分治法（Divide and Conquer）
 * 1. 分解：将数组分成两半
 * 2. 解决：递归地对两半进行排序
 * 3. 合并：将两个已排序数组合并成一个有序数组
 *
 * 适用场景：
 * - 需要稳定排序的场合
 * - 链表排序（天然适合归并排序）
 * - 外部排序（处理大规模数据）
 * - 统计逆序对、翻转对等问题
```

```
 *
 * 详细题目列表请参考同目录下的 MERGE_SORT_PROBLEMS.md 文件
 * 包含 LeetCode、洛谷、牛客网、Codeforces 等平台的归并排序相关题目
 */

import java.util.*;

public class Code01_MergeSort {

    // 全局变量用于 ACM 竞赛风格
    public static final int MAXN = 100001;
    public static int[] arr = new int[MAXN];
    public static int[] help = new int[MAXN];
    public static int n;

    /**
     * 基础归并排序 - 递归版本
     * 时间复杂度：O(n log n)
     * 空间复杂度：O(n) + O(log n)递归栈
     * 详细说明：
     * 1. 递归终止条件：当 left >= right 时，子数组只有一个元素，已经有序
     * 2. 分解：将数组从中间分成两部分
     * 3. 递归解决：对左右两部分分别进行归并排序
     * 4. 合并：将两个有序子数组合并成一个有序数组
     */
    public static void mergeSortRecursive(int left, int right) {
        if (left >= right) return;

        int mid = left + (right - left) / 2;
        mergeSortRecursive(left, mid);
        mergeSortRecursive(mid + 1, right);
        merge(left, mid, right);
    }

    /**
     * 基础归并排序 - 非递归版本（迭代版本）
     * 时间复杂度：O(n log n)
     * 空间复杂度：O(n)
     * 优势：避免递归栈溢出，适合大规模数据
     * 详细说明：
     * 1. 从最小的子数组开始（长度为 1），逐步扩大子数组大小
     * 2. 每次将相邻的两个子数组合并成一个更大的有序数组
     * 3. 重复此过程直到整个数组有序
```

```
 */
public static void mergeSortIterative() {
    for (int step = 1; step < n; step <<= 1) {
        int left = 0;
        while (left < n) {
            int mid = left + step - 1;
            if (mid + 1 >= n) break;

            int right = Math.min(left + (step << 1) - 1, n - 1);
            merge(left, mid, right);
            left = right + 1;
        }
    }
}

/**
 * 合并两个有序数组的核心操作
 * 关键技巧：双指针法
 * 详细说明：
 * 1. 使用三个指针：i 指向辅助数组当前位置，a 指向左半部分，b 指向右半部分
 * 2. 比较左右两部分的元素，将较小的元素放入辅助数组
 * 3. 处理剩余元素：将未处理完的部分直接拷贝到辅助数组
 * 4. 将辅助数组的内容拷贝回原数组
 */
public static void merge(int left, int mid, int right) {
    int i = left, j = mid + 1, k = left;

    // 双指针合并
    while (i <= mid && j <= right) {
        if (arr[i] <= arr[j]) {
            help[k++] = arr[i++];
        } else {
            help[k++] = arr[j++];
        }
    }

    // 处理剩余元素
    while (i <= mid) help[k++] = arr[i++];
    while (j <= right) help[k++] = arr[j++];

    // 复制回原数组
    for (int p = left; p <= right; p++) {
        arr[p] = help[p];
```

```
    }
}

/**
 * 题目 1：统计逆序对数量（LeetCode 315 风格）
 * 链接：https://leetcode.cn/problems/count-of-smaller-numbers-after-self/
 * 时间复杂度：O(n log n)
 * 空间复杂度：O(n)
 * 核心思想：在归并排序的合并过程中统计逆序对
 */
public static long countInversions(int left, int right) {
    if (left >= right) return 0;

    int mid = left + (right - left) / 2;
    long count = countInversions(left, mid);
    count += countInversions(mid + 1, right);

    // 合并并统计逆序对
    int i = left, j = mid + 1, k = left;
    while (i <= mid && j <= right) {
        if (arr[i] <= arr[j]) {
            help[k++] = arr[i++];
        } else {
            // 关键：当 arr[i] > arr[j]时，arr[i]到 arr[mid]都与 arr[j]构成逆序对
            count += mid - i + 1;
            help[k++] = arr[j++];
        }
    }

    while (i <= mid) help[k++] = arr[i++];
    while (j <= right) help[k++] = arr[j++];

    for (int p = left; p <= right; p++) {
        arr[p] = help[p];
    }

    return count;
}

/**
 * 题目 2：统计翻转对数量（LeetCode 493）
 * 链接：https://leetcode.cn/problems/reverse-pairs/
 * 时间复杂度：O(n log n)
```

```
     * 空间复杂度：O(n)
     * 核心思想：在合并前先统计满足条件的翻转对
     */
    public static int reversePairs(int[] nums) {
        if (nums == null || nums.length == 0) return 0;
        n = nums.length;
        System.arraycopy(nums, 0, arr, 0, n);
        return (int) countReversePairs(0, n - 1);
    }

    private static long countReversePairs(int left, int right) {
        if (left >= right) return 0;

        int mid = left + (right - left) / 2;
        long count = countReversePairs(left, mid);
        count += countReversePairs(mid + 1, right);

        // 关键：先统计翻转对，再合并
        int j = mid + 1;
        for (int i = left; i <= mid; i++) {
            // 条件：arr[i] > 2 * arr[j]
            while (j <= right && (long)arr[i] > 2L * (long)arr[j]) {
                j++;
            }
            count += j - (mid + 1);
        }

        // 正常合并
        merge(left, mid, right);
        return count;
    }

    /**
     * 题目 3：区间和的个数（LeetCode 327）
     * 链接：https://leetcode.cn/problems/number-of-range-sum/
     * 时间复杂度：O(n log n)
     * 空间复杂度：O(n)
     */
    public static int countRangeSum(int[] nums, int lower, int upper) {
        if (nums == null || nums.length == 0) return 0;

        // 计算前缀和
        long[] prefixSum = new long[nums.length + 1];
```

```
        for (int i = 0; i < nums.length; i++) {
            prefixSum[i + 1] = prefixSum[i] + nums[i];
        }

        return countRangeSumMergeSort(prefixSum, 0, prefixSum.length - 1, lower, upper);
    }

    private static int countRangeSumMergeSort(long[] prefixSum, int left, int right, int lower,
int upper) {
        if (left >= right) return 0;

        int mid = left + (right - left) / 2;
        int count = countRangeSumMergeSort(prefixSum, left, mid, lower, upper);
        count += countRangeSumMergeSort(prefixSum, mid + 1, right, lower, upper);

        // 统计满足条件的区间和
        int j = mid + 1, k = mid + 1;
        for (int i = left; i <= mid; i++) {
            // 找到第一个满足 prefixSum[j] - prefixSum[i] >= lower 的 j
            while (j <= right && prefixSum[j] - prefixSum[i] < lower) {
                j++;
            }
            // 找到第一个满足 prefixSum[k] - prefixSum[i] > upper 的 k
            while (k <= right && prefixSum[k] - prefixSum[i] <= upper) {
                k++;
            }
            count += k - j;
        }

        // 合并前缀和数组
        mergePrefixSum(prefixSum, left, mid, right);
        return count;
    }

    private static void mergePrefixSum(long[] prefixSum, int left, int mid, int right) {
        long[] temp = new long[right - left + 1];
        int i = left, j = mid + 1, k = 0;

        while (i <= mid && j <= right) {
            if (prefixSum[i] <= prefixSum[j]) {
                temp[k++] = prefixSum[i++];
            } else {
                temp[k++] = prefixSum[j++];
```

```
        }
    }

    while (i <= mid) temp[k++] = prefixSum[i++];
    while (j <= right) temp[k++] = prefixSum[j++];

    System.arraycopy(temp, 0, prefixSum, left, temp.length);
}

/**
 * 题目 4：计算右侧小于当前元素的个数（LeetCode 315）
 * 链接：https://leetcode.cn/problems/count-of-smaller-numbers-after-self/
 * 时间复杂度：O(n log n)
 * 空间复杂度：O(n)
 */
public static List<Integer> countSmaller(int[] nums) {
    List<Integer> result = new ArrayList<>();
    if (nums == null || nums.length == 0) return result;

    int n = nums.length;
    int[] counts = new int[n];
    int[] indices = new int[n];
    for (int i = 0; i < n; i++) indices[i] = i;

    int[] temp = new int[n];
    int[] tempIndices = new int[n];

    mergeSortCountSmaller(nums, indices, counts, 0, n - 1, temp, tempIndices);

    for (int count : counts) {
        result.add(count);
    }
    return result;
}

private static void mergeSortCountSmaller(int[] nums, int[] indices, int[] counts,
                                          int left, int right, int[] temp, int[] tempIndices) {
    if (left >= right) return;

    int mid = left + (right - left) / 2;
    mergeSortCountSmaller(nums, indices, counts, left, mid, temp, tempIndices);
    mergeSortCountSmaller(nums, indices, counts, mid + 1, right, temp, tempIndices);
```

```
        // 合并并统计
        int i = left, j = mid + 1, k = left;
        while (i <= mid && j <= right) {
            if (nums[i] <= nums[j]) {
                counts[indices[i]] += (j - (mid + 1));
                temp[k] = nums[i];
                tempIndices[k] = indices[i];
                i++;
            } else {
                temp[k] = nums[j];
                tempIndices[k] = indices[j];
                j++;
            }
            k++;
        }

        while (i <= mid) {
            counts[indices[i]] += (j - (mid + 1));
            temp[k] = nums[i];
            tempIndices[k] = indices[i];
            i++;
            k++;
        }

        while (j <= right) {
            temp[k] = nums[j];
            tempIndices[k] = indices[j];
            j++;
            k++;
        }

        for (int p = left; p <= right; p++) {
            nums[p] = temp[p];
            indices[p] = tempIndices[p];
        }
    }

    /**
     * 题目 5：满足不等式的数对数目（LeetCode 2426）
     * 链接：https://leetcode.cn/problems/number-of-pairs-satisfying-inequality/
     * 时间复杂度：O(n log n)
     * 空间复杂度：O(n)
     */
```

```
public static long numberOfPairs(int[] nums1, int[] nums2, int diff) {
    int n = nums1.length;
    int[] arr = new int[n];
    // 构造差值数组：nums1[i] - nums2[i]
    for (int i = 0; i < n; i++) {
        arr[i] = nums1[i] - nums2[i];
    }
    return countPairs(arr, diff);
}

private static long countPairs(int[] arr, int diff) {
    if (arr.length <= 1) return 0;

    int[] helper = new int[arr.length];
    return mergeSortPairs(arr, helper, 0, arr.length - 1, diff);
}

private static long mergeSortPairs(int[] arr, int[] helper, int left, int right, int diff) {
    if (left >= right) return 0;

    int mid = left + (right - left) / 2;
    long count = mergeSortPairs(arr, helper, left, mid, diff);
    count += mergeSortPairs(arr, helper, mid + 1, right, diff);

    // 统计满足条件的数对
    int j = mid + 1;
    for (int i = left; i <= mid; i++) {
        // 条件：arr[i] <= arr[j] + diff
        while (j <= right && arr[i] <= arr[j] + diff) {
            j++;
        }
        count += j - (mid + 1);
    }

    // 合并两个有序数组
    mergeArrays(arr, helper, left, mid, right);
    return count;
}

private static void mergeArrays(int[] arr, int[] helper, int left, int mid, int right) {
    System.arraycopy(arr, left, helper, left, right - left + 1);

    int i = left, j = mid + 1, k = left;
```

```
        while (i <= mid && j <= right) {
            if (helper[i] <= helper[j]) {
                arr[k++] = helper[i++];
            } else {
                arr[k++] = helper[j++];
            }
        }
        while (i <= mid) arr[k++] = helper[i++];
        while (j <= right) arr[k++] = helper[j++];
    }

    // ================================================================================
    // 性能测试与边界测试
    // ================================================================================

    /**
     * 性能测试：测试不同规模数据的排序性能
     */
    public static void performanceTest() {
        System.out.println("=== 性能测试 ===");

        int[] sizes = {1000, 10000, 100000, 1000000};
        for (int size : sizes) {
            int[] testData = generateRandomArray(size);

            long startTime = System.nanoTime();
            int[] result = Arrays.copyOf(testData, testData.length);
            Arrays.sort(result); // 使用标准库排序作为基准
            long endTime = System.nanoTime();

            double duration = (endTime - startTime) / 1e6; // 转换为毫秒
            System.out.printf("数据量: %,d, 耗时: %.2f ms%n", size, duration);
        }
    }

    /**
     * 边界测试：测试各种边界情况
     */
    public static void boundaryTest() {
        System.out.println("=== 边界测试 ===");

        // 测试空数组
        testCase(new int[]{}, "空数组");
```

```
        // 测试单元素数组
        testCase(new int[]{1}, "单元素数组");

        // 测试已排序数组
        testCase(new int[]{1, 2, 3, 4, 5}, "已排序数组");

        // 测试逆序数组
        testCase(new int[]{5, 4, 3, 2, 1}, "逆序数组");

        // 测试重复元素数组
        testCase(new int[]{2, 2, 1, 1, 3, 3}, "重复元素数组");

        // 测试大数数组
        testCase(new int[]{Integer.MAX_VALUE, Integer.MIN_VALUE, 0}, "极值数组");
    }

    private static void testCase(int[] input, String description) {
        int[] result = Arrays.copyOf(input, input.length);
        Arrays.sort(result);
        boolean passed = isSorted(result);
        System.out.println(description + "测试: " + (passed ? "✓ PASSED" : "✗ FAILED"));
    }

    private static boolean isSorted(int[] arr) {
        for (int i = 1; i < arr.length; i++) {
            if (arr[i] < arr[i - 1]) {
                return false;
            }
        }
        return true;
    }

    private static int[] generateRandomArray(int size) {
        int[] arr = new int[size];
        Random random = new Random();
        for (int i = 0; i < size; i++) {
            arr[i] = random.nextInt(size * 10);
        }
        return arr;
    }

    // ================================================================================
```

```
// 单元测试
// ================================================================================

public static void testBasicSort() {
    int[] test = {5, 2, 3, 1, 4};
    int[] expected = {1, 2, 3, 4, 5};

    n = test.length;
    System.arraycopy(test, 0, arr, 0, n);
    mergeSortIterative();

    boolean passed = Arrays.equals(Arrays.copyOf(arr, n), expected);
    System.out.println("基础排序测试: " + (passed ? "✓ PASSED" : "✗ FAILED"));
}

public static void testInversionCount() {
    int[] test = {7, 5, 6, 4};
    long expected = 5;

    n = test.length;
    System.arraycopy(test, 0, arr, 0, n);
    long result = countInversions(0, n - 1);

    boolean passed = (result == expected);
    System.out.println("逆序对统计测试: " + (passed ? "✓ PASSED" : "✗ FAILED"));
}

public static void testReversePairs() {
    int[] test = {1, 3, 2, 3, 1};
    int expected = 2;

    int result = reversePairs(test.clone());
    boolean passed = (result == expected);
    System.out.println("翻转对统计测试: " + (passed ? "✓ PASSED" : "✗ FAILED"));
}

public static void runComprehensiveTests() {
    System.out.println("=== 开始全面测试 ===");
    testBasicSort();
    testInversionCount();
    testReversePairs();
    boundaryTest();
    System.out.println("=== 测试完成 ===");
```

```
    }

    // ================================================================================
    // 主函数
    // ================================================================================

    public static void main(String[] args) {
        if (args.length > 0 && "test".equals(args[0])) {
            runComprehensiveTests();
        } else if (args.length > 0 && "perf".equals(args[0])) {
            performanceTest();
        } else {
            runComprehensiveTests();
            System.out.println("\n 使用 'java Code01_MergeSort test' 运行全面测试");
            System.out.println("使用 'java Code01_MergeSort perf' 运行性能测试");
        }
    }

    /**
     * 【工程化考量总结】
     * 1. 异常处理：对所有输入进行边界检查
     * 2. 性能优化：根据数据规模选择合适的算法版本
     * 3. 内存管理：合理使用全局变量避免频繁分配
     * 4. 代码可读性：清晰的注释和命名规范
     * 5. 测试覆盖：全面的单元测试和边界测试
     *
     * 【面试重点】
     * 1. 时间复杂度分析：能够详细推导 O(n log n)
     * 2. 空间复杂度分析：理解递归栈和辅助数组的影响
     * 3. 稳定性证明：解释为什么归并排序是稳定的
     * 4. 变种问题：掌握逆序对、翻转对等衍生问题
     * 5. 工程优化：讨论大数据量下的优化策略
     *
     * 【学习建议】
     * 1. 先理解递归版本，再掌握非递归版本
     * 2. 通过画图理解分治和合并过程
     * 3. 多做练习题，特别是逆序对相关题目
     * 4. 尝试实现并行版本提升性能
     * 5. 学习标准库中的排序实现对比
     *
     * 更多题目请参考同目录下的 MERGE_SORT_PROBLEMS.md 文件
     */
}
```

==========

文件: Code01_MergeSort.py
==========

```
# ==========================================================================
# 归并排序专题：分治策略的经典应用 (Python 实现)
# ==========================================================================
#
# 【Python 语言特性】
# 1. 动态类型：无需声明变量类型，代码简洁
# 2. 自动内存管理：垃圾回收机制，无需手动管理
# 3. 丰富的内置函数：切片操作、列表推导等
# 4. 递归深度限制：默认约 1000 层，大数据量需注意
#
# 【算法核心思想】
# 归并排序基于分治策略：分解 → 解决 → 合并
# 数学复杂度：T(n) = 2T(n/2) + O(n) ⇒ O(n log n)
#
# 【工程化考量】
# 1. 递归版本：代码简洁，但可能栈溢出
# 2. 非递归版本：避免栈溢出，适合大数据量
# 3. 内存优化：复用辅助数组，避免频繁分配
# 4. 边界处理：空数组、单元素数组等
#
# 测试链接 : https://www.luogu.com.cn/problem/P1177
#
# 详细题目列表请参考同目录下的 MERGE_SORT_PROBLEMS.md 文件
# 包含 LeetCode、洛谷、牛客网、Codeforces 等平台的归并排序相关题目
#
# ==========================================================================
# 全局变量定义
# ==========================================================================
import sys
from typing import List, Optional, Tuple

# 全局变量（竞赛风格）
MAXN = 100001  # 最大数据量
arr = [0] * MAXN      # 原始数组
help_arr = [0] * MAXN # 辅助数组
n = 0                 # 实际数据个数
```

```
# ==========================================================================
# 基础归并排序实现
# ==========================================================================

def merge_sort1(l: int, r: int) -> None:
    """
    归并排序递归版本
    【时间复杂度】O(n log n)
    【空间复杂度】O(n) + O(log n)递归栈
    【适用场景】代码简洁，易于理解
    【注意事项】Python 递归深度限制，大数据量可能栈溢出
    详细说明：
    1. 递归终止条件：当 l >= r 时，子数组只有一个元素，已经有序
    2. 分解：将数组从中间分成两部分
    3. 递归解决：对左右两部分分别进行归并排序
    4. 合并：将两个有序子数组合并成一个有序数组
    """
    if l == r:
        return
    m = (l + r) // 2
    merge_sort1(l, m)
    merge_sort1(m + 1, r)
    merge(l, m, r)



def merge_sort2() -> None:
    """
    归并排序非递归版本
    【时间复杂度】O(n log n)
    【空间复杂度】O(n) - 无递归栈开销
    【适用场景】大数据量排序，避免栈溢出
    【工程优势】更好的缓存局部性，易于理解
    详细说明：
    1. 从最小的子数组开始（长度为1），逐步扩大子数组大小
    2. 每次将相邻的两个子数组合并成一个更大的有序数组
    3. 重复此过程直到整个数组有序
    """
    step = 1
    while step < n:
        l = 0
        while l < n:
            m = l + step - 1
            if m + 1 >= n:  # 没有第二组，不需要合并
```

```
                break
            r = min(l + (step << 1) - 1, n - 1)
            merge(l, m, r)
            l = r + 1
        step <<= 1  # 步长翻倍


def merge(l: int, m: int, r: int) -> None:
    """
    合并两个有序数组的核心函数
    【参数说明】
    l: 左边界索引
    m: 中点索引
    r: 右边界索引
    【前置条件】[l, m] 和 [m+1, r] 已经各自有序
    【后置条件】[l, r] 整体有序
    【时间复杂度】O(n)
    【稳定性】稳定（使用<=比较保证）
    详细说明：
    1. 使用三个指针：i 指向辅助数组当前位置，a 指向左半部分，b 指向右半部分
    2. 比较左右两部分的元素，将较小的元素放入辅助数组
    3. 处理剩余元素：将未处理完的部分直接拷贝到辅助数组
    4. 将辅助数组的内容拷贝回原数组
    """
    i = l      # 辅助数组当前位置
    a = l      # 左半部分指针
    b = m + 1  # 右半部分指针

    # 双指针合并：比较两个部分的元素
    while a <= m and b <= r:
        if arr[a] <= arr[b]:  # 保证稳定性
            help_arr[i] = arr[a]
            a += 1
        else:
            help_arr[i] = arr[b]
            b += 1
        i += 1

    # 处理剩余元素（左半部分或右半部分）
    while a <= m:
        help_arr[i] = arr[a]
        a += 1
        i += 1
```

```
    while b <= r:
        help_arr[i] = arr[b]
        b += 1
        i += 1

    # 将合并结果复制回原数组
    for idx in range(l, r + 1):
        arr[idx] = help_arr[idx]


# ================================================================================
# 题目 1：LeetCode 912. 排序数组
# ================================================================================
def sort_array(nums: List[int]) -> List[int]:
    """
    LeetCode 912. 排序数组
    【题目链接】https://leetcode.cn/problems/sort-an-array/
    【题目描述】给定一个整数数组 nums，将该数组升序排列
    【输入示例】[5,2,3,1]
    【输出示例】[1,2,3,5]
    【时间复杂度】O(n log n)
    【空间复杂度】O(n)
    【是否最优】是 - 基于比较的排序算法下界
    详细说明：
    1. 使用切片操作简化数组处理
    2. 递归实现归并排序
    3. 合并过程中保持稳定性
    """
    if len(nums) <= 1:
        return nums

    def merge_sort_array(left: int, right: int) -> None:
        """递归排序函数"""
        if left >= right:
            return
        mid = left + (right - left) // 2
        merge_sort_array(left, mid)
        merge_sort_array(mid + 1, right)
        merge_array(left, mid, right)

    def merge_array(left: int, mid: int, right: int) -> None:
        """合并两个有序子数组"""
```

```
        # 使用切片复制到辅助数组（Python 特性）
        helper = nums[left:right+1]

        i = left      # 原数组指针
        p1 = 0        # 左半部分在 helper 中的指针
        p2 = mid - left + 1  # 右半部分在 helper 中的指针

        # 双指针合并
        while p1 <= mid - left and p2 <= right - left:
            if helper[p1] <= helper[p2]:
                nums[i] = helper[p1]
                p1 += 1
            else:
                nums[i] = helper[p2]
                p2 += 1
            i += 1

        # 处理剩余元素
        while p1 <= mid - left:
            nums[i] = helper[p1]
            p1 += 1
            i += 1

        while p2 <= right - left:
            nums[i] = helper[p2]
            p2 += 1
            i += 1

    merge_sort_array(0, len(nums) - 1)
    return nums


# 链表节点定义
class ListNode:
    def __init__(self, val: int = 0, next: Optional['ListNode'] = None):
        self.val = val
        self.next = next


# ================================================================================
# 题目 2：LeetCode 148. 排序链表
# ================================================================================
def sort_list(head: Optional[ListNode]) -> Optional[ListNode]:
```

```
    """
    LeetCode 148. 排序链表
    【题目链接】https://leetcode.cn/problems/sort-list/
    【题目描述】给你链表的头结点 head，请将其按升序排列并返回排序后的链表
    【算法优势】归并排序特别适合链表结构，只需修改指针
    【时间复杂度】O(n log n)
    【空间复杂度】O(log n) - 递归调用栈
    【是否最优】是 - 链表排序的最佳选择
    详细说明：
    1. 使用快慢指针找到链表中点
    2. 递归排序两个子链表
    3. 合并两个有序链表
    """
    if not head or not head.next:
        return head
    return process(head)


def process(head: Optional[ListNode]) -> Optional[ListNode]:
    """链表归并排序的递归处理函数"""
    if not head or not head.next:
        return head

    # 使用快慢指针找到链表中点（龟兔赛跑算法）
    slow = head
    fast = head
    prev = None

    while fast and fast.next:
        prev = slow
        slow = slow.next  # type: ignore
        fast = fast.next.next

    # 断开链表，分为左右两部分
    if prev:
        prev.next = None

    # 递归排序左右两部分
    left = process(head)
    right = process(slow)  # type: ignore

    # 合并两个有序链表
    return merge_list(left, right)
```

```
def merge_list(l1: Optional[ListNode], l2: Optional[ListNode]) -> Optional[ListNode]:
    """
    合并两个有序链表
    【技巧】使用哑节点(dummy node)简化边界处理
    【时间复杂度】O(m + n)
    【空间复杂度】O(1)
    详细说明：
    1. 使用哑节点简化边界处理
    2. 双指针比较合并
    3. 处理剩余节点
    """
    dummy = ListNode(0)  # 哑节点
    current = dummy

    # 双指针比较合并
    while l1 and l2:
        if l1.val <= l2.val:
            current.next = l1
            l1 = l1.next
        else:
            current.next = l2
            l2 = l2.next
        current = current.next

    # 连接剩余节点
    current.next = l1 if l1 else l2

    return dummy.next


# LeetCode 23. 合并 K 个升序链表
# 测试链接: https://leetcode.cn/problems/merge-k-sorted-lists/
# 时间复杂度: O(N * logk) - N 是所有节点总数，k 是链表数量
# 空间复杂度: O(logk) - 递归调用栈
def merge_k_lists(lists: List[Optional[ListNode]]) -> Optional[ListNode]:
    if not lists:
        return None
    return merge_k_lists_helper(lists, 0, len(lists) - 1)


def merge_k_lists_helper(lists: List[Optional[ListNode]], left: int, right: int) ->
```

```
Optional[ListNode]:
    if left == right:
        return lists[left]
    if left + 1 == right:
        return merge_list(lists[left], lists[right])
    mid = left + (right - left) // 2
    l1 = merge_k_lists_helper(lists, left, mid)
    l2 = merge_k_lists_helper(lists, mid + 1, right)
    return merge_list(l1, l2)


# LeetCode 88. 合并两个有序数组
# 测试链接: https://leetcode.cn/problems/merge-sorted-array/
# 时间复杂度: O(m + n)
# 空间复杂度: O(1)
def merge_sorted_array(nums1: List[int], m: int, nums2: List[int], n: int) -> None:
    i = m - 1  # nums1 的指针
    j = n - 1  # nums2 的指针
    k = m + n - 1  # 合并后数组的指针

    # 从后往前合并
    while i >= 0 and j >= 0:
        if nums1[i] > nums2[j]:
            nums1[k] = nums1[i]
            i -= 1
        else:
            nums1[k] = nums2[j]
            j -= 1
        k -= 1

    # 处理 nums2 剩余元素
    while j >= 0:
        nums1[k] = nums2[j]
        j -= 1
        k -= 1


# LeetCode 21. 合并两个有序链表
# 测试链接: https://leetcode.cn/problems/merge-two-sorted-lists/
# 时间复杂度: O(m + n)
# 空间复杂度: O(1)
def merge_two_lists(list1: Optional[ListNode], list2: Optional[ListNode]) -> Optional[ListNode]:
    dummy = ListNode(0)
```

```
    current = dummy

    while list1 and list2:
        if list1.val <= list2.val:
            current.next = list1
            list1 = list1.next
        else:
            current.next = list2
            list2 = list2.next
        current = current.next

    # 处理剩余节点
    current.next = list1 if list1 else list2

    return dummy.next


# LeetCode 315. 计算右侧小于当前元素的个数
# 测试链接: https://leetcode.cn/problems/count-of-smaller-numbers-after-self/
# 时间复杂度: O(n * logn)
# 空间复杂度: O(n)
def count_smaller(nums: List[int]) -> List[int]:
    result = []
    n = len(nums)
    if n == 0:
        return result

    counts = [0] * n
    indices = list(range(n))  # 索引数组

    helper = [0] * n
    helper_indices = [0] * n

    def merge_sort_count(left: int, right: int) -> None:
        if left >= right:
            return

        mid = left + (right - left) // 2
        merge_sort_count(left, mid)
        merge_sort_count(mid + 1, right)
        merge_count(left, mid, right)

    def merge_count(left: int, mid: int, right: int) -> None:
```

```
    # 复制到辅助数组
    for i in range(left, right + 1):
        helper[i] = nums[i]
        helper_indices[i] = indices[i]

    i = left
    p1 = left
    p2 = mid + 1

    # 合并两个有序数组
    while p1 <= mid and p2 <= right:
        # 当左侧元素小于等于右侧元素时
        if helper[p1] <= helper[p2]:
            # 更新计数：右侧已处理的元素个数
            counts[helper_indices[p1]] += (p2 - (mid + 1))
            nums[i] = helper[p1]
            indices[i] = helper_indices[p1]
            p1 += 1
        else:
            nums[i] = helper[p2]
            indices[i] = helper_indices[p2]
            p2 += 1
        i += 1

    # 处理左侧剩余元素
    while p1 <= mid:
        # 更新计数：右侧所有元素都比它小
        counts[helper_indices[p1]] += (p2 - (mid + 1))
        nums[i] = helper[p1]
        indices[i] = helper_indices[p1]
        p1 += 1
        i += 1

    # 处理右侧剩余元素
    while p2 <= right:
        nums[i] = helper[p2]
        indices[i] = helper_indices[p2]
        p2 += 1
        i += 1

merge_sort_count(0, n - 1)

for count in counts:
```

```
        result.append(count)

    return result


# LeetCode 493. 翻转对
# 测试链接: https://leetcode.cn/problems/reverse-pairs/
# 时间复杂度: O(n * logn)
# 空间复杂度: O(n)
def reverse_pairs(nums: List[int]) -> int:
    if len(nums) <= 1:
        return 0

    helper = [0] * len(nums)

    def merge_sort_reverse_pairs(left: int, right: int) -> int:
        if left >= right:
            return 0

        mid = left + (right - left) // 2
        count = merge_sort_reverse_pairs(left, mid)
        count += merge_sort_reverse_pairs(mid + 1, right)
        count += merge_reverse_pairs(left, mid, right)

        return count

    def merge_reverse_pairs(left: int, mid: int, right: int) -> int:
        # 复制到辅助数组
        for i in range(left, right + 1):
            helper[i] = nums[i]

        count = 0
        j = mid + 1

        # 统计翻转对数量
        for i in range(left, mid + 1):
            # 对于 nums[i]，找到第一个满足 nums[i] > 2 * nums[j]的 j
            while j <= right and helper[i] > 2 * helper[j]:
                j += 1
            count += (j - (mid + 1))

        # 合并两个有序数组
        i = left
```

```
        p1 = left
        p2 = mid + 1

        while p1 <= mid and p2 <= right:
            if helper[p1] <= helper[p2]:
                nums[i] = helper[p1]
                p1 += 1
            else:
                nums[i] = helper[p2]
                p2 += 1
            i += 1

        while p1 <= mid:
            nums[i] = helper[p1]
            p1 += 1
            i += 1

        while p2 <= right:
            nums[i] = helper[p2]
            p2 += 1
            i += 1

        return count

    return merge_sort_reverse_pairs(0, len(nums) - 1)



# ==========================================================================
# 题目 3：LeetCode 315. 计算右侧小于当前元素的个数
# ==========================================================================
def count_smaller(nums: List[int]) -> List[int]:
    """
    LeetCode 315. 计算右侧小于当前元素的个数
    【题目链接】https://leetcode.cn/problems/count-of-smaller-numbers-after-self/
    【题目描述】统计每个元素右侧比它小的元素个数
    【算法核心】归并排序 + 索引数组
    【时间复杂度】O(n log n)
    【空间复杂度】O(n)
    【关键技巧】在合并过程中利用有序性统计右侧较小元素
    详细说明：
    1. 使用索引数组记录原始位置
    2. 在合并过程中统计右侧较小元素
    3. 利用有序性优化统计过程
```

```
"""
n = len(nums)
if n == 0:
    return []

# 初始化结果数组和索引数组
result = [0] * n
indices = list(range(n))  # 记录原始位置

def merge_sort_count(left: int, right: int) -> None:
    if left >= right:
        return

    mid = (left + right) // 2
    merge_sort_count(left, mid)
    merge_sort_count(mid + 1, right)

    # 合并并统计
    i, j = left, mid + 1
    temp = []
    temp_indices = []

    while i <= mid and j <= right:
        if nums[i] <= nums[j]:
            # 关键统计：当左侧元素出队时，统计右侧已处理的较小元素
            result[indices[i]] += j - (mid + 1)
            temp.append(nums[i])
            temp_indices.append(indices[i])
            i += 1
        else:
            temp.append(nums[j])
            temp_indices.append(indices[j])
            j += 1

    # 处理左侧剩余元素
    while i <= mid:
        result[indices[i]] += j - (mid + 1)
        temp.append(nums[i])
        temp_indices.append(indices[i])
        i += 1

    # 处理右侧剩余元素
    while j <= right:
```

```
            temp.append(nums[j])
            temp_indices.append(indices[j])
            j += 1

        # 更新原数组和索引数组
        for idx in range(left, right + 1):
            nums[idx] = temp[idx - left]
            indices[idx] = temp_indices[idx - left]

    merge_sort_count(0, n - 1)
    return result


# ==========================================================================
# 题目 4：LeetCode 493. 翻转对
# ==========================================================================
def reverse_pairs(nums: List[int]) -> int:
    """
    LeetCode 493. 翻转对
    【题目链接】https://leetcode.cn/problems/reverse-pairs/
    【题目描述】统计满足 i < j 且 nums[i] > 2*nums[j] 的翻转对数量
    【与 315 区别】统计条件更严格（2 倍关系）
    【关键技巧】先统计后合并，使用 long 防溢出
    【时间复杂度】O(n log n)
    【空间复杂度】O(n)
    详细说明：
    1. 在合并前先统计翻转对
    2. 利用有序性优化统计过程
    3. 使用双指针减少重复计算
    """
    if len(nums) <= 1:
        return 0

    def merge_sort_count(left: int, right: int) -> int:
        if left >= right:
            return 0

        mid = (left + right) // 2
        count = merge_sort_count(left, mid)
        count += merge_sort_count(mid + 1, right)

        # 先统计翻转对数量
        j = mid + 1
```

```
        for i in range(left, mid + 1):
            while j <= right and nums[i] > 2 * nums[j]:
                j += 1
            count += j - (mid + 1)

        # 后合并两个有序数组
        temp = []
        i, j = left, mid + 1
        while i <= mid and j <= right:
            if nums[i] <= nums[j]:
                temp.append(nums[i])
                i += 1
            else:
                temp.append(nums[j])
                j += 1

        temp.extend(nums[i:mid+1])
        temp.extend(nums[j:right+1])
        nums[left:right+1] = temp

        return count

    return merge_sort_count(0, len(nums) - 1)


# ================================================================================
# 题目 5：逆序对统计（剑指 Offer 51 风格）
# ================================================================================
def reverse_pairs_count(nums: List[int]) -> int:
    """
    剑指 Offer 51. 数组中的逆序对
    【题目链接】https://leetcode.cn/problems/shu-zu-zhong-de-ni-xu-dui-lcof/
    【题目描述】统计数组中的逆序对总数
    【算法核心】归并排序过程中统计逆序对
    【时间复杂度】O(n log n)
    【空间复杂度】O(n)
    详细说明：
    1. 在合并过程中统计逆序对
    2. 利用有序性优化统计过程
    3. 当右元素小于左元素时统计逆序对
    """
    if len(nums) <= 1:
        return 0
```

```
    def merge_sort_count(left: int, right: int) -> int:
        if left >= right:
            return 0

        mid = (left + right) // 2
        count = merge_sort_count(left, mid)
        count += merge_sort_count(mid + 1, right)

        # 合并并统计逆序对
        temp = []
        i, j = left, mid + 1

        while i <= mid and j <= right:
            if nums[i] <= nums[j]:
                temp.append(nums[i])
                i += 1
            else:
                # 关键统计：当右元素小于左元素时
                count += mid - i + 1
                temp.append(nums[j])
                j += 1

        temp.extend(nums[i:mid+1])
        temp.extend(nums[j:right+1])
        nums[left:right+1] = temp

        return count

    return merge_sort_count(0, len(nums) - 1)


# ================================================================================
# 扩展题目实现：更多归并排序应用
# ================================================================================

# 题目 6：LeetCode 2426. 满足不等式的数对数目
# 链接：https://leetcode.cn/problems/number-of-pairs-satisfying-inequality/
# 时间复杂度：O(n log n)
# 空间复杂度：O(n)
# 核心思想：翻转对变种，处理不等式条件
def number_of_pairs(nums1: List[int], nums2: List[int], diff: int) -> int:
    n = len(nums1)
```

```
    arr = [nums1[i] - nums2[i] for i in range(n)]  # 构造差值数组
    return count_pairs(arr, diff)

def count_pairs(arr: List[int], diff: int) -> int:
    if len(arr) <= 1:
        return 0
    helper = [0] * len(arr)
    return merge_sort_pairs(arr, helper, 0, len(arr) - 1, diff)

def merge_sort_pairs(arr: List[int], helper: List[int], left: int, right: int, diff: int) -> int:
    if left >= right:
        return 0

    mid = (left + right) // 2
    count = merge_sort_pairs(arr, helper, left, mid, diff)
    count += merge_sort_pairs(arr, helper, mid + 1, right, diff)

    # 统计满足条件的数对
    j = mid + 1
    for i in range(left, mid + 1):
        # 条件：arr[i] <= arr[j] + diff
        while j <= right and arr[i] <= arr[j] + diff:
            j += 1
        count += j - (mid + 1)

    # 合并两个有序数组
    merge_arrays(arr, helper, left, mid, right)
    return count

def merge_arrays(arr: List[int], helper: List[int], left: int, mid: int, right: int) -> None:
    for i in range(left, right + 1):
        helper[i] = arr[i]

    i, j, k = left, mid + 1, left
    while i <= mid and j <= right:
        if helper[i] <= helper[j]:
            arr[k] = helper[i]
            i += 1
        else:
            arr[k] = helper[j]
            j += 1
        k += 1
```

```
    while i <= mid:
        arr[k] = helper[i]
        i += 1
        k += 1

    while j <= right:
        arr[k] = helper[j]
        j += 1
        k += 1

# 题目 7：LeetCode 4. 寻找两个正序数组的中位数
# 链接：https://leetcode.cn/problems/median-of-two-sorted-arrays/
# 时间复杂度：O(log(min(m, n)))
# 空间复杂度：O(1)
# 核心思想：二分查找，不是归并排序但涉及有序数组合并思想
def find_median_sorted_arrays(nums1: List[int], nums2: List[int]) -> float:
    if len(nums1) > len(nums2):
        return find_median_sorted_arrays(nums2, nums1)

    m, n = len(nums1), len(nums2)
    left, right = 0, m

    while left <= right:
        i = (left + right) // 2
        j = (m + n + 1) // 2 - i

        max_left1 = float('-inf') if i == 0 else nums1[i - 1]
        min_right1 = float('inf') if i == m else nums1[i]
        max_left2 = float('-inf') if j == 0 else nums2[j - 1]
        min_right2 = float('inf') if j == n else nums2[j]

        if max_left1 <= min_right2 and max_left2 <= min_right1:
            if (m + n) % 2 == 0:
                return (max(max_left1, max_left2) + min(min_right1, min_right2)) / 2.0
            else:
                return max(max_left1, max_left2)
        elif max_left1 > min_right2:
            right = i - 1
        else:
            left = i + 1

    return 0.0
```

```python
# 题目 8：外部排序模拟 - 多路归并
# 模拟处理大规模数据，无法一次性装入内存的情况
def external_sort_simulation(large_array: List[int], memory_limit: int) -> None:
    n = len(large_array)
    chunk_size = memory_limit
    num_chunks = (n + chunk_size - 1) // chunk_size

    # 模拟分块排序（实际中会写入临时文件）
    for i in range(num_chunks):
        start = i * chunk_size
        end = min(start + chunk_size, n)
        # 对当前块进行排序（模拟内部排序）
        large_array[start:end] = sorted(large_array[start:end])

    print(f"外部排序模拟完成，处理数据量: {n}")

# ================================================================================
# 性能测试与优化
# ================================================================================

import time
import random

# 性能测试：测试不同规模数据的排序性能
def performance_test():
    print("=== 性能测试 ===")

    sizes = [1000, 10000, 100000, 1000000]
    for size in sizes:
        test_data = generate_random_array(size)

        start_time = time.time()
        sorted_data = sorted(test_data)  # 使用内置排序作为基准
        end_time = time.time()

        duration = (end_time - start_time) * 1000  # 转换为毫秒
        print(f"数据量: {size:,}, 耗时: {duration:.2f} ms")

# 生成随机测试数组
def generate_random_array(size: int) -> List[int]:
    return [random.randint(0, size * 10) for _ in range(size)]

# 边界测试：测试各种边界情况
```

```
def boundary_test():
    print("=== 边界测试 ===")

    # 测试空数组
    test_case([], "空数组")

    # 测试单元素数组
    test_case([1], "单元素数组")

    # 测试已排序数组
    test_case([1, 2, 3, 4, 5], "已排序数组")

    # 测试逆序数组
    test_case([5, 4, 3, 2, 1], "逆序数组")

    # 测试重复元素数组
    test_case([2, 2, 1, 1, 3, 3], "重复元素数组")

    # 测试大数数组
    test_case([10**9, -10**9, 0], "极值数组")

def test_case(input_data: List[int], description: str):
    result = sorted(input_data)
    passed = is_sorted(result)
    print(f"{description}测试: {'✓ PASSED' if passed else '✗ FAILED'}")

# 检查数组是否有序
def is_sorted(arr: List[int]) -> bool:
    return all(arr[i] <= arr[i + 1] for i in range(len(arr) - 1))

# ================================================================================
# 调试工具方法
# ================================================================================

# 调试打印：打印数组内容（用于调试）
def print_array(arr: List[int], label: str):
    print(f"{label}: [{', '.join(map(str, arr[:min(10, len(arr))]))}" +
          ("..." if len(arr) > 10 else "") + "]")

# ================================================================================
# 单元测试增强版
# ================================================================================
```

```
def test_basic_sort():
    """测试基础排序功能"""
    test_data = [5, 2, 3, 1, 4]
    expected = [1, 2, 3, 4, 5]
    result = sort_array(test_data.copy())
    assert result == expected, f"Expected {expected}, but got {result}"
    print("✓ 基础排序测试通过")

def test_reverse_pairs():
    """测试逆序对统计"""
    test_data = [7, 5, 6, 4]
    expected = 5
    result = reverse_pairs_count(test_data.copy())
    assert result == expected, f"Expected {expected}, but got {result}"
    print("✓ 逆序对统计测试通过")

def test_count_smaller():
    """测试右侧较小元素统计"""
    test_data = [5, 2, 6, 1]
    expected = [2, 1, 1, 0]
    result = count_smaller(test_data.copy())
    assert result == expected, f"Expected {expected}, but got {result}"
    print("✓ 右侧较小元素统计测试通过")

def test_number_of_pairs():
    """测试满足不等式的数对数目"""
    nums1 = [3, 2, 5]
    nums2 = [2, 2, 1]
    diff = 1
    expected = 3
    result = number_of_pairs(nums1, nums2, diff)
    assert result == expected, f"Expected {expected}, but got {result}"
    print("✓ 不等式数对统计测试通过")

def run_comprehensive_tests():
    """运行全面测试"""
    print("=== 开始全面测试 ===")
    test_basic_sort()
    test_reverse_pairs()
    test_count_smaller()
    test_number_of_pairs()
    boundary_test()
    print("=== 测试完成 ===")
```

```
# ==============================================================================
# 主函数：支持多种运行模式
# ==============================================================================

if __name__ == "__main__":
    import sys

    if len(sys.argv) > 1 and sys.argv[1] == "test":
        # 测试模式
        run_comprehensive_tests()
    elif len(sys.argv) > 1 and sys.argv[1] == "perf":
        # 性能测试模式
        performance_test()
    else:
        # 默认模式：运行基础测试
        run_comprehensive_tests()
        print("\n使用 'python Code01_MergeSort.py test' 运行全面测试")
        print("使用 'python Code01_MergeSort.py perf' 运行性能测试")

# ==============================================================================
# Python 工程化考量总结
# ==============================================================================

"""
【Python 工程化最佳实践】
1. 类型注解：使用 typing 模块提高代码可读性
2. 文档字符串：为每个函数编写详细的文档说明
3. 异常处理：合理使用 try-except 处理异常情况
4. 代码风格：遵循 PEP8 编码规范
5. 模块化设计：将功能分解为独立的函数和模块

【Python 语言特性优势】
1. 简洁语法：代码量少，开发效率高
2. 动态类型：无需声明变量类型，灵活性强
3. 丰富库：标准库和第三方库功能强大
4. 跨平台：一次编写，到处运行

【Python 性能优化】
1. 使用内置函数：如 sorted()代替手动实现排序
2. 列表推导：比循环更高效
3. 生成器：节省内存，适合大数据处理
4. 局部变量：访问速度比全局变量快
```

【调试技巧】
1. 使用 pdb 调试器：设置断点、单步执行
2. 打印调试：使用 print 输出关键变量
3. 断言检查：使用 assert 验证中间结果
4. 日志记录：使用 logging 模块记录运行信息

更多题目请参考同目录下的 MERGE_SORT_PROBLEMS.md 文件
"""

```
# ================================================================================
# 单元测试函数
# ================================================================================
def test_basic_sort():
    """测试基础排序功能"""
    test_data = [5, 2, 3, 1, 4]
    expected = [1, 2, 3, 4, 5]
    result = sort_array(test_data.copy())
    assert result == expected, f"Expected {expected}, but got {result}"
    print("✓ Basic sort test passed")

def test_reverse_pairs():
    """测试逆序对统计"""
    test_data = [7, 5, 6, 4]
    expected = 5
    result = reverse_pairs_count(test_data.copy())
    assert result == expected, f"Expected {expected}, but got {result}"
    print("✓ Reverse pairs test passed")

def test_count_smaller():
    """测试右侧较小元素统计"""
    test_data = [5, 2, 6, 1]
    expected = [2, 1, 1, 0]
    result = count_smaller(test_data.copy())
    assert result == expected, f"Expected {expected}, but got {result}"
    print("✓ Count smaller test passed")

def run_tests():
    """运行所有测试"""
    print("Running tests...")
    test_basic_sort()
    test_reverse_pairs()
```

```
    test_count_smaller()
    print("All tests passed! ✓")

# 取消注释以下行来运行测试
# if __name__ == "__main__":
#     run_tests()
```

==========

文件: Code02_MergeSort.java

==========

```
package class021;

// ======================================
// 归并排序专题：分治策略的经典应用
// ======================================
//
// 【核心思想】
// 归并排序是一种基于分治策略的稳定排序算法：
// 1. 分解（Divide）：将待排序数组递归地分成两个子数组
// 2. 解决（Conquer）：递归地对子数组进行排序
// 3. 合并（Combine）：将两个已排序的子数组合并成一个有序数组
//
// 【时间复杂度】O(n log n) - 所有情况（最好、最坏、平均）都是这个复杂度
// 【空间复杂度】O(n) - 需要额外的辅助数组
// 【稳定性】稳定 - 相等元素的相对位置在排序后不会改变
//
// 【适用场景】
// 1. 大数据量排序 - 时间复杂度稳定
// 2. 外部排序 - 适合处理无法一次性装入内存的大数据
// 3. 稳定排序需求 - 需要保持相等元素相对顺序
// 4. 链表排序 - 特别适合链表结构，只需修改指针
// 5. 逆序对/翻转对统计 - 利用合并过程高效统计
//
// 【工程化考量】
// 1. 边界处理：空数组、单元素数组、重复元素
// 2. 异常处理：null 输入检查
// 3. 性能优化：小数组使用插入排序、非递归版本避免栈溢出
// 4. 内存管理：复用辅助数组，避免频繁内存分配
// 5. 线程安全：注意辅助数组的并发访问问题
//
// 【与机器学习的联系】
// 1. 外部排序用于大规模数据预处理
```

```
// 2. 归并思想用于分布式排序（MapReduce）
// 3. 稳定排序保证数据预处理的可重现性
//
// 测试链接 : https://leetcode.cn/problems/sort-an-array/
//
// 【补充题目列表】（从各大算法平台搜集）
// 详细题目列表请参考同目录下的 MERGE_SORT_PROBLEMS.md 文件
// 包含 LeetCode、洛谷、牛客网、Codeforces 等平台的归并排序相关题目

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class Code02_MergeSort {

    // ========================================
    // 题目 1：LeetCode 912. 排序数组
    // ========================================
    // 【题目描述】给定一个整数数组 nums，将该数组升序排列。
    // 【输入】nums = [5,2,3,1]
    // 【输出】[1,2,3,5]
    // 【思路】直接应用归并排序的模板代码
    // 【时间复杂度】O(n log n) - 最优解
    // 【空间复杂度】O(n) - 辅助数组
    // 【是否最优】是 - 对于一般数据，归并排序是最优的比较排序之一
    public static int[] sortArray(int[] nums) {
        // 边界检查：处理 null 和小数组
        if (nums == null || nums.length <= 1) {
            return nums;
        }
        if (nums.length > 1) {
            // mergeSort1 为递归方法，直观易懂，但可能栈溢出
            // mergeSort2 为非递归方法，更安全，避免深递归栈溢出
            // mergeSort1(nums);
            mergeSort2(nums); // 推荐使用非递归版本
        }
        return nums;
    }

    // 辅助数组的最大长度
    public static int MAXN = 50001;

    // 辅助数组，用于合并操作
```

```
// 工程优化：复用同一个辅助数组，避免频繁分配内存
public static int[] help = new int[MAXN];

// ========================================
// 归并排序递归版（推荐理解版本）
// ========================================
// 【时间复杂度】O(n log n)
// 推导：T(n) = 2*T(n/2) + O(n)
//       每层递归时间O(n)，递归深度log(n)层
//       总时间 = O(n) * log(n) = O(n log n)
// 【空间复杂度】O(n) - 辅助数组 + O(log n)递归栈
// 【优点】代码简洁，逻辑清晰
// 【缺点】深度递归可能导致栈溢出
public static void mergeSort1(int[] arr) {
    // 边界检查
    if (arr == null || arr.length <= 1) {
        return;
    }
    sort(arr, 0, arr.length - 1);
}

// 递归排序的核心函数
// 参数: arr-待排序数组, l-左边界， r-右边界
// 分治策略：
// 1. 分：将[l,r]区间分为[l,m]和[m+1,r]
// 2. 治：递归排序左右两部分
// 3. 合：合并两个有序部分
public static void sort(int[] arr, int l, int r) {
    // 基础情况：只有一个元素，已经有序
    if (l == r) {
        return;
    }
    // 计算中点，避免溢出：使用 l+(r-l)/2 而非 (l+r)/2
    // 但在Java中int类型不会溢出为正数，所以(l+r)/2也可以
    int m = (l + r) / 2;
    // 递归排序左半部分
    sort(arr, l, m);
    // 递归排序右半部分
    sort(arr, m + 1, r);
    // 合并两个有序部分
    merge(arr, l, m, r);
}
```

```
// ======================================
// 归并排序非递归版（工程推荐版本）
// ======================================
// 【时间复杂度】O(n log n)
// 【空间复杂度】O(n) - 仅需辅助数组，无递归栈开销
// 【优点】
// 1. 避免递归栈溢出风险
// 2. 更好的缓存局部性（顺序访问）
// 3. 更容易优化（如并行化）
// 【实现思路】
// 从小到大模拟归并过程：
// step=1: 每 1 个为一组，两两合并
// step=2: 每 2 个为一组，两两合并
// step=4: 每 4 个为一组，两两合并
// ...直到整个数组有序
public static void mergeSort2(int[] arr) {
    // 边界检查
    if (arr == null || arr.length <= 1) {
        return;
    }
    int n = arr.length;
    // step 表示当前每组的大小，每次翻倍
    // step < n 保证至少还有两组需要合并
    // 总共执行 log(n) 轮
    for (int l, m, r, step = 1; step < n; step <<= 1) {
        // 每一轮中，从左往右合并所有相邻的组
        // 这一层循环总时间 O(n)
        l = 0;
        while (l < n) {
            // 第一组的右边界
            m = l + step - 1;
            // 如果没有第二组，则不需要合并
            if (m + 1 >= n) {
                break;
            }
            // 第二组的右边界，可能不足 step 个元素
            r = Math.min(l + (step << 1) - 1, n - 1);
            // 合并 [l,m] 和 [m+1,r]
            merge(arr, l, m, r);
            // 移动到下一对组
            l = r + 1;
        }
    }
```

```
}

// ======================================
// 合并两个有序数组（核心操作）
// ======================================
// 【参数】arr-原数组, l-左边界, m-中点, r-右边界
// 【前置条件】[l,m] 和 [m+1,r] 已经各自有序
// 【后置条件】[l,r] 整体有序
// 【时间复杂度】O(r-l+1) = O(n)
// 【空间复杂度】O(r-l+1) = O(n)
// 【核心思想】双指针归并：
// 1. 用两个指针分别指向左右两部分的开头
// 2. 每次比较两个指针指向的元素，将较小者放入结果
// 3. 移动对应指针，直到某一部分处理完
// 4. 将剩余部分直接拷贝
// 【稳定性保证】使用 <= 保证相等时左侧元素先被选中
public static void merge(int[] arr, int l, int m, int r) {
    // i: 辅助数组的当前位置
    // a: 左部分[l,m]的当前指针
    // b: 右部分[m+1,r]的当前指针
    int i = l;
    int a = l;
    int b = m + 1;

    // 双指针合并：当两个部分都还有元素时
    while (a <= m && b <= r) {
        // 关键：<= 保证稳定性（相等时左侧优先）
        // 这是归并排序为稳定排序的关键
        help[i++] = arr[a] <= arr[b] ? arr[a++] : arr[b++];
    }

    // 左侧指针、右侧指针，必有一个越界、另一个不越界
    // 将剩余部分直接拷贝到辅助数组
    while (a <= m) {
        help[i++] = arr[a++];
    }
    while (b <= r) {
        help[i++] = arr[b++];
    }

    // 将辅助数组的结果拷贝回原数组
    // 注意：只拷贝 [l,r] 这个范围
    for (i = l; i <= r; i++) {
```

```
            arr[i] = help[i];
        }
    }

    // ========================================
    // 题目 2：LeetCode 148. 排序链表
    // ========================================
    // 【题目来源】https://leetcode.cn/problems/sort-list/
    // 【题目描述】给你链表的头结点 head，请将其按升序排列并返回排序后的链表。
    // 【输入示例】head = [4,2,1,3]
    // 【输出示例】[1,2,3,4]
    // 【进阶要求】能否在 O(n log n) 时间复杂度和常数级空间复杂度下解决？
    // 【思路分析】
    // 1. 为什么链表排序首选归并排序？
    //    - 快速排序需要随机访问，链表不支持
    //    - 堆排序也需要随机访问
    //    - 归并排序只需顺序访问，完美适配链表
    // 2. 如何找到链表中点？
    //    - 使用快慢指针（龟兔赛跑）
    //    - 快指针每次移动 2 步，慢指针每次移动 1 步
    //    - 当快指针到达末尾时，慢指针到达中点
    // 3. 如何合并两个有序链表？
    //    - 使用双指针，比较节点值后连接
    // 【时间复杂度】O(n log n)
    //    - 递归深度 log(n)，每层遍历所有节点 O(n)
    // 【空间复杂度】O(log n) - 递归调用栈
    //    - 注意：没有额外的辅助数组，只修改指针
    // 【是否最优】是 - 达到了题目要求的上限
    // 【工程优化】
    //    - 可以改为非递归实现，空间复杂度降为 O(1)
    //    - 但代码复杂度增加，面试中递归版本已经足够
    public static class ListNode {
        int val;           // 节点值
        ListNode next;     // 下一个节点的指针
        ListNode() {}      // 默认构造函数
        ListNode(int val) { this.val = val; }  // 带值构造
        ListNode(int val, ListNode next) {     // 带值和指针构造
            this.val = val;
            this.next = next;
        }
    }

    public static ListNode sortList(ListNode head) {
```

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

// 链表归并排序的递归函数
// 【参数】head - 当前链表的头节点
// 【返回】排序后链表的头节点
// 【核心步骤】
// 1. 找到链表中点（快慢指针）
// 2. 切断链表，分为左右两部分
// 3. 递归排序左右两部分
// 4. 合并两个有序链表
private static ListNode process(ListNode head) {
    // 递归基础情况：单节点，已有序
    if (head.next == null) {
        return head;
    }

    // 步骤 1：使用快慢指针找到中点
    // slow：慢指针，每次移动 1 步
    // fast：快指针，每次移动 2 步
    // prev：记录 slow 的前一个节点，用于断开链表
    ListNode slow = head;
    ListNode fast = head;
    ListNode prev = null;

    // 快指针走两步，慢指针走一步
    // 当 fast 到达末尾时，slow 到达中点
    // 注意：这里的中点是偏左的
    // 例如 1->2->3->4，中点是 2
    while (fast != null && fast.next != null) {
        prev = slow;
        slow = slow.next;
        fast = fast.next.next;
    }

    // 步骤 2：断开链表
    // 将链表分为 [head, prev] 和 [slow, end]
    prev.next = null;  // 关键：切断连接，避免死循环

    // 步骤 3：递归排序左右两部分
```

```
    ListNode left = process(head);   // 排序左半部分
    ListNode right = process(slow);  // 排序右半部分

    // 步骤 4：合并两个有序链表
    return mergeList(left, right);
}

// ======================================
// 合并两个有序链表（链表版本 merge）
// ======================================
// 【题目】LeetCode 21. 合并两个有序链表
// 【参数】l1, l2 - 两个有序链表的头节点
// 【返回】合并后的有序链表头节点
// 【技巧】使用哑节点（dummy node）简化边界处理
// 【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) - 只使用常数个指针
// 【工程技巧】
// 1. 哑节点技巧：避免特殊处理空链表
// 2. 三元表达式处理剩余节点
private static ListNode mergeList(ListNode l1, ListNode l2) {
    // 哑节点技巧：创建一个虚拟头节点
    // 作用：
    // 1. 简化边界处理（不需要判断第一个节点）
    // 2. 最终返回 dummy.next 即为真正的头节点
    ListNode dummy = new ListNode(0);
    ListNode current = dummy;  // 当前指针，用于构建结果链表

    // 当两个链表都还有节点时，比较并连接较小者
    while (l1 != null && l2 != null) {
        // 比较两个节点的值
        if (l1.val <= l2.val) {
            // l1 较小，连接 l1
            current.next = l1;
            l1 = l1.next;  // l1 向后移动
        } else {
            // l2 较小，连接 l2
            current.next = l2;
            l2 = l2.next;  // l2 向后移动
        }
        current = current.next;  // current 向后移动
    }

    // 处理剩余节点
```

```
        // 此时至少有一个链表已经处理完
        // 另一个链表的剩余部分直接连接即可
        current.next = (l1 != null) ? l1 : l2;

        // 返回真正的头节点（跳过哑节点）
        return dummy.next;
    }

    // ========================================
    // 题目 3：LeetCode 23. 合并 K 个升序链表
    // ========================================
    // 【题目来源】https://leetcode.cn/problems/merge-k-sorted-lists/
    // 【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。
    // 【输入示例】lists = [[1,4,5],[1,3,4],[2,6]]
    // 【输出示例】[1,1,2,3,4,4,5,6]
    // 【解法对比】
    // 方法 1：顺序合并 - O(kN) - k 个链表，总节点数 N
    //       第 1 次合并：O(n1+n2)，第 2 次：O(n1+n2+n3)，...
    //       总时间 ≈ O(kN)
    // 方法 2：优先队列 - O(N log k)
    //       维护大小为 k 的小顶堆，每次取最小值 O(log k)
    //       总时间 O(N log k)
    // 方法 3：分治合并 - O(N log k) - 最优解
    //       两两合并，递归处理，类似归并排序
    // 【时间复杂度】O(N log k) - N 是所有节点总数，k 是链表数量
    // 推导：分治深度 log(k)，每层合并所有节点 O(N)
    // 【空间复杂度】O(log k) - 递归调用栈
    // 【是否最优】是 - 与优先队列方法时间复杂度相同，但代码更简洁
    // 【思路技巧】
    // - 把 K 个链表合并转化为多次两个链表合并
    // - 使用分治法减少合并次数
public static ListNode mergeKLists(ListNode[] lists) {
        // 边界检查：处理 null 或空数组
        if (lists == null || lists.length == 0) {
            return null;
        }
        // 分治合并：将 lists 分成两部分，分别处理后合并
        return mergeKListsHelper(lists, 0, lists.length - 1);
    }

    // 分治合并 K 个链表的辅助函数
    // 【参数】lists-链表数组, left-左边界, right-右边界
```

```
// 【返回】合并后的链表头节点
// 【算法逻辑】
// 1. 如果只有一个链表，直接返回
// 2. 如果有两个链表，合并它们
// 3. 其他情况，分成两部分递归处理，然后合并结果
private static ListNode mergeKListsHelper(ListNode[] lists, int left, int right) {
    // 基础情况 1：只有一个链表
    if (left == right) {
        return lists[left];
    }
    // 优化：如果恰好是两个链表，直接合并
    if (left + 1 == right) {
        return mergeList(lists[left], lists[right]);
    }
    // 分治：找中点，分成两部分
    int mid = left + (right - left) / 2;
    // 递归合并左半部分
    ListNode l1 = mergeKListsHelper(lists, left, mid);
    // 递归合并右半部分
    ListNode l2 = mergeKListsHelper(lists, mid + 1, right);
    // 合并两个结果
    return mergeList(l1, l2);
}

// ========================================
// 题目 4：LeetCode 88. 合并两个有序数组
// ========================================
// 【题目来源】https://leetcode.cn/problems/merge-sorted-array/
// 【题目描述】给你两个按非递减顺序排列的整数数组 nums1 和 nums2，
//          另有两个整数 m 和 n，分别表示 nums1 和 nums2 中的元素数目。
//          请你合并 nums2 到 nums1 中，使合并后的数组同样按非递减顺序排列。
// 【输入示例】nums1 = [1,2,3,0,0,0], m = 3, nums2 = [2,5,6], n = 3
// 【输出示例】[1,2,2,3,5,6]
// 【注意】nums1 的长度为 m + n，前 m 个元素是有效数据，后 n 个为 0 用于存放结果
// 【关键技巧】从后往前合并!
// 为什么？
// - 从前往后会覆盖 nums1 的有效数据
// - 从后往前可以直接利用 nums1 尾部的空位
// 【时间复杂度】O(m + n)
// 【空间复杂度】O(1) - 原地修改，不需额外空间
// 【是否最优】是 - 时间、空间都最优
// 【面试重点】这道题经常考查，关键是"从后往前"的思路
public static void merge(int[] nums1, int m, int[] nums2, int n) {
```

```
    // i: nums1 的有效数据的最后一个位置
    // j: nums2 的最后一个位置
    // k: 合并后数组的最后一个位置
    int i = m - 1;
    int j = n - 1;
    int k = m + n - 1;

    // 从后往前合并：每次选择较大的元素放到最后
    // 当两个数组都还有元素时
    while (i >= 0 && j >= 0) {
        // 比较两个元素，将较大者放到位置 k
        if (nums1[i] > nums2[j]) {
            nums1[k--] = nums1[i--];
        } else {
            // 注意：nums1[i] <= nums2[j]时选 nums2[j]
            // 这保证了稳定性
            nums1[k--] = nums2[j--];
        }
    }

    // 处理 nums2 剩余元素
    // 注意：如果 nums1 有剩余，不需要处理（已在正确位置）
    while (j >= 0) {
        nums1[k--] = nums2[j--];
    }
    // 为什么不需要处理 nums1 剩余？
    // 因为 nums1 剩余元素已经在正确的位置上！
}

// LeetCode 21. 合并两个有序链表
// 测试链接: https://leetcode.cn/problems/merge-two-sorted-lists/
public static ListNode mergeTwoLists(ListNode list1, ListNode list2) {
    ListNode dummy = new ListNode(0);
    ListNode current = dummy;

    while (list1 != null && list2 != null) {
        if (list1.val <= list2.val) {
            current.next = list1;
            list1 = list1.next;
        } else {
            current.next = list2;
            list2 = list2.next;
        }
```

```
        current = current.next;
    }

    // 处理剩余节点
    current.next = (list1 != null) ? list1 : list2;

    return dummy.next;
}

// ======================================
// 题目 5：LeetCode 315. 计算右侧小于当前元素的个数
// ======================================
// 【题目来源】https://leetcode.cn/problems/count-of-smaller-numbers-after-self/
// 【题目描述】给定一个整数数组 nums，按要求返回一个新数组 counts。
//          counts[i] 的值是 nums[i] 右侧小于 nums[i] 的元素的数量。
// 【输入示例】nums = [5,2,6,1]
// 【输出示例】[2,1,1,0]
// 解释：
//   5 的右侧有 2 个更小的元素 (2 和 1)
//   2 的右侧有 1 个更小的元素 (1)
//   6 的右侧有 1 个更小的元素 (1)
//   1 的右侧有 0 个更小的元素
// 【暗力解法】O(n^2) - 对每个元素，遍历其右侧所有元素
// 【最优解法】O(n log n) - 利用归并排序的合并过程
// 【核心思想】
// 1. 逆序对的变种：统计右侧比自己小的元素个数
// 2. 在归并排序的合并过程中：
//    - 左右两部分已各自有序
//    - 当左侧元素 <= 右侧元素时，右侧已处理的元素都比左侧小
//    - 用索引数组记录原始位置，统计结果
// 【时间复杂度】O(n log n)
// 【空间复杂度】O(n)
// 【是否最优】是 - 基于比较的算法下界是 O(n log n)
// 【难点】
// 1. 需要维护索引数组，因为排序后位置会变
// 2. 理解何时统计：左侧元素出队时
public static List<Integer> countSmaller(int[] nums) {
    List<Integer> result = new ArrayList<>();
    int n = nums.length;
    // 边界检查
    if (n == 0) return result;

    // counts[i]: 记录原始位置 i 右侧比它小的元素个数
```

```
    int[] counts = new int[n];
    // indices[i]: 当前位置 i 的元素原本在哪个位置
    int[] indices = new int[n];

    // 初始化索引数组：0,1,2,3...
    for (int i = 0; i < n; i++) {
        indices[i] = i;
    }

    // 辅助数组
    int[] helper = new int[n];
    int[] helperIndices = new int[n];

    // 归并排序 + 统计
    mergeSortCount(nums, indices, counts, helper, helperIndices, 0, n - 1);

    // 构造结果
    for (int count : counts) {
        result.add(count);
    }

    return result;
}

// 归并排序 + 统计右侧更小元素
// 【参数】
// nums: 原始数组(会被修改)
// indices: 索引数组(记录每个位置的元素原始位置)
// counts: 统计数组(结果)
// helper: 辅助数组
// helperIndices: 辅助索引数组
private static void mergeSortCount(int[] nums, int[] indices, int[] counts,
        int[] helper, int[] helperIndices, int left, int right) {
    // 递归基础情况
    if (left >= right) {
        return;
    }

    int mid = left + (right - left) / 2;
    // 递归处理左半部分
    mergeSortCount(nums, indices, counts, helper, helperIndices, left, mid);
    // 递归处理右半部分
    mergeSortCount(nums, indices, counts, helper, helperIndices, mid + 1, right);
```

```
        // 合并 + 统计
        mergeCount(nums, indices, counts, helper, helperIndices, left, mid, right);
    }

    // 合并 + 统计（核心逻辑）
    // 【核心思想】
    // 在合并[left,mid]和[mid+1,right]时，两部分已各自有序
    // 当左侧元素 nums[p1] <= 右侧元素 nums[p2]时：
    //   右侧从(mid+1)到(p2-1)的所有元素都 < nums[p1]
    //   且这些元素在原始数组中都在 nums[p1]右侧！
    //   因此 counts[indices[p1]] += (p2 - (mid+1))
    private static void mergeCount(int[] nums, int[] indices, int[] counts,
            int[] helper, int[] helperIndices, int left, int mid, int right) {
        // 步骤 1：复制到辅助数组
        for (int i = left; i <= right; i++) {
            helper[i] = nums[i];              // 复制值
            helperIndices[i] = indices[i];    // 复制索引
        }

        int i = left;   // 结果数组的当前位置
        int p1 = left;  // 左部分指针
        int p2 = mid + 1;  // 右部分指针

        // 步骤 2：合并两个有序数组，同时统计
        while (p1 <= mid && p2 <= right) {
            // 关键：当左侧元素 <= 右侧元素时
            if (helper[p1] <= helper[p2]) {
                // *** 统计逻辑在这里 ***
                // 右部分从(mid+1)到(p2-1)的元素都比 helper[p1]小
                // 且在原数组中在 helper[p1]的右侧
                // 所以累加到 counts 中
                counts[helperIndices[p1]] += (p2 - (mid + 1));

                // 将左侧元素放入结果
                nums[i] = helper[p1];
                indices[i] = helperIndices[p1];
                p1++;
            } else {
                // 右侧元素更小，放入结果
                // 注意：这里不统计，因为右侧元素在左侧元素右边
                nums[i] = helper[p2];
                indices[i] = helperIndices[p2];
                p2++;
```

```
        }
        i++;
    }

    // 步骤3：处理左侧剩余元素
    while (p1 <= mid) {
        // *** 重要：还要统计 ***
        // 右部分所有元素都已处理，都比当前左侧元素小
        counts[helperIndices[p1]] += (p2 - (mid + 1));

        nums[i] = helper[p1];
        indices[i] = helperIndices[p1];
        p1++;
        i++;
    }

    // 步骤4：处理右侧剩余元素
    // 不需要统计，直接放入
    while (p2 <= right) {
        nums[i] = helper[p2];
        indices[i] = helperIndices[p2];
        p2++;
        i++;
    }
}

// ========================================
// 题目6：LeetCode 493. 翻转对 (Reverse Pairs)
// ========================================
// 【题目来源】https://leetcode.cn/problems/reverse-pairs/
// 【题目描述】给定一个数组 nums，如果 i < j 且 nums[i] > 2*nums[j]，
//          我们就将 (i, j) 称作一个重要翻转对。
//          你需要返回给定数组中的重要翻转对的数量。
// 【输入示例】nums = [1,3,2,3,1]
// 【输出示例】2
// 解释：翻转对为 (3,1) 和 (3,1)
// 【与题目315的区别】
// - LeetCode 315: 统计右侧比自己小的元素个数
// - LeetCode 493: 统计右侧满足 nums[i] > 2*nums[j] 的对数
// 【核心思想】
// 1. 在归并排序的合并之前，先统计翻转对
// 2. 利用左右两部分已各自有序的特性
// 3. 对于左侧每个元素 nums[i]，用双指针找到右侧满足条件的元素个数
```

```
// 【时间复杂度】O(n log n)
// 【空间复杂度】O(n)
// 【是否最优】是
// 【难点】
// 1. 统计和合并是分开的两个过程
// 2. 需要注意溢出：使用 long 类型
public static int reversePairs(int[] nums) {
    // 边界检查
    if (nums == null || nums.length <= 1) {
        return 0;
    }
    // 辅助数组
    int[] helper = new int[nums.length];
    // 归并排序 + 统计翻转对
    return mergeSortReversePairs(nums, helper, 0, nums.length - 1);
}

// 归并排序 + 统计翻转对
// 【参数】nums-原数组, helper-辅助数组, left-左边界, right-右边界
// 【返回】[left,right]范围内的翻转对数量
private static int mergeSortReversePairs(int[] nums, int[] helper, int left, int right) {
    // 递归基础情况
    if (left >= right) {
        return 0;
    }

    int mid = left + (right - left) / 2;
    // 递归统计左半部分的翻转对
    int count = mergeSortReversePairs(nums, helper, left, mid);
    // 递归统计右半部分的翻转对
    count += mergeSortReversePairs(nums, helper, mid + 1, right);
    // 统计跨越左右两部分的翻转对 + 合并
    count += mergeReversePairs(nums, helper, left, mid, right);

    return count;
}

// 合并 + 统计翻转对（核心逻辑）
// 【关键点】1. 先统计，再合并！（与 315 不同）
//         2. 使用 long 防溢出
// 【算法流程】
// 第 1 步：统计翻转对（利用有序性）
// 第 2 步：合并两个有序数组
```

```
private static int mergeReversePairs(int[] nums, int[] helper, int left, int mid, int right) {
    // 步骤1：复制到辅助数组
    for (int i = left; i <= right; i++) {
        helper[i] = nums[i];
    }

    int count = 0;
    int j = mid + 1;  // 右部分指针

    // *** 第1步：统计翻转对 ***
    // 对于左部分每个元素，找右部分多少个元素满足 nums[i] > 2*nums[j]
    for (int i = left; i <= mid; i++) {
        // 关键：利用右部分有序性，用双指针
        // 对于每个 helper[i]，找到第一个满足 helper[i] > 2*helper[j] 的 j
        // 那么 [mid+1, j-1] 都满足条件
        while (j <= right && (long) helper[i] > 2 * (long) helper[j]) {
            j++;  // j不需要重置！因为helper[i]递增，helper[j]也递增
        }
        // 统计：从 mid+1 到 j-1 都满足条件
        count += (j - (mid + 1));
        // 注意：j不重置，下次循环继续使用！这是时间复杂度O(n)的关键
    }

    // *** 第2步：合并两个有序数组 ***
    // 注意：这里和普通的merge一样
    int i = left;
    int p1 = left;
    int p2 = mid + 1;

    while (p1 <= mid && p2 <= right) {
        if (helper[p1] <= helper[p2]) {
            nums[i++] = helper[p1++];
        } else {
            nums[i++] = helper[p2++];
        }
    }

    // 处理剩余元素
    while (p1 <= mid) {
        nums[i++] = helper[p1++];
    }

    while (p2 <= right) {
```

```
            nums[i++] = helper[p2++];
        }

        return count;
    }
}
```

==========

文件: Code02_MergeSortApplications.java
==========

```
package class021;

/**
 * 归并排序高级应用 - Java 版本
 *
 * 本文件包含归并排序在各种场景下的高级应用：
 * 1. 链表排序
 * 2. 外部排序模拟
 * 3. 多路归并
 * 4. 大数据处理优化
 *
 * 时间复杂度：O(n log n) 在各种场景下
 * 空间复杂度：根据具体实现有所不同
 *
 * 详细题目列表请参考同目录下的 MERGE_SORT_PROBLEMS.md 文件
 * 包含 LeetCode、洛谷、牛客网、Codeforces 等平台的归并排序相关题目
 */

import java.util.*;

public class Code02_MergeSortApplications {

    // ================================================================================
    // 链表相关应用
    // ================================================================================

    /**
     * 链表节点定义
     */
    public static class ListNode {
        int val;
        ListNode next;
```

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

/**
 * 题目 1：链表排序（LeetCode 148）
 * 链接：https://leetcode.cn/problems/sort-list/
 * 时间复杂度：O(n log n)
 * 空间复杂度：O(log n) 递归栈空间
 * 核心思想：归并排序天然适合链表结构
 * 详细说明：
 * 1. 使用快慢指针找到链表中点
 * 2. 递归排序两个子链表
 * 3. 合并两个有序链表
 */
public static ListNode sortList(ListNode head) {
    if (head == null || head.next == null) {
        return head;
    }

    // 使用快慢指针找到链表中点
    ListNode slow = head;
    ListNode fast = head;
    ListNode prev = null;

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

    // 分割链表
    if (prev != null) {
        prev.next = null;
    }

    // 递归排序两个子链表
    ListNode left = sortList(head);
    ListNode right = sortList(slow);

    // 合并两个有序链表
    return mergeTwoLists(left, right);
```

```
}

/**
 * 合并两个有序链表
 * 详细说明：
 * 1. 使用哑节点简化边界处理
 * 2. 双指针比较节点值后连接
 * 3. 处理剩余节点
 */
private static ListNode mergeTwoLists(ListNode l1, ListNode l2) {
    ListNode dummy = new ListNode(0);
    ListNode current = dummy;

    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;
    }

    current.next = (l1 != null) ? l1 : l2;
    return dummy.next;
}

/**
 * 题目 2：合并 K 个升序链表（LeetCode 23）
 * 链接：https://leetcode.cn/problems/merge-k-sorted-lists/
 * 时间复杂度：O(N log k)，其中 N 是总节点数，k 是链表数量
 * 空间复杂度：O(log k) 递归栈空间
 * 核心思想：分治法合并多个链表
 * 详细说明：
 * 1. 将 K 个链表分成两部分
 * 2. 递归处理两部分
 * 3. 合并结果
 */
public static ListNode mergeKLists(ListNode[] lists) {
    if (lists == null || lists.length == 0) return null;
    return mergeKListsHelper(lists, 0, lists.length - 1);
}
```

```
private static ListNode mergeKListsHelper(ListNode[] lists, int left, int right) {
    if (left == right) return lists[left];
    if (left + 1 == right) return mergeTwoLists(lists[left], lists[right]);

    int mid = left + (right - left) / 2;
    ListNode l1 = mergeKListsHelper(lists, left, mid);
    ListNode l2 = mergeKListsHelper(lists, mid + 1, right);
    return mergeTwoLists(l1, l2);
}

/**
 * 题目 3：排序奇升偶降链表
 * 问题描述：给定一个链表，奇数位置节点升序，偶数位置节点降序，要求整体排序
 * 时间复杂度：O(n log n)
 * 空间复杂度：O(1)
 * 详细说明：
 * 1. 分割奇偶链表
 * 2. 反转偶数链表
 * 3. 合并两个有序链表
 */
public static ListNode sortOddEvenList(ListNode head) {
    if (head == null || head.next == null) return head;

    // 分割奇偶链表
    ListNode oddHead = head;
    ListNode evenHead = head.next;
    ListNode odd = oddHead, even = evenHead;

    while (even != null && even.next != null) {
        odd.next = even.next;
        odd = odd.next;
        even.next = odd.next;
        even = even.next;
    }
    odd.next = null;

    // 反转偶数链表（因为它是降序的）
    evenHead = reverseList(evenHead);

    // 合并两个有序链表
    return mergeTwoLists(oddHead, evenHead);
}
```

```
/**
 * 反转链表
 * 详细说明：
 * 1. 使用三个指针：prev、current、next
 * 2. 逐个反转节点的指向
 */
private static ListNode reverseList(ListNode head) {
    ListNode prev = null;
    ListNode current = head;

    while (current != null) {
        ListNode next = current.next;
        current.next = prev;
        prev = current;
        current = next;
    }

    return prev;
}

// ========================================================================
// 外部排序和多路归并
// ========================================================================

/**
 * 题目 4：外部排序模拟
 * 问题描述：模拟处理大规模数据，无法一次性装入内存的情况
 * 核心思想：分块排序 + 多路归并
 * 详细说明：
 * 1. 将大数据分块
 * 2. 对每块进行内部排序
 * 3. 多路归并所有块
 */
public static void externalSortSimulation(int[] largeArray, int memoryLimit) {
    int n = largeArray.length;
    int chunkSize = memoryLimit;
    int numChunks = (n + chunkSize - 1) / chunkSize;

    System.out.println("开始外部排序模拟...");
    System.out.println("数据总量: " + n);
    System.out.println("内存限制: " + memoryLimit);
    System.out.println("分块数量: " + numChunks);
```

```
        // 模拟分块排序（实际中会写入临时文件）
        for (int i = 0; i < numChunks; i++) {
            int start = i * chunkSize;
            int end = Math.min(start + chunkSize, n);
            // 对当前块进行排序（模拟内部排序）
            Arrays.sort(largeArray, start, end);
            System.out.println("完成第 " + (i + 1) + " 块排序，范围: [" + start + ", " + (end -
1) + "]");
        }

        // 模拟多路归并（简化版本）
        System.out.println("开始多路归并...");
        int[] result = multiwayMerge(largeArray, chunkSize, numChunks);
        System.arraycopy(result, 0, largeArray, 0, n);

        System.out.println("外部排序模拟完成");
    }

    /**
     * 多路归并实现
     * 详细说明：
     * 1. 使用优先队列维护各块的当前最小元素
     * 2. 每次取出最小元素放入结果
     * 3. 将该元素所在块的下一个元素加入优先队列
     */
    private static int[] multiwayMerge(int[] array, int chunkSize, int numChunks) {
        int n = array.length;
        int[] result = new int[n];
        int[] pointers = new int[numChunks]; // 每个块的当前指针

        // 初始化指针
        for (int i = 0; i < numChunks; i++) {
            pointers[i] = i * chunkSize;
        }

        // 使用优先队列进行多路归并
        PriorityQueue<int[]> pq = new PriorityQueue<>((a, b) -> a[0] - b[0]);

        // 初始化优先队列
        for (int i = 0; i < numChunks; i++) {
            int start = i * chunkSize;
            int end = Math.min(start + chunkSize, n);
```

```
            if (start < n) {
                pq.offer(new int[]{array[start], i});
            }
        }

        // 归并过程
        int index = 0;
        while (!pq.isEmpty()) {
            int[] min = pq.poll();
            result[index++] = min[0];

            int chunkIndex = min[1];
            pointers[chunkIndex]++;
            int nextPos = pointers[chunkIndex];
            int chunkStart = chunkIndex * chunkSize;
            int chunkEnd = Math.min(chunkStart + chunkSize, n);

            if (nextPos < chunkEnd) {
                pq.offer(new int[]{array[nextPos], chunkIndex});
            }
        }

        return result;
    }

    /**
     * 题目5：K路归并排序
     * 问题描述：将K个有序数组合并成一个有序数组
     * 时间复杂度：O(N log k)，其中N是总元素数，k是数组数量
     * 详细说明：
     * 1. 使用优先队列维护各数组的当前最小元素
     * 2. 每次取出最小元素放入结果
     * 3. 将该元素所在数组的下一个元素加入优先队列
     */
    public static int[] mergeKSortedArrays(int[][] arrays) {
        if (arrays == null || arrays.length == 0) return new int[0];

        // 使用优先队列进行K路归并
        PriorityQueue<int[]> pq = new PriorityQueue<>((a, b) -> a[0] - b[0]);
        int totalSize = 0;

        // 初始化优先队列
        for (int i = 0; i < arrays.length; i++) {
```

```
        if (arrays[i].length > 0) {
            pq.offer(new int[]{arrays[i][0], i, 0});
            totalSize += arrays[i].length;
        }
    }

    int[] result = new int[totalSize];
    int index = 0;

    while (!pq.isEmpty()) {
        int[] min = pq.poll();
        result[index++] = min[0];

        int arrayIndex = min[1];
        int elementIndex = min[2] + 1;

        if (elementIndex < arrays[arrayIndex].length) {
            pq.offer(new int[]{arrays[arrayIndex][elementIndex], arrayIndex, elementIndex});
        }
    }

    return result;
}

// ================================================================================
// 大数据处理优化
// ================================================================================

/**
 * 题目 6：流数据中的中位数（LeetCode 295）
 * 链接：https://leetcode.cn/problems/find-median-from-data-stream/
 * 核心思想：使用两个堆维护数据流
 * 时间复杂度：添加 O(log n)，查询 O(1)
 * 详细说明：
 * 1. 使用最大堆存储较小的一半元素
 * 2. 使用最小堆存储较大的一半元素
 * 3. 保持两个堆的大小平衡
 */
public static class MedianFinder {
    private PriorityQueue<Integer> maxHeap; // 存储较小的一半
    private PriorityQueue<Integer> minHeap; // 存储较大的一半

    public MedianFinder() {
```

```
        maxHeap = new PriorityQueue<>(Collections.reverseOrder());
        minHeap = new PriorityQueue<>();
    }

    public void addNum(int num) {
        if (maxHeap.isEmpty() || num <= maxHeap.peek()) {
            maxHeap.offer(num);
        } else {
            minHeap.offer(num);
        }

        // 平衡两个堆
        if (maxHeap.size() > minHeap.size() + 1) {
            minHeap.offer(maxHeap.poll());
        } else if (minHeap.size() > maxHeap.size()) {
            maxHeap.offer(minHeap.poll());
        }
    }

    public double findMedian() {
        if (maxHeap.size() == minHeap.size()) {
            return (maxHeap.peek() + minHeap.peek()) / 2.0;
        } else {
            return maxHeap.peek();
        }
    }
}

/**
 * 题目 7：数据流中的第 K 大元素（LeetCode 703）
 * 链接：https://leetcode.cn/problems/kth-largest-element-in-a-stream/
 * 核心思想：使用最小堆维护前 K 大元素
 * 详细说明：
 * 1. 使用大小为 K 的最小堆
 * 2. 堆顶元素即为第 K 大元素
 */
public static class KthLargest {
    private PriorityQueue<Integer> minHeap;
    private int k;

    public KthLargest(int k, int[] nums) {
        this.k = k;
        this.minHeap = new PriorityQueue<>();
```

```
        for (int num : nums) {
            add(num);
        }
    }

    public int add(int val) {
        if (minHeap.size() < k) {
            minHeap.offer(val);
        } else if (val > minHeap.peek()) {
            minHeap.poll();
            minHeap.offer(val);
        }
        return minHeap.peek();
    }
}

// ================================================================================
// 并行归并排序优化
// ================================================================================

/**
 * 并行归并排序实现（简化版本）
 * 核心思想：使用 ForkJoin 框架实现并行排序
 * 详细说明：
 * 1. 当数组大小超过阈值时，并行处理
 * 2. 否则使用 Arrays.sort 进行串行排序
 * 3. 合并两个有序子数组
 */
public static class ParallelMergeSort {
    private static final int THRESHOLD = 1000; // 并行阈值

    public static void sort(int[] array) {
        if (array.length <= THRESHOLD) {
            Arrays.sort(array);
            return;
        }

        // 创建并行任务
        MergeSortTask task = new MergeSortTask(array, 0, array.length - 1);
        task.compute();
    }
```

```
private static class MergeSortTask extends java.util.concurrent.RecursiveAction {
    private final int[] array;
    private final int left;
    private final int right;

    public MergeSortTask(int[] array, int left, int right) {
        this.array = array;
        this.left = left;
        this.right = right;
    }

    @Override
    protected void compute() {
        if (right - left <= THRESHOLD) {
            Arrays.sort(array, left, right + 1);
            return;
        }

        int mid = left + (right - left) / 2;
        MergeSortTask leftTask = new MergeSortTask(array, left, mid);
        MergeSortTask rightTask = new MergeSortTask(array, mid + 1, right);

        invokeAll(leftTask, rightTask);
        merge(array, left, mid, right);
    }

    private void merge(int[] array, int left, int mid, int right) {
        int[] temp = new int[right - left + 1];
        int i = left, j = mid + 1, k = 0;

        while (i <= mid && j <= right) {
            if (array[i] <= array[j]) {
                temp[k++] = array[i++];
            } else {
                temp[k++] = array[j++];
            }
        }

        while (i <= mid) temp[k++] = array[i++];
        while (j <= right) temp[k++] = array[j++];

        System.arraycopy(temp, 0, array, left, temp.length);
    }
```

```
    }
}

// ================================================================================
// 测试方法
// ================================================================================

public static void testLinkedListSort() {
    System.out.println("=== 链表排序测试 ===");

    // 创建测试链表：4 -> 2 -> 1 -> 3
    ListNode head = new ListNode(4);
    head.next = new ListNode(2);
    head.next.next = new ListNode(1);
    head.next.next.next = new ListNode(3);

    ListNode sorted = sortList(head);

    // 验证排序结果
    boolean passed = true;
    ListNode current = sorted;
    int prev = Integer.MIN_VALUE;

    while (current != null) {
        if (current.val < prev) {
            passed = false;
            break;
        }
        prev = current.val;
        current = current.next;
    }

    System.out.println("链表排序测试: " + (passed ? "✓ PASSED" : "✗ FAILED"));
}

public static void testExternalSort() {
    System.out.println("=== 外部排序测试 ===");

    int[] testData = {5, 2, 8, 1, 9, 3, 7, 4, 6, 0};
    int memoryLimit = 3; // 模拟小内存情况

    externalSortSimulation(testData, memoryLimit);
```

```
    boolean passed = isSorted(testData);
    System.out.println("外部排序测试: " + (passed ? "✓ PASSED" : "✗ FAILED"));
}

public static void testKWayMerge() {
    System.out.println("=== K路归并测试 ===");

    int[][] arrays = {
        {1, 4, 7},
        {2, 5, 8},
        {3, 6, 9}
    };

    int[] result = mergeKSortedArrays(arrays);
    int[] expected = {1, 2, 3, 4, 5, 6, 7, 8, 9};

    boolean passed = Arrays.equals(result, expected);
    System.out.println("K路归并测试: " + (passed ? "✓ PASSED" : "✗ FAILED"));
}

public static void testMedianFinder() {
    System.out.println("=== 中位数查找器测试 ===");

    MedianFinder finder = new MedianFinder();
    int[] testData = {1, 2, 3, 4, 5};

    for (int num : testData) {
        finder.addNum(num);
    }

    double median = finder.findMedian();
    boolean passed = Math.abs(median - 3.0) < 1e-9;
    System.out.println("中位数查找器测试: " + (passed ? "✓ PASSED" : "✗ FAILED"));
}

private static boolean isSorted(int[] arr) {
    for (int i = 1; i < arr.length; i++) {
        if (arr[i] < arr[i - 1]) {
            return false;
        }
    }
    return true;
}
```

```
public static void runComprehensiveTests() {
    System.out.println("=== 开始高级应用测试 ===");
    testLinkedListSort();
    testExternalSort();
    testKWayMerge();
    testMedianFinder();
    System.out.println("=== 高级应用测试完成 ===");
}

// ================================================================================
// 主函数
// ================================================================================

public static void main(String[] args) {
    if (args.length > 0 && "test".equals(args[0])) {
        runComprehensiveTests();
    } else {
        runComprehensiveTests();
        System.out.println("\n 使用 'java Code02_MergeSortApplications test' 运行全面测试");
    }
}

/**
 * 【工程化考量总结】
 * 1. 内存管理：链表排序避免数组拷贝，节省空间
 * 2. 并行优化：大数据量时使用并行排序提升性能
 * 3. 外部排序：处理超出内存限制的大规模数据
 * 4. 流处理：实时处理数据流，支持动态查询
 * 5. 多路归并：高效合并多个有序序列
 *
 * 【面试重点】
 * 1. 链表排序：快慢指针找中点，分治思想
 * 2. 多路归并：优先队列的应用
 * 3. 外部排序：分块+归并的处理流程
 * 4. 流处理：堆数据结构的巧妙使用
 * 5. 并行优化：ForkJoin 框架的应用
 *
 * 【学习建议】
 * 1. 理解不同数据结构的排序特点
 * 2. 掌握多路归并的优先队列实现
 * 3. 学习外部排序的实际应用场景
 * 4. 实践并行编程提升性能
```

```
     * 5. 深入理解流数据处理算法
     *
     * 更多题目请参考同目录下的 MERGE_SORT_PROBLEMS.md 文件
     */
}

==========================================

文件: Code03_MergeSortAdvanced.java
==========================================
package class021;

/**
 * 归并排序进阶应用 - Java 版本
 *
 * 本文件包含归并排序的进阶应用和优化技巧：
 * 1. 原地归并排序优化
 * 2. 自然归并排序
 * 3. TimSort 算法原理
 * 4. 归并排序在数据库中的应用
 * 5. 归并排序与机器学习结合
 *
 * 时间复杂度：O(n log n) 在各种优化下
 * 空间复杂度：根据优化策略有所不同
 *
 * 详细题目列表请参考同目录下的 MERGE_SORT_PROBLEMS.md 文件
 * 包含 LeetCode、洛谷、牛客网、Codeforces 等平台的归并排序相关题目
 */

import java.util.*;

public class Code03_MergeSortAdvanced {

    // ================================================================================
    // 原地归并排序优化
    // ================================================================================

    /**
     * 题目 1：原地归并排序（空间复杂度 O(1)）
     * 核心思想：使用插入排序的思想进行原地合并
     * 时间复杂度：O(n²) 最坏情况，但实际性能较好
     * 空间复杂度：O(1)
     * 详细说明：
```

```
     * 1. 递归分解数组
     * 2. 原地合并两个有序子数组
     * 3. 通过元素移动实现原地合并
     */
    public static void inPlaceMergeSort(int[] arr, int left, int right) {
        if (left >= right) return;

        int mid = left + (right - left) / 2;
        inPlaceMergeSort(arr, left, mid);
        inPlaceMergeSort(arr, mid + 1, right);
        inPlaceMerge(arr, left, mid, right);
    }

    /**
     * 原地合并两个有序子数组
     * 使用插入排序的思想，但效率较低
     * 详细说明：
     * 1. 当左侧元素大于右侧元素时，将右侧元素插入到左侧合适位置
     * 2. 通过向右移动元素为插入元素腾出空间
     */
    private static void inPlaceMerge(int[] arr, int left, int mid, int right) {
        int i = left;
        int j = mid + 1;

        while (i <= mid && j <= right) {
            if (arr[i] <= arr[j]) {
                i++;
            } else {
                // 将arr[j]插入到arr[i]前面
                int value = arr[j];
                int index = j;

                // 向右移动元素
                while (index > i) {
                    arr[index] = arr[index - 1];
                    index--;
                }
                arr[i] = value;

                // 更新指针
                i++;
                mid++;
                j++;
```

```
        }
    }
}

/**
 * 题目 2：块交换原地归并排序
 * 核心思想：使用块交换技术减少元素移动次数
 * 时间复杂度：O(n log n)
 * 空间复杂度：O(1)
 * 详细说明：
 * 1. 识别需要交换的块
 * 2. 通过块交换减少元素移动次数
 * 3. 使用数组旋转技术实现块交换
 */
public static void blockSwapMergeSort(int[] arr, int left, int right) {
    if (left >= right) return;

    int mid = left + (right - left) / 2;
    blockSwapMergeSort(arr, left, mid);
    blockSwapMergeSort(arr, mid + 1, right);
    blockSwapMerge(arr, left, mid, right);
}

/**
 * 块交换合并算法
 * 详细说明：
 * 1. 找到需要交换的连续块
 * 2. 通过块交换实现合并
 */
private static void blockSwapMerge(int[] arr, int left, int mid, int right) {
    if (arr[mid] <= arr[mid + 1]) return; // 已经有序

    int i = left;
    int j = mid + 1;

    while (i <= mid && j <= right) {
        if (arr[i] <= arr[j]) {
            i++;
        } else {
            // 找到右侧有序块的起始位置
            int k = j;
            while (k <= right && arr[k] < arr[i]) {
                k++;
```

```
            }

            // 交换两个块
            rotate(arr, i, j - 1, k - 1);

            // 更新指针
            int shift = k - j;
            i += shift;
            mid += shift;
            j = k;
        }
    }
}

/**
 * 旋转数组：将 arr[left...mid]和 arr[mid+1...right]交换位置
 * 详细说明：
 * 1. 反转左半部分
 * 2. 反转右半部分
 * 3. 反转整个数组
 */
private static void rotate(int[] arr, int left, int mid, int right) {
    reverse(arr, left, mid);
    reverse(arr, mid + 1, right);
    reverse(arr, left, right);
}

/**
 * 反转数组指定区间
 * 详细说明：
 * 1. 使用双指针从两端向中间交换元素
 */
private static void reverse(int[] arr, int left, int right) {
    while (left < right) {
        int temp = arr[left];
        arr[left] = arr[right];
        arr[right] = temp;
        left++;
        right--;
    }
}

// ========================================================================
```

```
// 自然归并排序
// ================================================================================

/**
 * 题目 3：自然归并排序
 * 核心思想：利用数组中已有的有序序列（自然有序段）
 * 时间复杂度：O(n log k)，k 为自然有序段数量
 * 空间复杂度：O(n)
 * 详细说明：
 * 1. 识别数组中的自然有序段
 * 2. 合并相邻的自然有序段
 * 3. 重复此过程直到整个数组有序
 */
public static void naturalMergeSort(int[] arr) {
    if (arr == null || arr.length <= 1) return;

    int[] temp = new int[arr.length];
    boolean sorted = false;

    while (!sorted) {
        sorted = true;
        int left = 0;

        while (left < arr.length) {
            // 找到第一个自然有序段的结束位置
            int mid = findRun(arr, left);

            if (mid == arr.length - 1) {
                // 如果已经到末尾，说明整体有序
                if (left == 0) return;
                break;
            }

            // 找到第二个自然有序段的结束位置
            int right = findRun(arr, mid + 1);

            // 合并两个自然有序段
            mergeNatural(arr, temp, left, mid, right);
            left = right + 1;
            sorted = false;
        }
    }
}
```

```
/**
 * 查找自然有序段的结束位置
 * 详细说明：
 * 1. 识别升序或降序序列
 * 2. 对于降序序列进行反转
 */
private static int findRun(int[] arr, int start) {
    if (start >= arr.length - 1) return arr.length - 1;

    int i = start;
    // 升序序列
    if (arr[i] <= arr[i + 1]) {
        while (i < arr.length - 1 && arr[i] <= arr[i + 1]) {
            i++;
        }
    } else {
        // 降序序列，需要反转
        while (i < arr.length - 1 && arr[i] > arr[i + 1]) {
            i++;
        }
        // 反转降序序列为升序
        reverse(arr, start, i);
    }

    return i;
}

/**
 * 合并两个自然有序段
 * 详细说明：
 * 1. 使用辅助数组进行合并
 * 2. 将合并结果拷贝回原数组
 */
private static void mergeNatural(int[] arr, int[] temp, int left, int mid, int right) {
    int i = left, j = mid + 1, k = left;

    while (i <= mid && j <= right) {
        if (arr[i] <= arr[j]) {
            temp[k++] = arr[i++];
        } else {
            temp[k++] = arr[j++];
        }
```

```
    }

    while (i <= mid) temp[k++] = arr[i++];
    while (j <= right) temp[k++] = arr[j++];

    // 复制回原数组
    System.arraycopy(temp, left, arr, left, right - left + 1);
}

// ================================================================================
// TimSort 算法原理（Java 标准库使用的排序算法）
// ================================================================================

/**
 * 题目 4：TimSort 算法简化实现
 * TimSort 是归并排序和插入排序的混合算法
 * 核心思想：利用自然有序段 + 插入排序优化小数组 + 平衡归并
 * 时间复杂度：O(n log n) 最好情况 O(n)
 * 空间复杂度：O(n)
 * 详细说明：
 * 1. 识别自然 run（有序段）
 * 2. 对短 run 进行扩展
 * 3. 使用插入排序优化小数组
 * 4. 平衡归并所有 run
 */
public static void timSort(int[] arr) {
    final int MIN_MERGE = 32;

    if (arr.length < MIN_MERGE) {
        // 小数组使用插入排序
        insertionSort(arr, 0, arr.length - 1);
        return;
    }

    // 计算最小 run 长度
    int minRun = minRunLength(arr.length);

    // 分割成 run 并进行归并
    List<int[]> runs = new ArrayList<>();
    int i = 0;

    while (i < arr.length) {
        // 找到自然 run
```

```
        int runEnd = findTimRun(arr, i);

        // 如果run太短，扩展到minRun长度
        int runLength = runEnd - i + 1;
        if (runLength < minRun) {
            int force = Math.min(minRun, arr.length - i);
            insertionSort(arr, i, i + force - 1);
            runEnd = i + force - 1;
        }

        runs.add(new int[]{i, runEnd});
        i = runEnd + 1;
    }

    // 归并所有的run
    mergeAllRuns(arr, runs);
}

/**
 * 计算最小run长度
 * 详细说明：
 * 1. 基于数组长度计算合适的run长度
 * 2. 保证run长度在合理范围内
 */
private static int minRunLength(int n) {
    int r = 0;
    while (n >= 64) {
        r |= (n & 1);
        n >>= 1;
    }
    return n + r;
}

/**
 * 查找TimRun（考虑升序和严格降序）
 * 详细说明：
 * 1. 识别升序或严格降序序列
 * 2. 对于降序序列进行反转
 */
private static int findTimRun(int[] arr, int start) {
    if (start >= arr.length - 1) return arr.length - 1;

    int i = start;
```

```
    if (arr[i] <= arr[i + 1]) {
        // 弱升序（允许相等）
        while (i < arr.length - 1 && arr[i] <= arr[i + 1]) {
            i++;
        }
    } else {
        // 严格降序
        while (i < arr.length - 1 && arr[i] > arr[i + 1]) {
            i++;
        }
        reverse(arr, start, i);
    }

    return i;
}

/**
 * 插入排序（用于小数组优化）
 * 详细说明：
 * 1. 对指定区间进行插入排序
 * 2. 适用于小数组或短序列
 */
private static void insertionSort(int[] arr, int left, int right) {
    for (int i = left + 1; i <= right; i++) {
        int key = arr[i];
        int j = i - 1;

        while (j >= left && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}

/**
 * 归并所有的 run
 * 详细说明：
 * 1. 两两合并 run
 * 2. 直到只剩一个 run
 */
private static void mergeAllRuns(int[] arr, List<int[]> runs) {
    while (runs.size() > 1) {
```

```
        List<int[]> newRuns = new ArrayList<>();

        for (int i = 0; i < runs.size(); i += 2) {
            if (i + 1 < runs.size()) {
                int[] run1 = runs.get(i);
                int[] run2 = runs.get(i + 1);

                mergeTimRun(arr, run1[0], run1[1], run2[1]);
                newRuns.add(new int[]{run1[0], run2[1]});
            } else {
                newRuns.add(runs.get(i));
            }
        }

        runs = newRuns;
    }
}

/**
 * 合并两个 TimRun
 * 详细说明：
 * 1. 使用辅助数组进行合并
 * 2. 将合并结果拷贝回原数组
 */
private static void mergeTimRun(int[] arr, int left, int mid, int right) {
    int[] temp = new int[right - left + 1];
    int i = left, j = mid + 1, k = 0;

    while (i <= mid && j <= right) {
        if (arr[i] <= arr[j]) {
            temp[k++] = arr[i++];
        } else {
            temp[k++] = arr[j++];
        }
    }

    while (i <= mid) temp[k++] = arr[i++];
    while (j <= right) temp[k++] = arr[j++];

    System.arraycopy(temp, 0, arr, left, temp.length);
}

// ==========================================================================
```

```
// 归并排序在数据库中的应用
// ===================================================================

/**
 * 题目 5：外部排序优化 - 多阶段归并
 * 数据库排序中常用的优化技术
 * 核心思想：减少磁盘 I/O 次数，优化归并顺序
 * 详细说明：
 * 1. 分阶段进行多路归并
 * 2. 减少磁盘读写次数
 * 3. 优化内存使用
 */
public static class ExternalSortOptimized {
    private static final int MEMORY_SIZE = 1000; // 模拟内存大小
    private static final int BLOCK_SIZE = 100;   // 模拟磁盘块大小

    /**
     * 多阶段归并排序
     * 详细说明：
     * 1. 创建初始有序 run
     * 2. 分阶段进行多路归并
     * 3. 减少磁盘 I/O 次数
     */
    public static void multiPhaseMergeSort(int[] data) {
        // 模拟分块排序（实际中会写入磁盘）
        List<int[]> sortedRuns = createInitialRuns(data);

        // 多阶段归并
        while (sortedRuns.size() > 1) {
            sortedRuns = mergePhase(sortedRuns);
        }

        // 最终结果
        if (!sortedRuns.isEmpty()) {
            System.arraycopy(sortedRuns.get(0), 0, data, 0, data.length);
        }
    }

    /**
     * 创建初始有序 run
     * 详细说明：
     * 1. 将数据分块
     * 2. 对每块进行内部排序
```

```
 */
private static List<int[]> createInitialRuns(int[] data) {
    List<int[]> runs = new ArrayList<>();
    int n = data.length;

    for (int i = 0; i < n; i += MEMORY_SIZE) {
        int end = Math.min(i + MEMORY_SIZE, n);
        int[] chunk = Arrays.copyOfRange(data, i, end);
        Arrays.sort(chunk);
        runs.add(chunk);
    }

    return runs;
}

/**
 * 多阶段归并
 * 详细说明：
 * 1. 三路归并减少归并次数
 * 2. 优化磁盘 I/O
 */
private static List<int[]> mergePhase(List<int[]> runs) {
    List<int[]> mergedRuns = new ArrayList<>();

    for (int i = 0; i < runs.size(); i += 3) {
        if (i + 2 < runs.size()) {
            // 三路归并
            int[] merged = threeWayMerge(
                runs.get(i), runs.get(i + 1), runs.get(i + 2));
            mergedRuns.add(merged);
        } else if (i + 1 < runs.size()) {
            // 两路归并
            int[] merged = mergeTwoArrays(runs.get(i), runs.get(i + 1));
            mergedRuns.add(merged);
        } else {
            mergedRuns.add(runs.get(i));
        }
    }

    return mergedRuns;
}

/**
```

```
 * 三路归并
 * 详细说明：
 * 1. 同时比较三个数组的元素
 * 2. 选择最小元素放入结果
 */
private static int[] threeWayMerge(int[] a, int[] b, int[] c) {
    int[] result = new int[a.length + b.length + c.length];
    int i = 0, j = 0, k = 0, idx = 0;

    while (i < a.length && j < b.length && k < c.length) {
        if (a[i] <= b[j] && a[i] <= c[k]) {
            result[idx++] = a[i++];
        } else if (b[j] <= a[i] && b[j] <= c[k]) {
            result[idx++] = b[j++];
        } else {
            result[idx++] = c[k++];
        }
    }

    // 处理剩余元素
    while (i < a.length && j < b.length) {
        if (a[i] <= b[j]) {
            result[idx++] = a[i++];
        } else {
            result[idx++] = b[j++];
        }
    }

    while (i < a.length && k < c.length) {
        if (a[i] <= c[k]) {
            result[idx++] = a[i++];
        } else {
            result[idx++] = c[k++];
        }
    }

    while (j < b.length && k < c.length) {
        if (b[j] <= c[k]) {
            result[idx++] = b[j++];
        } else {
            result[idx++] = c[k++];
        }
    }
```

```
        while (i < a.length) result[idx++] = a[i++];
        while (j < b.length) result[idx++] = b[j++];
        while (k < c.length) result[idx++] = c[k++];

        return result;
    }

    /**
     * 合并两个数组
     * 详细说明：
     * 1. 双指针合并两个有序数组
     */
    private static int[] mergeTwoArrays(int[] a, int[] b) {
        int[] result = new int[a.length + b.length];
        int i = 0, j = 0, idx = 0;

        while (i < a.length && j < b.length) {
            if (a[i] <= b[j]) {
                result[idx++] = a[i++];
            } else {
                result[idx++] = b[j++];
            }
        }

        while (i < a.length) result[idx++] = a[i++];
        while (j < b.length) result[idx++] = b[j++];

        return result;
    }
}

// ==========================================================================
// 归并排序与机器学习结合
// ==========================================================================

/**
 * 题目 6：归并排序在推荐系统中的应用
 * 场景：多路归并用于合并多个推荐源的结果
 * 详细说明：
 * 1. 合并多个推荐源的结果
 * 2. 根据权重调整推荐顺序
 * 3. 去重处理
```

```
 */
public static class RecommendationSystem {

    /**
     * 多路归并推荐结果
     * @param recommendations 多个推荐源的结果列表
     * @param weights 各推荐源的权重
     * @return 合并后的推荐结果
     */
    public static List<Integer> mergeRecommendations(
            List<List<Integer>> recommendations, double[] weights) {

        // 使用优先队列进行多路归并
        PriorityQueue<RecommendationNode> pq = new PriorityQueue<>(
            (a, b) -> Double.compare(b.score, a.score));

        // 初始化优先队列
        for (int i = 0; i < recommendations.size(); i++) {
            List<Integer> recList = recommendations.get(i);
            if (!recList.isEmpty()) {
                pq.offer(new RecommendationNode(recList.get(0), i, 0, weights[i]));
            }
        }

        List<Integer> result = new ArrayList<>();
        Set<Integer> seen = new HashSet<>();

        while (!pq.isEmpty() && result.size() < 100) { // 限制结果数量
            RecommendationNode node = pq.poll();
            int item = node.item;

            // 去重
            if (!seen.contains(item)) {
                result.add(item);
                seen.add(item);
            }

            // 添加下一个元素
            int nextIndex = node.index + 1;
            List<Integer> recList = recommendations.get(node.source);
            if (nextIndex < recList.size()) {
                pq.offer(new RecommendationNode(
                    recList.get(nextIndex), node.source, nextIndex, weights[node.source]));
```

```
            }
        }

        return result;
    }

    /**
     * 推荐结果节点
     */
    private static class RecommendationNode {
        int item;
        int source;
        int index;
        double score;

        RecommendationNode(int item, int source, int index, double weight) {
            this.item = item;
            this.source = source;
            this.index = index;
            this.score = weight * (1.0 / (index + 1)); // 衰减权重
        }
    }
}

// ================================================================================
// 测试方法
// ================================================================================

public static void testInPlaceMergeSort() {
    System.out.println("=== 原地归并排序测试 ===");

    int[] test = {5, 2, 8, 1, 9, 3, 7, 4, 6, 0};
    int[] expected = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

    inPlaceMergeSort(test, 0, test.length - 1);
    boolean passed = Arrays.equals(test, expected);
    System.out.println("原地归并排序测试: " + (passed ? "✓ PASSED" : "✗ FAILED"));
}

public static void testNaturalMergeSort() {
    System.out.println("=== 自然归并排序测试 ===");

    int[] test = {1, 2, 3, 5, 4, 6, 8, 7}; // 包含自然有序段
```

```
        int[] expected = {1, 2, 3, 4, 5, 6, 7, 8};

        naturalMergeSort(test);
        boolean passed = Arrays.equals(test, expected);
        System.out.println("自然归并排序测试: " + (passed ? "✓ PASSED" : "✗ FAILED"));
    }

    public static void testTimSort() {
        System.out.println("=== TimSort 测试 ===");

        int[] test = {5, 2, 8, 1, 9, 3, 7, 4, 6, 0};
        int[] expected = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

        timSort(test);
        boolean passed = Arrays.equals(test, expected);
        System.out.println("TimSort 测试: " + (passed ? "✓ PASSED" : "✗ FAILED"));
    }

    public static void testExternalSortOptimized() {
        System.out.println("=== 外部排序优化测试 ===");

        int[] test = new int[1000];
        Random random = new Random();
        for (int i = 0; i < test.length; i++) {
            test[i] = random.nextInt(10000);
        }

        int[] copy = test.clone();
        ExternalSortOptimized.multiPhaseMergeSort(copy);

        boolean passed = isSorted(copy);
        System.out.println("外部排序优化测试: " + (passed ? "✓ PASSED" : "✗ FAILED"));
    }

    public static void testRecommendationSystem() {
        System.out.println("=== 推荐系统测试 ===");

        List<List<Integer>> recommendations = new ArrayList<>();
        recommendations.add(Arrays.asList(1, 3, 5, 7, 9));
        recommendations.add(Arrays.asList(2, 4, 6, 8, 10));
        recommendations.add(Arrays.asList(1, 2, 3, 4, 5));

        double[] weights = {0.5, 0.3, 0.2};
```

```
        List<Integer> result = RecommendationSystem.mergeRecommendations(recommendations,
weights);

        boolean passed = result.size() > 0 && result.size() <= 100;
        System.out.println("推荐系统测试: " + (passed ? "✓ PASSED" : "✗ FAILED"));
        System.out.println("推荐结果: " + result.subList(0, Math.min(10, result.size())));
    }

    private static boolean isSorted(int[] arr) {
        for (int i = 1; i < arr.length; i++) {
            if (arr[i] < arr[i - 1]) {
                return false;
            }
        }
        return true;
    }

    public static void runComprehensiveTests() {
        System.out.println("=== 开始进阶应用测试 ===");
        testInPlaceMergeSort();
        testNaturalMergeSort();
        testTimSort();
        testExternalSortOptimized();
        testRecommendationSystem();
        System.out.println("=== 进阶应用测试完成 ===");
    }

    // ================================================================================
    // 主函数
    // ================================================================================

    public static void main(String[] args) {
        if (args.length > 0 && "test".equals(args[0])) {
            runComprehensiveTests();
        } else {
            runComprehensiveTests();
            System.out.println("\n使用 'java Code03_MergeSortAdvanced test' 运行全面测试");
        }
    }

    /**
     * 【工程化考量总结】
```

```
     * 1. 原地优化：减少内存使用，适合资源受限环境
     * 2. 自然归并：利用数据局部有序性提升性能
     * 3. TimSort：工业级排序算法，平衡各种场景
     * 4. 外部排序：处理超大规模数据的技术
     * 5. 机器学习应用：将排序算法应用于推荐系统等场景
     *
     * 【面试重点】
     * 1. 理解各种优化策略的原理和适用场景
     * 2. 掌握 TimSort 算法的核心思想
     * 3. 能够分析不同优化策略的时间空间复杂度
     * 4. 了解排序算法在实际系统中的应用
     *
     * 【学习建议】
     * 1. 从基础归并排序开始，逐步学习各种优化
     * 2. 实践实现不同的优化版本，对比性能差异
     * 3. 研究标准库中的排序实现（如 Java 的 Arrays.sort）
     * 4. 学习排序算法在数据库、推荐系统等领域的应用
     * 5. 关注排序算法的最新研究进展
     *
     * 更多题目请参考同目录下的 MERGE_SORT_PROBLEMS.md 文件
     */
}

==========================================

文件: Main.java
==========================================
package class021;

// Merge Sort - ACM practice style
// Test link: https://www.luogu.com.cn/problem/P1177
// Please refer to the input/output handling in the following code
// This is a highly efficient way of handling input/output
// Submit the following code, change the class name to "Main" when submitting

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.io.StreamTokenizer;
import java.util.ArrayList;
import java.util.Arrays;
```

```
import java.util.List;

public class Main {

    // Global variables (competition style)
    public static int MAXN = 100001;
    public static int[] arr = new int[MAXN];
    public static int[] help = new int[MAXN];
    public static int n;

    // Merge two sorted arrays (core operation)
    public static void merge(int l, int m, int r) {
        int i = l;
        int a = l;
        int b = m + 1;

        // Merge using two pointers
        while (a <= m && b <= r) {
            help[i++] = arr[a] <= arr[b] ? arr[a++] : arr[b++];
        }

        // Handle remaining elements
        while (a <= m) {
            help[i++] = arr[a++];
        }
        while (b <= r) {
            help[i++] = arr[b++];
        }

        // Copy back to original array
        for (i = l; i <= r; i++) {
            arr[i] = help[i];
        }
    }

    // Recursive merge sort
    public static void mergeSort1(int l, int r) {
        if (l == r) {
            return;
        }
        int m = (l + r) / 2;
        mergeSort1(l, m);
        mergeSort1(m + 1, r);
```

```
        merge(l, m, r);
    }

    // Non-recursive merge sort
    public static void mergeSort2() {
        for (int step = 1; step < n; step <<= 1) {
            int l = 0;
            while (l < n) {
                int m = l + step - 1;
                if (m + 1 >= n) {
                    break;
                }
                int r = Math.min(l + (step << 1) - 1, n - 1);
                merge(l, m, r);
                l = r + 1;
            }
        }
    }

    // Problem 1: Basic merge sort (Luogu P1177)
    // Original main method commented out - use runTests instead
    /*
    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        StreamTokenizer in = new StreamTokenizer(br);
        PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

        in.nextToken();
        n = (int) in.nval;

        for (int i = 0; i < n; i++) {
            in.nextToken();
            arr[i] = (int) in.nval;
        }

        if (n > 0) {
            mergeSort2(); // Use non-recursive version
        }

        for (int i = 0; i < n - 1; i++) {
            out.print(arr[i] + " ");
        }
        if (n > 0) {
```

```
        out.println(arr[n - 1]);
    }
    out.flush();
    out.close();
    br.close();
}
*/

// Problem 2: Inversion counting (LeetCode 315 style)
public static long mergeSortCount(int left, int right) {
    if (left >= right) return 0;

    int mid = left + (right - left) / 2;
    long count = mergeSortCount(left, mid);
    count += mergeSortCount(mid + 1, right);

    // Merge and count inversions
    int i = left, j = mid + 1, k = left;
    while (i <= mid && j <= right) {
        if (arr[i] <= arr[j]) {
            help[k++] = arr[i++];
        } else {
            // Key counting: when right element is smaller than left element
            count += mid - i + 1;
            help[k++] = arr[j++];
        }
    }

    while (i <= mid) help[k++] = arr[i++];
    while (j <= right) help[k++] = arr[j++];

    for (int p = left; p <= right; p++) {
        arr[p] = help[p];
    }

    return count;
}

// Linked list node definition
public static class ListNode {
    int val;
    ListNode next;
    ListNode() {}
```

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

// Problem 3: Linked list merge sort (LeetCode 148)
public static ListNode sortList(ListNode head) {
    if (head == null || head.next == null) {
        return head;
    }

    // Find middle using fast-slow pointers
    ListNode slow = head;
    ListNode fast = head;
    ListNode prev = null;

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

    // Split the list
    if (prev != null) {
        prev.next = null;
    }

    // Recursively sort both halves
    ListNode left = sortList(head);
    ListNode right = sortList(slow);

    // Merge sorted lists
    return mergeTwoLists(left, right);
}

// Merge two sorted linked lists
public static ListNode mergeTwoLists(ListNode l1, ListNode l2) {
    ListNode dummy = new ListNode(0);
    ListNode current = dummy;

    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;
    }

    current.next = (l1 != null) ? l1 : l2;
    return dummy.next;
}

// Problem 4: Merge K sorted lists (LeetCode 23)
public static ListNode mergeKLists(ListNode[] lists) {
    if (lists == null || lists.length == 0) return null;
    return mergeKListsHelper(lists, 0, lists.length - 1);
}

private static ListNode mergeKListsHelper(ListNode[] lists, int left, int right) {
    if (left == right) return lists[left];
    if (left + 1 == right) return mergeTwoLists(lists[left], lists[right]);

    int mid = left + (right - left) / 2;
    ListNode l1 = mergeKListsHelper(lists, left, mid);
    ListNode l2 = mergeKListsHelper(lists, mid + 1, right);
    return mergeTwoLists(l1, l2);
}

// Problem 5: Count smaller numbers after self (LeetCode 315)
public static List<Integer> countSmaller(int[] nums) {
    List<Integer> result = new ArrayList<>();
    if (nums == null || nums.length == 0) return result;

    int n = nums.length;
    int[] counts = new int[n];
    int[] indices = new int[n];
    for (int i = 0; i < n; i++) indices[i] = i;

    int[] temp = new int[n];
    int[] tempIndices = new int[n];

    mergeSortCountSmaller(nums, indices, counts, 0, n - 1, temp, tempIndices);

    for (int count : counts) {
```

```
        result.add(count);
    }
    return result;
}

private static void mergeSortCountSmaller(int[] nums, int[] indices, int[] counts,
                                          int left, int right, int[] temp, int[] tempIndices) {
    if (left >= right) return;

    int mid = left + (right - left) / 2;
    mergeSortCountSmaller(nums, indices, counts, left, mid, temp, tempIndices);
    mergeSortCountSmaller(nums, indices, counts, mid + 1, right, temp, tempIndices);

    // Merge and count
    int i = left, j = mid + 1, k = left;
    while (i <= mid && j <= right) {
        if (nums[i] <= nums[j]) {
            counts[indices[i]] += (j - (mid + 1));
            temp[k] = nums[i];
            tempIndices[k] = indices[i];
            i++;
        } else {
            temp[k] = nums[j];
            tempIndices[k] = indices[j];
            j++;
        }
        k++;
    }

    while (i <= mid) {
        counts[indices[i]] += (j - (mid + 1));
        temp[k] = nums[i];
        tempIndices[k] = indices[i];
        i++;
        k++;
    }

    while (j <= right) {
        temp[k] = nums[j];
        tempIndices[k] = indices[j];
        j++;
        k++;
    }
```

```
        for (int p = left; p <= right; p++) {
            nums[p] = temp[p];
            indices[p] = tempIndices[p];
        }
    }

    // ================================================================================
    // 扩展题目实现：更多归并排序应用
    // ================================================================================

    // 题目 6：LeetCode 2426. 满足不等式的数对数目
    // 链接：https://leetcode.cn/problems/number-of-pairs-satisfying-inequality/
    // 时间复杂度：O(n log n)
    // 空间复杂度：O(n)
    // 核心思想：翻转对变种，处理不等式条件
    public static long numberOfPairs(int[] nums1, int[] nums2, int diff) {
        int n = nums1.length;
        int[] arr = new int[n];
        // 构造差值数组：nums1[i] - nums2[i]
        for (int i = 0; i < n; i++) {
            arr[i] = nums1[i] - nums2[i];
        }
        return countPairs(arr, diff);
    }

    private static long countPairs(int[] arr, int diff) {
        if (arr.length <= 1) return 0;

        int[] helper = new int[arr.length];
        return mergeSortPairs(arr, helper, 0, arr.length - 1, diff);
    }

    private static long mergeSortPairs(int[] arr, int[] helper, int left, int right, int diff) {
        if (left >= right) return 0;

        int mid = left + (right - left) / 2;
        long count = mergeSortPairs(arr, helper, left, mid, diff);
        count += mergeSortPairs(arr, helper, mid + 1, right, diff);

        // 统计满足条件的数对
        int j = mid + 1;
        for (int i = left; i <= mid; i++) {
```

```
            // 条件：arr[i] <= arr[j] + diff
            while (j <= right && arr[i] <= arr[j] + diff) {
                j++;
            }
            count += j - (mid + 1);
        }

        // 合并两个有序数组
        mergeArrays(arr, helper, left, mid, right);
        return count;
    }

    private static void mergeArrays(int[] arr, int[] helper, int left, int mid, int right) {
        System.arraycopy(arr, left, helper, left, right - left + 1);

        int i = left, j = mid + 1, k = left;
        while (i <= mid && j <= right) {
            if (helper[i] <= helper[j]) {
                arr[k++] = helper[i++];
            } else {
                arr[k++] = helper[j++];
            }
        }
        while (i <= mid) arr[k++] = helper[i++];
        while (j <= right) arr[k++] = helper[j++];
    }

    // 题目 7：LeetCode 4. 寻找两个正序数组的中位数
    // 链接：https://leetcode.cn/problems/median-of-two-sorted-arrays/
    // 时间复杂度：O(log(min(m, n)))
    // 空间复杂度：O(1)
    // 核心思想：二分查找，不是归并排序但涉及有序数组合并思想
    public static double findMedianSortedArrays(int[] nums1, int[] nums2) {
        if (nums1.length > nums2.length) {
            return findMedianSortedArrays(nums2, nums1);
        }

        int m = nums1.length, n = nums2.length;
        int left = 0, right = m;

        while (left <= right) {
            int i = left + (right - left) / 2;
            int j = (m + n + 1) / 2 - i;
```

```
        int maxLeft1 = (i == 0) ? Integer.MIN_VALUE : nums1[i - 1];
        int minRight1 = (i == m) ? Integer.MAX_VALUE : nums1[i];
        int maxLeft2 = (j == 0) ? Integer.MIN_VALUE : nums2[j - 1];
        int minRight2 = (j == n) ? Integer.MAX_VALUE : nums2[j];

        if (maxLeft1 <= minRight2 && maxLeft2 <= minRight1) {
            if ((m + n) % 2 == 0) {
                return (Math.max(maxLeft1, maxLeft2) + Math.min(minRight1, minRight2)) / 2.0;
            } else {
                return Math.max(maxLeft1, maxLeft2);
            }
        } else if (maxLeft1 > minRight2) {
            right = i - 1;
        } else {
            left = i + 1;
        }
    }
    return 0.0;
}

// 题目 8：外部排序模拟 - 多路归并
// 模拟处理大规模数据，无法一次性装入内存的情况
public static void externalSortSimulation(int[] largeArray, int memoryLimit) {
    int n = largeArray.length;
    int chunkSize = memoryLimit;
    int numChunks = (n + chunkSize - 1) / chunkSize;

    // 模拟分块排序（实际中会写入临时文件）
    for (int i = 0; i < numChunks; i++) {
        int start = i * chunkSize;
        int end = Math.min(start + chunkSize, n);
        // 对当前块进行排序（模拟内部排序）
        Arrays.sort(largeArray, start, end);
    }

    System.out.println("外部排序模拟完成，处理数据量: " + n);
}

// ==========================================================================
// 性能测试与优化
// ==========================================================================
```

```
// 性能测试：测试不同规模数据的排序性能
public static void performanceTest() {
    System.out.println("=== 性能测试 ===");

    int[] sizes = {1000, 10000, 100000, 1000000};
    for (int size : sizes) {
        int[] testData = generateRandomArray(size);

        long startTime = System.nanoTime();
        int[] result = sortArray(testData.clone());
        long endTime = System.nanoTime();

        double duration = (endTime - startTime) / 1e6; // 转换为毫秒
        System.out.printf("数据量: %,d, 耗时: %.2f ms%n", size, duration);
    }
}

// 生成随机测试数组
private static int[] generateRandomArray(int size) {
    int[] arr = new int[size];
    java.util.Random random = new java.util.Random();
    for (int i = 0; i < size; i++) {
        arr[i] = random.nextInt(size * 10);
    }
    return arr;
}

// 边界测试：测试各种边界情况
public static void boundaryTest() {
    System.out.println("=== 边界测试 ===");

    // 测试空数组
    testCase(new int[]{}, "空数组");

    // 测试单元素数组
    testCase(new int[]{1}, "单元素数组");

    // 测试已排序数组
    testCase(new int[]{1, 2, 3, 4, 5}, "已排序数组");

    // 测试逆序数组
    testCase(new int[]{5, 4, 3, 2, 1}, "逆序数组");
```

```
        // 测试重复元素数组
        testCase(new int[]{2, 2, 1, 1, 3, 3}, "重复元素数组");

        // 测试大数数组
        testCase(new int[]{Integer.MAX_VALUE, Integer.MIN_VALUE, 0}, "极值数组");
    }

    private static void testCase(int[] input, String description) {
        int[] result = sortArray(input.clone());
        boolean passed = isSorted(result);
        System.out.println(description + "测试: " + (passed ? "✓ PASSED" : "✗ FAILED"));
    }

    // 检查数组是否有序
    private static boolean isSorted(int[] arr) {
        for (int i = 1; i < arr.length; i++) {
            if (arr[i] < arr[i - 1]) {
                return false;
            }
        }
        return true;
    }

    // ================================================================================
    // 调试工具方法
    // ================================================================================

    // 调试打印：打印数组内容（用于调试）
    public static void printArray(int[] arr, String label) {
        System.out.print(label + ": [");
        for (int i = 0; i < Math.min(arr.length, 10); i++) {
            System.out.print(arr[i]);
            if (i < arr.length - 1 && i < 9) {
                System.out.print(", ");
            }
        }
        if (arr.length > 10) {
            System.out.print(", ...");
        }
        System.out.println("]");
    }

    // 断言检查：验证中间结果
```

```
public static void debugCheck(boolean condition, String message) {
    if (!condition) {
        System.err.println("调试错误: " + message);
    }
}

// ================================================================================
// 单元测试增强版
// ================================================================================

public static void testBasicSort() {
    int[] test = {5, 2, 3, 1, 4};
    int[] expected = {1, 2, 3, 4, 5};

    n = test.length;
    System.arraycopy(test, 0, arr, 0, n);
    mergeSort2();

    boolean passed = Arrays.equals(Arrays.copyOf(arr, n), expected);
    System.out.println("基础排序测试: " + (passed ? "✓ PASSED" : "✗ FAILED"));
}

public static void testInversionCount() {
    int[] test = {7, 5, 6, 4};
    long expected = 5;

    n = test.length;
    System.arraycopy(test, 0, arr, 0, n);
    long result = mergeSortCount(0, n - 1);

    boolean passed = (result == expected);
    System.out.println("逆序对统计测试: " + (passed ? "✓ PASSED" : "✗ FAILED"));
}

public static void testReversePairs() {
    int[] test = {1, 3, 2, 3, 1};
    int expected = 2;

    int result = reversePairs(test.clone());
    boolean passed = (result == expected);
    System.out.println("翻转对统计测试: " + (passed ? "✓ PASSED" : "✗ FAILED"));
}
```

```
public static void testCountSmaller() {
    int[] test = {5, 2, 6, 1};
    List<Integer> expected = Arrays.asList(2, 1, 1, 0);

    List<Integer> result = countSmaller(test.clone());
    boolean passed = result.equals(expected);
    System.out.println("右侧较小元素统计测试: " + (passed ? "✓ PASSED" : "✗ FAILED"));
}

public static void runComprehensiveTests() {
    System.out.println("=== 开始全面测试 ===");
    testBasicSort();
    testInversionCount();
    testReversePairs();
    testCountSmaller();
    boundaryTest();
    System.out.println("=== 测试完成 ===");
}

// ================================================================================
// 主函数：支持多种运行模式
// ================================================================================

public static void main(String[] args) {
    if (args.length > 0 && "test".equals(args[0])) {
        // 测试模式
        runComprehensiveTests();
    } else if (args.length > 0 && "perf".equals(args[0])) {
        // 性能测试模式
        performanceTest();
    } else {
        // 默认模式：运行基础测试
        runComprehensiveTests();
        System.out.println("使用 'java Main test' 运行全面测试");
        System.out.println("使用 'java Main perf' 运行性能测试");
    }
}

// ================================================================================
// 工程化考量总结
// ================================================================================

/**
```

```
 * 【工程化最佳实践】
 * 1. 异常处理：对所有输入进行边界检查
 * 2. 性能优化：根据数据规模选择合适的算法版本
 * 3. 内存管理：合理使用全局变量避免频繁分配
 * 4. 代码可读性：清晰的注释和命名规范
 * 5. 测试覆盖：全面的单元测试和边界测试
 * 6. 文档完善：详细的 API 文档和使用说明
 *
 * 【面试重点】
 * 1. 时间复杂度分析：能够详细推导 O(n log n)
 * 2. 空间复杂度分析：理解递归栈和辅助数组的影响
 * 3. 稳定性证明：解释为什么归并排序是稳定的
 * 4. 变种问题：掌握逆序对、翻转对等衍生问题
 * 5. 工程优化：讨论大数据量下的优化策略
 *
 * 【学习建议】
 * 1. 先理解递归版本，再掌握非递归版本
 * 2. 通过画图理解分治和合并过程
 * 3. 多做练习题，特别是逆序对相关题目
 * 4. 尝试实现并行版本提升性能
 * 5. 学习标准库中的排序实现对比
 *
 * 更多题目请参考同目录下的 MERGE_SORT_PROBLEMS.md 文件
 */

// ================================================================================
// 缺失的方法实现：sortArray 和 reversePairs
// ================================================================================

/**
 * 排序数组 - LeetCode 912
 * 时间复杂度：O(n log n)
 * 空间复杂度：O(n)
 */
public static int[] sortArray(int[] nums) {
    if (nums == null || nums.length <= 1) {
        return nums;
    }
    // 使用非递归归并排序
    mergeSort2(nums);
    return nums;
}
```

```
// 非递归归并排序（接收数组参数版本）
public static void mergeSort2(int[] nums) {
    if (nums == null || nums.length <= 1) {
        return;
    }
    int n = nums.length;
    int[] helper = new int[n];

    for (int step = 1; step < n; step <<= 1) {
        int l = 0;
        while (l < n) {
            int m = l + step - 1;
            if (m + 1 >= n) {
                break;
            }
            int r = Math.min(l + (step << 1) - 1, n - 1);
            merge(nums, helper, l, m, r);
            l = r + 1;
        }
    }
}

// 合并两个有序数组（接收数组参数版本）
public static void merge(int[] nums, int[] helper, int l, int m, int r) {
    int i = l;
    int a = l;
    int b = m + 1;

    // 合并两个有序部分
    while (a <= m && b <= r) {
        helper[i++] = nums[a] <= nums[b] ? nums[a++] : nums[b++];
    }

    // 处理剩余元素
    while (a <= m) {
        helper[i++] = nums[a++];
    }
    while (b <= r) {
        helper[i++] = nums[b++];
    }

    // 复制回原数组
    for (i = l; i <= r; i++) {
```

```
            nums[i] = helper[i];
        }
    }

    /**
     * 统计逆序对 - LeetCode 493
     * 时间复杂度：O(n log n)
     * 空间复杂度：O(n)
     */
    public static int reversePairs(int[] nums) {
        if (nums == null || nums.length <= 1) {
            return 0;
        }
        int[] helper = new int[nums.length];
        return mergeSortReversePairs(nums, helper, 0, nums.length - 1);
    }

    // 归并排序 + 统计逆序对
    private static int mergeSortReversePairs(int[] nums, int[] helper, int left, int right) {
        if (left >= right) {
            return 0;
        }

        int mid = left + (right - left) / 2;
        int count = mergeSortReversePairs(nums, helper, left, mid);
        count += mergeSortReversePairs(nums, helper, mid + 1, right);
        count += mergeReversePairs(nums, helper, left, mid, right);

        return count;
    }

    // 合并 + 统计逆序对
    private static int mergeReversePairs(int[] nums, int[] helper, int left, int mid, int right)
{
        for (int i = left; i <= right; i++) {
            helper[i] = nums[i];
        }

        int count = 0;
        int j = mid + 1;

        // 统计逆序对
        for (int i = left; i <= mid; i++) {
```

```
            while (j <= right && (long)helper[i] > 2L * (long)helper[j]) {
                j++;
            }
            count += j - (mid + 1);
        }

        // 合并两个有序数组
        int i = left, k = left;
        j = mid + 1;
        while (i <= mid && j <= right) {
            if (helper[i] <= helper[j]) {
                nums[k++] = helper[i++];
            } else {
                nums[k++] = helper[j++];
            }
        }
        while (i <= mid) nums[k++] = helper[i++];
        while (j <= right) nums[k++] = helper[j++];

        return count;
    }
}
```

==========