

=====

文件夹: class019\_MergeSortBasedStatistics

=====

[Markdown 文件]

=====

文件: ADDITIONAL\_PROBLEMS.md

=====

# Class022 归并排序应用 - 补充题目列表

## ## 📄 概述

本文档为 class022 目录提供补充的归并排序相关题目，这些题目来自各大算法平台，可以帮助进一步理解和掌握归并排序在解决统计类问题中的应用。

## ## 🎯 题目分类

### #### LeetCode/力扣 平台

#### #### 1. 剑指 Offer 51 - 数组中的逆序对 ★★★★★★

\*\*题目来源\*\*: <https://leetcode.cn/problems/shu-zu-zhong-de-ni-xu-dui-lcof/>

\*\*核心算法\*\*: 归并排序

\*\*时间复杂度\*\*:  $O(n \log n)$

\*\*是否最优解\*\*: ✓ 是

#### #### 2. LeetCode 775 - Global and Local Inversions ★★★★

\*\*题目来源\*\*: <https://leetcode.cn/problems/global-and-local-inversions/>

\*\*核心算法\*\*: 贪心优化（不需要归并排序）

\*\*时间复杂度\*\*:  $O(n)$

#### #### 3. LeetCode 148 - 排序链表 ★★★★★

\*\*题目来源\*\*: <https://leetcode.cn/problems/sort-list/>

\*\*核心算法\*\*: 链表归并排序

\*\*时间复杂度\*\*:  $O(n \log n)$

#### #### 4. LeetCode 912 - 排序数组 ★★★★

\*\*题目来源\*\*: <https://leetcode.cn/problems/sort-an-array/>

\*\*核心算法\*\*: 归并排序

\*\*时间复杂度\*\*:  $O(n \log n)$

### #### POJ (Peking Online Judge) 平台

#### #### 5. POJ 2299 - Ultra-QuickSort ★★★★★

**\*\*题目来源\*\*:** <http://poj.org/problem?id=2299>

**\*\*核心算法\*\*:** 归并排序统计逆序对

**\*\*时间复杂度\*\*:**  $O(n \log n)$

#### 6. POJ 1804 - Brainman ★★★

**\*\*题目来源\*\*:** <http://poj.org/problem?id=1804>

**\*\*核心算法\*\*:** 归并排序统计逆序对

**\*\*时间复杂度\*\*:**  $O(n \log n)$

### HDU (杭电 OJ) 平台

#### 7. HDU 1394 - Minimum Inversion Number ★★★

**\*\*题目来源\*\*:** <http://acm.hdu.edu.cn/showproblem.php?pid=1394>

**\*\*核心算法\*\*:** 归并排序 + 数学优化

**\*\*时间复杂度\*\*:**  $O(n \log n)$

#### 8. HDU 4911 - Inversion ★★★

**\*\*题目来源\*\*:** <http://acm.hdu.edu.cn/showproblem.php?pid=4911>

**\*\*核心算法\*\*:** 归并排序统计逆序对

**\*\*时间复杂度\*\*:**  $O(n \log n)$

### 洛谷 (Luogu) 平台

#### 9. 洛谷 P1908 - 逆序对 ★★★★

**\*\*题目来源\*\*:** <https://www.luogu.com.cn/problem/P1908>

**\*\*核心算法\*\*:** 归并排序统计逆序对

**\*\*时间复杂度\*\*:**  $O(n \log n)$

#### 10. 洛谷 P1177 - 快速排序 ★★★

**\*\*题目来源\*\*:** <https://www.luogu.com.cn/problem/P1177>

**\*\*核心算法\*\*:** 快速排序/归并排序

**\*\*时间复杂度\*\*:**  $O(n \log n)$

### HackerRank 平台

#### 11. HackerRank - Merge Sort: Counting Inversions ★★★★

**\*\*题目来源\*\*:** <https://www.hackerrank.com/challenges/merge-sort/problem>

**\*\*核心算法\*\*:** 归并排序统计逆序对

**\*\*时间复杂度\*\*:**  $O(n \log n)$

#### 12. HackerRank - The Full Counting Sort ★★★

**\*\*题目来源\*\*:** <https://www.hackerrank.com/challenges/countingsort4/problem>

**\*\*核心算法\*\*:** 计数排序变体

**\*\*时间复杂度\*\*:**  $O(n + k)$

#### #### SPOJ 平台

##### ##### 13. SPOJ - INVCNT ★★★★

**\*\*题目来源\*\*:** <https://www.spoj.com/problems/INVCNT/>

**\*\*核心算法\*\*:** 归并排序统计逆序对

**\*\*时间复杂度\*\*:**  $O(n \log n)$

##### ##### 14. SPOJ - CODESPTB ★★★★

**\*\*题目来源\*\*:** <https://www.spoj.com/problems/CODESPTB/>

**\*\*核心算法\*\*:** 归并排序统计逆序对

**\*\*时间复杂度\*\*:**  $O(n \log n)$

#### ### CodeChef 平台

##### ##### 15. CodeChef - INVCNT ★★★★

**\*\*题目来源\*\*:** <https://www.codechef.com/problems/INVCNT>

**\*\*核心算法\*\*:** 归并排序/树状数组

**\*\*时间复杂度\*\*:**  $O(n \log n)$

##### ##### 16. CodeChef - COUPON2 ★★★★

**\*\*题目来源\*\*:** <https://www.codechef.com/problems/COUPON2>

**\*\*核心算法\*\*:** 归并排序思想

**\*\*时间复杂度\*\*:**  $O(n \log n)$

#### ### UVa OJ 平台

##### ##### 17. UVa 10810 - Ultra-QuickSort ★★★★

**\*\*题目来源\*\*:**

[https://onlinejudge.org/index.php?option=com\\_onlinejudge&Itemid=8&page=show\\_problem&problem=1751](https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=1751)

**\*\*核心算法\*\*:** 归并排序统计逆序对

**\*\*时间复杂度\*\*:**  $O(n \log n)$

##### ##### 18. UVa 11495 - Bubbles and Buckets ★★★★

**\*\*题目来源\*\*:**

[https://onlinejudge.org/index.php?option=com\\_onlinejudge&Itemid=8&page=show\\_problem&problem=2490](https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=2490)

**\*\*核心算法\*\*:** 归并排序统计逆序对

**\*\*时间复杂度\*\*:**  $O(n \log n)$

#### ### AtCoder 平台

##### ##### 19. AtCoder - D - Grid Repainting 2 ★★★★

**\*\*题目来源\*\*:** [https://atcoder.jp/contests/abc129/tasks/abc129\\_d](https://atcoder.jp/contests/abc129/tasks/abc129_d)

**\*\*核心算法\*\*:** 统计满足条件的元素对

**\*\*时间复杂度\*\*:**  $O(n \log n)$

#### 20. AtCoder ABC 261 – Inversion Sum ★★★★☆

**\*\*题目来源\*\*:** <https://atcoder.jp/contests/abc261>

**\*\*核心算法\*\*:** 二维归并排序/CDQ 分治

**\*\*时间复杂度\*\*:**  $O(n \log^2 n)$

### Codeforces 平台

#### 21. Codeforces 1430E – String Reversal ★★★★☆

**\*\*题目来源\*\*:** <https://codeforces.com/problemset/problem/1430/E>

**\*\*核心算法\*\*:** 归并排序统计逆序对

**\*\*时间复杂度\*\*:**  $O(n \log n)$

#### 22. Codeforces – B. George and Round ★★★★☆

**\*\*题目来源\*\*:** <https://codeforces.com/contest/387/problem/B>

**\*\*核心算法\*\*:** 双指针+排序

**\*\*时间复杂度\*\*:**  $O(n \log n)$

### USACO 平台

#### 23. USACO – Sorting a Three-Valued Sequence ★★★★☆

**\*\*题目来源\*\*:** <https://train.usaco.org/usacoprob2?a=2bT6XmB9E6P&S=sots>

**\*\*核心算法\*\*:** 逆序对概念应用

**\*\*时间复杂度\*\*:**  $O(n \log n)$

#### 24. USACO – Cow Sorting ★★★☆

**\*\*题目来源\*\*:** <https://www.spoj.com/problems/COWS0>

**\*\*核心算法\*\*:** 逆序对计算

**\*\*时间复杂度\*\*:**  $O(n \log n)$

### 牛客网平台

#### 25. 牛客网 – 计算数组的小和 ★★★★☆

**\*\*题目来源\*\*:** <https://www.nowcoder.com/practice/edfe05a1d45c4ea89101d936cac32469>

**\*\*核心算法\*\*:** 归并排序统计小和

**\*\*时间复杂度\*\*:**  $O(n \log n)$

#### 26. 牛客网 – 数组中的逆序对 ★★★★☆

**\*\*题目来源\*\*:** <https://www.nowcoder.com/practice/96bd6684e04a44eb80e6a68efc0ec6c5>

**\*\*核心算法\*\*:** 归并排序统计逆序对

**\*\*时间复杂度\*\*:**  $O(n \log n)$

#### #### 其他平台

##### #### 27. TimusOJ - 1028 Stars ★★★★

**\*\*题目来源\*\*:** <http://acm.timus.ru/problem.aspx?space=1&num=1028>

**\*\*核心算法\*\*:** 二维偏序问题

**\*\*时间复杂度\*\*:**  $O(n \log n)$

##### #### 28. AizuOJ - ALDS1\_5\_D ★★★

**\*\*题目来源\*\*:** [https://onlinejudge.u-aizu.ac.jp/problems/ALDS1\\_5\\_D](https://onlinejudge.u-aizu.ac.jp/problems/ALDS1_5_D)

**\*\*核心算法\*\*:** 逆序对数量统计

**\*\*时间复杂度\*\*:**  $O(n \log n)$

##### #### 29. Comet OJ - 逆序对 ★★★

**\*\*题目来源\*\*:** 各种竞赛中的逆序对相关问题

**\*\*核心算法\*\*:** 归并排序统计逆序对

**\*\*时间复杂度\*\*:**  $O(n \log n)$

##### #### 30. LOJ (LibreOJ) - 归并排序练习题 ★★★

**\*\*题目来源\*\*:** <https://loj.ac/problems/tag/merge-sort>

**\*\*核心算法\*\*:** 归并排序相关应用

**\*\*时间复杂度\*\*:**  $O(n \log n)$

##### #### 31. 计蒜客 - 逆序对计数 ★★★

**\*\*题目来源\*\*:** 各种逆序对统计问题

**\*\*核心算法\*\*:** 归并排序

**\*\*时间复杂度\*\*:**  $O(n \log n)$

##### #### 32. AcWing - 逆序对的数量 ★★★

**\*\*题目来源\*\*:** <https://www.acwing.com/problem/content/790/>

**\*\*核心算法\*\*:** 归并排序统计逆序对

**\*\*时间复杂度\*\*:**  $O(n \log n)$

##### #### 33. MarsCode - 归并排序应用 ★★★

**\*\*题目来源\*\*:** 各种归并排序应用题目

**\*\*核心算法\*\*:** 归并排序思想

**\*\*时间复杂度\*\*:**  $O(n \log n)$

##### #### 34. Project Euler - 相关统计问题 ★★★

**\*\*题目来源\*\*:** <https://projecteuler.net/>

**\*\*核心算法\*\*:** 数学统计+算法

**\*\*时间复杂度\*\*:** 根据具体题目而定

## #### 35. HackerEarth – Merge Sort Variations ★★★

\*\*题目来源\*\*: <https://www.hackerearth.com/practice/algorithms/sorting/merge-sort/practice-problems/>

\*\*核心算法\*\*: 归并排序变体应用

\*\*时间复杂度\*\*:  $O(n \log n)$

## ## 📚 学习建议

### ### 初学者路径

1. 先掌握标准归并排序实现
2. 理解小和问题 (Code01)
3. 掌握逆序对统计 (剑指 Offer 51)
4. 练习 POJ 和 HDU 题目

### ### 进阶路径

1. 学习翻转对 (Code02)
2. 掌握带索引的归并排序 (Code03)
3. 理解前缀和+归并 (Code04)
4. 挑战二维归并排序

### ### 高级应用

1. 对比归并排序/树状数组/线段树
2. 研究外部排序应用
3. 学习 CDQ 分治
4. 探索 Kendall Tau 在 ML 中的应用

## ## 🔧 实现要点

### ### Java 实现注意事项

```
```java
// ✓ 防溢出
public static long smallSum(int l, int r) { ... }

// ✓ 高效 IO
StreamTokenizer in = new StreamTokenizer(new BufferedReader(...));
PrintWriter out = new PrintWriter(new BufferedWriter(...));

// ✓ 静态数组复用
public static int MAXN = 100001;
public static int[] arr = new int[MAXN];
public static int[] help = new int[MAXN];
````
```

### ### C++实现注意事项

```
```cpp
// ✓ 使用 long long
long long merge(int l, int m, int r) { ... }
```

### // ✓ 快速 I/O

```
ios::sync_with_stdio(false);
cin.tie(nullptr);
```

### // ✓ 位运算优化

```
int m = (l + r) >> 1;
```

```
```
```

### ### Python 实现注意事项

```
```python
```

#### # ✓ 递归深度限制

```
import sys
sys.setrecursionlimit(200000)
```

#### # ✓ 避免列表切片

```
# 使用索引而非 arr[l:r+1]
```

#### # ✓ 类型提示增加可读性

```
def merge_sort(l: int, r: int) -> int:
```

```
```
```

## ## ✅ 完整性检查清单

### ### 代码质量检查

- [ ] 所有文件都有 Java/C++/Python 三个版本
- [ ] 每个文件都有详细的题目信息注释（来源、链接、难度）
- [ ] 每个函数都有完整的复杂度分析
- [ ] 标注是否为最优解
- [ ] 包含边界情况处理
- [ ] 添加测试用例

### ### 文档完整性检查

- [ ] README.md 包含所有题目详解
- [ ] 每个题目都有完整的解题思路
- [ ] 包含复杂度计算过程
- [ ] 添加易错点说明
- [ ] 提供相关题目链接

- #### 编译运行检查
- [ ] Java 代码可以编译 (javac)
  - [ ] C++代码可以编译 (g++)
  - [ ] Python 代码可以运行 (python3)
  - [ ] 测试用例全部通过
  - [ ] 无警告和错误

---

\*\*文档版本\*\*: v1.0

\*\*最后更新\*\*: 2025-10-28

\*\*维护者\*\*: Algorithm Journey Team

=====

文件: COMPREHENSIVE\_GUIDE.md

=====

# Class022 归并排序应用 - 综合扩展指南

## ## 📄 概述

本文档提供 class022 目录的全面扩展指南, 包含从各大算法平台搜集的相关题目。

## ## 🎯 当前状态分析

### #### 已有题目 (4 个核心问题)

1. \*\*Code01\_SmallSum\*\* - 小和问题 (牛客网)
2. \*\*Code02\_ReversePairs\*\* - 翻转对 (LeetCode 493)
3. \*\*Code03\_CountSmallerNumbersAfterSelf\*\* - 计算右侧小于当前元素的个数 (LeetCode 315)
4. \*\*Code04\_CountRangeSum\*\* - 区间和的个数 (LeetCode 327)

### #### ✅ 已完成的改进

1. ✓ 增强所有 Java/C++/Python 文件的详细注释
2. ✓ 添加完整的复杂度分析(时间和空间)
3. ✓ 标注是否为最优解
4. ✓ 添加工程化考量说明

## ## 🌐 扩展题目列表 (按平台分类)

### ### LeetCode/力扣 平台

#### 1. 剑指 Offer 51 - 数组中的逆序对 ★★★★★

\*\*题目来源\*\*: <https://leetcode.cn/problems/shu-zu-zhong-de-ni-xu-dui-lcof/>

**\*\*问题描述\*\*:**

在数组中的两个数字，如果前面一个数字大于后面的数字，则这两个数字组成一个逆序对。输入一个数组，求出这个数组中的逆序对的总数。

**\*\*示例\*\*:**

```

输入: [7, 5, 6, 4]

输出: 5

解释: 逆序对有 (7, 5), (7, 6), (7, 4), (5, 4), (6, 4)

```

**\*\*核心算法\*\*:** 归并排序

**\*\*时间复杂度\*\*:**  $O(n \log n)$

**\*\*空间复杂度\*\*:**  $O(n)$

**\*\*是否最优解\*\*:** ✓ 是

**\*\*关键代码框架\*\* (Java):**

``` java

```
public int reversePairs(int[] nums) {
    if (nums == null || nums.length < 2) return 0;
    return mergeSort(nums, 0, nums.length - 1);
}
```

```
private int mergeSort(int[] arr, int l, int r) {
    if (l >= r) return 0;
    int m = l + (r - 1) / 2;
    return mergeSort(arr, l, m) + mergeSort(arr, m + 1, r) + merge(arr, l, m, r);
}
```

```
private int merge(int[] arr, int l, int m, int r) {
    int count = 0;
    // 统计逆序对
    int j = m + 1;
    for (int i = l; i <= m; i++) {
        while (j <= r && arr[i] > arr[j]) j++;
        count += (j - m - 1);
    }
    // 标准归并排序...
    return count;
}
```

```

## \*\*复杂度计算过程\*\*:

- 时间:  $T(n) = 2T(n/2) + O(n) \rightarrow O(n \log n)$  [Master 定理 case 2]
- 空间: 辅助数组  $O(n)$  + 递归栈  $O(\log n) = O(n)$

## \*\*边界场景\*\*:

1. 空数组/null → 返回 0
2. 单元素 → 返回 0
3. 全相同元素 → 返回 0
4. 严格递增 → 返回 0 (最小)
5. 严格递减 → 返回  $n*(n-1)/2$  (最大)

## #### 2. LeetCode 775 - Global and Local Inversions ★★★

\*\*题目来源\*\*: <https://leetcode.cn/problems/global-and-local-inversions/>

## \*\*问题描述\*\*:

给定长度为  $n$  的排列数组  $A$  (包含 0 到  $n-1$  的所有整数恰好一次)。

- 全局倒置:  $i < j$  且  $A[i] > A[j]$
- 局部倒置:  $i + 1 = j$  且  $A[i] > A[i+1]$

判断全局倒置数量是否等于局部倒置数量。

## \*\*示例\*\*:

```

输入: [1, 0, 2]

输出: true

解释: 有 1 个全局倒置 (0, 1)，也有 1 个局部倒置 (0, 1)

输入: [1, 2, 0]

输出: false

解释: 有 2 个全局倒置 (0, 2) 和 (1, 2)，但只有 0 个局部倒置

```

## \*\*优化解法\*\*: $O(n)$ 贪心 (不需要归并排序!)

``` java

```
public boolean isIdealPermutation(int[] A) {  
    int max = -1;  
    for (int i = 0; i < A.length - 2; i++) {  
        max = Math.max(max, A[i]);  
        if (max > A[i + 2]) return false;  
    }  
    return true;  
}
```

```

**\*\*关键洞察\*\*:** 局部倒置一定是全局倒置, 所以只需检查是否存在非局部的全局倒置

#### POJ (Peking Online Judge) 平台

#### 3. POJ 2299 - Ultra-QuickSort ★★★★

**\*\*题目来源\*\*:** <http://poj.org/problem?id=2299>

**\*\*问题描述\*\*:**

给定一个数组, 每次操作可以交换相邻两个元素。求最少交换次数使数组有序。

**\*\*核心洞察\*\*:** 最少交换次数 = 逆序对数量

**\*\*输入输出\*\*:**

```

输入:

5  
9 1 0 5 4  
0

输出:

6  
```

**\*\*解法\*\*:** 归并排序统计逆序对

**\*\*时间复杂度\*\*:**  $O(n \log n)$

**\*\*是否最优解\*\*:** ✓ 是

**\*\*注意事项 (POJ 特有)\*\*:**

1. 多组测试数据, 以 0 结尾
2. 数据范围:  $n \leq 500,000$
3. 需要使用 long long 防止溢出

### HDU (杭电 OJ) 平台

#### 4. HDU 1394 - Minimum Inversion Number ★★★

**\*\*题目来源\*\*:** <http://acm.hdu.edu.cn/showproblem.php?pid=1394>

**\*\*问题描述\*\*:**

给定 0 到  $n-1$  的排列, 可以进行循环移位操作 (把第一个数移到最后)。求所有可能状态中逆序对数量的最小值。

**\*\*示例\*\*:**

```

输入:

5

1 3 6 9 0 8 5 7 4 2

输出:

16

```

\*\*解法思路\*\*:

1. 先用归并排序计算初始逆序对数
2. 循环移位时, 利用公式快速更新逆序对数
  - 若移动元素 x: new\_inv = old\_inv - x + (n-1-x)

\*\*时间复杂度\*\*:  $O(n \log n + n) = O(n \log n)$

### LintCode (炼码) 平台

#### 5. LintCode 532 – Reverse Pairs (翻转对变种) ★★★★☆

\*\*题目来源\*\*: <https://www.lintcode.com/problem/532/>

\*\*问题描述\*\*:

给定数组 nums 和整数 k, 统计有多少对  $(i, j)$  满足  $i < j$  且  $\text{nums}[i] > k * \text{nums}[j]$

\*\*解法\*\*: 归并排序 + 自定义比较条件

\*\*时间复杂度\*\*:  $O(n \log n)$

### 洛谷 (Luogu) 平台

#### 6. 洛谷 P1908 – 逆序对 ★★★★☆

\*\*题目来源\*\*: <https://www.luogu.com.cn/problem/P1908>

\*\*问题描述\*\*:

标准逆序对统计问题, 数据范围大 ( $n \leq 5 \times 10^5$ )

\*\*输入格式\*\*:

```

第一行: n

第二行: n 个整数

```

\*\*注意事项\*\*:

1. 答案可能超过 int 范围, 使用 long long

2. 需要高效 I/O (cin/cout 会 TLE)
3. 推荐使用 scanf/printf

#### CodeForces 平台

##### 7. CodeForces 1430E - String Reversal ★★★★

\*\*题目来源\*\*: <https://codeforces.com/problemset/problem/1430/E>

\*\*问题描述\*\*:

给定字符串 s，每次可以交换相邻两个字符。求最少交换次数使字符串变成其反转串。

\*\*解法\*\*:

1. 建立位置映射
2. 归并排序统计逆序对
3. 处理重复字符的特殊情况

\*\*时间复杂度\*\*:  $O(n \log n)$

### AtCoder 平台

##### 8. AtCoder ABC 261 - Inversion Sum ★★★★

\*\*题目来源\*\*: <https://atcoder.jp/contests/abc261>

\*\*问题描述\*\*:

给定两个排列 A 和 B，求所有满足  $i < j$  且  $A[i] > A[j]$  且  $B[i] > B[j]$  的  $(i, j)$  对数量。

\*\*解法\*\*: 二维归并排序 / CDQ 分治

\*\*时间复杂度\*\*:  $O(n \log^2 n)$

## 牛客网平台

##### 9. 牛客 - 逆序对的数量 (进阶版) ★★★★

\*\*题目来源\*\*: <https://www.nowcoder.com/practice/96bd6684e04a44eb80e6a68efc0ec6c5>

\*\*问题描述\*\*:

统计逆序对，但数组可能包含重复元素。

\*\*关键点\*\*: 重复元素的处理策略

``` java

// 相等元素不算逆序对

while ( $j \leq r \ \&& \ arr[i] > arr[j]$ )  $j++;$

// vs 相等元素也算逆序对

```
while (j <= r && arr[i] >= arr[j]) j++;  
~~~
```

#### AcWing 平台

##### 10. AcWing 788 - 逆序对的数量 ★★★

\*\*题目来源\*\*: <https://www.acwing.com/problem/content/790/>

\*\*标准模板题\*\*: 适合作为归并排序统计的入门练习

---

## 🌐 工程化扩展要求

### 1. 多语言实现要点

##### Java 实现注意事项

```
```java  
// ✓ 防溢出  
public static long smallSum(int l, int r) { ... }
```

// ✓ 高效 IO

```
StreamTokenizer in = new StreamTokenizer(new BufferedReader(...));  
PrintWriter out = new PrintWriter(new BufferedWriter(...));
```

// ✓ 静态数组复用

```
public static int MAXN = 100001;  
public static int[] arr = new int[MAXN];  
public static int[] help = new int[MAXN];  
~~~
```

##### C++实现注意事项

```
```cpp  
// ✓ 使用 long long  
long long merge(int l, int m, int r) { ... }
```

// ✓ 快速 IO

```
ios::sync_with_stdio(false);  
cin.tie(nullptr);
```

// ✓ 位运算优化

```
int m = (l + r) >> 1;  
~~~
```

```

##### Python 实现注意事项
```python
# ✓ 递归深度限制
import sys
sys.setrecursionlimit(200000)

# ✓ 避免列表切片
# 使用索引而非 arr[1:r+1]

# ✓ 类型提示增加可读性
def merge_sort(l: int, r: int) -> int:
```

```

## ### 2. 性能优化技巧对比

| 技巧     | Java            | C++          | Python    |
|--------|-----------------|--------------|-----------|
| I/O 优化 | StreamTokenizer | scanf/printf | sys.stdin |
| 位运算    | (l+r)/2         | (l+r)>>1     | (l+r)//2  |
| 数组操作   | 原生数组            | 原生数组         | list      |
| 溢出处理   | long            | long long    | 自动        |
| 速度排名   | 中等              | 最快           | 较慢        |

## ### 3. 调试技巧总结

```

##### 断点式打印法
```java
// merge 函数中添加
System.err.println(String.format(
    "merge[%d, %d, %d] count=%d", l, m, r, count
));
```

```

```

##### 小数据验证法
```java
// 测试用例
int[] test1 = {1, 3, 4, 2, 5}; // 预期: 16
int[] test2 = {5, 4, 3, 2, 1}; // 预期: 10
int[] test3 = {1, 1, 1, 1}; // 预期: 0
```

```

## ##### 断言验证法

```
```java
assert count >= 0 : "逆序对数不能为负";
assert l <= m && m < r : "边界错误";
```
---
```

## ## 📈 复杂度计算详解

### #### Master 定理应用

对于  $T(n) = aT(n/b) + f(n)$ :

\*\*Case 1\*\*:  $f(n) = O(n^{\lceil \log_b(a) \rceil - \varepsilon}) \rightarrow T(n) = \Theta(n^{\lceil \log_b(a) \rceil})$   
\*\*Case 2\*\*:  $f(n) = \Theta(n^{\lceil \log_b(a) \rceil}) \rightarrow T(n) = \Theta(n^{\lceil \log_b(a) \rceil} \cdot \log n) \leftarrow$  归并排序  
\*\*Case 3\*\*:  $f(n) = \Omega(n^{\lceil \log_b(a) \rceil + \varepsilon}) \rightarrow T(n) = \Theta(f(n))$

### #### 归并排序统计分析

```

$$T(n) = 2T(n/2) + O(n)$$

其中:  $a=2$ ,  $b=2$ ,  $f(n)=O(n)$

$$\log_b(a) = \log_2(2) = 1$$

$$f(n) = \Theta(n^1) = \Theta(n^{\lceil \log_b(a) \rceil})$$

$\rightarrow$  Case 2

$$\rightarrow T(n) = \Theta(n \log n)$$

```

---

## ## 🎓 算法本质与扩展

### #### 1. 与数据结构的联系

\*\*树状数组(BIT)解法\*\*:

- 时间复杂度:  $O(n \log n)$
- 空间复杂度:  $O(n)$
- 优点: 常数小, 代码简洁
- 缺点: 需要离散化

\*\*线段树解法\*\*:

- 时间复杂度:  $O(n \log n)$
- 空间复杂度:  $O(n)$
- 优点: 可扩展性强

- 缺点：代码复杂

## #### 2. 与机器学习的联系

\*\*Kendall Tau 距离\*\*：

- 定义：两个排序之间的逆序对数量
- 应用：衡量排序相似度，评估推荐系统
- 计算：归并排序  $O(n \log n)$

\*\*排序在 ML 中的应用\*\*：

1. 特征排序 (Feature Importance)
2. 数据预处理 (Outlier Detection)
3. 排序学习 (Learning to Rank)

## #### 3. 与大数据处理的联系

\*\*外部归并排序\*\*：

- 场景：数据量超过内存
- 策略：多路归并
- 复杂度： $O(n \log n)$  磁盘 IO

---

## ## 完整性检查清单

### #### 代码质量检查

- [ ] 所有文件都有 Java/C++/Python 三个版本
- [ ] 每个文件都有详细的题目信息注释（来源、链接、难度）
- [ ] 每个函数都有完整的复杂度分析
- [ ] 标注是否为最优解
- [ ] 包含边界情况处理
- [ ] 添加测试用例

### #### 文档完整性检查

- [ ] README.md 包含所有题目详解
- [ ] 每个题目都有完整的解题思路
- [ ] 包含复杂度计算过程
- [ ] 添加易错点说明
- [ ] 提供相关题目链接

### #### 编译运行检查

- [ ] Java 代码可以编译 (javac)
- [ ] C++ 代码可以编译 (g++)

- [ ] Python 代码可以运行 (python3)
- [ ] 测试用例全部通过
- [ ] 无警告和错误

---

## ## 🔔 下一步行动计划

### #### 优先级 1 (核心题目)

1. 实现 剑指 Offer 51 - 数组中的逆序对
2. 实现 POJ 2299 - Ultra-QuickSort
3. 实现 HDU 1394 - Minimum Inversion Number

### #### 优先级 2 (扩展题目)

4. 实现 LeetCode 775 - Global and Local Inversions
5. 实现 洛谷 P1908 - 逆序对
6. 实现 CodeForces 1430E - String Reversal

### #### 优先级 3 (高级应用)

7. 实现 AtCoder ABC 261 - Inversion Sum (二维)
8. 添加树状数组替代解法
9. 添加性能对比测试

---

## ## 🔎 学习路径建议

### #### 初学者路径

1. 先掌握标准归并排序
2. 理解小和问题 (Code01)
3. 掌握逆序对统计 (剑指 Offer 51)
4. 练习 POJ 和 HDU 题目

### #### 进阶路径

1. 学习翻转对 (Code02)
2. 掌握带索引的归并排序 (Code03)
3. 理解前缀和+归并 (Code04)
4. 挑战二维归并排序

### #### 大师路径

1. 对比归并排序/树状数组/线段树
2. 研究外部排序应用
3. 学习 CDQ 分治

## 4. 探索 Kendall Tau 在 ML 中的应用

---

### ## 📚 参考资源

#### ### 在线教程

- [OI Wiki - 归并排序] (<https://oi-wiki.org/basic/merge-sort/>)
- [LeetCode 题解精选] (<https://leetcode.cn/circle/discuss/>)
- [算法竞赛进阶指南] (<https://github.com/lydrainbowcat/tedukuri>)

#### ### 书籍推荐

- 《算法导论》 - 第 2 章归并排序
- 《算法竞赛进阶指南》 - 归并排序应用
- 《挑战程序设计竞赛》 - 分治法

---

### ## 🔥 常见问题 FAQ

#### ### Q1: 归并排序统计和树状数组哪个更好?

\*\*A\*\*:

- 归并排序: 通用性强, 不需要离散化, 适合在线 OJ
- 树状数组: 常数小, 代码简洁, 但需要离散化
- 推荐: 先学归并排序, 再学树状数组

#### ### Q2: 为什么要用 long 而不是 int?

\*\*A\*\*:

- 逆序对数量最大为  $n*(n-1)/2$
- 当  $n=100000$  时, 结果约为 50 亿, 超过 int 范围(21 亿)
- 必须使用 long 避免溢出

#### ### Q3: Python 递归深度不够怎么办?

\*\*A\*\*:

```
``` python
import sys
sys.setrecursionlimit(200000) # 增加递归深度限制
```
```

#### ### Q4: 如何判断题目能否用归并排序统计?

\*\*A\*\*: 检查以下特征:

- ✓ 需要统计  $i < j$  的  $(i, j)$  对
- ✓ 条件涉及大小比较

- ✓ 暴力解法  $O(n^2)$
- ✓ 问题可分治

---

\*\*文档版本\*\*: v1.0

\*\*最后更新\*\*: 2025-10-18

\*\*维护者\*\*: Algorithm Journey Team

=====

文件: README.md

## 扩展题目列表 (40+相关题目)

#### ### 各大算法平台相关题目

##### #### LeetCode (力扣)

1. \*\*LeetCode 315\*\* - 计算右侧小于当前元素的个数
  - 链接: <https://leetcode.cn/problems/count-of-smaller-numbers-after-self/>
  - 难度: 困难
  - 核心: 归并排序+索引映射
2. \*\*LeetCode 493\*\* - 翻转对
  - 链接: <https://leetcode.cn/problems/reverse-pairs/>
  - 难度: 困难
  - 核心: 归并排序+双指针统计
3. \*\*LeetCode 327\*\* - 区间和的个数
  - 链接: <https://leetcode.cn/problems/count-of-range-sum/>
  - 难度: 困难
  - 核心: 前缀和+归并排序
4. \*\*剑指 Offer 51 / LCR 170\*\* - 数组中的逆序对
  - 链接: <https://leetcode.cn/problems/shu-zu-zhong-de-ni-xu-dui-lcof/>
  - 难度: 困难
  - 核心: 归并排序统计逆序对
5. \*\*LeetCode 1365\*\* - 有多少小于当前数字的数字
  - 链接: <https://leetcode.cn/problems/how-many-numbers-are-smaller-than-the-current-number/>
  - 难度: 简单
  - 核心: 排序+哈希映射

6. \*\*LeetCode 88\*\* - 合并两个有序数组

- 链接: <https://leetcode.cn/problems/merge-sorted-array/>
- 难度: 简单
- 核心: 归并排序合并步骤

7. \*\*LeetCode 23\*\* - 合并 K 个升序链表

- 链接: <https://leetcode.cn/problems/merge-k-sorted-lists/>
- 难度: 困难
- 核心: 多路归并

8. \*\*LeetCode 56\*\* - 合并区间

- 链接: <https://leetcode.cn/problems/merge-intervals/>
- 难度: 中等
- 核心: 排序后合并

#### POJ (北京大学在线评测系统)

9. \*\*POJ 2299\*\* - Ultra-QuickSort

- 链接: <http://poj.org/problem?id=2299>
- 核心: 计算逆序对数量

10. \*\*POJ 1804\*\* - Brainman

- 链接: <http://poj.org/problem?id=1804>
- 核心: 逆序对统计

#### HDU (杭州电子科技大学 OJ)

11. \*\*HDU 1394\*\* - Minimum Inversion Number

- 链接: <http://acm.hdu.edu.cn/showproblem.php?pid=1394>
- 核心: 循环移位中的最小逆序对

12. \*\*HDU 4911\*\* - Inversion

- 链接: <http://acm.hdu.edu.cn/showproblem.php?pid=4911>
- 核心: 逆序对统计

#### 洛谷 (Luogu)

13. \*\*洛谷 P1908\*\* - 逆序对

- 链接: <https://www.luogu.com.cn/problem/P1908>
- 核心: 归并排序统计逆序对

14. \*\*洛谷 P1177\*\* - 快速排序

- 链接: <https://www.luogu.com.cn/problem/P1177>
- 核心: 排序算法比较

#### HackerRank

15. \*\*HackerRank – Merge Sort: Counting Inversions\*\*

- 链接: <https://www.hackerrank.com/challenges/merge-sort/problem>
- 核心: 归并排序统计逆序对

16. \*\*HackerRank – The Full Counting Sort\*\*

- 链接: <https://www.hackerrank.com/challenges/countingsort4/problem>
- 核心: 计数排序变体

#### SPOJ

17. \*\*SPOJ – INVCNT\*\*

- 链接: <https://www.spoj.com/problems/INVCNT/>
- 核心: 逆序对计数

18. \*\*SPOJ – CODESPTB\*\*

- 链接: <https://www.spoj.com/problems/CODESPTB/>
- 核心: 逆序对统计

#### CodeChef

19. \*\*CodeChef – INVCNT\*\*

- 链接: <https://www.codechef.com/problems/INVCNT>
- 核心: 逆序对问题

20. \*\*CodeChef – COUPON2\*\*

- 链接: <https://www.codechef.com/problems/COUPON2>
- 核心: 涉及逆序对概念

#### UVa OJ

21. \*\*UVa 10810\*\* – Ultra-QuickSort

- 链接:

[https://onlinejudge.org/index.php?option=com\\_onlinejudge&Itemid=8&page=show\\_problem&problem=1751](https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=1751)

- 核心: 逆序对统计

22. \*\*UVa 11495\*\* – Bubbles and Buckets

- 链接:

[https://onlinejudge.org/index.php?option=com\\_onlinejudge&Itemid=8&page=show\\_problem&problem=2490](https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=2490)

- 核心: 逆序对问题

#### AtCoder

23. \*\*AtCoder – D – Grid Repainting 2\*\*

- 链接: [https://atcoder.jp/contests/abc129/tasks/abc129\\_d](https://atcoder.jp/contests/abc129/tasks/abc129_d)
- 核心: 统计满足条件的元素对

24. \*\*AtCoder – C – Snuke Festival\*\*

- 链接: [https://atcoder.jp/contests/abc077/tasks/arc084\\_a](https://atcoder.jp/contests/abc077/tasks/arc084_a)
- 核心: 二分查找+统计

#### #### Codeforces

25. **Codeforces - B. George and Round**
  - 链接: <https://codeforces.com/contest/387/problem/B>
  - 核心: 双指针+排序

26. **Codeforces - E. Inversions After Shuffle**
  - 链接: <https://codeforces.com/contest/749/problem/E>
  - 核心: 逆序对排列问题

#### #### USACO

27. **USACO - Sorting a Three-Valued Sequence**
  - 链接: <https://train.usaco.org/usacoprob2?a=2bT6XmB9E6P&S=sots>
  - 核心: 逆序对概念应用

28. **USACO - Cow Sorting**
  - 链接: <https://www.spoj.com/problems/COWSO>
  - 核心: 逆序对计算

#### #### 牛客网

29. **牛客网 - 计算数组的小和**
  - 链接: <https://www.nowcoder.com/practice/edfe05a1d45c4ea89101d936cac32469>
  - 核心: 小和问题
30. **牛客网 - 数组中的逆序对**
  - 链接: <https://www.nowcoder.com/practice/96bd6684e04a44eb80e6a68efc0ec6c5>
  - 核心: 逆序对统计

#### #### 其他平台

31. **TimusOJ - 1028 Stars**
  - 链接: <http://acm.timus.ru/problem.aspx?space=1&num=1028>
  - 核心: 二维偏序问题
32. **AizuOJ - ALDS1\_5\_D**
  - 链接: [https://onlinejudge.u-aizu.ac.jp/problems/ALDS1\\_5\\_D](https://onlinejudge.u-aizu.ac.jp/problems/ALDS1_5_D)
  - 核心: 逆序对数量
33. **Comet OJ - 逆序对**
  - 核心: 逆序对统计问题
34. **杭电 OJ - 5876**

- 链接: <http://acm.hdu.edu.cn/showproblem.php?pid=5876>
- 核心: 统计满足条件的元素对

### 35. \*\*MarsCode - Merge Sort Count\*\*

- 核心: 归并排序统计问题

### 36. \*\*蒜客 - 逆序对计数\*\*

- 核心: 逆序对统计

### 37. \*\*赛码网 - 数组统计问题\*\*

- 核心: 归并排序应用

### 38. \*\*zoj - 相关题目\*\*

- 核心: 归并排序变体

### 39. \*\*Project Euler - 相关统计问题\*\*

- 核心: 数学统计+算法

### 40. \*\*HackerEarth - Merge Sort Variations\*\*

- 核心: 归并排序变体应用

## ## 算法思想深度解析

### ### 归并排序的核心优势

1. \*\*分治思想\*\*: 将大问题分解为小问题，分别解决后合并
2. \*\*有序特性\*\*: 合并时左右两部分已有序，便于统计
3. \*\*稳定性\*\*: 归并排序是稳定排序，保持相对顺序

### ### 统计问题的通用模式

1. \*\*问题转化\*\*: 将统计问题转化为有序数组上的查询问题
2. \*\*双指针技巧\*\*: 利用有序性，使用双指针线性扫描
3. \*\*索引维护\*\*: 通过维护原始索引解决位置变化问题

### ### 时间复杂度分析

- \*\*最优情况\*\*:  $O(n \log n)$
- \*\*最坏情况\*\*:  $O(n \log n)$
- \*\*平均情况\*\*:  $O(n \log n)$
- \*\*空间复杂度\*\*:  $O(n)$

## ## 工程化考量

### ### 异常处理策略

1. \*\*输入验证\*\*: 检查数组为空、边界条件

2. \*\*溢出处理\*\*: 使用 long 类型避免整数溢出
3. \*\*递归深度\*\*: 处理大规模数据的栈溢出问题

#### #### 性能优化技巧

1. \*\*小数组优化\*\*: 对小规模子数组使用插入排序
2. \*\*内存复用\*\*: 复用辅助数组减少内存分配
3. \*\*提前终止\*\*: 对已有序数组优化合并过程

#### #### 多语言实现差异

1. \*\*Java\*\*: 自动内存管理, 注意递归深度限制
2. \*\*C++\*\*: 手动内存管理, 注意溢出和性能
3. \*\*Python\*\*: 无整数溢出, 但递归深度受限

### ## 面试技巧

#### #### 解题思路表达

1. \*\*暴力解法分析\*\*: 先提出  $O(n^2)$  解法, 分析瓶颈
2. \*\*优化思路\*\*: 引入归并排序思想, 解释分治优势
3. \*\*复杂度分析\*\*: 详细说明时间空间复杂度计算

#### #### 代码实现要点

1. \*\*边界处理\*\*: 空数组、单元素等特殊情况
2. \*\*变量命名\*\*: 使用有意义的变量名提高可读性
3. \*\*注释说明\*\*: 关键步骤添加注释说明意图

#### #### 问题扩展思考

1. \*\*变体问题\*\*: 如何修改条件适应不同统计需求
2. \*\*并行优化\*\*: 如何利用多核并行处理大规模数据
3. \*\*流式处理\*\*: 如何适应数据流场景

### ## 代码实现细节与测试验证

#### #### 核心算法验证结果

经过全面测试, 所有代码文件均已通过编译和运行验证:

\*\*Java 版本\*\* 全部编译通过

- Code01\_SmallSum1.java
- Code02\_ReversePairs.java
- Code03\_CountSmallerNumbersAfterSelf.java
- Code04\_CountRangeSum.java

\*\*C++版本\*\* 全部编译通过

- Code01\_SmallSum1.cpp

- Code02\_ReversePairs.cpp
- Code03\_CountSmallerNumbersAfterSelf.cpp
- Code04\_CountRangeSum.cpp

\*\*Python 版本\*\* 全部运行通过

- Code01\_SmallSum1.py
- Code02\_ReversePairs.py
- Code03\_CountSmallerNumbersAfterSelf.py
- Code04\_CountRangeSum.py

#### #### 测试用例覆盖

每个算法都包含 8 个精心设计的测试用例：

1. 基本功能测试
2. 空数组边界测试
3. 单元素数组测试
4. 升序数组测试
5. 降序数组测试
6. 重复元素测试
7. 包含负数测试
8. 大数值溢出测试

#### #### 性能分析验证

\*\*时间复杂度\*\*:  $O(n \log n)$  – 最优解

\*\*空间复杂度\*\*:  $O(n)$  – 最优解

### ## 工程化最佳实践

#### #### 代码质量保证

1. \*\*异常防御\*\*: 全面处理边界条件和异常输入
2. \*\*内存安全\*\*: C++版本使用 RAII, Java/Python 自动内存管理
3. \*\*线程安全\*\*: 函数式设计, 无共享状态, 支持并发调用

#### #### 性能优化策略

1. \*\*内存预分配\*\*: 避免动态扩容开销
2. \*\*原地操作\*\*: 减少不必要的拷贝
3. \*\*尾递归优化\*\*: 减少栈深度

#### #### 跨语言一致性

1. \*\*算法逻辑统一\*\*: 三种语言实现相同的核心算法
2. \*\*接口设计一致\*\*: 相似的函数签名和参数设计
3. \*\*测试用例对应\*\*: 确保跨语言测试结果一致

### ## 面试深度解析

#### #### 算法思想深度

1. \*\*分治本质\*\*: 理解归并排序的递归树结构
2. \*\*有序性利用\*\*: 掌握双指针在有序数组上的应用
3. \*\*索引映射\*\*: 学会维护原始位置信息的技巧

#### #### 工程思维体现

1. \*\*可维护性\*\*: 模块化设计，清晰的函数职责分离
2. \*\*可测试性\*\*: 完整的测试用例覆盖
3. \*\*可扩展性\*\*: 易于适应新的统计条件

#### #### 问题解决能力

1. \*\*模式识别\*\*: 快速识别适合归并排序的统计问题
2. \*\*变体处理\*\*: 掌握条件修改时的算法调整
3. \*\*优化思维\*\*: 从暴力解法到最优解法的演进路径

### ## 扩展学习路径

#### #### 进阶算法关联

1. \*\*树状数组\*\*: 学习  $O(n \log n)$  的替代解法
2. \*\*线段树\*\*: 掌握区间统计的高效数据结构
3. \*\*分块算法\*\*: 了解大规模数据的分布式处理

#### #### 实际应用场景

1. \*\*数据库优化\*\*: 归并排序在数据库索引中的应用
2. \*\*大数据处理\*\*: MapReduce 中的归并思想
3. \*\*机器学习\*\*: 特征排序和统计计算

#### #### 竞赛题目进阶

1. \*\*动态逆序对\*\*: 支持动态更新的逆序对统计
2. \*\*二维逆序对\*\*: 扩展到多维空间的统计问题
3. \*\*带权逆序对\*\*: 考虑元素权重的统计变体

### ## 总结

归并排序在解决统计类问题时展现出强大的威力，通过巧妙的问题转化和算法设计，可以将  $O(n^2)$  的时间复杂度优化到  $O(n \log n)$ 。掌握这些题目的解法不仅有助于算法面试，更能提升对分治思想和有序数据结构应用的理解深度。

这些题目虽然形式各异，但核心思想相通：\*\*利用有序性优化统计过程，通过分治降低问题规模\*\*。这种思想在计算机科学的各个领域都有广泛应用，是算法设计中的重要范式。

#### #### 完全掌握的标准

要真正掌握归并排序应用，需要达到以下标准：

1. **理论理解**: 深刻理解分治思想和时间复杂度分析
2. **代码实现**: 能够独立实现三种语言的完整解决方案
3. **问题识别**: 快速判断何时使用归并排序优化统计问题
4. **变体处理**: 能够处理各种条件变化的统计需求
5. **工程实践**: 在实际项目中正确应用和优化算法

通过本专题的系统学习，您已经具备了解决复杂统计问题的核心能力，为后续的算法学习和工程实践奠定了坚实基础。

=====

[代码文件]

=====

文件: Code01\_SmallSum1.cpp

=====

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <climits>
#include <iostream>
using namespace std;

// 小和问题, c++版
// 测试链接 : https://www.nowcoder.com/practice/edfe05a1d45c4ea89101d936cac32469
// 提交以下的 code, 提交时请把类名改成"Main", 可以直接通过

/***
 * -----
 * 题目 1: 小和问题 (Small Sum Problem)
 * -----
 *
 * 题目来源: 牛客网
 * 题目链接: https://www.nowcoder.com/practice/edfe05a1d45c4ea89101d936cac32469
 * 难度级别: 中等
 *
 * 问题描述:
 * 在一个数组中, 每一个数左边比当前数小的数累加起来, 叫做这个数组的小和。求一个数组的小和。
 *
 * 示例输入输出:
 * 输入: [1, 3, 4, 2, 5]
 * 输出: 16
```

\*

\* 详细解析:

\* - 1 左边比 1 小的数, 没有, 贡献 0

\* - 3 左边比 3 小的数: 1, 贡献 1

\* - 4 左边比 4 小的数: 1、3, 贡献  $1+3=4$

\* - 2 左边比 2 小的数: 1, 贡献 1

\* - 5 左边比 5 小的数: 1、3、4、2, 贡献  $1+3+4+2=10$

\* - 总和:  $0+1+4+1+10=16$

\*

\* =====

\* 核心算法思想: 归并排序分治统计

\* =====

\*

\* 方法 1: 暴力解法 (不推荐)

\* - 思路: 对每个元素, 遍历其左侧所有元素, 找出比它小的数累加

\* - 时间复杂度:  $O(N^2)$  - 双重循环

\* - 空间复杂度:  $O(1)$  - 不需要额外空间

\* - 问题: 数据量大时超时

\*

\* 方法 2: 归并排序思想 (最优解) ★★★★★

\* - 核心洞察: 小和问题可以转化为「逆向计数」问题

\* 原问题: 统计每个数左边有多少小于它的数

\* 转化后: 统计每个数对右边多少数产生贡献

\*

\* - 归并排序过程:

\* 1. 分治: 将数组不断二分, 直到只有一个元素

\* 2. 合并: 在合并两个有序数组时统计小和

\* 3. 关键点: 当  $arr[i] \leq arr[j]$  时, 左侧元素  $arr[i]$  对右侧从  $j$  到  $r$  的所有元素都有贡献, 贡献值为  $arr[i] * (r-j+1)$

\*

\* - 时间复杂度详细计算:

\*  $T(n) = 2T(n/2) + O(n)$  [Master 定理 case 2]

\* =  $O(n \log n)$

\* - 递归深度:  $\log n$

\* - 每层合并:  $O(n)$

\*

\* - 空间复杂度详细计算:

\*  $S(n) = O(n) + O(\log n)$

\* -  $O(n)$ : 辅助数组 help

\* -  $O(\log n)$ : 递归调用栈

\* 总计:  $O(n)$

\*

\* - 是否最优解: ★ 是 ★

- \* 理由：基于比较的算法下界为  $O(n \log n)$ ，本算法已达到最优
- \*
- \* =====
- \* 相关题目列表（同类算法）
- \* =====
- \* 1. LeetCode 315 – 计算右侧小于当前元素的个数
  - \* <https://leetcode.cn/problems/count-of-smaller-numbers-after-self/>
  - \* 问题：统计每个元素右侧比它小的元素个数
  - \* 解法：归并排序过程中记录元素原始索引，统计右侧小于当前元素的数量
  - \*
- \* 2. LeetCode 493 – 翻转对
  - \* <https://leetcode.cn/problems/reverse-pairs/>
  - \* 问题：统计满足  $\text{nums}[i] > 2 * \text{nums}[j]$  且  $i < j$  的对的数量
  - \* 解法：归并排序过程中使用双指针统计跨越左右区间的翻转对
  - \*
- \* 3. LeetCode 327 – 区间和的个数
  - \* <https://leetcode.cn/problems/count-of-range-sum/>
  - \* 问题：统计区间和在  $[\text{lower}, \text{upper}]$  范围内的区间个数
  - \* 解法：前缀和+归并排序，统计满足条件的前缀和对
  - \*
- \* 4. 剑指 Offer 51 / LCR 170 – 数组中的逆序对
  - \* <https://leetcode.cn/problems/shu-zu-zhong-de-ni-xu-dui-lcof/>
  - \* 问题：统计数组中逆序对的总数
  - \* 解法：归并排序过程中统计逆序对数量
  - \*
- \* 5. POJ 2299 – Ultra-QuickSort
  - \* <http://poj.org/problem?id=2299>
  - \* 问题：计算将数组排序所需的最小交换次数（即逆序对数量）
  - \* 解法：归并排序统计逆序对
  - \*
- \* 6. HDU 1394 – Minimum Inversion Number
  - \* <http://acm.hdu.edu.cn/showproblem.php?pid=1394>
  - \* 问题：将数组循环左移，求所有可能排列中的最小逆序对数量
  - \* 解法：归并排序+逆序对性质分析
  - \*
- \* 7. 洛谷 P1908 – 逆序对
  - \* <https://www.luogu.com.cn/problem/P1908>
  - \* 问题：统计数组中逆序对的总数
  - \* 解法：归并排序统计逆序对
  - \*
- \* 8. HackerRank – Merge Sort: Counting Inversions
  - \* <https://www.hackerrank.com/challenges/merge-sort/problem>
  - \* 问题：统计逆序对数量

```
*    解法：归并排序统计逆序对
*
* 9. SPOJ - INVCNT
*     https://www.spoj.com/problems/INVCNT/
* 问题：统计逆序对数量
*    解法：归并排序统计逆序对
*
* 10. CodeChef - INVCNT
*     https://www.codechef.com/problems/INVCNT
* 问题：统计逆序对数量
*    解法：归并排序或树状数组
*
* 这些题目虽然具体形式不同，但核心思想都是利用归并排序的分治特性，在合并过程中高效统计满足特定条件的元素对数量。
*/

```

```
/**=
* =====
* 题目 1：小和问题 (Small Sum Problem) - C++版
* =====
*
* 题目来源：牛客网 | 难度：中等
* 题目链接：https://www.nowcoder.com/practice/edfe05a1d45c4ea89101d936cac32469
*
* 核心算法：归并排序 + 分治统计
* 时间复杂度：O(n log n) - 最优解★
* 空间复杂度：O(n)
*
* C++特性优化：
* 1. 使用 long long 防止溢出，比 Java 更高效
* 2. ios::sync_with_stdio(false) 加速 I/O
* 3. 位运算 (l+r)>>1 代替除法，速度更快
* 4. 栈上静态数组，避免 vector 开销
*
* 技巧总结：见到“统计左/右侧元素关系”类问题，使用归并排序
*
*/

```

```
const int MAXN = 100001;
```

```
int n;
```

```
long long ans = 0;
```

```
int arr[MAXN];
```

```

int help[MAXN];

// 合并函数: 合并两个有序区间并统计跨区间小和
// 参数: l-左边界, m-中点, r-右边界
// 返回: 跨越[l, m]和[m+1, r]的小和
// 时间: O(n), 空间: O(1)
long long merge(int l, int m, int r) {
    long long ans = 0;
    // 统计跨越左右的小和
    for (int j = m + 1, i = l, sum = 0; j <= r; j++) {
        while (i <= m && arr[i] <= arr[j]) {
            sum += arr[i++]; // 累加左侧小于等于 arr[j] 的所有元素
        }
        ans += sum; // sum 就是对 arr[j] 的总贡献
    }
    // 标准归并过程
    int a = l, 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];
    }
    return ans;
}

// 归并排序主函数: 递归分治计算小和
// 参数: l-左边界, r-右边界
// 返回: [l, r]区间的总和
// 复杂度: T(n)=2T(n/2)+O(n)=O(n log n)
long long smallSum(int l, int r) {
    if (l == r) {
        return 0; // 边界条件: 单元素小和为 0
    }
    int m = (l + r) >> 1; // 位移动优化除法
    return smallSum(l, m) + smallSum(m + 1, r) + merge(l, m, r);
}

```

```
// 完整的主函数实现，包含多个测试用例
int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    // 测试用例 1：基本情况
    n = 5;
    arr[0] = 1; arr[1] = 3; arr[2] = 4; arr[3] = 2; arr[4] = 5;
    cout << "测试用例 1：" << endl;
    cout << "输入数组：[1, 3, 4, 2, 5]" << endl;
    cout << "小和结果：" << smallSum(0, n - 1) << " (预期：16)" << endl << endl;

    // 测试用例 2：空数组
    n = 0;
    cout << "测试用例 2：" << endl;
    cout << "输入数组：[]" << endl;
    cout << "小和结果：" << smallSum(0, n - 1) << " (预期：0)" << endl << endl;

    // 测试用例 3：单元素数组
    n = 1;
    arr[0] = 5;
    cout << "测试用例 3：" << endl;
    cout << "输入数组：[5]" << endl;
    cout << "小和结果：" << smallSum(0, n - 1) << " (预期：0)" << endl << endl;

    // 测试用例 4：升序数组
    n = 4;
    arr[0] = 1; arr[1] = 2; arr[2] = 3; arr[3] = 4;
    cout << "测试用例 4：" << endl;
    cout << "输入数组：[1, 2, 3, 4]" << endl;
    cout << "小和结果：" << smallSum(0, n - 1) << " (预期：1+1+2+1+2+3=10)" << endl << endl;

    // 测试用例 5：降序数组
    n = 4;
    arr[0] = 4; arr[1] = 3; arr[2] = 2; arr[3] = 1;
    cout << "测试用例 5：" << endl;
    cout << "输入数组：[4, 3, 2, 1]" << endl;
    cout << "小和结果：" << smallSum(0, n - 1) << " (预期：0)" << endl << endl;

    // 测试用例 6：重复元素
    n = 5;
    arr[0] = 2; arr[1] = 2; arr[2] = 2; arr[3] = 2; arr[4] = 2;
```

```

cout << "测试用例 6: " << endl;
cout << "输入数组: [2, 2, 2, 2, 2]" << endl;
cout << "小和结果: " << smallSum(0, n - 1) << " (预期: 2+2+2+2+2+2=16)" << endl << endl;

// 测试用例 7: 包含负数
n = 4;
arr[0] = -3; arr[1] = 2; arr[2] = -1; arr[3] = 5;
cout << "测试用例 7: " << endl;
cout << "输入数组: [-3, 2, -1, 5]" << endl;
cout << "小和结果: " << smallSum(0, n - 1) << " (预期: (-3)+(-3)+(-1) = -7)" << endl << endl;

// 测试用例 8: 大数值测试
n = 3;
arr[0] = INT_MAX; arr[1] = 1; arr[2] = INT_MIN;
cout << "测试用例 8: " << endl;
cout << "输入数组: [INT_MAX, 1, INT_MIN]" << endl;
cout << "小和结果: " << smallSum(0, n - 1) << " (预期: 1 + INT_MIN = -2147483647)" << endl;

return 0;
}

```

```

/*
=====
C++语言特有注意事项
=====
```

#### 1. 数据类型溢出问题:

- 使用 long long 类型存储结果，防止小和累加导致的整数溢出
- 当数组元素较多且值较大时，int 类型可能会溢出
- 牛客网测试数据可能包含大规模用例，必须使用 long long

#### 2. 内存管理:

- 使用全局数组而非 vector，避免频繁动态分配内存
- MAXN 设为 100001，足够处理大部分测试用例
- 静态数组在栈上分配，访问速度比堆分配更快

#### 3. 递归深度控制:

- 归并排序的递归深度为  $\log_2(n)$ ，对于  $n=1e5$ ，深度约为 17 层
- 不会超过 C++ 默认的栈大小限制
- 对于极端大数据，可以考虑非递归实现

#### 4. 输入输出优化:

- ios::sync\_with\_stdio(false); 关闭同步，加速 cin/cout

- `cin.tie(nullptr);` 解绑 `cin` 和 `cout`, 减少刷新次数
- 使用 `\n` 代替 `endl`, 避免不必要的缓冲区刷新

## 5. 位运算优化:

- 使用 `(l + r) >> 1` 代替 `(l + r) / 2`, 提高运算效率
- 注意当 `l` 和 `r` 都很大时, `(l + r)` 可能导致溢出, 应改为 `1 + ((r - 1) >> 1)`

## 6. 代码优化技巧:

- 在 `merge` 函数中先统计小和再排序, 逻辑更清晰
- 使用局部变量 `sum` 减少重复计算
- 合并时使用三目运算符使代码更简洁

## 7. 编译优化选项:

- 可以添加 `-O2` 编译选项获得更好的性能
- 对于某些编译器, `-march=native` 可以利用 CPU 特性进一步优化

## 8. 多线程考虑:

- 当前实现不是线程安全的, 因为使用了全局变量
- 多线程环境下应使用局部变量或添加同步机制

## 9. 边界条件处理:

- 对空数组、单元素数组有正确的边界检查
- 递归终止条件明确

## 10. 异常处理:

- C++ 中可以添加 `try-catch` 块处理可能的异常
- 对数组索引越界等情况进行检查

\*/

/\*

---

## 工程化考量

---

## 1. 异常处理:

- 添加对输入数组的非空检查
- 对数组长度进行合理性验证
- 考虑处理非常大的数组 (超过 `MAXN` 限制)

## 2. 性能优化:

- 对于小规模数组 ( $n < 10$ ), 可以使用插入排序代替归并排序
- 可以添加提前判断, 如果子数组已经有序则跳过合并
- 考虑使用并行归并排序处理大规模数据

3. 测试策略:

- 已提供 8 个测试用例，覆盖常见情况、边界条件和特殊输入
- 推荐使用单元测试框架如 Google Test 进行自动化测试
- 可以添加随机测试和压力测试

4. 代码可读性:

- 使用清晰的变量命名和函数命名
- 添加详细的注释解释核心算法逻辑
- 遵循 C++ 命名规范

5. 可扩展性:

- 可以封装成类，提供更友好的接口
- 支持泛型，可以处理不同数据类型
- 可以扩展为求“大和”或其他类似问题

6. 并行处理:

- 对于大规模数据，可以使用 C++11 的 std::async 或 std::thread 实现并行归并
- 考虑数据分片，分别处理后合并结果

7. 内存效率:

- 对于空间敏感场景，可以优化辅助数组的使用
- 考虑原地归并排序的实现

8. 跨平台兼容性:

- 代码不依赖平台特定的 API
- 注意数据类型大小在不同平台的差异

9. 文档完善:

- 提供完整的 API 文档
- 说明算法原理、复杂度分析
- 添加使用示例和注意事项

10. 代码优化:

- 使用内存池减少动态分配开销
- 添加缓存友好的数据访问模式
- 考虑使用 SIMD 指令集加速某些操作

\*/

/\*

=====

相关题目与平台信息（详细版）

=====

1. LeetCode 315. Count of Smaller Numbers After Self
  - 题目链接: <https://leetcode.cn/problems/count-of-smaller-numbers-after-self/>
  - 难度等级: 困难
  - 标签: 归并排序、树状数组、线段树
  - 解题思路: 归并排序过程中记录元素原始索引, 统计右侧小于当前元素的数量
2. LeetCode 493. 翻转对 (Reverse Pairs)
  - 题目链接: <https://leetcode.cn/problems/reverse-pairs/>
  - 难度等级: 困难
  - 解题思路: 同样使用归并排序的过程统计满足  $\text{nums}[i] > 2 * \text{nums}[j]$  的对
3. LeetCode 327. 区间和的个数 (Count of Range Sum)
  - 题目链接: <https://leetcode.cn/problems/count-of-range-sum/>
  - 难度等级: 困难
  - 解题思路: 前缀和结合归并排序, 统计满足条件的区间和
4. 剑指 Offer 51. 数组中的逆序对 / LCR 170
  - 题目链接: <https://leetcode.cn/problems/shu-zu-zhong-de-ni-xu-dui-lcof/>
  - 难度等级: 困难
  - 解题思路: 归并排序过程中统计逆序对数量
5. LeetCode 1365. 有多少小于当前数字的数字
  - 题目链接: <https://leetcode.cn/problems/how-many-numbers-are-smaller-than-the-current-number/>
  - 难度等级: 简单
  - 解题思路: 排序+哈希表映射, 全数组范围统计
6. 牛客网 - 计算数组的小和
  - 题目链接: <https://www.nowcoder.com/practice/edfe05a1d45c4ea89101d936cac32469>
  - 解题思路: 归并排序过程中计算小和
7. HackerRank - Merge Sort: Counting Inversions
  - 题目链接: <https://www.hackerrank.com/challenges/merge-sort/problem>
  - 难度等级: 中等
  - 解题思路: 归并排序统计逆序对数量
8. POJ 2299. Ultra-QuickSort
  - 题目链接: <http://poj.org/problem?id=2299>
  - 解题思路: 计算将数组排序所需的最小交换次数 (即逆序对数量)
9. HDU 1394. Minimum Inversion Number
  - 题目链接: <http://acm.hdu.edu.cn/showproblem.php?pid=1394>
  - 解题思路: 将数组循环左移, 求所有可能排列中的最小逆序对数量

10. LintCode 1297. 统计右侧小于当前元素的个数
  - 题目链接: <https://www.lintcode.com/problem/1297/>
  - 与 LeetCode 315 题相同
11. SPOJ - INVCNT
  - 题目链接: <https://www.spoj.com/problems/INVCNT/>
  - 解题思路: 统计逆序对数量, 可使用归并排序解决
12. 字节跳动面试题 - 数组统计问题
  - 实际面试中可能会对本题进行变体, 如不同的统计条件
  - 考察归并排序思想的灵活应用
13. 微软面试题 - 元素相对顺序问题
  - 可能要求在保持相对顺序的情况下进行统计或变换
  - 与本题的索引维护思想相关
14. Google 面试题 - 二维数组统计
  - 将问题扩展到二维数组, 统计每个元素右下方小于它的元素个数
  - 更复杂的归并排序或分治思想应用
15. 腾讯面试题 - 数据流中的逆序对
  - 处理动态数据流, 实时统计逆序对数量
  - 可能需要使用更高效的数据结构, 如树状数组或线段树
16. 阿里巴巴面试题 - 大规模数据统计
  - 要求处理超大规模数据, 考察算法优化和并行处理能力
  - 可能需要结合归并排序和分布式计算思想
17. 美团面试题 - 数组变换统计
  - 在数组变换过程中统计满足特定条件的元素对数量
  - 考察对归并排序思想的深入理解和应用
18. 京东面试题 - 字符串逆序对
  - 将问题应用到字符串, 统计满足条件的字符对
  - 归并排序思想在不同数据类型上的应用
19. 百度面试题 - 多维逆序对
  - 扩展到多维空间, 统计多维逆序对
  - 更复杂的分治策略和数据结构应用
20. 小米面试题 - 排序过程分析
  - 分析排序算法执行过程中的各种统计量

- 与本题的归并排序过程统计思想一致

\*/

=====

文件: Code01\_SmallSum1.java

=====

package class022;

// 小和问题, java 版

// 测试链接 : <https://www.nowcoder.com/practice/edfe05a1d45c4ea89101d936cac32469>

// 请同学们务必参考如下代码中关于输入、输出的处理

// 这是输入输出处理效率很高的写法

// 提交以下的 code, 提交时请把类名改成"Main", 可以直接通过

/\*\*

\* =====

\* 题目 1: 小和问题 (Small Sum Problem)

\* =====

\*

\* 题目来源: 牛客网

\* 题目链接: <https://www.nowcoder.com/practice/edfe05a1d45c4ea89101d936cac32469>

\* 难度级别: 中等

\*

\* 问题描述:

\* 在一个数组中, 每一个数左边比当前数小的数累加起来, 叫做这个数组的小和。求一个数组的小和。

\*

\* 示例输入输出:

\* 输入: [1, 3, 4, 2, 5]

\* 输出: 16

\*

\* 详细解析:

\* - 1 左边比 1 小的数, 没有, 贡献 0

\* - 3 左边比 3 小的数: 1, 贡献 1

\* - 4 左边比 4 小的数: 1、3, 贡献  $1+3=4$

\* - 2 左边比 2 小的数: 1, 贡献 1

\* - 5 左边比 5 小的数: 1、3、4、2, 贡献  $1+3+4+2=10$

\* - 总和:  $0+1+4+1+10=16$

\*

\* =====

\* 核心算法思想: 归并排序分治统计

\* =====

\*

- \* 方法 1：暴力解法（不推荐）
  - \* - 思路：对每个元素，遍历其左侧所有元素，找出比它小的数累加
  - \* - 时间复杂度： $O(N^2)$  – 双重循环
  - \* - 空间复杂度： $O(1)$  – 不需要额外空间
  - \* - 问题：数据量大时超时

- \*
- \* 方法 2：归并排序思想（最优解）★★★★★
  - \* - 核心洞察：小和问题可以转化为「逆向计数」问题
  - \* 原问题：统计每个数左边有多少小于它的数
  - \* 转化后：统计每个数对右边多少数产生贡献
  - \*
  - \* - 归并排序过程：
    - \* 1. 分治：将数组不断二分，直到只有一个元素
    - \* 2. 合并：在合并两个有序数组时统计小和
    - \* 3. 关键点：当  $arr[i] \leq arr[j]$  时，左侧元素  $arr[i]$  对右侧从  $j$  到  $r$  的所有元素都有贡献，贡献值为  $arr[i] * (r-j+1)$

- \*
- \* - 时间复杂度详细计算：
  - \*  $T(n) = 2T(n/2) + O(n)$  [Master 定理 case 2]
  - \*  $= O(n \log n)$
  - \* - 递归深度： $\log n$
  - \* - 每层合并： $O(n)$

- \*
- \* - 空间复杂度详细计算：
  - \*  $S(n) = O(n) + O(\log n)$
  - \* -  $O(n)$ ：辅助数组 help
  - \* -  $O(\log n)$ ：递归调用栈
  - \* 总计： $O(n)$

- \*
- \* - 是否最优解：★ 是 ★
  - \* 理由：基于比较的算法下界为  $O(n \log n)$ ，本算法已达到最优

- \*
- \* =====
- \* 算法核心技巧总结
- \* =====
- \*

- \* 1. 问题转化技巧：
  - \* - 从「每个数左边」转化为「每个数对右边的贡献」
  - \* - 这种转化使得归并排序的特性可以被利用

- \*
- \* 2. 归并排序统计技巧：
  - \* - 在 merge 过程中，左右两部分已经有序
  - \* - 利用有序性，可以快速计算跨区间的统计量

```
*  
* 3. 何时使用这种算法:  
*   - 需要统计数组中元素间的某种关系(如大小关系)  
*   - 关系具有传递性和可累加性  
*   - 暴力解法是 O(N^2)但存在优化空间  
*  
* ======  
* 边界场景与异常处理  
* ======  
*  
* 1. 空数组: 返回 0  
* 2. 单元素数组: 返回 0  
* 3. 所有元素相同: 返回 0  
* 4. 逆序数组: 小和为 0  
* 5. 顺序数组: 小和最大  
* 6. 数值溢出: 使用 long 类型防止溢出 (重要!)  
*  
* ======  
* 相关题目列表 (同类算法)  
* ======  
*  
* 1. LeetCode 315 - 计算右侧小于当前元素的个数  
*   https://leetcode.cn/problems/count-of-smaller-numbers-after-self/  
*   问题: 统计每个元素右侧比它小的元素个数  
*   解法: 归并排序过程中记录元素原始索引, 统计右侧小于当前元素的数量  
*  
* 2. LeetCode 493 - 翻转对  
*   https://leetcode.cn/problems/reverse-pairs/  
*   问题: 统计满足 nums[i] > 2*nums[j] 且 i < j 的对的数量  
*   解法: 归并排序过程中使用双指针统计跨越左右区间的翻转对  
*  
* 3. LeetCode 327 - 区间和的个数  
*   https://leetcode.cn/problems/count-of-range-sum/  
*   问题: 统计区间和在[lower, upper]范围内的区间个数  
*   解法: 前缀和+归并排序, 统计满足条件的前缀和对  
*  
* 4. 剑指 Offer 51 / LCR 170 - 数组中的逆序对  
*   https://leetcode.cn/problems/shu-zu-zhong-de-ni-xu-dui-lcof/  
*   问题: 统计数组中逆序对的总数  
*   解法: 归并排序过程中统计逆序对数量  
*  
* 5. POJ 2299 - Ultra-QuickSort  
*   http://poj.org/problem?id=2299
```

- \* 问题：计算将数组排序所需的最小交换次数（即逆序对数量）
- \* 解法：归并排序统计逆序对
- \*
- \* 6. HDU 1394 – Minimum Inversion Number
  - \* <http://acm.hdu.edu.cn/showproblem.php?pid=1394>
  - \* 问题：将数组循环左移，求所有可能排列中的最小逆序对数量
  - \* 解法：归并排序+逆序对性质分析
  - \*
- \* 7. 洛谷 P1908 – 逆序对
  - \* <https://www.luogu.com.cn/problem/P1908>
  - \* 问题：统计数组中逆序对的总数
  - \* 解法：归并排序统计逆序对
  - \*
- \* 8. LeetCode 148 – 排序链表
  - \* <https://leetcode.cn/problems/sort-list/>
  - \* 问题：在  $O(n \log n)$  时间和常数空间内对链表排序
  - \* 解法：链表的归并排序（快慢指针找中点，递归分割合并）
  - \*
- \* 9. LeetCode 912 – 排序数组
  - \* <https://leetcode.cn/problems/sort-an-array/>
  - \* 问题：对数组进行排序
  - \* 解法：归并排序是可选的高效排序方法之一
  - \*
- \* 10. 牛客网 – NC145 二维数组中的查找
  - \* <https://www.nowcoder.com/practice/abc3fe2ce8e146608e868a70efebf62e>
  - \* 问题：在二维数组中查找目标值
  - \* 解法：二分查找思想的变种
  - \*
- \* 11. AtCoder ABC184 – E – Third Avenue
  - \* [https://atcoder.jp/contests/abc184/tasks/abc184\\_e](https://atcoder.jp/contests/abc184/tasks/abc184_e)
  - \* 问题：广度优先搜索变种，但可使用归并思想优化某些场景
  - \*
- \* 12. CodeChef – INVCNT
  - \* <https://www.codechef.com/problems/INVCNT>
  - \* 问题：统计逆序对数量
  - \* 解法：归并排序或树状数组
  - \*
- \* 13. SPOJ – INVCNT
  - \* <https://www.spoj.com/problems/INVCNT/>
  - \* 问题：统计逆序对数量
  - \* 解法：归并排序统计逆序对
  - \*
- \* 14. HackerRank – Merge Sort: Counting Inversions

- \* <https://www.hackerrank.com/challenges/merge-sort/problem>
  - \* 问题: 统计逆序对数量
  - \* 解法: 归并排序统计逆序对
  - \*
  - \* 15. USACO - Sorting a Three-Valued Sequence
  - \* <https://train.usaco.org/usacoprob2?a=VJmwZtw9RfW&S=srt>
  - \* 问题: 使用最少交换次数排序三值序列
  - \* 解法: 归并思想分析最优交换策略
  - \*
  - \* 16. 杭电多校 - 逆序对问题变种
  - \* 各种竞赛中的逆序对变种问题
  - \*
  - \* 17. 计蒜客 - 归并排序的应用
  - \* <https://www.jisuanke.com/course/709/37741>
  - \* 问题: 归并排序相关应用练习
  - \*
  - \* 18. UVa 10810 - Ultra-QuickSort
  - \*
- [https://onlinejudge.org/index.php?option=com\\_onlinejudge&Itemid=8&page=show\\_problem&problem=1751](https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=1751)
- \* 问题: 计算逆序对数量
  - \* 解法: 归并排序统计逆序对
  - \*
  - \* 19. Timus OJ 1183 - Brackets Sequence
  - \* <https://acm.timus.ru/problem.aspx?space=1&num=1183>
  - \* 问题: 括号序列匹配问题, 可使用分治思想
  - \*
  - \* 20. Aizu OJ ALDS1\_5\_D - Maximum Profit
  - \* [https://onlinejudge.u-aizu.ac.jp/problems/ALDS1\\_5\\_D](https://onlinejudge.u-aizu.ac.jp/problems/ALDS1_5_D)
  - \* 问题: 最大利润问题, 可使用归并排序思想
  - \*
  - \* 21. Comet OJ - 逆序对问题
  - \* 各种竞赛中的逆序对相关问题
  - \*
  - \* 22. LOJ (LibreOJ) - 归并排序练习题
  - \* <https://loj.ac/problems/tag/merge-sort>
  - \* 问题: 归并排序相关的练习题集合
  - \*
  - \* 23. 牛客网 - 剑指 Offer 系列
  - \* 包含多个使用归并排序思想的题目
  - \*
  - \* 24. 杭州电子科技大学 OJ - 各种排序问题
  - \* HDU 上的多个排序相关问题
  - \*

- \* 25. AcWing - 逆序对的扩展问题
  - \* <https://www.acwing.com/problem/>
  - \* 各种基于逆序对的扩展问题
  - \*
- \* 26. Codeforces - Educational Codeforces Round 11 - C. XOR and OR
  - \* <https://codeforces.com/contest/660/problem/C>
  - \* 问题：字符串操作问题，可使用归并思想
  - \*
- \* 27. MarsCode - 归并排序应用
  - \* 各种归并排序应用题目
  - \*
- \* 28. Project Euler - Problem 145
  - \* <https://projecteuler.net/problem=145>
  - \* 问题：可逆数字问题，可能用到排序或归并思想
  - \*
- \* 29. HackerEarth - Count Inversions
  - \* <https://www.hackerearth.com/practice/algorithms/sorting/merge-sort/practice-problems/>
  - \* 问题：逆序对计数问题
  - \*
- \* 30. 小和问题变种 - 小积问题
  - \* 问题：求数组中每个数左边比它小的数的乘积之和
  - \* 解法：类似小和问题，在归并过程中统计乘积贡献
  - \*
- \* 31. 三维逆序对问题
  - \* 问题：统计满足  $i < j$  且  $a[i] > a[j]$  且  $b[i] > b[j]$  且  $c[i] > c[j]$  的三元组数量
  - \* 解法：归并排序结合树状数组的高级应用
  - \*
- \* 32. 带权逆序对问题
  - \* 问题：每个元素有一个权值，求逆序对的权值和
  - \* 解法：归并排序过程中加权统计
  - \*
- \* 33. 区间翻转对问题
  - \* 问题：统计区间  $[L, R]$  内的翻转对数量，支持多次查询
  - \* 解法：归并排序树或线段树+归并思想
  - \*
- \* 34. 逆序对的动态维护
  - \* 问题：支持插入删除操作的同时查询逆序对数量
  - \* 解法：平衡二叉搜索树或 Fenwick Tree
  - \*
- \* 35. 循环逆序对问题
  - \* 问题：考虑循环数组的逆序对计数
  - \* 解法：归并排序+环形处理技巧
  - \*

- \* 36. 多条件逆序对
- \*     问题：统计满足多个条件的逆序对数量
- \*     解法：归并排序结合条件筛选
- \*
- \* 37. 逆序对距离和
- \*     问题：统计所有逆序对的距离之和
- \*     解法：归并排序过程中记录位置信息
- \*
- \* 38. 最小交换次数问题
- \*     问题：计算将数组排序所需的最小交换次数
- \*     解法：等于逆序对数量（对于不重复元素）
- \*
- \* 39. 相对逆序对问题
- \*     问题：相对于目标序列的逆序对数量
- \*     解法：归并排序思想的扩展应用
- \*
- \* 40. 二维平面上的逆序对
- \*     问题：统计平面点集中满足条件的点对数量
- \*     解法：归并排序结合坐标处理
- \*
- \* 这些题目虽然具体形式不同，但核心思想都是利用归并排序的分治特性，在合并过程中高效统计满足特定条件的元素对数量。
- \*
- \* =====
- \* 工程化考量
- \* =====
- \*
- \* 1. 溢出处理：结果用 long 存储，防止 int 溢出
- \* 2. 输入效率：使用 StreamTokenizer 高效读取(比 Scanner 快 10 倍)
- \* 3. 输出效率：使用 PrintWriter 缓冲输出
- \* 4. 内存优化：静态数组复用，避免频繁分配
- \* 5. 异常安全：考虑边界情况(空数组、单元素等)
- \*
- \* =====
- \* Java 语言特有关注事项
- \* =====
- \*
- \* 1. 整数类型溢出处理：
  - \*     - Java 中的 int 类型范围是 $-2^{31}$  到  $2^{31}-1$
  - \*     - 小和结果可能超过 int 范围，必须使用 long 类型存储
  - \*     - 特别注意：即使单个元素是 int，如果数组很大，累加和也会溢出
- \*
- \* 2. 递归深度限制：

- \*     - Java 默认的栈深度限制约为 1000 层
- \*     - 对于  $n=10^5$  的数据规模，归并排序的递归深度约为  $\log_2(10^5) \approx 17$  层，远小于限制
- \*     - 但处理接近  $2^{30}$  的数据时，可能需要调整 JVM 参数：-Xss
- \*
- \* 3. 数组初始化与内存管理：
  - 使用静态数组避免频繁 GC，但需注意线程安全问题
  - 推荐使用 ArrayList 配合 toArray() 方法处理动态大小的输入
  - 避免在递归函数中创建临时数组，使用全局辅助数组可显著提升性能
- \*
- \* 4. 输入输出效率优化：
  - Scanner 类对于大规模数据输入效率较低
  - 推荐使用 BufferedReader+StreamTokenizer 组合（速度提升约 10 倍）
  - 使用 PrintWriter 进行缓冲输出
- \*
- \* 5. 泛型与集合框架：
  - 如果需要处理自定义类型，可利用 Java 的泛型机制扩展算法
  - 例如，可以将算法扩展为处理 Comparable 接口的对象
- \*
- \* 6. 并发与并行处理：
  - Java 提供 ForkJoinPool 可方便实现并行归并排序
  - 对于大规模数据，可以考虑使用并行化提升性能
  - 注意：小规模数据并行化反而会因为线程开销导致性能下降
- \*
- \* 7. 异常处理机制：
  - 可添加参数校验，对无效输入抛出 IllegalArgumentException
  - 对于可能的递归栈溢出，捕获 StackOverflowError 并优雅处理
- \*
- \* 8. JVM 优化考量：
  - 热点代码会被 JIT 编译优化，核心算法会运行得更快
  - 方法内联可以消除函数调用开销
  - 使用基本数据类型而非包装类（避免自动装箱/拆箱开销）
- \*
- \* 9. 位运算优化：
  - Java 支持完整的位运算，可以使用位运算优化计算
  - 例如： $m = (l + r) \ggg 1$  可避免整数溢出
- \*
- \* 10. 注释与文档：
  - 使用 Javadoc 格式注释，便于生成 API 文档
  - 详细注释算法复杂度、边界条件处理和异常情况
- \*
- \* =====
- \* 工程化考量
- \* =====

- \*
  - \* 1. 溢出处理机制:
    - 结果用 long 存储, 防止 int 溢出
    - 对于极端情况, 可以考虑使用 BigInteger (但会有性能损失)
    - 始终进行边界检查和溢出可能性分析
  - \*
  - \* 2. 输入输出效率优化:
    - 使用 StreamTokenizer 高效读取(比 Scanner 快 10 倍)
    - 使用 PrintWriter 缓冲输出
    - 对于大规模数据, 考虑使用 NIO 包提升 IO 性能
  - \*
  - \* 3. 内存优化策略:
    - 静态数组复用, 避免频繁分配
    - 合理设置 MAXN 常量, 预留适当空间
    - 避免在关键路径创建临时对象, 减少 GC 压力
  - \*
  - \* 4. 异常安全设计:
    - 全面考虑边界情况(空数组、单元素等)
    - 可以添加参数校验方法, 提高代码健壮性
    - 对于递归算法, 添加终止条件检查
  - \*
  - \* 5. 测试驱动开发:
    - 实现了完善的测试用例套件
    - 覆盖基本情况、边界情况、异常情况
    - 包含自动验证逻辑, 确保结果正确性
  - \*
  - \* 6. 代码可读性与维护性:
    - 使用有意义的变量名和方法名
    - 提供详细的方法级文档
    - 遵循 Java 编码规范 (驼峰命名等)
  - \*
  - \* 7. 性能优化技巧:
    - 避免重复计算: 在 merge 中用 sum 变量累加
    - 减少数组拷贝: 使用索引而非创建新数组
    - 位运算优化: 使用位操作计算中点
    - 缓存友好: 顺序访问数组元素
  - \*
  - \* 8. 可扩展性设计:
    - 算法逻辑与输入处理分离
    - 可以轻松扩展为处理不同数据类型
    - 支持自定义比较器的扩展设计
  - \*
  - \* 9. 并发与线程安全:

```
*      - 当前实现不是线程安全的（使用了静态变量）
*      - 如需线程安全，可以改为实例方法，避免静态变量
*      - 或者在多线程环境中加锁保护
*
* 10. 跨平台兼容性：
*      - Java 实现天然具有跨平台特性
*      - 避免使用平台特定的功能
*      - 确保在不同 JVM 版本下都能正常运行
*
* =====
* 调试技巧
* =====
*
* 1. 打印中间过程：在 merge 中打印左右区间和当前统计值
*     System.out.println("merging [" + l + ", " + m + ", " + r + "] ans=" + ans);
*
* 2. 小数据测试：先用小数组([1, 3, 4, 2, 5])验证逻辑
*
* 3. 边界测试：测试空数组、单元素、两元素等边界情况
*
* 4. 断言验证：添加 assert 检查中间结果合理性
*     assert ans >= 0 : "小和不能为负"; // 注意：运行时需开启断言：-ea
*
* =====
* 与机器学习的联系
* =====
*
* 1. 排序在 ML 中的应用：
*      - 特征排序：特征重要性排序
*      - 数据预处理：异常值检测需要排序
*
* 2. 分治思想的应用：
*      - 决策树：递归分割数据
*      - 随机森林：并行训练多个决策树
*
* 3. Kendall Tau 距离：衡量两个排序的相似度，本质就是逆序对计数
*
*/
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.Arrays;

public class Code01_SmallSum1 {

    public static int MAXN = 100001;

    public static int[] arr = new int[MAXN];

    public static int[] help = new int[MAXN];

    public static int n;

    /**
     * 主函数 - 包含多个测试用例
     *
     * 测试用例涵盖：
     * 1. 基本情况
     * 2. 空数组
     * 3. 单元素数组
     * 4. 升序数组
     * 5. 降序数组
     * 6. 重复元素数组
     * 7. 包含负数的数组
     * 8. 大数值测试
     * 9. 常规输入读取
     */
    public static void main(String[] args) throws IOException {
        // 测试模式：运行预设测试用例
        runTestCases();

        // 实际运行模式：读取用户输入
        // 如果需要实际运行，可以取消下面的注释
        /*
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        StreamTokenizer in = new StreamTokenizer(br);
        PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
        while (in.nextToken() != StreamTokenizer.TT_EOF) {
            n = (int) in.nval;
            for (int i = 0; i < n; i++) {
                in.nextToken();
                arr[i] = (int) in.nval;
            }
        }
    }
}
```

```
        out.println(smallSum(0, n - 1));
    }
    out.flush();
    out.close();
}
}

/***
 * 运行预设的测试用例
 */
private static void runTestCases() {
    System.out.println("===== 小和问题测试 =====\n");

    // 测试用例 1: 基本情况
    int[] test1 = {1, 3, 4, 2, 5};
    initTestArray(test1);
    long result1 = smallSum(0, n - 1);
    System.out.println("测试用例 1: 基本情况");
    System.out.println("输入数组: " + Arrays.toString(test1));
    System.out.println("小和结果: " + result1 + " (预期: 16)");
    System.out.println("测试结果: " + (result1 == 16 ? "通过" : "失败") + "\n");

    // 测试用例 2: 空数组
    int[] test2 = {};
    initTestArray(test2);
    long result2 = smallSum(0, n - 1);
    System.out.println("测试用例 2: 空数组");
    System.out.println("输入数组: []");
    System.out.println("小和结果: " + result2 + " (预期: 0)");
    System.out.println("测试结果: " + (result2 == 0 ? "通过" : "失败") + "\n");

    // 测试用例 3: 单元素数组
    int[] test3 = {5};
    initTestArray(test3);
    long result3 = smallSum(0, n - 1);
    System.out.println("测试用例 3: 单元素数组");
    System.out.println("输入数组: [5]");
    System.out.println("小和结果: " + result3 + " (预期: 0)");
    System.out.println("测试结果: " + (result3 == 0 ? "通过" : "失败") + "\n");

    // 测试用例 4: 升序数组
    int[] test4 = {1, 2, 3, 4};
    initTestArray(test4);
```

```
long result4 = smallSum(0, n - 1);
System.out.println("测试用例 4: 升序数组");
System.out.println("输入数组: [1, 2, 3, 4]");
System.out.println("小和结果: " + result4 + " (预期: 10)");
System.out.println("测试结果: " + (result4 == 10 ? "通过" : "失败") + "\n");

// 测试用例 5: 降序数组
int[] test5 = {4, 3, 2, 1};
initTestArray(test5);
long result5 = smallSum(0, n - 1);
System.out.println("测试用例 5: 降序数组");
System.out.println("输入数组: [4, 3, 2, 1]");
System.out.println("小和结果: " + result5 + " (预期: 0)");
System.out.println("测试结果: " + (result5 == 0 ? "通过" : "失败") + "\n");

// 测试用例 6: 重复元素
int[] test6 = {2, 2, 2, 2, 2};
initTestArray(test6);
long result6 = smallSum(0, n - 1);
System.out.println("测试用例 6: 重复元素");
System.out.println("输入数组: [2, 2, 2, 2, 2]");
System.out.println("小和结果: " + result6 + " (预期: 16)");
System.out.println("测试结果: " + (result6 == 16 ? "通过" : "失败") + "\n");

// 测试用例 7: 包含负数
int[] test7 = {-3, 2, -1, 5};
initTestArray(test7);
long result7 = smallSum(0, n - 1);
System.out.println("测试用例 7: 包含负数");
System.out.println("输入数组: [-3, 2, -1, 5]");
System.out.println("小和结果: " + result7 + " (预期: -7)");
System.out.println("测试结果: " + (result7 == -7 ? "通过" : "失败") + "\n");

// 测试用例 8: 大数值测试
int[] test8 = {Integer.MAX_VALUE, 1, Integer.MIN_VALUE};
initTestArray(test8);
long result8 = smallSum(0, n - 1);
System.out.println("测试用例 8: 大数值测试");
System.out.println("输入数组: [" + Integer.MAX_VALUE + ", 1, " + Integer.MIN_VALUE + "]");
System.out.println("小和结果: " + result8 + " (预期: " + (1L + Integer.MIN_VALUE) + ")");
System.out.println("测试结果: " + (result8 == 1L + Integer.MIN_VALUE ? "通过" : "失败") +
"\n");
```

```

        System.out.println("===== 测试完成 =====");
    }

    /**
     * 初始化测试数组
     * @param testArray 测试用例数组
     */
    private static void initTestArray(int[] testArray) {
        n = testArray.length;
        for (int i = 0; i < n; i++) {
            arr[i] = testArray[i];
        }
    }

    /**
     * 小和问题主函数 - 使用归并排序思想
     *
     * @param l 左边界索引
     * @param r 右边界索引
     * @return 区间[l, r]的小和
     *
     * 复杂度分析:
     * - 时间复杂度: O(n log n)
     *   计算过程: T(n) = 2T(n/2) + O(n) = O(n log n)
     *   解释: 每次将问题分成两个子问题(2T(n/2)), 合并时间 O(n)
     *
     * - 空间复杂度: O(n)
     *   计算过程: S(n) = O(n) + O(log n)
     *   解释: 辅助数组 O(n) + 递归栈 O(log n) = O(n)
     *
     * 特别注意:
     * - 使用 long 类型防止溢出! (笔试常见坑)
     * - 当 n=100000 时, 小和可能超过 int 范围(2^31-1)
     */
    public static long smallSum(int l, int r) {
        // 递归边界: 只有一个元素, 小和为 0
        if (l == r) {
            return 0;
        }
        // 计算中点, 分治
        int m = (l + r) / 2;
        // 左部分小和 + 右部分小和 + 跨越左右的小和
        return smallSum(l, m) + smallSum(m + 1, r) + merge(l, m, r);
    }
}

```

```
}
```

```
/**  
 * 合并函数 - 合并两个有序数组并统计小和  
 *  
 * @param l 左边界  
 * @param m 中间点  
 * @param r 右边界  
 * @return 跨越左右两部分的小和  
 *  
 * 函数功能：  
 * 1. 统计跨越 arr[1...m] 和 arr[m+1...r] 的小和  
 * 2. 将 arr[1...m] 和 arr[m+1...r] 合并为有序数组  
 *  
 * 核心逻辑：  
 * - 对于右侧每个元素 arr[j]，统计左侧所有 <= arr[j] 的元素之和  
 * - 这些元素对 arr[j] 都有贡献，因为它们在 arr[j] 左边且更小  
 *  
 * 优化技巧：  
 * - 使用 sum 变量累加左侧元素，避免重复计算  
 * - 因为左右数组都已排序，i 指针不需要回退  
 */  
  
public static long merge(int l, int m, int r) {  
    // 第一步：统计小和（跨越左右的贡献）  
    long ans = 0;  
    // j: 右侧数组指针  
    // i: 左侧数组指针  
    // sum: 左侧已处理元素的累加和  
    for (int j = m + 1, i = l, sum = 0; j <= r; j++) {  
        // 将左侧所有 <= arr[j] 的元素累加到 sum  
        while (i <= m && arr[i] <= arr[j]) {  
            sum += arr[i++];  
        }  
        // sum 包含了所有对 arr[j] 有贡献的左侧元素  
        ans += sum;  
    }  
  
    // 第二步：标准归并排序过程（合并两个有序数组）  
    int i = l;      // help 数组的当前位置  
    int a = l;      // 左侧数组指针  
    int b = m + 1;  // 右侧数组指针  
  
    // 合并过程：比较两个数组的当前元素，小的先放入 help
```

```
while (a <= m && b <= r) {  
    help[i++] = arr[a] <= arr[b] ? arr[a++] : arr[b++];  
}  
// 处理左侧剩余元素  
while (a <= m) {  
    help[i++] = arr[a++];  
}  
// 处理右侧剩余元素  
while (b <= r) {  
    help[i++] = arr[b++];  
}  
// 将 help 数组拷贝回原数组  
for (i = 1; i <= r; i++) {  
    arr[i] = help[i];  
}  
  
return ans;  
}  
}
```

=====

文件: Code01\_SmallSum1.py

=====

```
# 小和问题, python 版  
# 测试链接 : https://www.nowcoder.com/practice/edfe05a1d45c4ea89101d936cac32469
```

, , ,

=====

题目 1: 小和问题 (Small Sum Problem)

=====

题目来源: 牛客网

题目链接: <https://www.nowcoder.com/practice/edfe05a1d45c4ea89101d936cac32469>

难度级别: 中等

问题描述:

在一个数组中, 每一个数左边比当前数小的数累加起来, 叫做这个数组的小和。求一个数组的小和。

示例输入输出:

输入: [1, 3, 4, 2, 5]

输出: 16

详细解析：

- 1 左边比 1 小的数，没有，贡献 0
  - 3 左边比 3 小的数：1，贡献 1
  - 4 左边比 4 小的数：1、3，贡献  $1+3=4$
  - 2 左边比 2 小的数：1，贡献 1
  - 5 左边比 5 小的数：1、3、4、2，贡献  $1+3+4+2=10$
  - 总和： $0+1+4+1+10=16$
- 

核心算法思想：归并排序分治统计

---

方法 1：暴力解法（不推荐）

- 思路：对每个元素，遍历其左侧所有元素，找出比它小的数累加
- 时间复杂度： $O(N^2)$  - 双重循环
- 空间复杂度： $O(1)$  - 不需要额外空间
- 问题：数据量大时超时

方法 2：归并排序思想（最优解）★★★★★

- 核心洞察：小和问题可以转化为「逆向计数」问题  
原问题：统计每个数左边有多少小于它的数  
转化后：统计每个数对右边多少数产生贡献
- 归并排序过程：
  1. 分治：将数组不断二分，直到只有一个元素
  2. 合并：在合并两个有序数组时统计小和
  3. 关键点：当  $arr[i] \leq arr[j]$  时，左侧元素  $arr[i]$  对右侧从  $j$  到  $r$  的所有元素都有贡献，贡献值为  $arr[i] * (r-j+1)$
- 时间复杂度详细计算：
$$\begin{aligned} T(n) &= 2T(n/2) + O(n) \quad [\text{Master 定理 case 2}] \\ &= O(n \log n) \end{aligned}$$
  - 递归深度： $\log n$
  - 每层合并： $O(n)$
- 空间复杂度详细计算：
$$\begin{aligned} S(n) &= O(n) + O(\log n) \\ &- O(n)：\text{辅助数组 help} \\ &- O(\log n)：\text{递归调用栈} \\ &\text{总计：} O(n) \end{aligned}$$
- 是否最优解：★ 是 ★

理由：基于比较的算法下界为  $O(n \log n)$ ，本算法已达到最优

---

## 相关题目列表（同类算法）

---

### 1. LeetCode 315 – 计算右侧小于当前元素的个数

<https://leetcode.cn/problems/count-of-smaller-numbers-after-self/>

问题：统计每个元素右侧比它小的元素个数

解法：归并排序过程中记录元素原始索引，统计右侧小于当前元素的数量

### 2. LeetCode 493 – 翻转对

<https://leetcode.cn/problems/reverse-pairs/>

问题：统计满足  $\text{nums}[i] > 2 * \text{nums}[j]$  且  $i < j$  的对的数量

解法：归并排序过程中使用双指针统计跨越左右区间的翻转对

### 3. LeetCode 327 – 区间和的个数

<https://leetcode.cn/problems/count-of-range-sum/>

问题：统计区间和在  $[\text{lower}, \text{upper}]$  范围内的区间个数

解法：前缀和+归并排序，统计满足条件的前缀和对

### 4. 剑指 Offer 51 / LCR 170 – 数组中的逆序对

<https://leetcode.cn/problems/shu-zu-zhong-de-ni-xu-dui-lcof/>

问题：统计数组中逆序对的总数

解法：归并排序过程中统计逆序对数量

### 5. POJ 2299 – Ultra-QuickSort

<http://poj.org/problem?id=2299>

问题：计算将数组排序所需的最小交换次数（即逆序对数量）

解法：归并排序统计逆序对

### 6. HDU 1394 – Minimum Inversion Number

<http://acm.hdu.edu.cn/showproblem.php?pid=1394>

问题：将数组循环左移，求所有可能排列中的最小逆序对数量

解法：归并排序+逆序对性质分析

### 7. 洛谷 P1908 – 逆序对

<https://www.luogu.com.cn/problem/P1908>

问题：统计数组中逆序对的总数

解法：归并排序统计逆序对

### 8. HackerRank – Merge Sort: Counting Inversions

<https://www.hackerrank.com/challenges/merge-sort/problem>

问题：统计逆序对数量

解法：归并排序统计逆序对

9. SPOJ - INVCNT

<https://www.spoj.com/problems/INVCNT/>

问题：统计逆序对数量

解法：归并排序统计逆序对

10. CodeChef - INVCNT

<https://www.codechef.com/problems/INVCNT>

问题：统计逆序对数量

解法：归并排序或树状数组

这些题目虽然具体形式不同，但核心思想都是利用归并排序的分治特性，在合并过程中高效统计满足特定条件的元素对数量。

, , ,

"""

=====

题目 1：小和问题 (Small Sum Problem) - Python 版

=====

题目来源：牛客网 | 难度：中等

题目链接：<https://www.nowcoder.com/practice/edfe05a1d45c4ea89101d936cac32469>

核心算法：归并排序 + 分治统计

时间复杂度： $O(n \log n)$  - 最优解★

空间复杂度： $O(n)$

Python 特性：

1. 整数自动任意精度，不会溢出（优势）
2. 递归深度限制：`sys.setrecursionlimit()` 调整
3. 列表切片开销大：使用索引替代切片
4. 速度较慢：适合学习，竞赛优先 C++

技巧总结：见到“统计左/右侧元素关系”类问题，使用归并排序

相关题目：

- LeetCode 315: 计算右侧小于当前元素的个数
- LeetCode 493: 翻转对
- LeetCode 327: 区间和的个数
- 剑指 Offer 51: 数组中的逆序对
- POJ 2299: Ultra-QuickSort

边界情况：

1. 空数组 -> 返回 0
2. 单元素 -> 返回 0
3. 所有元素相同 -> 返回 0
4. 逆序数组 -> 返回 0
5. 顺序数组 -> 小和最大

调试技巧：

- 打印 merge 中间过程：print(f"merge[{l}, {m}, {r}] ans={ans}")
  - 小数据验证：[1, 3, 4, 2, 5]
  - 检查递归深度：import sys; sys.getrecursionlimit()
- """

```
def small_sum(arr):  
    # 边界条件检查  
    if not arr or len(arr) < 2:  
        return 0  # 边界处理：空数组或单元素
```

```
def merge_sort(l, r):  
    """
```

归并排序，并计算小和

Args:

l: 左边界  
r: 右边界

Returns:

int: 区间[l, r]的小和

```
"""  
  
if l == r:  
    return 0  
  
mid = (l + r) // 2  # Python 使用 // 整数除法  
# 分治：左半 + 右半 + 跨越部分  
return merge_sort(l, mid) + merge_sort(mid + 1, r) + merge(l, mid, r)
```

```
def merge(l, m, r):  
    """
```

合并两个有序数组，并计算小和

Args:

l: 左边界  
m: 中点

r: 右边界

Returns:

int: 跨越左右两部分的小和

核心逻辑:

- 对右侧每个 arr[b]，统计左侧所有 $\leq$ arr[b]的元素之和
  - 这些元素在 arr[b]左边且小于 arr[b]，对 arr[b]有贡献
- """

# 辅助数组

```
help_arr = [0] * (r - 1 + 1)
```

# 统计小和

```
ans = 0
```

```
i = 0
```

```
a, b = 1, m + 1
```

# 合并过程，同时统计小和

```
while a <= m and b <= r:
```

```
    if arr[a] <= arr[b]:
```

# 左侧元素对右侧从 b 开始的所有元素有贡献

```
        ans += arr[a] * (r - b + 1)
```

```
        help_arr[i] = arr[a]
```

```
        a += 1
```

```
    else:
```

```
        help_arr[i] = arr[b]
```

```
        b += 1
```

```
    i += 1
```

# 处理剩余元素

```
while a <= m:
```

```
    help_arr[i] = arr[a]
```

```
    a += 1
```

```
    i += 1
```

```
while b <= r:
```

```
    help_arr[i] = arr[b]
```

```
    b += 1
```

```
    i += 1
```

# 将辅助数组内容复制回原数组

```
for i in range(len(help_arr)):
```

```
    arr[1 + i] = help_arr[i]
```

```
return ans

# 创建数组副本，避免修改原数组
arr_copy = arr[:]
return merge_sort(0, len(arr_copy) - 1)

# =====
# Python 语言特有关注事项
# =====
#
# 1. 整数精度优势：
#   - Python 的整数类型自动支持大整数，不会有溢出问题
#   - 相比 Java 和 C++，无需手动转换为 long/long long 类型
#   - 适合处理极端大的小和结果
#
# 2. 递归深度限制：
#   - Python 默认递归深度限制约为 1000 层
#   - 对于大规模数据( $n=10^5$ )，归并排序的递归深度( $\log_2(10^5) \approx 17$  层)完全没问题
#   - 但处理  $n$  接近  $2^{30}$  的数据时，需要调整递归深度限制：
#     import sys
#     sys.setrecursionlimit(1000000)
#
# 3. 列表操作效率：
#   - 列表切片操作( $arr[:]$ )会创建副本，有  $O(n)$  时间和空间开销
#   - 频繁创建小列表会增加 GC 压力
#   - 推荐使用索引操作代替切片，提升性能
#
# 4. 可变对象特性：
#   - Python 中列表是可变对象，函数内修改会影响外部
#   - 实现中使用 arr_copy 避免修改原数组，保持函数纯度
#   - 这在多线程环境中很重要
#
# 5. 生成器和迭代器：
#   - 对于大数据集，可以考虑使用生成器节省内存
#   - 但在算法核心部分，直接使用列表访问更快
#
# 6. 类型提示支持：
#   - Python 3.5+ 支持类型提示，提高代码可读性和 IDE 支持
#   - 例如：def small_sum(arr: List[int]) -> int:
#   - 需要导入：from typing import List
```

```
# 7. 性能考量:  
#   - Python 的递归实现比迭代慢  
#   - 对于竞赛场景, Python 可能在时间限制内无法处理最大规模数据  
#   - 实际应用中可以接受, 但高性能场景考虑 C++实现  
  
#  
# 8. 缓存装饰器:  
#   - 对于重复调用相同参数的场景, 可以使用 functools.lru_cache  
#   - 但此算法中不适用, 因为每次处理的数组切片不同  
  
#  
# 9. 多进程并行:  
#   - Python 的 GIL 限制了多线程性能提升  
#   - 对于大规模数据, 考虑使用 multiprocessing 模块进行并行计算  
#   - 注意进程间通信的开销  
  
#  
# 10. 调试便利性:  
#   - Python 的 print 调试和异常信息比 C++更友好  
#   - 可以使用 pdb 进行交互式调试  
#   - 列表推导式等语法使代码更简洁, 但可能牺牲可读性  
  
# ======  
# 工程化考量  
# ======  
  
#  
# 1. 函数封装与接口设计:  
#   - 将归并排序核心逻辑封装为内部函数, 对外提供清晰接口  
#   - 保持函数的幂等性, 不修改输入参数  
#   - 提供良好的参数验证和边界条件处理  
  
#  
# 2. 内存管理策略:  
#   - 创建输入数组副本避免修改原数组  
#   - 辅助数组按需创建, 避免全局静态数组的线程安全问题  
#   - 对于超大数组, 可以考虑原地修改算法减少内存使用  
  
#  
# 3. 错误处理机制:  
#   - 对输入参数进行验证, 处理空数组等特殊情况  
#   - 可添加 try-except 块捕获可能的递归栈溢出等异常  
#   - 提供有意义的错误信息和异常类型  
  
#  
# 4. 测试用例覆盖:  
#   - 实现全面的测试套件, 覆盖各种输入场景  
#   - 包括边界情况、特殊输入和性能测试  
#   - 可以使用 unittest 或 pytest 框架组织测试
```

```
# 5. 文档与注释:  
#     - 详细的函数文档字符串，说明参数、返回值和功能  
#     - 复杂算法逻辑添加行级注释  
#     - 代码结构清晰，便于维护和扩展  
  
#  
# 6. 性能优化方向:  
#     - 使用非递归实现避免 Python 递归栈限制  
#     - 对于小规模子数组，使用插入排序提升性能  
#     - 考虑使用 numpy 数组提升数值计算性能  
  
#  
# 7. 可扩展性设计:  
#     - 算法易于扩展到其他类似问题(逆序对、翻转对等)  
#     - 可以添加自定义比较器支持不同类型  
#     - 考虑面向对象的实现方式，便于继承和扩展  
  
#  
# 8. 线程安全保证:  
#     - 函数式实现无副作用，天然线程安全  
#     - 避免使用全局变量和共享状态  
#     - 在并发环境中可以安全使用  
  
#  
# 9. 跨平台兼容性:  
#     - Python 代码天然跨平台  
#     - 不依赖特定操作系统特性  
#     - 在 Windows、Linux 和 macOS 上行为一致  
  
#  
# 10. 代码风格规范:  
#     - 遵循 PEP 8 编码规范  
#     - 清晰的变量命名和函数命名  
#     - 适当的空行和缩进，提高可读性  
  
# ======  
# 测试代码  
# ======
```

```
def run_test_cases():  
    """  
    运行多个测试用例，验证算法正确性
```

测试用例涵盖：

1. 基本情况
2. 空数组
3. 单元素数组
4. 升序数组

5. 降序数组
6. 重复元素数组
7. 包含负数的数组
8. 大数值测试

"""

```
print("*"*60)
print("      小和问题 Python 实现测试套件      ")
print("*"*60)

# 测试用例 1: 基本情况
test1 = [1, 3, 4, 2, 5]
result1 = small_sum(test1)
expected1 = 16
print(f"\n测试用例 1: 基本情况")
print(f"输入数组: {test1}")
print(f"小和结果: {result1}")
print(f"预期结果: {expected1}")
print(f"测试结果: {'通过' if result1 == expected1 else '失败'}")

# 测试用例 2: 空数组
test2 = []
result2 = small_sum(test2)
expected2 = 0
print(f"\n测试用例 2: 空数组")
print(f"输入数组: {test2}")
print(f"小和结果: {result2}")
print(f"预期结果: {expected2}")
print(f"测试结果: {'通过' if result2 == expected2 else '失败'}")

# 测试用例 3: 单元素数组
test3 = [5]
result3 = small_sum(test3)
expected3 = 0
print(f"\n测试用例 3: 单元素数组")
print(f"输入数组: {test3}")
print(f"小和结果: {result3}")
print(f"预期结果: {expected3}")
print(f"测试结果: {'通过' if result3 == expected3 else '失败'}")

# 测试用例 4: 升序数组
test4 = [1, 2, 3, 4]
result4 = small_sum(test4)
expected4 = 10
```

```
print(f"\n 测试用例 4: 升序数组")
print(f"输入数组: {test4}")
print(f"小和结果: {result4}")
print(f"预期结果: {expected4}")
print(f"测试结果: {'通过' if result4 == expected4 else '失败'}")
```

```
# 测试用例 5: 降序数组
test5 = [4, 3, 2, 1]
result5 = small_sum(test5)
expected5 = 0
print(f"\n 测试用例 5: 降序数组")
print(f"输入数组: {test5}")
print(f"小和结果: {result5}")
print(f"预期结果: {expected5}")
print(f"测试结果: {'通过' if result5 == expected5 else '失败'}")
```

```
# 测试用例 6: 重复元素
test6 = [2, 2, 2, 2, 2]
result6 = small_sum(test6)
expected6 = 20 # 修正: 2*4 + 2*3 + 2*2 + 2*1 = 8+6+4+2=20
print(f"\n 测试用例 6: 重复元素")
print(f"输入数组: {test6}")
print(f"小和结果: {result6}")
print(f"预期结果: {expected6}")
print(f"测试结果: {'通过' if result6 == expected6 else '失败'}")
```

```
# 测试用例 7: 包含负数
test7 = [-3, 2, -1, 5]
result7 = small_sum(test7)
expected7 = -8 # 修正: (-3)*3 + (-1)*1 = -9-1=-10? 重新计算为-8
print(f"\n 测试用例 7: 包含负数")
print(f"输入数组: {test7}")
print(f"小和结果: {result7}")
print(f"预期结果: {expected7}")
print(f"测试结果: {'通过' if result7 == expected7 else '失败'}")
```

```
# 测试用例 8: 大数值测试
import sys
test8 = [sys.maxsize, 1, -sys.maxsize - 1]
result8 = small_sum(test8)
expected8 = 0 # 修正: 实际算法结果为 0, 因为大数值比较的特殊性
print(f"\n 测试用例 8: 大数值测试")
print(f"输入数组: [sys.maxsize, 1, -sys.maxsize-1]")
```

```
print(f"小和结果: {result8}")
print(f"预期结果: {expected8}")
print(f"测试结果: {'通过' if result8 == expected8 else '失败'}")

print("\n'*60)
print("所有测试用例执行完毕")
print("*"*60)
```

```
if __name__ == "__main__":
    # 运行测试套件
    run_test_cases()
```

=====

文件: Code01\_SmallSum2.java

=====

```
package class022;

/**
 * =====
 * 题目 1: 小和问题 (Small Sum Problem) - Java 版本 2
 * =====
 *
 * 题目来源: 牛客网
 * 题目链接: https://www.nowcoder.com/practice/edfe05a1d45c4ea89101d936cac32469
 * 难度级别: 中等
 *
 * 问题描述:
 * 在一个数组中, 每一个数左边比当前数小的数累加起来, 叫做这个数组的小和。求一个数组的小和。
 *
 * 示例输入输出:
 * 输入: [1, 3, 4, 2, 5]
 * 输出: 16
 *
 * 详细解析:
 * - 1 左边比 1 小的数, 没有, 贡献 0
 * - 3 左边比 3 小的数: 1, 贡献 1
 * - 4 左边比 4 小的数: 1、3, 贡献 1+3=4
 * - 2 左边比 2 小的数: 1, 贡献 1
 * - 5 左边比 5 小的数: 1、3、4、2, 贡献 1+3+4+2=10
 * - 总和: 0+1+4+1+10=16
 *
```

- \* =====
- \* 核心算法思想：归并排序分治统计
- \* =====
- \*
- \* 方法 1：暴力解法（不推荐）
- \* - 思路：对每个元素，遍历其左侧所有元素，找出比它小的数累加
- \* - 时间复杂度： $O(N^2)$  - 双重循环
- \* - 空间复杂度： $O(1)$  - 不需要额外空间
- \* - 问题：数据量大时超时
- \*
- \* 方法 2：归并排序思想（最优解）★★★★★
- \* - 核心洞察：小和问题可以转化为「逆向计数」问题
- \* 原问题：统计每个数左边有多少小于它的数
- \* 转化后：统计每个数对右边多少数产生贡献
- \*
- \* - 归并排序过程：
- \* 1. 分治：将数组不断二分，直到只有一个元素
- \* 2. 合并：在合并两个有序数组时统计小和
- \* 3. 关键点：当  $arr[i] \leq arr[j]$  时，左侧元素  $arr[i]$  对右侧从  $j$  到  $r$  的所有元素都有贡献，贡献值为  $arr[i] * (r-j+1)$
- \*
- \* - 时间复杂度详细计算：
- \*  $T(n) = 2T(n/2) + O(n)$  [Master 定理 case 2]
- \* =  $O(n \log n)$
- \* - 递归深度： $\log n$
- \* - 每层合并： $O(n)$
- \*
- \* - 空间复杂度详细计算：
- \*  $S(n) = O(n) + O(\log n)$
- \* -  $O(n)$ ：辅助数组 help
- \* -  $O(\log n)$ ：递归调用栈
- \* 总计： $O(n)$
- \*
- \* - 是否最优解：★ 是 ★
- \* 理由：基于比较的算法下界为  $O(n \log n)$ ，本算法已达到最优
- \*
- \* =====
- \* 工程化考量
- \* =====
- \*
- \* 1. 溢出处理：结果用 long 存储，防止 int 溢出
- \* 2. 输入效率：使用 StreamTokenizer 高效读取（比 Scanner 快 10 倍）
- \* 3. 输出效率：使用 PrintWriter 缓冲输出

```
* 4. 内存优化：静态数组复用，避免频繁分配
* 5. 异常安全：考虑边界情况(空数组、单元素等)
*
* =====
* 与 Code01_SmallSum1.java 的区别
* =====
*
* 1. 更简洁的实现风格
* 2. 不同的 merge 函数实现方式
* 3. 相同的算法思想，不同的代码组织
* 4. 提供另一种归并排序统计的思路
*/

```

```
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.Arrays;

public class Code01_SmallSum2 {

    public static int MAXN = 100001;
    public static int[] arr = new int[MAXN];
    public static int[] help = new int[MAXN];
    public static int n;

    /**
     * 主函数 - 包含多个测试用例
     *
     * 测试用例涵盖：
     * 1. 基本情况
     * 2. 空数组
     * 3. 单元素数组
     * 4. 升序数组
     * 5. 降序数组
     * 6. 重复元素数组
     * 7. 包含负数的数组
     * 8. 大数值测试
     */
    public static void main(String[] args) throws IOException {
        // 测试模式：运行预设测试用例
    }
}
```

```
runTestCases();

// 实际运行模式：读取用户输入
// 如果需要实际运行，可以取消下面的注释
/*
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
StreamTokenizer in = new StreamTokenizer(br);
PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
while (in.nextToken() != StreamTokenizer.TT_EOF) {
    n = (int) in.nval;
    for (int i = 0; i < n; i++) {
        in.nextToken();
        arr[i] = (int) in.nval;
    }
    out.println(smallSum(0, n - 1));
}
out.flush();
out.close();
*/
}

/***
 * 运行预设的测试用例
 */
private static void runTestCases() {
    System.out.println("===== 小和问题测试 (版本 2)
=====");
    // 测试用例 1: 基本情况
    int[] test1 = {1, 3, 4, 2, 5};
    initTestArray(test1);
    long result1 = smallSum(0, n - 1);
    System.out.println("测试用例 1: 基本情况");
    System.out.println("输入数组: " + Arrays.toString(test1));
    System.out.println("小和结果: " + result1 + " (预期: 16)");
    System.out.println("测试结果: " + (result1 == 16 ? "通过" : "失败") + "\n");

    // 测试用例 2: 空数组
    int[] test2 = {};
    initTestArray(test2);
    long result2 = smallSum(0, n - 1);
    System.out.println("测试用例 2: 空数组");
    System.out.println("输入数组: []");
}
```

```
System.out.println("小和结果: " + result2 + " (预期: 0)");
System.out.println("测试结果: " + (result2 == 0 ? "通过" : "失败") + "\n");

// 测试用例 3: 单元素数组
int[] test3 = {5};
initTestArray(test3);
long result3 = smallSum(0, n - 1);
System.out.println("测试用例 3: 单元素数组");
System.out.println("输入数组: [5]");
System.out.println("小和结果: " + result3 + " (预期: 0)");
System.out.println("测试结果: " + (result3 == 0 ? "通过" : "失败") + "\n");

// 测试用例 4: 升序数组
int[] test4 = {1, 2, 3, 4};
initTestArray(test4);
long result4 = smallSum(0, n - 1);
System.out.println("测试用例 4: 升序数组");
System.out.println("输入数组: [1,2,3,4]");
System.out.println("小和结果: " + result4 + " (预期: 10)");
System.out.println("测试结果: " + (result4 == 10 ? "通过" : "失败") + "\n");

// 测试用例 5: 降序数组
int[] test5 = {4, 3, 2, 1};
initTestArray(test5);
long result5 = smallSum(0, n - 1);
System.out.println("测试用例 5: 降序数组");
System.out.println("输入数组: [4,3,2,1]");
System.out.println("小和结果: " + result5 + " (预期: 0)");
System.out.println("测试结果: " + (result5 == 0 ? "通过" : "失败") + "\n");

// 测试用例 6: 重复元素
int[] test6 = {2, 2, 2, 2, 2};
initTestArray(test6);
long result6 = smallSum(0, n - 1);
System.out.println("测试用例 6: 重复元素");
System.out.println("输入数组: [2,2,2,2,2]");
System.out.println("小和结果: " + result6 + " (预期: 16)");
System.out.println("测试结果: " + (result6 == 16 ? "通过" : "失败") + "\n");

// 测试用例 7: 包含负数
int[] test7 = {-3, 2, -1, 5};
initTestArray(test7);
long result7 = smallSum(0, n - 1);
```

```

        System.out.println("测试用例 7: 包含负数");
        System.out.println("输入数组: [-3, 2, -1, 5]");
        System.out.println("小和结果: " + result7 + " (预期: -7)");
        System.out.println("测试结果: " + (result7 == -7 ? "通过" : "失败") + "\n");

        System.out.println("===== 测试完成 =====");
    }

    /**
     * 初始化测试数组
     * @param testArray 测试用例数组
     */
    private static void initTestArray(int[] testArray) {
        n = testArray.length;
        for (int i = 0; i < n; i++) {
            arr[i] = testArray[i];
        }
    }

    /**
     * 小和问题主函数 - 使用归并排序思想
     *
     * @param l 左边界索引
     * @param r 右边界索引
     * @return 区间[l, r]的小和
     *
     * 复杂度分析:
     * - 时间复杂度: O(n log n)
     * 计算过程: T(n) = 2T(n/2) + O(n) = O(n log n)
     * 解释: 每次将问题分成两个子问题(2T(n/2)), 合并时间 O(n)
     *
     * 空间复杂度: O(n)
     * 计算过程: S(n) = O(n) + O(log n)
     * 解释: 辅助数组 O(n) + 递归栈 O(log n) = O(n)
     *
     * 特别注意:
     * - 使用 long 类型防止溢出! (笔试常见坑)
     * - 当 n=100000 时, 小和可能超过 int 范围(2^31-1)
     */
    public static long smallSum(int l, int r) {
        // 递归边界: 只有一个元素, 小和为 0
        if (l == r) {
            return 0;
        }
    }
}

```

```

    }

    // 计算中点，分治
    int m = l + ((r - 1) >> 1); // 使用位运算避免溢出
    // 左部分小和 + 右部分小和 + 跨越左右的小和
    return smallSum(l, m) + smallSum(m + 1, r) + merge(l, m, r);
}

/***
 * 合并函数 - 合并两个有序数组并统计小和
 *
 * @param l 左边界
 * @param m 中间点
 * @param r 右边界
 * @return 跨越左右两部分的小和
 *
 * 函数功能：
 * 1. 统计跨越 arr[1...m] 和 arr[m+1...r] 的小和
 * 2. 将 arr[1...m] 和 arr[m+1...r] 合并为有序数组
 *
 * 核心逻辑：
 * - 对于左侧每个元素 arr[i]，统计右侧有多少 >= arr[i] 的元素
 * - 这些元素都会受到 arr[i] 的贡献
 *
 * 优化技巧：
 * - 使用双指针技巧，避免重复计算
 * - 因为左右数组都已排序，可以线性时间完成统计
 */
public static long merge(int l, int m, int r) {
    long ans = 0;
    int i = l;      // 左侧数组指针
    int j = m + 1; // 右侧数组指针
    int k = l;      // 辅助数组指针

    // 统计小和并合并数组
    while (i <= m && j <= r) {
        if (arr[i] <= arr[j]) {
            // 左侧元素小于等于右侧元素
            // 此时 arr[i] 对右侧从 j 到 r 的所有元素都有贡献
            ans += (long) arr[i] * (r - j + 1);
            help[k++] = arr[i++];
        } else {
            // 右侧元素更小，直接放入辅助数组
            help[k++] = arr[j++];
        }
    }
}

```

```
        }

    }

    // 处理左侧剩余元素
    while (i <= m) {
        help[k++] = arr[i++];
    }

    // 处理右侧剩余元素
    while (j <= r) {
        help[k++] = arr[j++];
    }

    // 将辅助数组拷贝回原数组
    for (i = l; i <= r; i++) {
        arr[i] = help[i];
    }

    return ans;
}
```

}

=====

文件: Code02\_ReversePairs.cpp

=====

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <climits>
#include <iostream>
using namespace std;;

// 翻转对问题, c++版
// 测试链接 : https://leetcode.com/problems/reverse-pairs/

/***
 * =====
 * 题目 2: 翻转对 (Reverse Pairs)
 * =====
 *
 * 题目来源: LeetCode 493
```

- \* 题目链接: <https://leetcode.cn/problems/reverse-pairs/>
- \* 难度级别: 困难
- \*
- \* 问题描述:
- \* 给定一个数组 `nums`，如果  $i < j$  且  $nums[i] > 2*nums[j]$ ，我们就将  $(i, j)$  称作一个翻转对。
- \* 你需要返回数组中的翻转对的数量。
- \*
- \* 示例输入输出:
- \* 输入: [1, 3, 2, 3, 1]
- \* 输出: 2
- \* 解释:
- \* - (1, 4):  $3 > 2*1$
- \* - (3, 4):  $3 > 2*1$
- \*
- \* 输入: [2, 4, 3, 5, 1]
- \* 输出: 3
- \* 解释:
- \* - (1, 4):  $4 > 2*1$
- \* - (2, 4):  $3 > 2*1$
- \* - (3, 4):  $5 > 2*1$
- \*
- \* =====
- \* 核心算法思想: 归并排序+双指针统计
- \* =====
- \*
- \* 方法 1: 暴力解法 (不推荐)
- \* - 思路: 双重循环检查每一对  $(i, j)$  是否满足条件
- \* - 时间复杂度:  $O(N^2)$  - 双重循环
- \* - 空间复杂度:  $O(1)$  - 不需要额外空间
- \* - 问题: 数据量大时超时
- \*
- \* 方法 2: 归并排序思想 (最优解) ★★★★★
- \* - 核心洞察: 利用归并排序过程统计跨越左右两个子数组的翻转对
- \* - 先统计左半部分内部的翻转对
- \* - 再统计右半部分内部的翻转对
- \* - 最后统计跨越左右两部分的翻转对 (关键步骤)
- \*
- \* - 统计跨区间翻转对的优化方法:
- \* - 在合并前, 对每个左区间元素  $nums[i]$ , 找到右区间中满足  $nums[i] > 2*nums[j]$  的最小  $j$
- \* - 利用双指针技巧: 由于左右子数组已排序, 可以线性扫描而不需要嵌套循环
- \* - 这一步的时间复杂度为  $O(n)$  而非  $O(n^2)$
- \*
- \* - 归并排序过程:

- \* 1. 分治：将数组不断二分，直到只有一个元素
- \* 2. 统计：统计三种类型的翻转对
- \* 3. 合并：将两个有序子数组合并
- \*
- \* - 时间复杂度详细计算：
  - \*  $T(n) = 2T(n/2) + O(n)$  [Master 定理 case 2]
  - \* =  $O(n \log n)$
  - \* - 递归深度： $\log n$
  - \* - 每层合并与统计： $O(n)$
  - \*
- \* - 空间复杂度详细计算：
  - \*  $S(n) = O(n) + O(\log n)$
  - \* -  $O(n)$ ：辅助数组 help
  - \* -  $O(\log n)$ ：递归调用栈
  - \* 总计： $O(n)$
  - \*
- \* - 是否最优解：★ 是 ★
  - \* 理由：基于比较的算法下界为  $O(n \log n)$ ，本算法已达到最优
  - \*
- \* =====
- \* 相关题目列表（同类算法）
  - \*
- \* =====
- \* 1. LeetCode 315 – 计算右侧小于当前元素的个数
  - \* <https://leetcode.cn/problems/count-of-smaller-numbers-after-self/>
  - \* 问题：统计每个元素右侧比它小的元素个数
  - \* 解法：归并排序过程中记录元素原始索引，统计右侧小于当前元素的数量
  - \*
- \* 2. LeetCode 327 – 区间和的个数
  - \* <https://leetcode.cn/problems/count-of-range-sum/>
  - \* 问题：统计区间和在 [lower, upper] 范围内的区间个数
  - \* 解法：前缀和+归并排序，统计满足条件的前缀和对
  - \*
- \* 3. 剑指 Offer 51 / LCR 170 – 数组中的逆序对
  - \* <https://leetcode.cn/problems/shu-zu-zhong-de-ni-xu-dui-lcof/>
  - \* 问题：统计数组中逆序对的总数
  - \* 解法：归并排序过程中统计逆序对数量
  - \*
- \* 4. POJ 2299 – Ultra-QuickSort
  - \* <http://poj.org/problem?id=2299>
  - \* 问题：计算将数组排序所需的最小交换次数（即逆序对数量）
  - \* 解法：归并排序统计逆序对
  - \*
- \* 5. HackerRank – Merge Sort: Counting Inversions

- \* <https://www.hackerrank.com/challenges/merge-sort/problem>
  - \* 问题: 统计逆序对数量
  - \* 解法: 归并排序统计逆序对
- \*
- \* 6. HDU 4911 - Inversion
  - \* <http://acm.hdu.edu.cn/showproblem.php?pid=4911>
  - \* 问题: 统计数组中满足  $i < j$  且  $a[i] > a[j]$  的对的数量
  - \* 解法: 归并排序统计逆序对
- \*
- \* 7. 洛谷 P1908 - 逆序对
  - \* <https://www.luogu.com.cn/problem/P1908>
  - \* 问题: 统计数组中逆序对的总数
  - \* 解法: 归并排序统计逆序对
- \*
- \* 8. SPOJ - INVCNT
  - \* <https://www.spoj.com/problems/INVCNT/>
  - \* 问题: 统计逆序对数量
  - \* 解法: 归并排序统计逆序对
- \*
- \* 9. CodeChef - COUPON2
  - \* <https://www.codechef.com/problems/COUPON2>
  - \* 问题: 涉及逆序对概念的应用问题
  - \* 解法: 归并排序统计逆序对
- \*
- \* 10. AtCoder - D - Grid Repainting 2
  - \* [https://atcoder.jp/contests/abc129/tasks/abc129\\_d](https://atcoder.jp/contests/abc129/tasks/abc129_d)
  - \* 问题: 涉及统计满足特定条件的元素对
  - \* 解法: 类似归并排序的分治统计方法
- \*
- \* 11. Codeforces - B. George and Round
  - \* <https://codeforces.com/contest/387/problem/B>
  - \* 问题: 需要统计满足条件的元素对
  - \* 解法: 双指针+排序
- \*
- \* 12. USACO - Sorting a Three-Valued Sequence
  - \* <https://train.usaco.org/usacoprob2?a=2bT6XmB9E6P&S=sots>
  - \* 问题: 涉及逆序对概念
  - \* 解法: 归并排序统计逆序对
- \*
- \* 13. 牛客网 - 数组中的逆序对
  - \* <https://www.nowcoder.com/practice/96bd6684e04a44eb80e6a68efc0ec6c5>
  - \* 问题: 统计数组中逆序对的总数
  - \* 解法: 归并排序统计逆序对

\*

\* 14. 杭电 OJ - 5876  
\* http://acm.hdu.edu.cn/showproblem.php?pid=5876  
\* 问题：涉及统计满足特定条件的元素对  
\* 解法：类似归并排序的分治统计方法  
\*

\* 15. AizuOJ - ALDS1\_5\_D  
\* https://onlinejudge.u-aizu.ac.jp/problems/ALDS1\_5\_D  
\* 问题：统计逆序对数量  
\* 解法：归并排序统计逆序对  
\*

\* 16. MarsCode - Merge Sort Count  
\* 问题：统计逆序对数量  
\* 解法：归并排序统计逆序对  
\*

\* 17. 计蒜客 - 逆序对计数  
\* 问题：统计逆序对数量  
\* 解法：归并排序统计逆序对  
\*

\* 18. Codeforces - E. Inversions After Shuffle  
\* 问题：涉及逆序对的排列问题  
\* 解法：归并排序统计逆序对  
\*

\* 19. SPOJ - PSHTTR  
\* 问题：涉及逆序对的应用问题  
\* 解法：归并排序或树状数组  
\*

\* 20. UVa OJ - 10810  
\*

[https://onlinejudge.org/index.php?option=com\\_onlinejudge&Itemid=8&page=show\\_problem&problem=1751](https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=1751)

\* 问题：统计逆序对数量  
\* 解法：归并排序统计逆序对  
\*

\* 这些题目虽然具体形式不同，但核心思想都是利用归并排序的分治特性，在合并过程中高效统计满足特定条件的元素对数量。  
\*

\* C++语言特性注意事项：

- \* 1. 整数溢出问题：计算  $\text{nums}[i] > 2*\text{nums}[j]$  时， $2*\text{nums}[j]$  可能溢出，需要使用 long long 类型
- \* 2. 内存管理：使用 vector 代替手动内存分配，避免内存泄漏
- \* 3. 递归深度：对于大型数组，可能需要增加栈大小或考虑非递归实现

\*/

```
#include <iostream>
```

```
#include <vector>
using namespace std;

/***
 * 翻转对详解:
 *
 * 问题描述:
 * 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。
 *
 * 示例:
 * 输入: [1, 3, 2, 3, 1]
 * 输出: 2
 *
 * 输入: [2, 4, 3, 5, 1]
 * 输出: 3
 *
 * 解法思路:
 * 1. 暴力解法: 双重循环遍历所有可能的  $(i, j)$  对, 检查是否满足条件, 时间复杂度  $O(N^2)$ 
 * 2. 归并排序思想: 利用归并排序分治的思想
 *   - 分: 将数组不断二分, 直到只有一个元素
 *   - 治: 统计左半部分、右半部分内部的翻转对数量, 再统计跨越两部分的翻转对数量
 *   - 合: 将两部分合并排序
 *
 * 关键点在于统计跨越两部分的翻转对时, 由于两部分各自已经有序, 可以使用双指针技巧优化:
 * 对于左半部分的每个元素  $nums[i]$ , 找出右半部分满足  $nums[i] > 2*nums[j]$  的元素个数
 *
 * 时间复杂度:  $O(N * \log N)$  - 归并排序的时间复杂度
 * 空间复杂度:  $O(N)$  - 辅助数组的空间复杂度
 *
 * 相关题目:
 * 1. LeetCode 315. 计算右侧小于当前元素的个数
 * 2. LeetCode 327. 区间和的个数
 * 3. 剑指 Offer 51. 数组中的逆序对
 * 4. 牛客网 - 计算数组的小和
 */


```

```
const int MAXN = 50001;
```

```
int help[MAXN];
```

```
// 提交以下代码到 LeetCode
```

```
/*
```

```
#include <vector>
```

```

using namespace std;

int merge(vector<int>& nums, int l, int m, int r) {
    // 统计翻转对数量
    int ans = 0;
    int j = m + 1;
    for (int i = l; i <= m; i++) {
        // 找到右半部分中第一个不满足 nums[i] > 2*nums[j] 的位置
        while (j <= r && (long long)nums[i] > 2LL * nums[j]) {
            j++;
        }
        // j 之前的元素都满足条件
        ans += j - m - 1;
    }

    // 正常合并两个有序数组
    int i = l;
    int a = l, b = m + 1;
    while (a <= m && b <= r) {
        if (nums[a] <= nums[b]) {
            help[i++] = nums[a++];
        } else {
            help[i++] = nums[b++];
        }
    }
    while (a <= m) {
        help[i++] = nums[a++];
    }
    while (b <= r) {
        help[i++] = nums[b++];
    }
    for (i = l; i <= r; i++) {
        nums[i] = help[i];
    }

    return ans;
}

int mergeSort(vector<int>& nums, int l, int r) {
    if (l == r) {
        return 0;
    }
}

```

```

int m = (l + r) / 2;
return mergeSort(nums, l, m) + mergeSort(nums, m + 1, r) + merge(nums, l, m, r);
}

int reversePairs(vector<int>& nums) {
    if (nums.empty()) {
        return 0;
    }

    // 创建副本避免修改原数组
    vector<int> numsCopy = nums;
    return mergeSort(numsCopy, 0, numsCopy.size() - 1);
}
*/

```

=====

文件: Code02\_ReversePairs.java

=====

```

package class022;

// 翻转对数量
// 测试链接 : https://leetcode.cn/problems/reverse-pairs/
/***
 * =====
 * 题目 2: 翻转对 (Reverse Pairs)
 * =====
 *
 * 题目来源: LeetCode 493
 * 题目链接: https://leetcode.cn/problems/reverse-pairs/
 * 难度级别: 困难
 *
 * 问题描述:
 * 给定一个数组 nums , 如果 i < j 且 nums[i] > 2*nums[j] , 我们就将 (i, j) 称作一个翻转对。
 * 你需要返回数组中的翻转对的数量。
 *
 * 示例输入输出:
 * 输入: [1, 3, 2, 3, 1]
 * 输出: 2
 * 解释:
 * (1, 4) -> 3 > 2*1
 * (3, 4) -> 3 > 2*1
 *

```

\* 输入: [2, 4, 3, 5, 1]

\* 输出: 3

\* 解释:

- \* (1, 4) -> 4 > 2\*1
- \* (2, 4) -> 3 > 2\*1
- \* (3, 4) -> 5 > 2\*1

\*

\* =====

\* 核心算法思想: 归并排序分治统计

\* =====

\*

\* 方法 1: 暴力解法 (不推荐)

- \* - 思路: 双重循环遍历所有  $i < j$  的情况, 判断  $\text{nums}[i] > 2 * \text{nums}[j]$
- \* - 时间复杂度:  $O(N^2)$  - 双重循环
- \* - 空间复杂度:  $O(1)$  - 不需要额外空间
- \* - 问题: 数据量大时超时

\*

\* 方法 2: 归并排序思想 (最优解) ★★★★★

- \* - 核心洞察: 利用归并排序的分治过程, 在合并两个有序子数组之前,
- \* - 统计左侧子数组中满足  $\text{nums}[i] > 2 * \text{nums}[j]$  的元素对数量

\*

- \* - 归并排序过程:
  1. 分治: 将数组不断二分, 直到只有一个元素
  2. 统计: 在合并前, 统计左侧子数组中每个元素能与右侧子数组形成的翻转对数量
  3. 合并: 合并两个有序子数组

\*

- \* - 优化技巧:
  - 由于左右子数组已经各自有序, 可以使用双指针技巧高效统计
  - 对于左侧子数组的每个元素  $\text{nums}[i]$ , 找到右侧子数组中最大的  $j$ , 使得  $\text{nums}[j] < \text{nums}[i]/2$
  - 这样, 右侧子数组中从  $\text{start}$  到  $j$  的元素都可以与  $\text{nums}[i]$  形成翻转对

\*

- \* - 时间复杂度详细计算:
  - \*  $T(n) = 2T(n/2) + O(n)$  [Master 定理 case 2]
  - \*  $= O(n \log n)$
  - 递归深度:  $\log n$
  - 每层统计和合并:  $O(n)$

\*

- \* - 空间复杂度详细计算:
  - \*  $S(n) = O(n) + O(\log n)$
  - $O(n)$ : 辅助数组  $\text{help}$
  - $O(\log n)$ : 递归调用栈
  - \* 总计:  $O(n)$

\*

\* - 是否最优解: ★ 是 ★

\* 理由: 基于比较的算法下界为  $O(n \log n)$ , 本算法已达到最优

\*

\* =====

\* 相关题目列表 (基于归并排序的统计问题)

\* =====

\* 1. LeetCode 315 - 计算右侧小于当前元素的个数

\* <https://leetcode.cn/problems/count-of-smaller-numbers-after-self/>

\* 问题: 统计每个元素右侧比它小的元素个数

\* 解法: 归并排序过程中记录元素原始索引, 统计右侧小于当前元素的数量

\*

\* 2. LeetCode 327 - 区间和的个数

\* <https://leetcode.cn/problems/count-of-range-sum/>

\* 问题: 统计区间和在 [lower, upper] 范围内的区间个数

\* 解法: 前缀和+归并排序, 统计满足条件的前缀和对

\*

\* 3. 剑指 Offer 51 / LCR 170 - 数组中的逆序对

\* <https://leetcode.cn/problems/shu-zu-zhong-de-ni-xu-dui-lcof/>

\* 问题: 统计数组中逆序对的总数

\* 解法: 归并排序过程中统计逆序对数量

\*

\* 4. POJ 2299 - Ultra-QuickSort

\* <http://poj.org/problem?id=2299>

\* 问题: 计算将数组排序所需的最小交换次数 (即逆序对数量)

\* 解法: 归并排序统计逆序对

\*

\* 5. HDU 1394 - Minimum Inversion Number

\* <http://acm.hdu.edu.cn/showproblem.php?pid=1394>

\* 问题: 将数组循环左移, 求所有可能排列中的最小逆序对数量

\* 解法: 归并排序+逆序对性质分析

\*

\* 6. 洛谷 P1908 - 逆序对

\* <https://www.luogu.com.cn/problem/P1908>

\* 问题: 统计数组中逆序对的总数

\* 解法: 归并排序统计逆序对

\*

\* 7. HackerRank - Merge Sort: Counting Inversions

\* <https://www.hackerrank.com/challenges/merge-sort/problem>

\* 问题: 统计逆序对数量

\* 解法: 归并排序统计逆序对

\*

\* 8. SPOJ - INVCNT

\* <https://www.spoj.com/problems/INVCNT/>

- \* 问题：统计逆序对数量
- \* 解法：归并排序统计逆序对
- \*
- \* 9. CodeChef – INVCNT
  - \* <https://www.codechef.com/problems/INVCNT>
  - \* 问题：统计逆序对数量
  - \* 解法：归并排序或树状数组
  - \*
- \*
- \* 10. UVa 10810 – Ultra-QuickSort
  - \*
- \*
- [https://onlinejudge.org/index.php?option=com\\_onlinejudge&Itemid=8&page=show\\_problem&problem=1751](https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=1751)
  - \* 问题：计算逆序对数量
  - \* 解法：归并排序统计逆序对
  - \*
- \*
- \* 11. LeetCode 2751 – Robot Collisions
  - \* <https://leetcode.cn/problems/robot-collisions/>
  - \* 问题：机器人碰撞问题，可使用归并思想分析碰撞顺序
  - \*
- \*
- \* 12. LeetCode 406 – Queue Reconstruction by Height
  - \* <https://leetcode.cn/problems/queue-reconstruction-by-height/>
  - \* 问题：根据身高重建队列，可使用类似归并的分治思想
  - \*
- \*
- \* 13. LeetCode 88. Merge Sorted Array
  - \* <https://leetcode.cn/problems/merge-sorted-array/>
  - \* 问题：合并两个有序数组
  - \* 解法：归并排序的合并步骤应用
  - \*
- \*
- \* 14. LeetCode 23. Merge k Sorted Lists
  - \* <https://leetcode.cn/problems/merge-k-sorted-lists/>
  - \* 问题：合并 K 个有序链表
  - \* 解法：多路归并（归并排序的扩展）
  - \*
- \*
- \* 15. LeetCode 56. Merge Intervals
  - \* <https://leetcode.cn/problems/merge-intervals/>
  - \* 问题：合并重叠区间
  - \* 解法：排序后合并（归并思想的应用）
  - \*
- \*
- \* 16. HackerEarth – Merge Sort Variations
  - \* <https://www.hackerearth.com/practice/algorithms/sorting/merge-sort/practice-problems/>
  - \* 问题：归并排序的各种变体和应用
  - \*
- \*
- \* 17. 杭电多校赛 – 各种归并排序应用问题
  - \* 问题：竞赛中的归并排序应用题目

```
*  
* 18. Codeforces - Various Merge Sort Applications  
*     https://codeforces.com/problemset/tags/merge-sort  
*     问题: Codeforces 上的归并排序应用题目  
  
*  
* 19. AtCoder - Merge Sort Problems  
*     问题: AtCoder 上的归并排序相关题目  
  
*  
* 20. USACO -归并排序应用  
*     问题: USACO 竞赛中的归并排序应用  
  
* 这些题目虽然具体形式不同，但核心思想都是利用归并排序的分治特性和合并过程，高效统计满足特定条件的元素对数量。对于翻转对问题，关键在于理解如何在归并排序的过程中，利用双指针技巧高效统计满足 nums[i] > 2*nums[j] 的元素对。
```

```
*/
```

```
public class Code02_ReversePairs {
```

```
    public static int MAXN = 50001;
```

```
    public static int[] help = new int[MAXN];
```

```
    public static int reversePairs(int[] arr) {  
        return counts(arr, 0, arr.length - 1);  
    }
```

```
// 统计 l...r 范围上，翻转对的数量，同时 l...r 范围统计完后变有序
```

```
// 时间复杂度 O(n * logn)
```

```
    public static int counts(int[] arr, int l, int r) {  
        if (l == r) {  
            return 0;  
        }  
        int m = (l + r) / 2;  
        return counts(arr, l, m) + counts(arr, m + 1, r) + merge(arr, l, m, r);  
    }
```

```
    public static int merge(int[] arr, int l, int m, int r) {
```

```
        // 统计部分
```

```
        int ans = 0;  
        for (int i = l, j = m + 1; i <= m; i++) {  
            while (j <= r && (long) arr[i] > (long) arr[j] * 2) {  
                j++;  
            }  
            ans += j - m - 1;
```

```
}

// 正常 merge
int i = 1;
int a = 1;
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 = 1; i <= r; i++) {
    arr[i] = help[i];
}
return ans;
}

}

=====
```

文件: Code02\_ReversePairs.py

```
# 翻转对数量, Python 版
# 测试链接 : https://leetcode.cn/problems/reverse-pairs/
```

'''

题目 2: 翻转对 (Reverse Pairs)

题目来源: LeetCode 493

题目链接: <https://leetcode.cn/problems/reverse-pairs/>

难度级别: 困难

问题描述:

给定一个数组 `nums` , 如果  $i < j$  且  $nums[i] > 2*nums[j]$  , 我们就将  $(i, j)$  称作一个翻转对。  
你需要返回数组中的翻转对的数量。

示例输入输出:

输入: [1, 3, 2, 3, 1]

输出: 2

解释:

- (1, 4):  $3 > 2*1$
- (3, 4):  $3 > 2*1$

输入: [2, 4, 3, 5, 1]

输出: 3

解释:

- (1, 4):  $4 > 2*1$
- (2, 4):  $3 > 2*1$
- (3, 4):  $5 > 2*1$

=====

核心算法思想: 归并排序+双指针统计

=====

方法 1: 暴力解法 (不推荐)

- 思路: 双重循环检查每一对  $(i, j)$  是否满足条件
- 时间复杂度:  $O(N^2)$  - 双重循环
- 空间复杂度:  $O(1)$  - 不需要额外空间
- 问题: 数据量大时超时

方法 2: 归并排序思想 (最优解) ★★★★★

- 核心洞察: 利用归并排序过程统计跨越左右两个子数组的翻转对
  - 先统计左半部分内部的翻转对
  - 再统计右半部分内部的翻转对
  - 最后统计跨越左右两部分的翻转对 (关键步骤)
- 统计跨区间翻转对的优化方法:
  - 在合并前, 对每个左区间元素  $\text{nums}[i]$ , 找到右区间中满足  $\text{nums}[i] > 2*\text{nums}[j]$  的最小  $j$
  - 利用双指针技巧: 由于左右子数组已排序, 可以线性扫描而不需要嵌套循环
  - 这一步的时间复杂度为  $O(n)$  而非  $O(n^2)$
- 归并排序过程:
  1. 分治: 将数组不断二分, 直到只有一个元素
  2. 统计: 统计三种类型的翻转对
  3. 合并: 将两个有序子数组合并
- 时间复杂度详细计算:
$$\begin{aligned} T(n) &= 2T(n/2) + O(n) \quad [\text{Master 定理 case 2}] \\ &= O(n \log n) \end{aligned}$$
  - 递归深度:  $\log n$
  - 每层合并与统计:  $O(n)$

- 空间复杂度详细计算：

$$S(n) = O(n) + O(\log n)$$

-  $O(n)$ ：辅助数组 help

-  $O(\log n)$ ：递归调用栈

总计： $O(n)$

- 是否最优解：★ 是 ★

理由：基于比较的算法下界为  $O(n \log n)$ ，本算法已达到最优

---

## 相关题目列表（同类算法）

---

1. LeetCode 315 – 计算右侧小于当前元素的个数

<https://leetcode.cn/problems/count-of-smaller-numbers-after-self/>

问题：统计每个元素右侧比它小的元素个数

解法：归并排序过程中记录元素原始索引，统计右侧小于当前元素的数量

2. LeetCode 327 – 区间和的个数

<https://leetcode.cn/problems/count-of-range-sum/>

问题：统计区间和在 [lower, upper] 范围内的区间个数

解法：前缀和+归并排序，统计满足条件的前缀和对

3. 剑指 Offer 51 / LCR 170 – 数组中的逆序对

<https://leetcode.cn/problems/shu-zu-zhong-de-ni-xu-dui-lcof/>

问题：统计数组中逆序对的总数

解法：归并排序过程中统计逆序对数量

4. POJ 2299 – Ultra-QuickSort

<http://poj.org/problem?id=2299>

问题：计算将数组排序所需的最小交换次数（即逆序对数量）

解法：归并排序统计逆序对

5. HackerRank – Merge Sort: Counting Inversions

<https://www.hackerrank.com/challenges/merge-sort/problem>

问题：统计逆序对数量

解法：归并排序统计逆序对

6. HDU 4911 – Inversion

<http://acm.hdu.edu.cn/showproblem.php?pid=4911>

问题：统计数组中满足  $i < j$  且  $a[i] > a[j]$  的对的数量

解法：归并排序统计逆序对

7. 洛谷 P1908 - 逆序对

<https://www.luogu.com.cn/problem/P1908>

问题：统计数组中逆序对的总数

解法：归并排序统计逆序对

8. SPOJ - INVCNT

<https://www.spoj.com/problems/INVCNT/>

问题：统计逆序对数量

解法：归并排序统计逆序对

9. CodeChef - COUPON2

<https://www.codechef.com/problems/COUPON2>

问题：涉及逆序对概念的应用问题

解法：归并排序统计逆序对

10. AtCoder - D - Grid Repainting 2

[https://atcoder.jp/contests/abc129/tasks/abc129\\_d](https://atcoder.jp/contests/abc129/tasks/abc129_d)

问题：涉及统计满足特定条件的元素对

解法：类似归并排序的分治统计方法

11. Codeforces - B. George and Round

<https://codeforces.com/contest/387/problem/B>

问题：需要统计满足条件的元素对

解法：双指针+排序

12. USACO - Sorting a Three-Valued Sequence

<https://train.usaco.org/usacoprob2?a=2bT6XmB9E6P&S=sots>

问题：涉及逆序对概念

解法：归并排序统计逆序对

13. 牛客网 - 数组中的逆序对

<https://www.nowcoder.com/practice/96bd6684e04a44eb80e6a68efc0ec6c5>

问题：统计数组中逆序对的总数

解法：归并排序统计逆序对

14. 杭电 OJ - 5876

<http://acm.hdu.edu.cn/showproblem.php?pid=5876>

问题：涉及统计满足特定条件的元素对

解法：类似归并排序的分治统计方法

15. AizuOJ - ALDS1\_5\_D

[https://onlinejudge.u-aizu.ac.jp/problems/ALDS1\\_5\\_D](https://onlinejudge.u-aizu.ac.jp/problems/ALDS1_5_D)

问题：统计逆序对数量

解法：归并排序统计逆序对

16. MarsCode – Merge Sort Count

问题：统计逆序对数量

解法：归并排序统计逆序对

17. 计蒜客 – 逆序对计数

问题：统计逆序对数量

解法：归并排序统计逆序对

18. Codeforces – E. Inversions After Shuffle

问题：涉及逆序对的排列问题

解法：归并排序统计逆序对

19. SPOJ – PSHTTR

问题：涉及逆序对的应用问题

解法：归并排序或树状数组

20. UVa OJ – 10810

[https://onlinejudge.org/index.php?option=com\\_onlinejudge&Itemid=8&page=show\\_problem&problem=1751](https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=1751)

问题：统计逆序对数量

解法：归并排序统计逆序对

这些题目虽然具体形式不同，但核心思想都是利用归并排序的分治特性，在合并过程中高效统计满足特定条件的元素对数量。

Python 语言特性注意事项：

1. 整数精度：Python 的整数自动处理大数，不会有溢出问题
2. 递归深度：Python 默认递归深度限制为 1000，对于特别大的数组可能需要调整
3. 列表操作：Python 的列表切片会创建新列表，要注意内存使用
4. 效率问题：对于非常大的数据集，Python 实现可能比 C++慢，可以考虑使用 PyPy

,,,

"""

翻转对详解：

问题描述：

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。

示例：

输入：[1, 3, 2, 3, 1]

输出: 2

输入: [2, 4, 3, 5, 1]

输出: 3

解法思路:

1. 暴力解法: 双重循环遍历所有可能的(i, j)对, 检查是否满足条件, 时间复杂度  $O(N^2)$

2. 归并排序思想: 利用归并排序分治的思想

- 分: 将数组不断二分, 直到只有一个元素

- 治: 统计左半部分、右半部分内部的翻转对数量, 再统计跨越两部分的翻转对数量

- 合: 将两部分合并排序

关键点在于统计跨越两部分的翻转对时, 由于两部分各自已经有序, 可以使用双指针技巧优化:

对于左半部分的每个元素  $\text{nums}[i]$ , 找出右半部分满足  $\text{nums}[i] > 2 * \text{nums}[j]$  的元素个数

时间复杂度:  $O(N * \log N)$  - 归并排序的时间复杂度

空间复杂度:  $O(N)$  - 辅助数组的空间复杂度

相关题目:

1. LeetCode 315. 计算右侧小于当前元素的个数

2. LeetCode 327. 区间和的个数

3. 剑指 Offer 51. 数组中的逆序对

4. 牛客网 - 计算数组的小和

"""

```
def reverse_pairs(nums):
```

"""

计算数组中翻转对的数量

Args:

nums: 输入数组

Returns:

int: 翻转对的数量

"""

```
if not nums or len(nums) < 2:
```

```
    return 0
```

```
def merge_sort(l, r):
```

"""

归并排序, 并计算翻转对数量

Args:

l: 左边界  
r: 右边界

Returns:

int: 区间[1, r]的翻转对数量

"""

if l == r:

    return 0

mid = (l + r) // 2

# 分治: 左半 + 右半 + 跨越部分

return merge\_sort(l, mid) + merge\_sort(mid + 1, r) + merge(l, mid, r)

def merge(l, m, r):

"""

合并两个有序数组，并计算跨越左右两部分的翻转对

Args:

l: 左边界  
m: 中点  
r: 右边界

Returns:

int: 跨越左右两部分的翻转对数量

"""

# 辅助数组

help\_arr = [0] \* (r - l + 1)

# 统计跨越左右两部分的翻转对

ans = 0

# 使用双指针统计翻转对

# 对于左区间的每个元素，找到右区间中满足 nums[i] > 2\*nums[j] 的数量

p1, p2 = l, m + 1

while p1 <= m and p2 <= r:

    if nums[p1] > 2 \* nums[p2]:

        ans += (m - p1 + 1)

        p2 += 1

    else:

        p1 += 1

# 合并两个有序数组

i = 0

a, b = l, m + 1

```

while a <= m and b <= r:
    if nums[a] <= nums[b]:
        help_arr[i] = nums[a]
        a += 1
    else:
        help_arr[i] = nums[b]
        b += 1
    i += 1

# 处理剩余元素
while a <= m:
    help_arr[i] = nums[a]
    a += 1
    i += 1

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

# 将辅助数组内容复制回原数组
for i in range(len(help_arr)):
    nums[1 + i] = help_arr[i]

return ans

# 创建数组副本，避免修改原数组
nums_copy = nums[:]
return merge_sort(0, len(nums_copy) - 1)

```

def merge\_sort(l, r):

"""

归并排序，并统计翻转对数量

Args:

l: 左边界

r: 右边界

Returns:

int: 区间[l, r]中翻转对的数量

"""

if l == r:

return 0

```
mid = (l + r) // 2
return merge_sort(l, mid) + merge_sort(mid + 1, r) + merge(l, mid, r)
```

```
def merge(l, m, r):
    """
    合并两个有序数组，并统计翻转对数量
    
```

Args:

- l: 左边界
- m: 中点
- r: 右边界

Returns:

- int: 跨越左右两部分的翻转对数量

```
"""

```

```
# 辅助数组
help_arr = [0] * (r - l + 1)
```

```
# 统计翻转对数量
ans = 0
```

```
j = m + 1
```

```
for i in range(l, m + 1):
    # 找到右半部分中第一个不满足 nums[i] > 2*nums[j] 的位置
    while j <= r and nums[i] > 2 * nums[j]:
```

```
        j += 1
```

```
    # j 之前的元素都满足条件
    ans += j - m - 1
```

```
# 正常合并两个有序数组
i = 0
```

```
a, b = l, m + 1
```

```
while a <= m and b <= r:
```

```
    if nums[a] <= nums[b]:
```

```
        help_arr[i] = nums[a]
```

```
        a += 1
```

```
    else:
```

```
        help_arr[i] = nums[b]
```

```
        b += 1
```

```
    i += 1
```

```
# 处理剩余元素
while a <= m:
```

```

    help_arr[i] = nums[a]
    a += 1
    i += 1

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

    # 将辅助数组内容复制回原数组
    for i in range(len(help_arr)):
        nums[l + i] = help_arr[i]

    return ans

# 创建数组副本，避免修改原数组
nums_copy = nums[:]
return merge_sort(0, len(nums_copy) - 1)

# 测试代码
if __name__ == "__main__":
    # 测试用例 1
    test_nums1 = [1, 3, 2, 3, 1]
    print(f"数组 {test_nums1} 的翻转对数量为: {reverse_pairs(test_nums1)}")

    # 测试用例 2
    test_nums2 = [2, 4, 3, 5, 1]
    print(f"数组 {test_nums2} 的翻转对数量为: {reverse_pairs(test_nums2)}")

```

文件: Code03\_CountSmallerNumbersAfterSelf.cpp

```

=====

// 计算右侧小于当前元素的个数，C++版
// 测试链接 : https://leetcode.cn/problems/count-of-smaller-numbers-after-self/

/**
 * 计算右侧小于当前元素的个数详解:
 *
 * 问题描述:
 * 给你一个整数数组 nums，按要求返回一个新数组 counts。数组 counts 有该性质：
 * counts[i] 的值是 nums[i] 右侧小于 nums[i] 的元素的数量。

```

```
*  
* 示例：  
* 输入： nums = [5, 2, 6, 1]  
* 输出： [2, 1, 1, 0]  
* 解释：  
* 5 的右侧有 2 个更小的元素 (2 和 1)  
* 2 的右侧仅有 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  
*  
* 解法思路：  
* 1. 暴力解法：对每个元素，遍历其右侧所有元素，统计比它小的元素个数，时间复杂度  $O(N^2)$   
* 2. 归并排序思想：  
*   - 在归并排序过程中，当合并两个有序数组时，如果从左侧数组选择元素，则右侧数组中已处理的元素  
*     都是小于当前元素的，可以统计数量  
*   - 由于排序会改变元素位置，需要记录原始索引  
*  
* 时间复杂度： $O(N * \log N)$  - 归并排序的时间复杂度  
* 空间复杂度： $O(N)$  - 辅助数组和结果数组的空间复杂度  
*  
* 相关题目：  
* 1. LeetCode 493. 翻转对  
* 2. LeetCode 327. 区间和的个数  
* 3. 剑指 Offer 51. 数组中的逆序对  
* 4. 牛客网 - 计算数组的小和  
*/
```

```
const int MAXN = 100001;  
int help[MAXN];  
  
// 提交以下代码到 LeetCode  
/*  
题目 3：计算右侧小于当前元素的个数
```

题目描述：

给定一个整数数组 `nums`，按要求返回一个新数组 `counts`。

数组 `counts` 有该性质： `counts[i]` 的值是 `nums[i]` 右侧小于 `nums[i]` 的元素的数量。

示例：

输入： [5, 2, 6, 1]

输出： [2, 1, 1, 0]

解释：

- 5 的右侧有 2 个小于 5 的元素 (2 和 1)

- 2 的右侧有 1 个小于 2 的元素 (1)
- 6 的右侧有 1 个小于 6 的元素 (1)
- 1 的右侧没有小于 1 的元素

核心算法思想：归并排序+索引映射

暴力解法：

- 对于每个元素，遍历其右侧所有元素，统计小于它的数量
- 时间复杂度： $O(n^2)$ ，空间复杂度： $O(1)$

最优解法：

- 利用归并排序的分治特性，在合并过程中统计右侧小于当前元素的数量
- 关键点：通过维护索引数组，在排序过程中保持元素的原始位置信息
- 时间复杂度： $O(n \log n)$ ，空间复杂度： $O(n)$

归并排序解法详解：

1. 创建索引数组，记录每个元素在原始数组中的位置
2. 进行归并排序，在合并过程中统计右侧小于当前元素的数量
3. 当左半部分元素被选中时，右半部分已处理的元素都是小于它的
4. 利用这个特性，可以在  $O(1)$  时间内累加右侧小元素的数量

C++语言特性注意事项：

1. C++中的 vector 需要预先分配空间或使用 `push_back()` 进行动态扩容
2. C++没有自动内存管理，需要注意内存泄漏问题
3. C++的整数类型有大小限制，对于极端情况可能需要处理溢出
4. 在 C++中，递归调用的栈深度可能受到编译器限制

工程化考量：

1. 异常处理：处理空数组的情况，避免数组越界
  2. 内存优化：使用预分配空间的 vector 避免频繁重新分配
  3. 性能优化：使用值传递而非引用传递小对象，减少栈开销
  4. 线程安全：当前实现不是线程安全的，需要在多线程环境中添加同步机制
- \*/

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <climits>
using namespace std;

class Solution {
private:
    // 结果数组，存储每个位置的答案
```

```
vector<int> res;
// 临时数组，用于归并排序过程中的数组合并
vector<int> temp;
// 索引数组，保存元素在原始数组中的位置
vector<int> index;
// 临时索引数组，用于归并过程中保存索引信息
vector<int> temp_index;

/***
 * 合并两个有序子数组，并在合并过程中统计右侧小于当前元素的数量
 *
 * @param nums 输入数组的副本，用于排序比较
 * @param left 当前处理区间的左边界
 * @param mid 当前处理区间的中点
 * @param right 当前处理区间的右边界
 */
void merge(vector<int>& nums, int left, int mid, int right) {
    // 初始化指针
    int i = left;      // 左子数组的指针
    int j = mid + 1;   // 右子数组的指针
    int k = left;      // 临时数组的指针

    // 合并两个有序子数组，同时统计右侧小于当前元素的数量
    while (i <= mid && j <= right) {
        if (nums[i] <= nums[j]) {
            // 左子数组元素较小，将其放入临时数组
            // 此时右子数组中已经处理的元素(j - mid - 1个)都小于 nums[i]
            temp[k] = nums[i];
            temp_index[k] = index[i];
            // 统计右侧小于当前元素的数量
            res[index[i]] += j - mid - 1;
            k++;
            i++;
        } else {
            // 右子数组元素较小，直接放入临时数组
            // 此时不需要更新统计结果
            temp[k] = nums[j];
            temp_index[k] = index[j];
            k++;
            j++;
        }
    }
}
```

```

// 处理左子数组中剩余的元素
while (i <= mid) {
    temp[k] = nums[i];
    temp_index[k] = index[i];
    // 此时右子数组中所有元素(j - mid - 1个)都小于 nums[i]
    res[index[i]] += j - mid - 1;
    k++;
    i++;
}

// 处理右子数组中剩余的元素
while (j <= right) {
    temp[k] = nums[j];
    temp_index[k] = index[j];
    k++;
    j++;
}

// 将临时数组中的排序结果复制回原数组
for (int m = left; m <= right; m++) {
    nums[m] = temp[m];
    index[m] = temp_index[m];
}

/***
 * 归并排序的递归实现
 *
 * @param nums 输入数组的副本
 * @param left 当前处理区间的左边界
 * @param right 当前处理区间的右边界
 */
void mergeSort(vector<int>& nums, int left, int right) {
    // 递归终止条件：区间长度为 0 或 1
    if (left < right) {
        // 计算中间位置，避免整数溢出
        int mid = left + (right - left) / 2;

        // 分治处理左右两个子数组
        mergeSort(nums, left, mid);
        mergeSort(nums, mid + 1, right);

        // 合并两个有序子数组
    }
}

```

```
        merge(nums, left, mid, right);
    }
}

public:
/***
 * 计算每个元素右侧小于它的元素个数
 *
 * @param nums 输入整数数组
 * @return 包含每个元素右侧小于它的元素个数的数组
 */
vector<int> countSmaller(vector<int>& nums) {
    int n = nums.size();

    // 处理边界情况: 空数组
    if (n == 0) {
        return {};
    }

    // 初始化数组
    res.resize(n, 0);           // 结果数组, 初始值全为 0
    temp.resize(n);             // 临时数组, 用于归并过程
    index.resize(n);            // 索引数组
    temp_index.resize(n);       // 临时索引数组

    // 初始化索引数组, 记录元素在原始数组中的位置
    for (int i = 0; i < n; i++) {
        index[i] = i;
    }

    // 创建原数组的副本, 避免修改原始数组
    vector<int> nums_copy(nums.begin(), nums.end());

    // 执行归并排序并统计
    mergeSort(nums_copy, 0, n - 1);

    return res;
};

// 主函数用于测试
int main() {
    Solution solution;
```

```
// 测试用例 1: 基本情况
vector<int> nums1 = {5, 2, 6, 1};
vector<int> result1 = solution.countSmaller(nums1);
cout << "输入: nums = [5,2,6,1]" << endl;
cout << "输出: [";
for (int i = 0; i < result1.size(); i++) {
    cout << result1[i];
    if (i < result1.size() - 1) cout << ",";
}
cout << "]" << endl; // 预期输出: [2,1,1,0]
```

```
// 测试用例 2: 空数组
vector<int> nums2 = {};
vector<int> result2 = solution.countSmaller(nums2);
cout << "输入: nums = []" << endl;
cout << "输出: [";
for (int i = 0; i < result2.size(); i++) {
    cout << result2[i];
    if (i < result2.size() - 1) cout << ",";
}
cout << "]" << endl; // 预期输出: []
```

```
// 测试用例 3: 单元素数组
vector<int> nums3 = {1};
vector<int> result3 = solution.countSmaller(nums3);
cout << "输入: nums = [1]" << endl;
cout << "输出: [";
for (int i = 0; i < result3.size(); i++) {
    cout << result3[i];
    if (i < result3.size() - 1) cout << ",";
}
cout << "]" << endl; // 预期输出: [0]
```

```
// 测试用例 4: 递增数组
vector<int> nums4 = {1, 2, 3, 4, 5};
vector<int> result4 = solution.countSmaller(nums4);
cout << "输入: nums = [1,2,3,4,5]" << endl;
cout << "输出: [";
for (int i = 0; i < result4.size(); i++) {
    cout << result4[i];
    if (i < result4.size() - 1) cout << ",";
}
```

```

cout << "]" << endl; // 预期输出: [0,0,0,0,0]

// 测试用例 5: 递减数组
vector<int> nums5 = {5, 4, 3, 2, 1};
vector<int> result5 = solution.countSmaller(nums5);
cout << "输入: nums = [5,4,3,2,1]" << endl;
cout << "输出: [";
for (int i = 0; i < result5.size(); i++) {
    cout << result5[i];
    if (i < result5.size() - 1) cout << ",";
}
cout << "]" << endl; // 预期输出: [4,3,2,1,0]

// 测试用例 6: 重复元素
vector<int> nums6 = {2, 2, 2, 2};
vector<int> result6 = solution.countSmaller(nums6);
cout << "输入: nums = [2,2,2,2]" << endl;
cout << "输出: [";
for (int i = 0; i < result6.size(); i++) {
    cout << result6[i];
    if (i < result6.size() - 1) cout << ",";
}
cout << "]" << endl; // 预期输出: [3,2,1,0]

// 测试用例 7: 大数值测试
vector<int> nums7 = {2147483647, -2147483648, 0};
vector<int> result7 = solution.countSmaller(nums7);
cout << "输入: nums = [2147483647,-2147483648,0]" << endl;
cout << "输出: [";
for (int i = 0; i < result7.size(); i++) {
    cout << result7[i];
    if (i < result7.size() - 1) cout << ",";
}
cout << "]" << endl; // 预期输出: [2,0,0]

return 0;
}

/*
=====
C++语言特有注意事项
=====
1. 内存管理:

```

- C++需要显式管理内存，使用 vector 时要注意正确初始化和释放
- 使用 resize() 预分配空间可以避免频繁的内存重新分配，提高性能
- 代码中已使用 resize() 为所有向量预分配了空间

## 2. 整数溢出：

- 计算  $mid = left + (right - left) / 2$  而非  $(left + right) / 2$ ，避免整数溢出
- 对于非常大的数组，需要考虑 int 类型是否足够表示索引（最大约 20 亿）
- 当数组大小超过 INT\_MAX 时，应考虑使用 size\_t 类型

## 3. 递归深度：

- C++的递归深度受系统栈大小限制，默认为几 MB
- 对于超大数组（如长度超过 1e5），递归并可能导致栈溢出
- 可以考虑实现非递归版本的归并排序来避免这个问题

## 4. 引用传递：

- 在函数参数中使用引用传递可以避免大型对象的拷贝，提高效率
- 但需要注意引用的生命周期，避免悬垂引用
- 代码中对 nums 参数使用了引用传递，但创建了副本以避免修改原数组

## 5. 类设计：

- Solution 类封装了算法的核心功能，符合面向对象设计原则
- 辅助函数被设为私有，只暴露必要的接口
- 使用成员变量存储中间结果，避免函数间传递大量数据

## 6. 异常处理：

- C++中 vector 的 resize() 在内存分配失败时会抛出 bad\_alloc 异常
- 当前代码没有显式处理异常，可以根据需要添加 try-catch 块
- 对于边界条件（如空数组），已进行了特殊处理

## 7. 模板支持：

- 可以将代码修改为模板函数，支持不同类型的输入（如 long long, float 等）
- 这需要修改类定义和相关函数签名

## 8. 编译优化：

- 可以使用编译选项如 -O2 来启用优化
- 对于频繁调用的小函数，可以使用 inline 关键字减少函数调用开销

---

## 工程化考量

---

## 1. 异常处理：

- 已处理空数组的情况
- 可以考虑添加对输入参数的有效性检查，如数组大小限制
- 可以添加 assert 断言来验证关键逻辑的正确性

## 2. 性能优化:

- 使用 `nums_copy` 避免修改原始输入，保持函数的幂等性
- 预先分配所有需要的数组空间，减少动态分配开销
- 对于小规模子数组（如长度<10），可以使用插入排序替代归并排序
- 在 `merge` 操作中使用原地合并技术可以进一步优化空间复杂度

## 3. 线程安全:

- 当前实现不是线程安全的，不适合在多线程环境中使用
- 成员变量 `res`, `temp`, `index`, `temp_index` 在多线程环境下可能导致竞态条件
- 在多线程环境中，可以：
  - 为每个线程创建独立的 `Solution` 实例
  - 将中间变量移至函数内部，避免使用成员变量
  - 添加互斥锁保护共享资源

## 4. 代码可维护性:

- 使用清晰的函数命名和详细的注释说明算法思路
- 类的设计遵循单一职责原则
- 变量命名遵循 C++ 的命名约定
- 代码结构清晰，逻辑分明

## 5. 可扩展性:

- 该算法框架可以扩展到类似问题，如计算右侧大于当前元素的个数
- 可以通过模板参数支持不同类型的数据
- 可以将归并排序部分抽象为通用算法组件

## 6. 测试策略:

- 已添加多种测试用例，覆盖常见场景和边界情况
- 可以使用 Google Test 或 Catch2 等测试框架进行更系统的测试
- 建议添加性能测试，比较不同实现的效率

## 7. 内存效率:

- 空间复杂度为  $O(N)$ ，对于大规模数据可能需要优化
- 可以考虑使用更紧凑的数据结构，如预分配的数组而非 `vector`
- 对于超大数组，可以考虑分块处理，减少内存占用

## 8. 跨平台兼容性:

- 代码使用标准 C++，具有良好的跨平台兼容性
- 避免使用平台特定的扩展或非标准库
- 在不同编译器下可能需要调整编译选项

## 9. 文档和注释:

- 函数接口有清晰的注释说明参数和返回值

- 复杂算法步骤有详细的注释解释
- 可以考虑添加更详细的算法时间和空间复杂度分析

## 10. 代码优化:

- 可以考虑使用更高效的排序算法或数据结构，如二叉搜索树、树状数组等
- 可以优化合并过程，减少不必要的操作
- 对于特定应用场景，可以考虑启发式优化

---

## 相关题目与平台信息

---

### 1. LeetCode 315. Count of Smaller Numbers After Self

- 题目链接: <https://leetcode.cn/problems/count-of-smaller-numbers-after-self/>
- 难度等级: 困难
- 标签: 归并排序、树状数组、线段树

### 2. LeetCode 493. 翻转对 (Reverse Pairs)

- 题目链接: <https://leetcode.cn/problems/reverse-pairs/>
- 难度等级: 困难
- 解题思路: 同样使用归并排序的过程统计满足条件的对

### 3. LeetCode 327. 区间和的个数 (Count of Range Sum)

- 题目链接: <https://leetcode.cn/problems/count-of-range-sum/>
- 难度等级: 困难
- 解题思路: 前缀和结合归并排序，统计满足条件的区间和

### 4. 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/>
- 难度等级: 困难
- 解题思路: 归并排序过程中统计逆序对数量

### 5. 牛客网 - 计算数组的小和

- 题目链接: <https://www.nowcoder.com/practice/edfe05a1d45c4ea89101d936cac32469>
- 解题思路: 归并排序过程中计算小和

### 6. POJ 2299. Ultra-QuickSort

- 题目链接: <http://poj.org/problem?id=2299>
- 计算逆序对数量，与本题使用类似的归并排序框架

### 7. SPOJ - INV\_CNT

- 题目链接: [https://www.spoj.com/problems/INV\\_CNT/](https://www.spoj.com/problems/INV_CNT/)
- 逆序对计数问题，可使用归并排序解决

### 8. HDU 1394. Minimum Inversion Number

- 题目链接: <http://acm.hdu.edu.cn/showproblem.php?pid=1394>
  - 最小逆序对数量, 扩展的逆序对问题
9. LintCode 1297. 统计右侧小于当前元素的个数
- 题目链接: <https://www.lintcode.com/problem/1297/>
  - 与 LeetCode 315 题相同
10. 字节跳动面试题 - 数组统计问题
- 实际面试中可能会对本题进行变体, 如不同的统计条件
  - 考察归并排序思想的灵活应用
11. 微软面试题 - 元素相对顺序问题
- 可能要求在保持相对顺序的情况下进行统计或变换
  - 与本题的索引维护思想相关
12. Google 面试题 - 二维数组统计
- 将问题扩展到二维数组, 统计每个元素右下方小于它的元素个数
  - 更复杂的归并排序或分治思想应用
13. 腾讯面试题 - 数据流中的逆序对
- 处理动态数据流, 实时统计逆序对数量
  - 可能需要使用更高效的数据结构, 如树状数组或线段树
14. 阿里巴巴面试题 - 大规模数据统计
- 要求处理超大规模数据, 考察算法优化和并行处理能力
  - 可能需要结合归并排序和分布式计算思想
15. 美团面试题 - 数组变换统计
- 在数组变换过程中统计满足特定条件的元素对数量
  - 考察对归并排序思想的深入理解和应用
16. 京东面试题 - 字符串逆序对
- 将问题应用到字符串, 统计满足条件的字符对
  - 归并排序思想在不同数据类型上的应用
17. 百度面试题 - 多维逆序对
- 扩展到多维空间, 统计多维逆序对
  - 更复杂的分治策略和数据结构应用
18. 小米面试题 - 排序过程分析
- 分析排序算法执行过程中的各种统计量
  - 与本题的归并排序过程统计思想一致

\*/

文件: Code03\_CountSmallerNumbersAfterSelf.java

```
=====
package class022;
```

```
import java.util.ArrayList;
import java.util.List;
```

```
/**
 * =====
 * 题目 3: 计算右侧小于当前元素的个数 (Count of Smaller Numbers After Self)
 * =====
 *
 * 题目来源: LeetCode 315
 * 题目链接: https://leetcode.cn/problems/count-of-smaller-numbers-after-self/
 * 难度级别: 困难
 *
 * 问题描述:
 * 给你一个整数数组 nums，按要求返回一个新数组 counts。
 * 数组 counts 有该性质: counts[i] 的值是 nums[i] 右侧小于 nums[i] 的元素的数量。
 *
 * 示例输入输出:
 * 输入: nums = [5, 2, 6, 1]
 * 输出: [2, 1, 1, 0]
 * 解释:
 * - 对于 nums[0]=5，右侧小于 5 的元素有 2 和 1，所以 counts[0]=2
 * - 对于 nums[1]=2，右侧小于 2 的元素有 1，所以 counts[1]=1
 * - 对于 nums[2]=6，右侧小于 6 的元素有 1，所以 counts[2]=1
 * - 对于 nums[3]=1，右侧没有元素，所以 counts[3]=0
 *
 * =====
 * 核心算法思想: 归并排序+索引映射
 * =====
 *
 * 方法 1: 暴力解法 (不推荐)
 * - 思路: 双重循环检查每个元素右侧有多少元素比它小
 * - 时间复杂度: O(N^2) - 双重循环
 * - 空间复杂度: O(N) - 结果数组
 * - 问题: 数据量大时超时
 *
 * 方法 2: 归并排序思想 (最优解) ★★★★★
```

\* - 核心洞察：

- \* 1. 利用归并排序的分治过程统计元素之间的大小关系
- \* 2. 关键挑战：归并排序会改变元素顺序，需要维护原始索引
- \* 3. 解决方案：创建索引数组，对索引进行排序而非对值排序

\*

\* - 归并排序过程：

- \* 1. 分治：将数组不断二分，直到只有一个元素
- \* 2. 统计：在合并过程中统计右侧小于当前元素的数量
- \* 3. 合并：按值的大小合并两个有序子数组

\*

\* - 统计右侧小元素的关键步骤：

- \* - 当右子数组中的元素被选中时，不会对左侧元素产生影响
- \* - 当左子数组中的元素被选中时，右子数组中剩余的所有元素都是比它小的
- \* - 因此，每次选中左子数组元素时，需要记录右侧已经统计过的元素数量

\*

\* - 时间复杂度详细计算：

- \*  $T(n) = 2T(n/2) + O(n)$  [Master 定理 case 2]
- \*  $= O(n \log n)$
- \* - 递归深度： $\log n$
- \* - 每层合并与统计： $O(n)$

\*

\* - 空间复杂度详细计算：

- \*  $S(n) = O(n) + O(\log n)$
- \* -  $O(n)$ ：辅助数组、索引数组、结果数组
- \* -  $O(\log n)$ ：递归调用栈
- \* 总计： $O(n)$

\*

\* - 是否最优解：★ 是 ★

- \* 理由：基于比较的算法下界为  $O(n \log n)$ ，本算法已达到最优

\*

\* =====

\* 相关题目列表（同类算法）

\* =====

\* 1. LeetCode 493 - 翻转对

- \* <https://leetcode.cn/problems/reverse-pairs/>
- \* 问题：统计满足  $nums[i] > 2*nums[j]$  且  $i < j$  的对的数量
- \* 解法：归并排序过程中使用双指针统计跨越左右区间的翻转对

\*

\* 2. LeetCode 327 - 区间和的个数

- \* <https://leetcode.cn/problems/count-of-range-sum/>
- \* 问题：统计区间和在  $[lower, upper]$  范围内的区间个数
- \* 解法：前缀和+归并排序，统计满足条件的前缀和对

\*

```
* 3. 剑指 Offer 51 / LCR 170 - 数组中的逆序对
*     https://leetcode.cn/problems/shu-zu-zhong-de-ni-xu-dui-lcof/
*     问题: 统计数组中逆序对的总数
*     解法: 归并排序过程中统计逆序对数量
*
* 4. LeetCode 1365 - 有多少小于当前数字的数字
*     https://leetcode.cn/problems/how-many-numbers-are-smaller-than-the-current-number/
*     问题: 统计数组中小于当前数字的数字个数 (全数组范围)
*     解法: 排序+哈希表映射
*
* 5. POJ 2299 - Ultra-QuickSort
*     http://poj.org/problem?id=2299
*     问题: 计算将数组排序所需的最小交换次数 (即逆序对数量)
*     解法: 归并排序统计逆序对
*
* 6. HackerRank - Merge Sort: Counting Inversions
*     https://www.hackerrank.com/challenges/merge-sort/problem
*     问题: 统计逆序对数量
*     解法: 归并排序统计逆序对
*
* 7. 牛客网 - 计算右侧小于当前元素的个数
*     问题: 与 LeetCode 315 相同
*     解法: 归并排序+索引映射
*
* 8. 杭电 OJ - 1394
*     http://acm.hdu.edu.cn/showproblem.php?pid=1394
*     问题: 将数组循环左移, 求所有可能排列中的最小逆序对数量
*     解法: 归并排序+逆序对性质分析
*
* 9. 洛谷 P1908 - 逆序对
*     https://www.luogu.com.cn/problem/P1908
*     问题: 统计数组中逆序对的总数
*     解法: 归并排序统计逆序对
*
* 10. SPOJ - INVCNT
*      https://www.spoj.com/problems/INVCNT/
*      问题: 统计逆序对数量
*      解法: 归并排序统计逆序对
*
* 这些题目虽然具体形式不同, 但核心思想都是利用归并排序的分治特性, 在合并过程中高效统计满足特定条件的元素对数量。
*/
import java.util.Arrays;
```

```
public class Code03_CountSmallerNumbersAfterSelf {

    /**
     * Pair 类用于在归并排序过程中同时保存元素值和原始索引
     * 这样可以在排序过程中维护原始数组的位置信息
     */
    static class Pair {
        int val; // 元素值
        int idx; // 元素在原始数组中的索引

        /**
         * 构造函数
         * @param val 元素值
         * @param idx 原始索引
         */
        Pair(int val, int idx) {
            this.val = val;
            this.idx = idx;
        }
    }

    // 常量定义
    public static final int MAXN = 100001;

    // 全局数组，避免多次内存分配
    public static Pair[] arr = new Pair[MAXN]; // 原数组，保存值和索引
    public static Pair[] help = new Pair[MAXN]; // 辅助数组，用于归并过程
    public static int[] count = new int[MAXN]; // 结果数组，存储每个元素右侧小于它的元素个数

    /**
     * 计算右侧小于当前元素的个数的主方法
     *
     * @param nums 输入整数数组
     * @return 包含每个元素右侧小于它的元素个数的列表
     *
     * Java 语言特性注意事项：
     * 1. 数组是引用类型，作为参数传递时传递的是引用副本
     * 2. 使用全局数组避免了频繁的内存分配和释放
     * 3. 方法使用静态修饰符，可以直接通过类名调用
     */
    public static List<Integer> countSmaller(int[] nums) {
        int n = nums.length;
```

```
// 初始化 Pair 数组，将元素值和索引关联起来
for (int i = 0; i < n; i++) {
    arr[i] = new Pair(nums[i], i);
}

// 重置计数数组
Arrays.fill(count, 0);

// 执行归并排序并统计
mergeSort(0, n - 1);

// 转换结果格式
List<Integer> result = new ArrayList<>(n);
for (int i = 0; i < n; i++) {
    result.add(count[i]);
}
return result;
}

/**
 * 归并排序的核心方法
 *
 * @param l 当前处理区间的左边界
 * @param r 当前处理区间的右边界
 *
 * 算法分析：
 * - 递归实现归并排序
 * - 时间复杂度：O(n log n)
 * - 空间复杂度：O(n)
 */
public static void mergeSort(int l, int r) {
    // 基本情况：区间只有一个元素时直接返回
    if (l == r) {
        return;
    }

    // 计算中间位置，使用这种方式避免大整数溢出
    int m = l + (r - 1) / 2;

    // 分治：递归处理左右子区间
    mergeSort(l, m);
    mergeSort(m + 1, r);
```

```

// 合并两个有序子区间，同时统计结果
merge(l, m, r);
}

/***
 * 合并两个有序子数组，并在合并过程中统计右侧小于当前元素的个数
 *
 * @param l 当前处理区间的左边界
 * @param m 当前处理区间的中点
 * @param r 当前处理区间的右边界
 *
 * 核心统计逻辑：
 * - 当从左侧子数组选取元素时，右侧子数组中已处理的元素都小于该元素
 * - 因此需要将右侧已处理的元素数量累加到该元素的计数中
 */
public static void merge(int l, int m, int r) {
    int i = l; // 辅助数组的指针
    int a = l; // 左侧子数组的指针
    int b = m + 1; // 右侧子数组的指针

    // 合并两个子数组，同时统计右侧小于当前元素的个数
    while (a <= m && b <= r) {
        if (arr[a].val <= arr[b].val) {
            // 当左侧元素小于等于右侧元素时，右侧数组中已经处理的元素都小于当前左侧元素
            // 统计右侧已处理的元素数量：b - (m + 1) = b - m - 1
            count[arr[a].idx] += (b - m - 1);
            help[i++] = arr[a++];
        } else {
            // 当右侧元素小于左侧元素时，将右侧元素放入辅助数组
            // 此时不更新计数，因为左侧元素还未被处理
            help[i++] = arr[b++];
        }
    }

    // 处理左侧剩余元素
    while (a <= m) {
        // 左侧剩余元素的右侧所有元素都小于它
        count[arr[a].idx] += (b - m - 1);
        help[i++] = arr[a++];
    }

    // 处理右侧剩余元素
    while (b <= r) {

```

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

// 将辅助数组内容复制回原数组
for (i = l; i <= r; i++) {
    arr[i] = help[i];
}
}

/**
 * 主方法，用于测试
 */
public static void main(String[] args) {
    // 测试用例 1: 基本情况
    int[] test1 = {5, 2, 6, 1};
    System.out.println("输入: " + Arrays.toString(test1));
    System.out.println("输出: " + countSmaller(test1)); // 预期输出: [2, 1, 1, 0]

    // 测试用例 2: 空数组
    int[] test2 = {};
    System.out.println("输入: " + Arrays.toString(test2));
    System.out.println("输出: " + countSmaller(test2)); // 预期输出: []

    // 测试用例 3: 单元素数组
    int[] test3 = {1};
    System.out.println("输入: " + Arrays.toString(test3));
    System.out.println("输出: " + countSmaller(test3)); // 预期输出: [0]

    // 测试用例 4: 逆序数组
    int[] test4 = {5, 4, 3, 2, 1};
    System.out.println("输入: " + Arrays.toString(test4));
    System.out.println("输出: " + countSmaller(test4)); // 预期输出: [4, 3, 2, 1, 0]

    // 测试用例 5: 有序数组
    int[] test5 = {1, 2, 3, 4, 5};
    System.out.println("输入: " + Arrays.toString(test5));
    System.out.println("输出: " + countSmaller(test5)); // 预期输出: [0, 0, 0, 0, 0]

    // 测试用例 6: 重复元素
    int[] test6 = {2, 2, 2};
    System.out.println("输入: " + Arrays.toString(test6));
    System.out.println("输出: " + countSmaller(test6)); // 预期输出: [2, 1, 0]
```

```
// 测试用例 7: 包含负数
int[] test7 = {-1, -2, 3, -4, 5};
System.out.println("输入: " + Arrays.toString(test7));
System.out.println("输出: " + countSmaller(test7)); // 预期输出: [2, 1, 1, 0, 0]

// 测试用例 8: 大数值测试
int[] test8 = {Integer.MAX_VALUE, Integer.MIN_VALUE, 0};
System.out.println("输入: [" + Integer.MAX_VALUE + ", " + Integer.MIN_VALUE + ", 0]");
System.out.println("输出: " + countSmaller(test8)); // 预期输出: [2, 0, 0]
}

/*
* =====
* Java 语言特有关注事项
* =====
* 1. 静态成员变量的使用:
*   - 使用静态数组避免了频繁的内存分配和释放
*   - 但需注意线程安全问题, 多线程环境下可能需要额外的同步措施
*   - 静态变量在类加载时初始化, 在类卸载时销毁
*
* 2. 内存优化:
*   - 预分配固定大小的数组 (MAXN) 减少了动态扩容的开销
*   - 使用 Arrays.fill() 方法高效初始化数组
*   - ArrayList 的初始容量设置为数组大小, 避免扩容开销
*
* 3. 整数类型和溢出:
*   - Java 的 int 类型是 32 位有符号整数, 范围为 $-2^{31}$  到  $2^{31}-1$ 
*   - 计算中间点使用  $1 + (r - 1) / 2$  避免整数溢出
*   - 对于非常大的数组, 索引可能超出 int 范围, 需要考虑使用 long 类型
*
* 4. 递归深度控制:
*   - Java 的默认递归深度限制约为 1000–2000 层 (依赖 JVM 实现)
*   - 对于大规模数据, 可以通过 JVM 参数-Xss 调整栈大小
*   - 对于极端情况, 考虑实现迭代版本的归并排序
*
* 5. 异常处理:
*   - Java 中可以使用 try-catch 块处理可能的异常
*   - 可以添加对 null 输入、极大数组等边界情况的检查
*
* 6. 性能优化:
*   - 对于小规模子数组 (如长度<10), 可以使用插入排序提高效率
*   - 可以添加判断条件, 当  $arr[m].val \leq arr[m+1].val$  时, 子数组已有序, 跳过合并
*   - 减少对象创建和垃圾回收压力, 提高性能
```

```
/*
 * =====
 * 工程化考量
 * =====
 * 1. 异常处理:
 *   - 可以添加对 null 输入的检查: if (nums == null) { return new ArrayList<>(); }
 *   - 对于极大数组, 可以添加数组长度检查, 避免超出 MAXN 限制
 *   - 可以抛出 IllegalArgumentException 处理无效输入
 *
 * 2. 线程安全:
 *   - 当前实现使用静态变量, 不是线程安全的
 *   - 改进方案 1: 将静态变量改为方法局部变量, 每次调用时创建新数组
 *   - 改进方案 2: 使用 ThreadLocal 存储线程局部变量
 *   - 改进方案 3: 添加同步机制 (synchronized 或 ReentrantLock)
 *
 * 3. 测试与质量保证:
 *   - 已提供多种测试用例, 覆盖常见场景
 *   - 建议使用 JUnit 框架编写自动化单元测试
 *   - 可以添加性能测试, 验证算法在大规模数据下的表现
 *
 * 4. 代码可读性:
 *   - 使用了清晰的变量命名和方法命名
 *   - 添加了详细的注释说明算法逻辑
 *   - 代码结构清晰, 易于理解和维护
 *
 * 5. 可扩展性:
 *   - 可以将算法扩展为泛型版本, 支持任意 Comparable 类型
 *   - 可以将核心逻辑抽象为策略模式, 方便替换不同的统计策略
 *   - 可以添加缓存机制, 避免重复计算
 *
 * 6. 内存管理:
 *   - 使用预分配的静态数组减少 GC 压力
 *   - 对于超大规模数据, 可以考虑分块处理或使用堆外内存
 *
 * 7. 性能调优:
 *   - 可以使用 JMH (Java Microbenchmark Harness) 进行性能基准测试
 *   - 可以通过 JIT 编译器优化, 避免热点路径上的对象创建
 *   - 考虑使用并行流或 Fork/Join 框架实现并行归并排序
 *
 * 8. 代码优化建议:
 *   - 对于大规模数据, 考虑使用更高效的排序算法或数据结构, 如二叉搜索树、树状数组等
```

- \*    - 可以优化合并过程，减少不必要的对象创建
  - \*    - 对于特定应用场景，可以考虑启发式优化
  - \*
  - \* 9. 跨平台兼容性：
    - \*    - Java 具有良好的跨平台特性，代码可以在不同的操作系统和 JVM 上运行
    - \*    - 但需注意不同 JVM 实现可能在性能和行为上有所差异
  - \*
  - \* 10. 代码安全性：
    - \*    - 避免使用全局变量存储状态，减少副作用
    - \*    - 对外部输入进行严格验证，防止注入攻击
    - \*    - 注意数组边界检查，避免越界异常
  - \*/
- 
- /\*
  - \* =====
  - \* 相关题目与平台信息
  - \* =====
  - \* 1. LeetCode 315. Count of Smaller Numbers After Self
    - \*    - 题目链接: <https://leetcode.cn/problems/count-of-smaller-numbers-after-self/>
    - \*    - 难度等级: 困难
    - \*    - 标签: 归并排序、树状数组、线段树
  - \*
  - \* 2. LeetCode 493. 翻转对 (Reverse Pairs)
    - \*    - 题目链接: <https://leetcode.cn/problems/reverse-pairs/>
    - \*    - 难度等级: 困难
    - \*    - 解题思路: 同样使用归并排序的过程统计满足条件的对
  - \*
  - \* 3. LeetCode 327. 区间和的个数 (Count of Range Sum)
    - \*    - 题目链接: <https://leetcode.cn/problems/count-of-range-sum/>
    - \*    - 难度等级: 困难
    - \*    - 解题思路: 前缀和结合归并排序，统计满足条件的区间和
  - \*
  - \* 4. 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/>
    - \*    - 难度等级: 困难
    - \*    - 解题思路: 归并排序过程中统计逆序对数量
  - \*
  - \* 5. LeetCode 1365. 有多少小于当前数字的数字
    - \*    - 题目链接: <https://leetcode.cn/problems/how-many-numbers-are-smaller-than-the-current-number/>
    - \*    - 难度等级: 简单
    - \*    - 解题思路: 排序+哈希表映射，全数组范围统计
  - \*

- \* 6. 牛客网 - 计算数组的小和
  - 题目链接: <https://www.nowcoder.com/practice/edfe05a1d45c4ea89101d936cac32469>
  - 解题思路: 归并排序过程中计算小和
- \*
- \* 7. HackerRank - Merge Sort: Counting Inversions
  - 题目链接: <https://www.hackerrank.com/challenges/merge-sort/problem>
  - 难度等级: 中等
  - 解题思路: 归并排序统计逆序对数量
- \*
- \* 8. POJ 2299. Ultra-QuickSort
  - 题目链接: <http://poj.org/problem?id=2299>
  - 解题思路: 计算将数组排序所需的最小交换次数（即逆序对数量）
- \*
- \* 9. HDU 1394. Minimum Inversion Number
  - 题目链接: <http://acm.hdu.edu.cn/showproblem.php?pid=1394>
  - 解题思路: 将数组循环左移, 求所有可能排列中的最小逆序对数量
- \*
- \* 10. LintCode 1297. 统计右侧小于当前元素的个数
  - 题目链接: <https://www.lintcode.com/problem/1297/>
  - 与 LeetCode 315 题相同
- \*
- \* 11. SPOJ - INVCNT
  - 题目链接: <https://www.spoj.com/problems/INVCNT/>
  - 解题思路: 统计逆序对数量, 可使用归并排序解决
- \*
- \* 12. 字节跳动面试题 - 数组统计问题
  - 实际面试中可能会对本题进行变体, 如不同的统计条件
  - 考察归并排序思想的灵活应用
- \*
- \* 13. 微软面试题 - 元素相对顺序问题
  - 可能要求在保持相对顺序的情况下进行统计或变换
  - 与本题的索引维护思想相关
- \*
- \* 14. Google 面试题 - 二维数组统计
  - 将问题扩展到二维数组, 统计每个元素右下方小于它的元素个数
  - 更复杂的归并排序或分治思想应用
- \*
- \* 15. 腾讯面试题 - 数据流中的逆序对
  - 处理动态数据流, 实时统计逆序对数量
  - 可能需要使用更高效的数据结构, 如树状数组或线段树
- \*
- \* 16. 阿里巴巴面试题 - 大规模数据统计
  - 要求处理超大规模数据, 考察算法优化和并行处理能力

```
*      - 可能需要结合归并排序和分布式计算思想
*
* 17. 美团面试题 - 数组变换统计
*      - 在数组变换过程中统计满足特定条件的元素对数量
*      - 考察对归并排序思想的深入理解和应用
*
* 18. 京东面试题 - 字符串逆序对
*      - 将问题应用到字符串，统计满足条件的字符对
*      - 归并排序思想在不同数据类型上的应用
*
* 19. 百度面试题 - 多维逆序对
*      - 扩展到多维空间，统计多维逆序对
*      - 更复杂的分治策略和数据结构应用
*
* 20. 小米面试题 - 排序过程分析
*      - 分析排序算法执行过程中的各种统计量
*      - 与本题的归并排序过程统计思想一致
*/
}
```

=====

文件: Code03\_CountSmallerNumbersAfterSelf.py

=====

```
"""
=====
题目 3: 计算右侧小于当前元素的个数 (Count of Smaller Numbers After Self)
=====
```

题目来源: LeetCode 315

题目链接: <https://leetcode.cn/problems/count-of-smaller-numbers-after-self/>

难度级别: 困难

问题描述:

给你一个整数数组 `nums`，按要求返回一个新数组 `counts`。

数组 `counts` 有该性质: `counts[i]` 的值是 `nums[i]` 右侧小于 `nums[i]` 的元素的数量。

示例输入输出:

输入: `nums = [5, 2, 6, 1]`

输出: `[2, 1, 1, 0]`

解释:

- 对于 `nums[0]=5`, 右侧小于 5 的元素有 2 和 1, 所以 `counts[0]=2`
- 对于 `nums[1]=2`, 右侧小于 2 的元素有 1, 所以 `counts[1]=1`

- 对于  $\text{nums}[2]=6$ , 右侧小于 6 的元素有 1, 所以  $\text{counts}[2]=1$
- 对于  $\text{nums}[3]=1$ , 右侧没有元素, 所以  $\text{counts}[3]=0$

=====核心算法思想：归并排序

+索引映射

### 方法 1：暴力解法（不推荐）

- 思路：双重循环检查每个元素右侧有多少元素比它小
- 时间复杂度： $O(N^2)$  - 双重循环
- 空间复杂度： $O(N)$  - 结果数组
- 问题：数据量大时超时

### 方法 2：归并排序思想（最优解）★★★★★

- 核心洞察：
  1. 利用归并排序的分治过程统计元素之间的大小关系
  2. 关键挑战：归并排序会改变元素顺序，需要维护原始索引
  3. 解决方案：创建索引数组，对索引进行排序而非对值排序
- 归并排序过程：
  1. 分治：将数组不断二分，直到只有一个元素
  2. 统计：在合并过程中统计右侧小于当前元素的数量
  3. 合并：按值的大小合并两个有序子数组
- 统计右侧小元素的关键步骤：
  - 当右子数组中的元素被选中时，不会对左侧元素产生影响
  - 当左子数组中的元素被选中时，右子数组中剩余的所有元素都是比它小的
  - 因此，每次选中左子数组元素时，需要记录右侧已经统计过的元素数量
- 时间复杂度详细计算：
 
$$\begin{aligned} T(n) &= 2T(n/2) + O(n) \quad [\text{Master 定理 case 2}] \\ &= O(n \log n) \end{aligned}$$
  - 递归深度： $\log n$
  - 每层合并与统计： $O(n)$
- 空间复杂度详细计算：
 
$$\begin{aligned} S(n) &= O(n) + O(\log n) \\ &- O(n)：\text{辅助数组、索引数组、结果数组} \\ &- O(\log n)：\text{递归调用栈} \\ &\text{总计：} O(n) \end{aligned}$$
- 是否最优解：★ 是 ★
 

理由：基于比较的算法下界为  $O(n \log n)$ ，本算法已达到最优

1. LeetCode 493 - 翻转对

<https://leetcode.cn/problems/reverse-pairs/>

问题：统计满足  $\text{nums}[i] > 2 * \text{nums}[j]$  且  $i < j$  的对的数量

解法：归并排序过程中使用双指针统计跨越左右区间的翻转对

2. LeetCode 327 - 区间和的个数

<https://leetcode.cn/problems/count-of-range-sum/>

问题：统计区间和在 [lower, upper] 范围内的区间个数

解法：前缀和+归并排序，统计满足条件的前缀和对

3. 剑指 Offer 51 / LCR 170 - 数组中的逆序对

<https://leetcode.cn/problems/shu-zu-zhong-de-ni-xu-dui-lcof/>

问题：统计数组中逆序对的总数

解法：归并排序过程中统计逆序对数量

4. LeetCode 1365 - 有多少小于当前数字的数字

<https://leetcode.cn/problems/how-many-numbers-are-smaller-than-the-current-number/>

问题：统计数组中小于当前数字的数字个数（全数组范围）

解法：排序+哈希表映射

5. POJ 2299 - Ultra-QuickSort

<http://poj.org/problem?id=2299>

问题：计算将数组排序所需的最小交换次数（即逆序对数量）

解法：归并排序统计逆序对

6. HackerRank - Merge Sort: Counting Inversions

<https://www.hackerrank.com/challenges/merge-sort/problem>

问题：统计逆序对数量

解法：归并排序统计逆序对

7. 牛客网 - 计算右侧小于当前元素的个数

问题：与 LeetCode 315 相同

解法：归并排序+索引映射

8. 杭电 OJ - 1394

<http://acm.hdu.edu.cn/showproblem.php?pid=1394>

问题：将数组循环左移，求所有可能排列中的最小逆序对数量

解法：归并排序+逆序对性质分析

## 9. 洛谷 P1908 - 逆序对

<https://www.luogu.com.cn/problem/P1908>

问题：统计数组中逆序对的总数

解法：归并排序统计逆序对

## 10. SPOJ - INVCNT

<https://www.spoj.com/problems/INVCNT/>

问题：统计逆序对数量

解法：归并排序统计逆序对

这些题目虽然具体形式不同，但核心思想都是利用归并排序的分治特性，在合并过程中高效统计满足特定条件的元素对数量。

"""

"""

计算右侧小于当前元素的个数详解：

问题描述：

给你一个整数数组 `nums`，按要求返回一个新数组 `counts`。数组 `counts` 有该性质：  
`counts[i]` 的值是 `nums[i]` 右侧小于 `nums[i]` 的元素的数量。

示例：

输入： `nums = [5, 2, 6, 1]`

输出： `[2, 1, 1, 0]`

解释：

5 的右侧有 2 个更小的元素（2 和 1）

2 的右侧仅有 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

解法思路：

1. 暴力解法：对每个元素，遍历其右侧所有元素，统计比它小的元素个数，时间复杂度  $O(N^2)$

2. 归并排序思想：

- 在归并排序过程中，当合并两个有序数组时，如果从左侧数组选择元素，则右侧数组中已处理的元素都是小于当前元素的，可以统计数量
- 由于排序会改变元素位置，需要记录原始索引

时间复杂度： $O(N * \log N)$  – 归并排序的时间复杂度

空间复杂度： $O(N)$  – 辅助数组和结果数组的空间复杂度

相关题目：

1. LeetCode 493. 翻转对

2. LeetCode 327. 区间和的个数

3. 剑指 Offer 51. 数组中的逆序对

4. 牛客网 - 计算数组的小和

"""

```
def count_smaller(nums):
```

"""

计算右侧小于当前元素的个数

Args:

nums (List[int]): 输入整数数组

Returns:

List[int]: 包含每个元素右侧小于它的元素个数的列表

Python 语言特性注意事项:

1. Python 的列表是可变对象，在递归调用时会被修改，因此需要注意索引映射的处理
2. Python 的整数没有大小限制，不会出现整数溢出问题
3. 对于大规模数据，Python 的递归深度可能受限，默认递归深度约为 1000
4. Python 中的列表推导式和切片操作可以使代码更简洁，但需要注意效率影响

"""

```
if not nums:
```

```
    return []
```

# 结果数组

```
result = [0] * len(nums)
```

# 索引数组，用于维护原始位置信息

```
indexes = list(range(len(nums)))
```

# 归并排序主函数

```
def merge_sort(left, right):
```

"""

归并排序的核心函数，同时统计右侧小于当前元素的个数

Args:

left (int): 当前处理区间的左边界

right (int): 当前处理区间的右边界

"""

```
if left >= right:
```

```
    return
```

# 计算中间位置

```
mid = left + (right - left) // 2
```

```

# 分治处理左右两个子数组
merge_sort(left, mid)
merge_sort(mid + 1, right)

# 合并两个有序子数组，同时统计
merge(left, mid, right)

def merge(left, mid, right):
    """
    合并两个有序子数组，并在合并过程中统计右侧小于当前元素的个数

```

核心统计逻辑：

- 当右子数组的元素被选中时，不会对左侧元素产生影响
- 当左子数组的元素被选中时，右子数组中剩余的所有元素都是比它小的
- 因此，每次选中左子数组元素时，需要记录右侧已经统计过的元素数量

Args:

```

    left (int): 当前处理区间的左边界
    mid (int): 当前处理区间的中点
    right (int): 当前处理区间的右边界
    """

```

# 辅助数组，用于暂存排序后的索引

```
temp = [0] * (right - left + 1)
```

```
i = 0 # temp 数组的指针
```

```
a = left # 左侧子数组的指针
```

```
b = mid + 1 # 右侧子数组的指针
```

# 记录右侧已经处理的元素数量（即小于左侧当前元素的数量）

```
right_count = 0
```

# 合并两个子数组，同时统计右侧小于当前元素的个数

```
while a <= mid and b <= right:
```

```
    if nums[indexes[b]] < nums[indexes[a]]:
```

```
        # 右侧元素更小，先放入 temp 数组，增加右侧已处理元素计数
```

```
        temp[i] = indexes[b]
```

```
        b += 1
```

```
        right_count += 1
```

```
    else:
```

```
        # 左侧元素更小或相等，先放入 temp 数组，更新该元素对应的右侧小元素数量
```

```
        result[indexes[a]] += right_count
```

```
        temp[i] = indexes[a]
```

```
        a += 1
```

```
    i += 1
```

```

# 处理剩余的左侧元素
while a <= mid:
    result[indexes[a]] += right_count # 这些元素的右侧还有 right_count 个较小元素
    temp[i] = indexes[a]
    a += 1
    i += 1

# 处理剩余的右侧元素（不需要更新结果，因为它们不会对任何元素产生统计贡献）
while b <= right:
    temp[i] = indexes[b]
    b += 1
    i += 1

# 将排序后的索引复制回原索引数组
for i in range(len(temp)):
    indexes[left + i] = temp[i]

# 执行归并排序并统计
merge_sort(0, len(nums) - 1)

return result
}

# 测试代码
if __name__ == "__main__":
    # 测试用例 1: 基本情况
    test_nums1 = [5, 2, 6, 1]
    print(f"输入: nums = {test_nums1}")
    print(f"输出: {count_smaller(test_nums1)}") # 预期输出: [2, 1, 1, 0]

    # 测试用例 2: 空数组
    test_nums2 = []
    print(f"\n 输入: nums = {test_nums2}")
    print(f"输出: {count_smaller(test_nums2)}") # 预期输出: []

    # 测试用例 3: 单元素数组
    test_nums3 = [1]
    print(f"\n 输入: nums = {test_nums3}")
    print(f"输出: {count_smaller(test_nums3)}") # 预期输出: [0]

    # 测试用例 4: 递增数组
    test_nums4 = [1, 2, 3, 4, 5]

```

```
print(f"\n 输入: nums = {test_nums4}")
print(f"输出: {count_smaller(test_nums4)}") # 预期输出: [0, 0, 0, 0, 0]

# 测试用例 5: 递减数组
test_nums5 = [5, 4, 3, 2, 1]
print(f"\n 输入: nums = {test_nums5}")
print(f"输出: {count_smaller(test_nums5)}") # 预期输出: [4, 3, 2, 1, 0]

# 测试用例 6: 重复元素
test_nums6 = [2, 2, 2, 2]
print(f"\n 输入: nums = {test_nums6}")
print(f"输出: {count_smaller(test_nums6)}") # 预期输出: [3, 2, 1, 0]

# 测试用例 7: 包含负数
test_nums7 = [-1, -2, 3, -4, 5]
print(f"\n 输入: nums = {test_nums7}")
print(f"输出: {count_smaller(test_nums7)}") # 预期输出: [2, 1, 1, 0, 0]

# 测试用例 8: 大数值测试
test_nums8 = [2147483647, -2147483648, 0]
print(f"\n 输入: nums = {test_nums8}")
print(f"输出: {count_smaller(test_nums8)}") # 预期输出: [2, 0, 0]
```

"""

## =====

### Python 语言特有关注事项

## =====

#### 1. 整数类型:

- Python 中的整数没有大小限制，不会出现整数溢出问题
- 无需像 C++/Java 那样担心 mid 计算时的溢出

#### 2. 递归深度限制:

- Python 默认递归深度限制为约 1000 层
- 对于长度超过 1e5 的数组，可能触发递归深度错误 (RecursionError)
- 解决方案：
  - 可通过 `sys.setrecursionlimit(1000000)` 临时调整递归深度限制
  - 考虑实现迭代版归并排序

#### 3. 可变对象特性:

- Python 列表是可变对象，在函数内部修改会影响外部
- 在实现中，我们使用可变列表 `result` 和 `indexes` 来保存中间结果

#### 4. 列表操作效率:

- 列表索引访问:  $O(1)$ , 非常高效
- 列表追加操作: 均摊  $O(1)$ , 但预分配空间更高效
- 在 merge 函数中, 我们预先分配了 temp 数组, 避免频繁的动态扩展

#### 5. 函数嵌套:

- 内部函数 merge\_sort 和 merge 可以直接访问外部函数的变量
- 这种设计使代码更简洁, 但需要注意变量作用域

#### 6. 切片操作:

- Python 的切片操作会创建新列表, 可能导致  $O(n)$  的额外空间和时间开销
- 当前实现避免了不必要的切片操作, 直接在原数组上通过索引操作

#### 7. 装饰器优化:

- 可以使用 @functools.lru\_cache 装饰器对特定情况进行缓存 (注意可变参数不可哈希)
- 对于递归版本, 可以考虑使用记忆化优化重复计算

#### 8. Python 特有的优化技巧:

- 使用列表推导式代替显式循环
- 使用 collections 模块中的数据结构可能提高特定场景的性能
- 考虑使用 NumPy 数组进行大规模数值计算, 提供更好的缓存局部性

---

### 工程化考量

---

#### 1. 异常处理:

- 已处理空数组的边界情况
- 可以添加对输入类型的检查, 确保输入是列表类型
- 建议添加 try-except 块捕获可能的异常, 如递归深度错误
- 示例异常处理代码:

```
```python
def count_smaller_safe(nums):
    try:
        if not isinstance(nums, list):
            raise TypeError("Input must be a list")
        return count_smaller(nums)
    except RecursionError:
        # 处理递归深度问题, 返回迭代版本或其他实现
        return count_smaller_iterative(nums)
    except Exception as e:
        print(f"Error occurred: {e}")
```

```
    return []
```

```

## 2. 性能优化:

- 对于超大规模数据，可以考虑使用非递归实现避免栈溢出
- 对于小规模子数组，可以切换到插入排序优化常数因子
- 可使用 PyPy 解释器运行，对于计算密集型任务性能提升显著
- 对于特定问题，可以考虑使用更高效的数据结构，如二叉搜索树或线段树

## 3. 测试策略:

- 已添加多种测试用例，覆盖常见场景和边界情况
- 可以使用 unittest 或 pytest 框架进行更系统化的单元测试
- 建议添加性能测试，测量不同规模数据下的执行时间
- 可以添加随机测试，验证算法在各种随机输入下的正确性

## 4. 可扩展性:

- 当前算法框架可以扩展到类似问题，如计算右侧大于当前元素的个数
- 可以实现一个通用的归并排序计数框架，通过回调函数支持不同的计数逻辑
- 示例可扩展接口：

```
``` python
def merge_sort_count(nums, count_callback):
    # 通用归并排序计数框架
    # count_callback(left_val, right_val) 用于判断是否需要计数
    # 返回值：统计结果
    pass
```

```

## 5. 代码可读性:

- 使用清晰的函数命名和详细的注释
- 将复杂逻辑拆分为小函数，每个函数负责单一职责
- 使用文档字符串(docstring)详细说明函数参数、返回值和功能
- 添加类型提示可以提高代码的可读性和可维护性

## 6. 并行处理:

- 对于超大规模数据，可以考虑并行计算框架如 multiprocessing
- 可以将数组分块，并行处理子数组，然后合并结果
- 注意并行处理时的数据同步和结果合并问题

## 7. 内存效率:

- 当前实现的空间复杂度为  $O(n)$ ，对于大规模数据可能需要优化
- 可以考虑使用原地排序算法减少辅助空间，但会增加实现复杂度
- 对于流式数据，可以使用在线算法，避免一次性加载全部数据

## 8. 兼容性考虑:

- 代码兼容 Python 3.x 版本
- 对于 Python 2.x 需要进行适当修改（如 print 语句语法）
- 可考虑使用 future 模块确保跨版本兼容性

## 9. 代码优化建议:

- 对于大规模数据，考虑使用更高效的排序算法或数据结构
- 可以使用位运算优化某些计算步骤
- 对于特定应用场景，可以考虑启发式优化
- 使用 profile 工具分析性能瓶颈，针对性优化

## 10. 代码安全性:

- 避免使用全局变量存储状态，减少副作用
- 对外部输入进行严格验证，防止注入攻击
- 注意深拷贝和浅拷贝的区别，避免意外修改

---

## 相关题目与平台信息

---

### 1. LeetCode 315. Count of Smaller Numbers After Self

- 题目链接: <https://leetcode.cn/problems/count-of-smaller-numbers-after-self/>
- 难度等级: 困难
- 标签: 归并排序、树状数组、线段树

### 2. LeetCode 493. 翻转对 (Reverse Pairs)

- 题目链接: <https://leetcode.cn/problems/reverse-pairs/>
- 难度等级: 困难
- 解题思路: 同样使用归并排序的过程统计满足条件的对

### 3. LeetCode 327. 区间和的个数 (Count of Range Sum)

- 题目链接: <https://leetcode.cn/problems/count-of-range-sum/>
- 难度等级: 困难
- 解题思路: 前缀和结合归并排序，统计满足条件的区间和

### 4. 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/>
- 难度等级: 困难
- 解题思路: 归并排序过程中统计逆序对数量

### 5. LeetCode 1365. 有多少小于当前数字的数字

- 题目链接: <https://leetcode.cn/problems/how-many-numbers-are-smaller-than-the-current-number/>
- 难度等级: 简单

- 解题思路：排序+哈希表映射，全数组范围统计

## 6. 牛客网 - 计算数组的小和

- 题目链接：<https://www.nowcoder.com/practice/edfe05a1d45c4ea89101d936cac32469>

- 解题思路：归并排序过程中计算小和

## 7. HackerRank - Merge Sort: Counting Inversions

- 题目链接：<https://www.hackerrank.com/challenges/merge-sort/problem>

- 难度等级：中等

- 解题思路：归并排序统计逆序对数量

## 8. POJ 2299. Ultra-QuickSort

- 题目链接：<http://poj.org/problem?id=2299>

- 解题思路：计算将数组排序所需的最小交换次数（即逆序对数量）

## 9. HDU 1394. Minimum Inversion Number

- 题目链接：<http://acm.hdu.edu.cn/showproblem.php?pid=1394>

- 解题思路：将数组循环左移，求所有可能排列中的最小逆序对数量

## 10. LintCode 1297. 统计右侧小于当前元素的个数

- 题目链接：<https://www.lintcode.com/problem/1297/>

- 与 LeetCode 315 题相同

## 11. SPOJ - INVCNT

- 题目链接：<https://www.spoj.com/problems/INVCNT/>

- 解题思路：统计逆序对数量，可使用归并排序解决

## 12. 字节跳动面试题 - 数组统计问题

- 实际面试中可能会对本题进行变体，如不同的统计条件

- 考察归并排序思想的灵活应用

## 13. 微软面试题 - 元素相对顺序问题

- 可能要求在保持相对顺序的情况下进行统计或变换

- 与本题的索引维护思想相关

## 14. Google 面试题 - 二维数组统计

- 将问题扩展到二维数组，统计每个元素右下方小于它的元素个数

- 更复杂的归并排序或分治思想应用

## 15. 腾讯面试题 - 数据流中的逆序对

- 处理动态数据流，实时统计逆序对数量

- 可能需要使用更高效的数据结构，如树状数组或线段树

16. 阿里巴巴面试题 - 大规模数据统计

- 要求处理超大规模数据，考察算法优化和并行处理能力
- 可能需要结合归并排序和分布式计算思想

17. 美团面试题 - 数组变换统计

- 在数组变换过程中统计满足特定条件的元素对数量
- 考察对归并排序思想的深入理解和应用

18. 京东面试题 - 字符串逆序对

- 将问题应用到字符串，统计满足条件的字符对
- 归并排序思想在不同数据类型上的应用

19. 百度面试题 - 多维逆序对

- 扩展到多维空间，统计多维逆序对
- 更复杂的分治策略和数据结构应用

20. 小米面试题 - 排序过程分析

- 分析排序算法执行过程中的各种统计量
- 与本题的归并排序过程统计思想一致

"""

文件: Code04\_CountRangeSum.cpp

// 区间和的个数, C++版

// 测试链接 : <https://leetcode.cn/problems/count-of-range-sum/>

/\*\*

\* 区间和的个数详解:

\*

\* 问题描述:

\* 给你一个整数数组 nums 以及两个整数 lower 和 upper 。

\* 求数组中，值位于范围 [lower, upper] （包含 lower 和 upper）之内的 区间和的个数 。

\* 区间和 S(i, j) 表示在 nums 中，位置从 i 到 j 的元素之和，包含 i 和 j (i ≤ j)。

\*

\* 示例:

\* 输入: nums = [-2, 5, -1], lower = -2, upper = 2

\* 输出: 3

\* 解释: 存在三个区间: [0,0]、[2,2] 和 [0,2] , 对应的区间和分别是: -2 、 -1 、 2 。

\*

\* 解法思路:

\* 1. 暴力解法: 计算所有可能区间的和, 检查是否在范围内, 时间复杂度  $O(N^3)$  或  $O(N^2)$  (使用前缀和优

化)

\* 2. 归并排序思想:

\* - 使用前缀和将问题转换为: 找到满足条件  $\text{prefixSum}[j] - \text{prefixSum}[i] \in [\text{lower}, \text{upper}]$  的  $(i, j)$  对数量

\* - 在归并排序过程中, 对于左半部分的每个前缀和  $\text{prefixSum}[i]$ , 在右半部分找到满足条件的  $\text{prefixSum}[j]$  数量

\* - 由于两部分各自有序, 可以使用双指针技巧优化查找过程

\*

\* 时间复杂度:  $O(N * \log N)$  - 归并排序的时间复杂度

\* 空间复杂度:  $O(N)$  - 前缀和数组和辅助数组的空间复杂度

\*/

```
#include <vector>
#include <iostream>
using namespace std;

class Solution {
private:
    // 归并排序, 在排序过程中统计满足条件的区间和个数
    int mergeSort(vector<long long>& arr, int left, int right, int lower, int upper) {
        if (left == right) {
            return 0;
        }

        // 计算中间位置, 避免整数溢出
        int mid = left + (right - left) / 2;

        // 递归处理左右两部分, 并累加统计结果
        int count = mergeSort(arr, left, mid, lower, upper) +
                    mergeSort(arr, mid + 1, right, lower, upper);

        // 统计跨越左右两部分的满足条件的区间和个数
        // 使用双指针技巧: l 和 r 分别指向右半部分中满足条件的起始和结束位置
        int l = mid + 1, r = mid + 1;
        for (int i = left; i <= mid; i++) {
            // 找到右半部分中第一个满足  $\text{arr}[j] - \text{arr}[i] \geq \text{lower}$  的位置 l
            while (l <= right && arr[l] - arr[i] < lower) {
                l++;
            }
            // 找到右半部分中第一个满足  $\text{arr}[j] - \text{arr}[i] > \text{upper}$  的位置 r
            while (r <= right && arr[r] - arr[i] <= upper) {
                r++;
            }
        }
    }
}
```

```

        // 区间[1, r-1]中的元素都满足条件，累加数量
        count += (r - 1);
    }

    // 合并两个有序数组
    vector<long long> help(right - left + 1);
    int i = 0;           // 辅助数组的索引
    int a = left, b = mid + 1; // 左半部分和右半部分的起始索引

    // 合并过程
    while (a <= mid && b <= right) {
        if (arr[a] <= arr[b]) {
            help[i++] = arr[a++];
        } else {
            help[i++] = arr[b++];
        }
    }

    // 处理剩余元素
    while (a <= mid) {
        help[i++] = arr[a++];
    }
    while (b <= right) {
        help[i++] = arr[b++];
    }

    // 将辅助数组中的元素复制回原数组
    for (i = 0; i < help.size(); i++) {
        arr[left + i] = help[i];
    }

    return count;
}

public:
    int countRangeSum(vector<int>& nums, int lower, int upper) {
        int n = nums.size();
        // 边界情况处理
        if (n == 0) {
            return 0;
        }

        // 计算前缀和数组，使用 long long 避免整数溢出

```

```

vector<long long> prefixSum(n + 1, 0);
for (int i = 0; i < n; i++) {
    prefixSum[i + 1] = prefixSum[i] + nums[i];
}

// 调用归并排序并统计结果
return mergeSort(prefixSum, 0, n, lower, upper);
}

};

// 主函数用于测试
int main() {
    Solution solution;

    // 测试用例 1: 基本情况
    vector<int> nums1 = {-2, 5, -1};
    int lower1 = -2, upper1 = 2;
    cout << "输入: nums = [-2, 5, -1], lower = -2, upper = 2" << endl;
    cout << "输出: " << solution.countRangeSum(nums1, lower1, upper1) << endl; // 预期输出: 3

    // 测试用例 2: 空数组
    vector<int> nums2 = {};
    int lower2 = 0, upper2 = 0;
    cout << "输入: nums = [], lower = 0, upper = 0" << endl;
    cout << "输出: " << solution.countRangeSum(nums2, lower2, upper2) << endl; // 预期输出: 0

    // 测试用例 3: 单元素数组
    vector<int> nums3 = {0};
    int lower3 = 0, upper3 = 0;
    cout << "输入: nums = [0], lower = 0, upper = 0" << endl;
    cout << "输出: " << solution.countRangeSum(nums3, lower3, upper3) << endl; // 预期输出: 1

    // 测试用例 4: 全为正数的数组
    vector<int> nums4 = {1, 2, 3, 4};
    int lower4 = 5, upper4 = 10;
    cout << "输入: nums = [1, 2, 3, 4], lower = 5, upper = 10" << endl;
    cout << "输出: " << solution.countRangeSum(nums4, lower4, upper4) << endl; // 预期输出: 4

    // 测试用例 5: 全为负数的数组
    vector<int> nums5 = {-4, -3, -2, -1};
    int lower5 = -6, upper5 = -2;
    cout << "输入: nums = [-4, -3, -2, -1], lower = -6, upper = -2" << endl;
    cout << "输出: " << solution.countRangeSum(nums5, lower5, upper5) << endl; // 预期输出: 4

```

```
// 测试用例 6: 大数值测试
vector<int> nums6 = {2147483647, -2147483647};
int lower6 = -2, upper6 = 2;
cout << "输入: nums = [2147483647, -2147483647], lower = -2, upper = 2" << endl;
cout << "输出: " << solution.countRangeSum(nums6, lower6, upper6) << endl; // 预期输出: 1

return 0;
}

/*
=====
C++语言特有关注事项
=====
```

### 1. 整数溢出问题:

- C++中的 int 类型通常是 32 位，范围为 $-2^{31}$  到  $2^{31}-1$
- 前缀和可能会超出 int 的范围，必须使用 long long 类型
- 示例代码中已使用 vector<long long> 存储前缀和，避免溢出
- 计算中间位置时使用 left + (right - left)/2 而非(left + right)/2，也是为了避免溢出

### 2. 内存管理:

- C++需要显式管理内存，但本例中使用 vector 自动管理内存
- 辅助数组 help 在每次 mergeSort 调用时创建，递归深度为  $O(\log N)$ ，总内存占用为  $O(N)$
- 对于超大规模数据，可以考虑使用全局辅助数组复用，减少内存分配开销

### 3. 递归深度限制:

- C++默认的栈空间有限（通常为几 MB）
- 对于非常大的数组（如百万级别），递归可能导致栈溢出
- 可以考虑实现非递归版本的归并排序

### 4. 性能优化:

- C++允许更细粒度的内存控制，可以预分配内存以提高性能
- 对于小规模子数组（如长度<10），可以使用插入排序替代归并排序
- 可以考虑使用 std::sort 作为优化，尽管它不符合题目的思想要求

### 5. 类设计:

- 封装为 Solution 类，符合 LeetCode 的常见接口设计
- 将 mergeSort 作为私有成员函数，countRangeSum 作为公有接口
- 这种设计提高了代码的封装性和可维护性

### 6. 异常安全:

- 代码使用 STL 容器，具有基本的异常安全性
- vector 的内存分配失败会抛出 std::bad\_alloc 异常

- 可以考虑添加更多的异常处理代码

\*/

/\*

## 工程化考量

### 1. 异常处理:

- 已添加对空数组的处理
- 可以添加对无效输入的检查，如 `lower > upper` 的情况
- 考虑添加断言来验证输入参数的有效性

### 2. 性能优化:

- 对于小规模数组（如长度<100），可以使用  $O(N^2)$  的暴力解法
- 可以使用内存池技术优化频繁的内存分配
- 可以考虑使用 SIMD 指令集加速归并过程

### 3. 测试策略:

- 已提供多种测试用例，覆盖常见场景
- 可以使用 Google Test 或 Catch2 等测试框架进行更系统的测试
- 建议添加性能测试和边界条件测试

### 4. 代码可读性:

- 使用了清晰的类和函数命名
- 添加了详细的注释说明算法思路和关键步骤
- 遵循了 C++ 的编码规范

### 5. 可扩展性:

- 可以将归并排序部分抽象为通用算法
- 可以支持自定义的比较函数和统计逻辑
- 考虑扩展支持其他类型的数组（如 `float`、`double`）

### 6. 并行处理:

- 对于超大规模数据，可以考虑使用 C++17 的并行算法
- 可以使用 OpenMP 或 TBB 库实现并行化归并排序
- 注意并行处理的同步开销

### 7. 边界情况处理:

- 空数组：返回 0
- 单元素数组：直接检查该元素是否在范围内
- 全部为相同元素：优化计算逻辑
- 大数值：使用 `long long` 类型避免溢出

8. 跨平台兼容性:
  - 代码使用标准 C++, 具有良好的跨平台兼容性
  - 避免使用平台特定的扩展
  - 注意不同编译器的实现差异

\*/

/\*

---

## 相关题目与平台信息

---

### 1. LeetCode 327. Count of Range Sum

- 题目链接: <https://leetcode.cn/problems/count-of-range-sum/>
- 难度等级: 困难
- 标签: 归并排序、前缀和、二分查找

### 2. LeetCode 493. 翻转对 (Reverse Pairs)

- 题目链接: <https://leetcode.cn/problems/reverse-pairs/>
- 难度等级: 困难
- 解题思路: 同样使用归并排序的过程统计满足条件的对

### 3. LeetCode 315. 计算右侧小于当前元素的个数 (Count of Smaller Numbers After Self)

- 题目链接: <https://leetcode.cn/problems/count-of-smaller-numbers-after-self/>
- 难度等级: 困难
- 解题思路: 类似的归并排序框架, 统计右侧较小的元素个数

### 4. 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/>
- 难度等级: 困难
- 解题思路: 归并排序过程中统计逆序对数量

### 5. 牛客网 - 计算数组的小和

- 题目链接: <https://www.nowcoder.com/practice/edfe05a1d45c4ea89101d936cac32469>
- 解题思路: 归并排序过程中计算小和

### 6. LintCode 1497. 区间和的个数

- 题目链接: <https://www.lintcode.com/problem/1497/>
- 与 LeetCode 327 题相同

### 7. POJ 2299 - Ultra-QuickSort

- 题目链接: <http://poj.org/problem?id=2299>
- 计算逆序对数量, 与本题使用类似的归并排序框架

### 8. SPOJ - INVCNT

- 题目链接: <https://www.spoj.com/problems/INVCNT/>
  - 逆序对计数问题, 可使用归并排序解决
9. HDU 4911 - Inversion
- 题目链接: <http://acm.hdu.edu.cn/showproblem.php?pid=4911>
  - 扩展的逆序对问题, 有不同的约束条件
10. 字节跳动面试题 - 前缀和区间统计
- 实际面试中可能会对本题进行变体, 如不同的范围条件或附加约束
11. 微软面试题 - 子数组和范围查询
- 可能需要处理多次查询操作
  - 考察前缀和和数据结构的应用
12. Google 面试题 - 二维区间和计数
- 将问题扩展到二维数组
  - 更复杂的前缀和和排序应用
13. 腾讯面试题 - 最大区间和个数
- 寻找满足特定和的最大区间数
  - 与本题有类似的前缀和思想
14. 阿里巴巴面试题 - 区间和统计优化
- 对本题进行性能优化, 要求处理超大规模数据
  - 考察并行计算和算法优化能力

\*/

=====

文件: Code04\_CountRangeSum.java

=====

```
package class022;

// 区间和的个数
// 测试链接 : https://leetcode.cn/problems/count-of-range-sum/
import java.util.*;

/**
 * 区间和的个数详解:
 *
 * 问题描述:
 * 给你一个整数数组 nums 以及两个整数 lower 和 upper 。
 * 求数组中, 值位于范围 [lower, upper] (包含 lower 和 upper) 之内的 区间和的个数 。
```

\* 区间和  $S(i, j)$  表示在  $\text{nums}$  中，位置从  $i$  到  $j$  的元素之和，包含  $i$  和  $j$  ( $i \leq j$ )。

\*

\* 示例：

\* 输入:  $\text{nums} = [-2, 5, -1]$ ,  $\text{lower} = -2$ ,  $\text{upper} = 2$

\* 输出: 3

\* 解释: 存在三个区间:  $[0, 0]$ 、 $[2, 2]$  和  $[0, 2]$ ，对应的区间和分别是:  $-2$ 、 $-1$ 、 $2$ 。

\*

\* 解法思路:

\* 1. 暴力解法: 计算所有可能区间的和, 检查是否在范围内, 时间复杂度  $O(N^3)$  或  $O(N^2)$  (使用前缀和优化)

\* 2. 归并排序思想:

\* - 使用前缀和将问题转换为: 找到满足条件  $\text{prefixSum}[j] - \text{prefixSum}[i] \in [\text{lower}, \text{upper}]$  的  $(i, j)$  对数量

\* - 在归并排序过程中, 对于左半部分的每个前缀和  $\text{prefixSum}[i]$ , 在右半部分找到满足条件的  $\text{prefixSum}[j]$  数量

\* - 由于两部分各自有序, 可以使用双指针技巧优化查找过程

\*

\* 时间复杂度:  $O(N * \log N)$  - 归并排序的时间复杂度

\* 空间复杂度:  $O(N)$  - 前缀和数组和辅助数组的空间复杂度

\*

\* 相关题目:

\* 1. LeetCode 493. 翻转对

\* 2. LeetCode 315. 计算右侧小于当前元素的个数

\* 3. 剑指 Offer 51. 数组中的逆序对

\* 4. 牛客网 - 计算数组的小和

\*/

```
public class Code04_CountRangeSum {
```

```
    public static int countRangeSum(int[] nums, int lower, int upper) {
        int n = nums.length;
        // 计算前缀和数组
        long[] prefixSum = new long[n + 1];
        for (int i = 0; i < n; i++) {
            prefixSum[i + 1] = prefixSum[i] + nums[i];
        }

        return mergeSort(prefixSum, 0, n, lower, upper);
    }
```

```
// 归并排序, 在排序过程中统计满足条件的区间和个数
```

```
    public static int mergeSort(long[] arr, int left, int right, int lower, int upper) {
        if (left == right) {
            return 0;
        }
```

```

}

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

// 统计跨越左右两部分的满足条件的区间和个数
int l = mid + 1, r = mid + 1;
for (int i = left; i <= mid; i++) {
    // 找到右半部分中第一个满足 arr[j] - arr[i] >= lower 的位置 l
    while (l <= right && arr[l] - arr[i] < lower) {
        l++;
    }
    // 找到右半部分中第一个满足 arr[j] - arr[i] > upper 的位置 r
    while (r <= right && arr[r] - arr[i] <= upper) {
        r++;
    }
    // 区间[l, r-1]中的元素都满足条件
    count += (r - l);
}

// 合并两个有序数组
long[] help = new long[right - left + 1];
int i = 0;
int a = left, b = mid + 1;
while (a <= mid && b <= right) {
    help[i++] = arr[a] <= arr[b] ? arr[a++] : arr[b++];
}
while (a <= mid) {
    help[i++] = arr[a++];
}
while (b <= right) {
    help[i++] = arr[b++];
}
for (i = 0; i < help.length; i++) {
    arr[left + i] = help[i];
}

return count;
}

/**
 * 主方法，用于测试

```

```
/*
public static void main(String[] args) {
    // 测试用例 1: 基本情况
    int[] test1 = {-2, 5, -1};
    int lower1 = -2, upper1 = 2;
    System.out.println("输入: nums = " + Arrays.toString(test1) + ", lower = " + lower1 + ", "
upper = " + upper1);
    System.out.println("输出: " + countRangeSum(test1, lower1, upper1)); // 预期输出: 3

    // 测试用例 2: 空数组
    int[] test2 = {};
    int lower2 = 0, upper2 = 0;
    System.out.println("输入: nums = " + Arrays.toString(test2) + ", lower = " + lower2 + ", "
upper = " + upper2);
    System.out.println("输出: " + countRangeSum(test2, lower2, upper2)); // 预期输出: 0

    // 测试用例 3: 全为正数的数组
    int[] test3 = {1, 2, 3, 4};
    int lower3 = 5, upper3 = 10;
    System.out.println("输入: nums = " + Arrays.toString(test3) + ", lower = " + lower3 + ", "
upper = " + upper3);
    System.out.println("输出: " + countRangeSum(test3, lower3, upper3)); // 预期输出: 4

    // 测试用例 4: 全为负数的数组
    int[] test4 = {-4, -3, -2, -1};
    int lower4 = -6, upper4 = -2;
    System.out.println("输入: nums = " + Arrays.toString(test4) + ", lower = " + lower4 + ", "
upper = " + upper4);
    System.out.println("输出: " + countRangeSum(test4, lower4, upper4)); // 预期输出: 4

    // 测试用例 5: 大数值测试
    int[] test5 = {Integer.MAX_VALUE, -Integer.MAX_VALUE};
    int lower5 = -2, upper5 = 2;
    System.out.println("输入: nums = [MAX, -MAX], lower = " + lower5 + ", upper = " +
upper5);
    System.out.println("输出: " + countRangeSum(test5, lower5, upper5)); // 预期输出: 1
}

/*
 * =====
 * Java 语言特有关注事项
 * =====
 * 1. 数据类型溢出问题:

```

```
*      - 前缀和可能会超出 int 的范围，因此使用 long 类型存储前缀和
*      - 当数组长度较大且元素值较大时，普通 int 类型会导致溢出错误
*      - 例如：当处理包含 Integer.MAX_VALUE 或 Integer.MIN_VALUE 的数组时
*
* 2. 递归深度限制：
*      - Java 的默认栈深度约为 1000–2000 层
*      - 对于极大数组（如长度>1e5），可能导致栈溢出
*      - 可以通过 JVM 参数-Xss 调整栈大小
*
* 3. 内存分配：
*      - help 数组在每次递归调用时都会重新创建，这会增加 GC 压力
*      - 对于频繁调用的场景，可以考虑使用静态辅助数组复用内存
*
* 4. 线程安全性：
*      - 当前实现是无状态的，多线程调用 countRangeSum 方法是线程安全的
*      - 但 mergeSort 方法的局部变量不会被多线程共享，因此整体是线程安全的
*
* 5. 装箱和拆箱：
*      - 本实现避免了自动装箱/拆箱操作，提高了性能
*      - 直接使用基本类型进行计算，减少了对象创建
*/

```

```
/*
* =====
* 工程化考量
* =====
* 1. 异常处理：
*      - 可以添加对 null 输入数组的检查
*      - 可以处理极端情况下的数组长度（如 Integer.MAX_VALUE）
*      - 对于 lower > upper 的情况，可以直接返回 0 并给出警告
*
* 2. 性能优化：
*      - 对于小规模数组（如长度<100），可以考虑使用暴力解法 ( $O(N^2)$ )，常数项更小
*      - 可以使用静态辅助数组避免重复创建 help 数组
*      - 考虑使用迭代版本的归并排序避免栈溢出
*
* 3. 测试策略：
*      - 已提供多种测试用例，包括基本情况、边界情况和特殊输入
*      - 建议补充压力测试，验证算法在大规模数据下的性能
*      - 可以使用参数化测试覆盖更多的输入组合
*
* 4. 可扩展性：
*      - 该算法框架可以扩展应用于其他类似的区间查询问题

```

- \*    - 可以将双指针部分抽象为独立函数，提高代码复用性
  - \*    - 考虑添加缓存机制，避免对相同输入的重复计算
  - \*
  - \* 5. 代码可读性：
    - \*    - 方法命名清晰（countRangeSum、mergeSort）
    - \*    - 变量命名直观（prefixSum、help、count 等）
    - \*    - 添加了详细的注释解释算法思路和关键步骤
  - \*
  - \* 6. 并行处理：
    - \*    - 对于超大规模数据集，可以考虑使用 Fork/Join 框架实现并行归并排序
    - \*    - 但需要注意任务划分的粒度，避免过多的线程创建开销
  - \*
  - \* 7. 边界情况处理：
    - \*    - 空数组：返回 0（没有区间）
    - \*    - lower > upper：返回 0（没有符合条件的区间）
    - \*    - 单元素数组：检查该元素是否在 [lower, upper] 范围内
    - \*    - 大数值：使用 long 类型避免溢出
  - \*/
- /\*
- \* =====
- \* 相关题目与平台信息
- \* =====
- \* 1. LeetCode 327. Count of Range Sum
    - \*    - 题目链接: <https://leetcode.cn/problems/count-of-range-sum/>
    - \*    - 难度等级: 困难
    - \*    - 标签: 归并排序、前缀和、二分查找
  - \*
  - \* 2. LeetCode 493. 翻转对 (Reverse Pairs)
    - \*    - 题目链接: <https://leetcode.cn/problems/reverse-pairs/>
    - \*    - 难度等级: 困难
    - \*    - 解题思路: 同样使用归并排序的过程统计满足条件的对
  - \*
  - \* 3. LeetCode 315. 计算右侧小于当前元素的个数 (Count of Smaller Numbers After Self)
    - \*    - 题目链接: <https://leetcode.cn/problems/count-of-smaller-numbers-after-self/>
    - \*    - 难度等级: 困难
    - \*    - 解题思路: 类似的归并排序框架，统计右侧较小的元素个数
  - \*
  - \* 4. 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/>
    - \*    - 难度等级: 困难
    - \*    - 解题思路: 归并排序过程中统计逆序对数量
  - \*

```
* 5. 牛客网 - 计算数组的小和
*   - 题目链接: https://www.nowcoder.com/practice/edfe05a1d45c4ea89101d936cac32469
*   - 解题思路: 归并排序过程中计算小和
*
* 6. LintCode 1497. 区间和的个数
*   - 题目链接: https://www.lintcode.com/problem/1497/
*   - 与 LeetCode 327 题相同
*
* 7. 华为机试 - 区间和统计
*   - 类似问题, 但可能有不同的输入输出格式要求
*
* 8. 字节跳动面试题 - 前缀和区间统计
*   - 实际面试中可能会对本题进行变体, 如不同的范围条件或附加约束
*/
}
```

=====

文件: Code04\_CountRangeSum.py

=====

```
# 区间和的个数, Python 版
# 测试链接 : https://leetcode.cn/problems/count-of-range-sum/
```

"""

区间和的个数详解:

问题描述:

给你一个整数数组 `nums` 以及两个整数 `lower` 和 `upper`。

求数组中, 值位于范围  $[lower, upper]$  (包含 `lower` 和 `upper`) 之内的 区间和的个数。

区间和  $S(i, j)$  表示在 `nums` 中, 位置从  $i$  到  $j$  的元素之和, 包含  $i$  和  $j$  ( $i \leq j$ )。

示例:

输入: `nums = [-2, 5, -1]`, `lower = -2`, `upper = 2`

输出: 3

解释: 存在三个区间:  $[0, 0]$ 、 $[2, 2]$  和  $[0, 2]$  , 对应的区间和分别是:  $-2$ 、 $-1$ 、 $2$ 。

解法思路:

1. 暴力解法: 计算所有可能区间的和, 检查是否在范围内, 时间复杂度  $O(N^3)$  或  $O(N^2)$  (使用前缀和优化)

2. 归并排序思想:

- 使用前缀和将问题转换为: 找到满足条件  $\text{prefixSum}[j] - \text{prefixSum}[i] \in [lower, upper]$  的  $(i, j)$  对数量

- 在归并排序过程中, 对于左半部分的每个前缀和 `prefixSum[i]`, 在右半部分找到满足条件的 `prefixSum[j]` 数量

- 由于两部分各自有序，可以使用双指针技巧优化查找过程

时间复杂度:  $O(N * \log N)$  - 归并排序的时间复杂度

空间复杂度:  $O(N)$  - 前缀和数组和辅助数组的空间复杂度

相关题目:

1. LeetCode 493. 翻转对
2. LeetCode 315. 计算右侧小于当前元素的个数
3. 剑指 Offer 51. 数组中的逆序对
4. 牛客网 - 计算数组的小和

"""

```
def count_range_sum(nums, lower, upper):
```

```
    """
```

计算数组中区间和在指定范围内的个数

Args:

nums: 输入数组

lower: 范围下界

upper: 范围上界

Returns:

int: 区间和在指定范围内的个数

```
    """
```

```
if not nums:
```

```
    return 0
```

```
n = len(nums)
```

```
# 计算前缀和数组
```

```
prefix_sum = [0] * (n + 1)
```

```
for i in range(n):
```

```
    prefix_sum[i + 1] = prefix_sum[i] + nums[i]
```

```
def merge_sort(left, right):
```

```
    """
```

归并排序，在排序过程中统计满足条件的区间和个数

Args:

left: 左边界

right: 右边界

Returns:

int: 区间 [left, right] 中满足条件的区间和个数

```

"""
if left == right:
    return 0

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

# 统计跨越左右两部分的满足条件的区间和个数
l_ptr = mid + 1
r_ptr = mid + 1
for i in range(left, mid + 1):
    # 找到右半部分中第一个满足 prefix_sum[j] - prefix_sum[i] >= lower 的位置
    while l_ptr <= right and prefix_sum[l_ptr] - prefix_sum[i] < lower:
        l_ptr += 1
    # 找到右半部分中第一个满足 prefix_sum[j] - prefix_sum[i] > upper 的位置
    while r_ptr <= right and prefix_sum[r_ptr] - prefix_sum[i] <= upper:
        r_ptr += 1
    # 区间[l_ptr, r_ptr-1]中的元素都满足条件
    count += (r_ptr - l_ptr)

# 合并两个有序数组
help_arr = [0] * (right - left + 1)
i = 0
a, b = left, mid + 1
while a <= mid and b <= right:
    if prefix_sum[a] <= prefix_sum[b]:
        help_arr[i] = prefix_sum[a]
        a += 1
    else:
        help_arr[i] = prefix_sum[b]
        b += 1
    i += 1

while a <= mid:
    help_arr[i] = prefix_sum[a]
    a += 1
    i += 1

while b <= right:
    help_arr[i] = prefix_sum[b]
    b += 1
    i += 1

```

```
for i in range(len(help_arr)):
    prefix_sum[left + i] = help_arr[i]

return count

return merge_sort(0, n)

# 测试代码
if __name__ == "__main__":
    # 测试用例 1: 基本情况
    test_nums1 = [-2, 5, -1]
    lower1, upper1 = -2, 2
    print(f"输入: nums = {test_nums1}, lower = {lower1}, upper = {upper1}")
    print(f"输出: {count_range_sum(test_nums1, lower1, upper1)}") # 预期输出: 3

    # 测试用例 2: 空数组
    test_nums2 = []
    lower2, upper2 = 0, 0
    print(f"输入: nums = {test_nums2}, lower = {lower2}, upper = {upper2}")
    print(f"输出: {count_range_sum(test_nums2, lower2, upper2)}") # 预期输出: 0

    # 测试用例 3: 单元素数组
    test_nums3 = [0]
    lower3, upper3 = 0, 0
    print(f"输入: nums = {test_nums3}, lower = {lower3}, upper = {upper3}")
    print(f"输出: {count_range_sum(test_nums3, lower3, upper3)}") # 预期输出: 1

    # 测试用例 4: 全为正数的数组
    test_nums4 = [1, 2, 3, 4]
    lower4, upper4 = 5, 10
    print(f"输入: nums = {test_nums4}, lower = {lower4}, upper = {upper4}")
    print(f"输出: {count_range_sum(test_nums4, lower4, upper4)}") # 预期输出: 4

    # 测试用例 5: 全为负数的数组
    test_nums5 = [-4, -3, -2, -1]
    lower5, upper5 = -6, -2
    print(f"输入: nums = {test_nums5}, lower = {lower5}, upper = {upper5}")
    print(f"输出: {count_range_sum(test_nums5, lower5, upper5)}") # 预期输出: 4

    # 测试用例 6: 大数值测试
    test_nums6 = [2147483647, -2147483647]
    lower6, upper6 = -2, 2
```

```
print(f"输入: nums = [2147483647, -2147483647], lower = {lower6}, upper = {upper6}")
print(f"输出: {count_range_sum(test_nums6, lower6, upper6)}") # 预期输出: 1
```

"""

=====

Python 语言特有关注事项

=====

1. 整数类型:

- Python 的整数没有固定大小限制，不会出现整数溢出问题
- 与 Java/C++不同，无需显式使用 long 类型来存储大数值
- 但仍需注意浮点数精度问题（本题不涉及）

2. 递归深度限制:

- Python 的默认递归深度限制约为 1000
- 对于非常大的数组，可能会抛出 RecursionError
- 可以通过 sys.setrecursionlimit() 调整递归深度限制
- 也可以考虑实现非递归版本的归并排序

3. 列表操作:

- Python 列表的动态扩容机制可能带来性能开销
- 预先分配列表大小可以提高性能（如代码中已做的那样）
- 列表切片操作会创建新的列表，应尽量避免在递归中频繁使用

4. 闭包特性:

- merge\_sort 函数作为嵌套函数可以直接访问外部函数的变量
- 这种闭包结构使代码更简洁，但要注意变量作用域

5. 内存管理:

- Python 的垃圾回收机制自动管理内存，但递归调用可能导致内存消耗增加
- 对于非常大的数据集，需要考虑内存使用效率

6. 性能优化:

- Python 中的递归实现比迭代实现慢
- 可以考虑使用更底层的优化，如 NumPy 数组操作
- 对于小规模数据，可以使用更简单的暴力解法

"""

"""

=====

工程化考量

=====

1. 异常处理:

- 已添加对空数组的处理

- 可以添加对无效输入的检查，如  $\text{lower} > \text{upper}$  的情况
- 可以考虑添加类型检查，确保输入参数类型正确

## 2. 性能优化：

- 对于小规模数组（如长度<100），可以使用  $O(N^2)$  的暴力解法
- 可以使用 `itertools` 模块中的工具函数简化实现
- 考虑使用记忆化或缓存机制优化重复计算

## 3. 测试策略：

- 已提供多种测试用例，覆盖常见场景
- 可以使用 `unittest` 或 `pytest` 框架进行更系统的测试
- 建议添加随机测试用例和边界值测试

## 4. 代码可读性：

- 使用了清晰的函数和变量命名
- 添加了详细的注释说明算法思路
- 函数参数和返回值都有明确的文档字符串

## 5. 可扩展性：

- 可以将核心逻辑抽象为可配置的类
- 可以扩展支持流数据处理
- 考虑添加进度显示，用于处理大规模数据

## 6. 并行处理：

- 对于超大规模数据，可以考虑使用 `multiprocessing` 模块并行化处理
- 但需要注意 Python 的 GIL 对多线程的限制
- 可以使用 `concurrent.futures` 库简化并行任务管理

## 7. 边界情况处理：

- 空数组：返回 0
- 单元素数组：检查该元素是否在范围内
- 全部为相同元素：优化计算逻辑
- 大数值：Python 无需特殊处理

"""

"""

## 相关题目与平台信息

### 1. LeetCode 327. Count of Range Sum

- 题目链接：<https://leetcode.cn/problems/count-of-range-sum/>
- 难度等级：困难
- 标签：归并排序、前缀和、二分查找

2. LeetCode 493. 翻转对 (Reverse Pairs)
  - 题目链接: <https://leetcode.cn/problems/reverse-pairs/>
  - 难度等级: 困难
  - 解题思路: 同样使用归并排序的过程统计满足条件的对
3. LeetCode 315. 计算右侧小于当前元素的个数 (Count of Smaller Numbers After Self)
  - 题目链接: <https://leetcode.cn/problems/count-of-smaller-numbers-after-self/>
  - 难度等级: 困难
  - 解题思路: 类似的归并排序框架, 统计右侧较小的元素个数
4. 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/>
  - 难度等级: 困难
  - 解题思路: 归并排序过程中统计逆序对数量
5. 牛客网 - 计算数组的小和
  - 题目链接: <https://www.nowcoder.com/practice/edfe05a1d45c4ea89101d936cac32469>
  - 解题思路: 归并排序过程中计算小和
6. LintCode 1497. 区间和的个数
  - 题目链接: <https://www.lintcode.com/problem/1497/>
  - 与 LeetCode 327 题相同
7. HackerRank - Sum of Intervals
  - 类似问题, 但可能有不同的约束条件
  - 可能需要处理重叠区间
8. CodeChef - Range Sum Query
  - 可能需要多次查询不同区间的和
  - 可能需要预处理优化
9. 华为机试 - 区间和统计
  - 类似问题, 但可能有不同的输入输出格式要求
10. 字节跳动面试题 - 前缀和区间统计
  - 实际面试中可能会对本题进行变体, 如不同的范围条件或附加约束
11. POJ 2299 - Ultra-QuickSort
  - 题目链接: <http://poj.org/problem?id=2299>
  - 计算逆序对数量, 与本题使用类似的归并排序框架
12. SPOJ - INVCNT

- 题目链接: <https://www.spoj.com/problems/INVCNT/>
- 逆序对计数问题, 可使用归并排序解决

### 13. 计蒜客 - 区间和查询

- 可能包含多个查询操作
- 考察前缀和和二分查找的应用

### 14. AtCoder - Range Sum Query

- 类似的区间查询问题, 可能有不同的约束条件

### 15. USACO - Cow Sorting

- 问题涉及逆序对的计算, 与本题思路相关

"""

文件: Code05\_ReversePairsLCR170.cpp

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <climits>
#include <iostream>
using namespace std;

/***
 * =====
 * 题目 5: 剑指 Offer 51 - 数组中的逆序对 (LCR 170) - C++版
 * =====
 *
 * 题目来源: 剑指 Offer / LCR 170
 * 题目链接: https://leetcode.cn/problems/shu-zu-zhong-de-ni-xu-dui-lcof/
 * 难度级别: 困难
 *
 * 问题描述:
 * 在数组中的两个数字, 如果前面一个数字大于后面的数字, 则这两个数字组成一个逆序对。
 * 输入一个数组, 求出这个数组中的逆序对的总数。
 *
 * 示例输入输出:
 * 输入: [7, 5, 6, 4]
 * 输出: 5
 * 解释: 逆序对有(7, 5), (7, 6), (7, 4), (5, 4), (6, 4)
 *
```

\* =====

\* 核心算法思想：归并排序分治统计

\* =====

\*

\* 方法 1：暴力解法（不推荐）

\* - 思路：双重循环检查每一对  $(i, j)$  是否满足  $i < j$  且  $\text{nums}[i] > \text{nums}[j]$

\* - 时间复杂度： $O(N^2)$  - 双重循环

\* - 空间复杂度： $O(1)$  - 不需要额外空间

\* - 问题：数据量大时超时

\*

\* 方法 2：归并排序思想（最优解）★★★★★

\* - 核心洞察：利用归并排序过程统计跨越左右两个子数组的逆序对

\* - 先统计左半部分内部的逆序对

\* - 再统计右半部分内部的逆序对

\* - 最后统计跨越左右两部分的逆序对（关键步骤）

\*

\* - 统计跨区间逆序对的优化方法：

\* - 在合并前，对每个左区间元素  $\text{nums}[i]$ ，找到右区间中满足  $\text{nums}[i] > \text{nums}[j]$  的元素个数

\* - 利用双指针技巧：由于左右子数组已排序，可以线性扫描而不需要嵌套循环

\* - 这一步的时间复杂度为  $O(n)$  而非  $O(n^2)$

\*

\* - 归并排序过程：

\* 1. 分治：将数组不断二分，直到只有一个元素

\* 2. 统计：统计三种类型的逆序对

\* 3. 合并：将两个有序子数组合并

\*

\* - 时间复杂度详细计算：

\*  $T(n) = 2T(n/2) + O(n)$  [Master 定理 case 2]

\*  $= O(n \log n)$

\* - 递归深度： $\log n$

\* - 每层合并与统计： $O(n)$

\*

\* - 空间复杂度详细计算：

\*  $S(n) = O(n) + O(\log n)$

\* -  $O(n)$ ：辅助数组 help

\* -  $O(\log n)$ ：递归调用栈

\* 总计： $O(n)$

\*

\* - 是否最优解：★ 是 ★

\* 理由：基于比较的算法下界为  $O(n \log n)$ ，本算法已达到最优

\*

\* =====

\* 相关题目列表（同类算法）

\* =====

\* 1. LeetCode 493 - 翻转对  
\* <https://leetcode.cn/problems/reverse-pairs/>  
\* 问题: 统计满足  $\text{nums}[i] > 2 * \text{nums}[j]$  且  $i < j$  的对的数量  
\* 解法: 归并排序过程中使用双指针统计跨越左右区间的翻转对  
\*

\* 2. LeetCode 315 - 计算右侧小于当前元素的个数  
\* <https://leetcode.cn/problems/count-of-smaller-numbers-after-self/>  
\* 问题: 统计每个元素右侧比它小的元素个数  
\* 解法: 归并排序过程中记录元素原始索引, 统计右侧小于当前元素的数量  
\*

\* 3. LeetCode 327 - 区间和的个数  
\* <https://leetcode.cn/problems/count-of-range-sum/>  
\* 问题: 统计区间和在 [lower, upper] 范围内的区间个数  
\* 解法: 前缀和+归并排序, 统计满足条件的前缀和对  
\*

\* 4. POJ 2299 - Ultra-QuickSort  
\* <http://poj.org/problem?id=2299>  
\* 问题: 计算将数组排序所需的最小交换次数 (即逆序对数量)  
\* 解法: 归并排序统计逆序对  
\*

\* 5. HDU 1394 - Minimum Inversion Number  
\* <http://acm.hdu.edu.cn/showproblem.php?pid=1394>  
\* 问题: 将数组循环左移, 求所有可能排列中的最小逆序对数量  
\* 解法: 归并排序+逆序对性质分析  
\*

\* 6. 洛谷 P1908 - 逆序对  
\* <https://www.luogu.com.cn/problem/P1908>  
\* 问题: 统计数组中逆序对的总数  
\* 解法: 归并排序统计逆序对  
\*

\* 7. HackerRank - Merge Sort: Counting Inversions  
\* <https://www.hackerrank.com/challenges/merge-sort/problem>  
\* 问题: 统计逆序对数量  
\* 解法: 归并排序统计逆序对  
\*

\* 8. SPOJ - INVCNT  
\* <https://www.spoj.com/problems/INVCNT/>  
\* 问题: 统计逆序对数量  
\* 解法: 归并排序统计逆序对  
\*

\* 9. CodeChef - INVCNT  
\* <https://www.codechef.com/problems/INVCNT>

```
* 问题：统计逆序对数量
* 解法：归并排序或树状数组
*
* 这些题目虽然具体形式不同，但核心思想都是利用归并排序的分治特性，在合并过程中高效统计满足特定条件的元素对数量。
*/

```

```
const int MAXN = 50001;
int help[MAXN];

/***
 * 计算数组中逆序对的数量
 *
 * @param nums 输入数组
 * @return 逆序对的数量
 */
int reversePairs(vector<int>& nums) {
    if (nums.empty() || nums.size() < 2) {
        return 0;
    }

    // 创建副本避免修改原数组
    vector<int> numsCopy = nums;
    return mergeSort(numsCopy, 0, numsCopy.size() - 1);
}

/***
 * 归并排序，在排序过程中统计逆序对数量
 *
 * @param arr 输入数组
 * @param l 左边界
 * @param r 右边界
 * @return 区间[l, r]中的逆序对数量
 */
int mergeSort(vector<int>& arr, int l, int r) {
    if (l == r) {
        return 0;
    }

    int m = (l + r) / 2;
    return mergeSort(arr, l, m) + mergeSort(arr, m + 1, r) + merge(arr, l, m, r);
}
```

```

/***
 * 合并两个有序数组，并统计跨越左右两部分的逆序对
 *
 * @param arr 输入数组
 * @param l 左边界
 * @param m 中点
 * @param r 右边界
 * @return 跨越左右两部分的逆序对数量
 */
int merge(vector<int>& arr, int l, int m, int r) {
    // 统计逆序对数量
    int ans = 0;
    int j = m + 1;
    for (int i = l; i <= m; i++) {
        // 找到右半部分中第一个不满足 arr[i] > arr[j] 的位置
        while (j <= r && arr[i] > arr[j]) {
            j++;
        }
        // j 之前的元素都满足条件，即与 arr[i] 构成逆序对
        ans += (j - m - 1);
    }

    // 正常合并两个有序数组
    int i = l;
    int a = l, 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];
    }

    return ans;
}

/***
 * 主函数，用于测试

```

```
/*
int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    // 测试用例 1: 基本情况
    vector<int> test1 = {7, 5, 6, 4};
    cout << "输入: [7,5,6,4]" << endl;
    cout << "输出: " << reversePairs(test1) << endl; // 预期输出: 5

    // 测试用例 2: 空数组
    vector<int> test2 = {};
    cout << "输入: []" << endl;
    cout << "输出: " << reversePairs(test2) << endl; // 预期输出: 0

    // 测试用例 3: 单元素数组
    vector<int> test3 = {1};
    cout << "输入: [1]" << endl;
    cout << "输出: " << reversePairs(test3) << endl; // 预期输出: 0

    // 测试用例 4: 升序数组
    vector<int> test4 = {1, 2, 3, 4};
    cout << "输入: [1,2,3,4]" << endl;
    cout << "输出: " << reversePairs(test4) << endl; // 预期输出: 0

    // 测试用例 5: 降序数组
    vector<int> test5 = {4, 3, 2, 1};
    cout << "输入: [4,3,2,1]" << endl;
    cout << "输出: " << reversePairs(test5) << endl; // 预期输出: 6

    // 测试用例 6: 重复元素
    vector<int> test6 = {2, 2, 2, 2};
    cout << "输入: [2,2,2,2]" << endl;
    cout << "输出: " << reversePairs(test6) << endl; // 预期输出: 0

    // 测试用例 7: 包含负数
    vector<int> test7 = {-1, -2, 3, -4};
    cout << "输入: [-1,-2,3,-4]" << endl;
    cout << "输出: " << reversePairs(test7) << endl; // 预期输出: 4

    return 0;
}
```

```
/*
```

```
=====
```

## C++语言特有关注事项

```
=====
```

### 1. 数据类型溢出问题:

- 逆序对数量可能超过 int 范围, 但题目约束范围内不会溢出
- 对于极端情况, 可以使用 long long 类型存储结果
- 当数组元素较多且值较大时, int 类型可能会溢出

### 2. 内存管理:

- 使用全局数组而非 vector, 避免频繁动态分配内存
- MAXN 设为 50001, 足够处理大部分测试用例
- 静态数组在栈上分配, 访问速度比堆分配更快

### 3. 递归深度控制:

- 归并排序的递归深度为  $\log_2(n)$ , 对于 n=50000, 深度约为 16 层
- 不会超过 C++默认的栈大小限制
- 对于极端大数据, 可以考虑非递归实现

### 4. 输入输出优化:

- `ios::sync_with_stdio(false);` 关闭同步, 加速 `cin/cout`
- `cin.tie(nullptr);` 解绑 `cin` 和 `cout`, 减少刷新次数
- 使用 \n 代替 endl, 避免不必要的缓冲区刷新

### 5. 位运算优化:

- 使用 `(l + r) >> 1` 代替 `(l + r) / 2`, 提高运算效率
- 注意当 l 和 r 都很大时, `(l + r)` 可能导致溢出, 应改为 `1 + ((r - 1) >> 1)`

### 6. 代码优化技巧:

- 在 merge 函数中先统计逆序对再排序, 逻辑更清晰
- 合并时使用三目运算符使代码更简洁

### 7. 编译优化选项:

- 可以添加 `-O2` 编译选项获得更好的性能
- 对于某些编译器, `-march=native` 可以利用 CPU 特性进一步优化

### 8. 多线程考虑:

- 当前实现不是线程安全的, 因为使用了全局变量
- 多线程环境下应使用局部变量或添加同步机制

### 9. 边界条件处理:

- 对空数组、单元素数组有正确的边界检查

- 递归终止条件明确

## 10. 异常处理:

- C++中可以添加 try-catch 块处理可能的异常
- 对数组索引越界等情况进行检查

\*/

=====

文件: Code05\_ReversePairsLCR170.java

=====

```
package class022;

import java.util.*;

/**
 * =====
 * 题目 5: 剑指 Offer 51 - 数组中的逆序对 (LCR 170)
 * =====
 *
 * 题目来源: 剑指 Offer / LCR 170
 * 题目链接: https://leetcode.cn/problems/shu-zu-zhong-de-ni-xu-dui-lcof/
 * 难度级别: 困难
 *
 * 问题描述:
 * 在数组中的两个数字, 如果前面一个数字大于后面的数字, 则这两个数字组成一个逆序对。
 * 输入一个数组, 求出这个数组中的逆序对的总数。
 *
 * 示例输入输出:
 * 输入: [7, 5, 6, 4]
 * 输出: 5
 * 解释: 逆序对有(7, 5), (7, 6), (7, 4), (5, 4), (6, 4)
 *
 * 核心算法思想: 归并排序分治统计
 *
 *
 * 方法 1: 暴力解法 (不推荐)
 * - 思路: 双重循环检查每一对 (i, j) 是否满足 i < j 且 nums[i] > nums[j]
 * - 时间复杂度: O(N^2) - 双重循环
 * - 空间复杂度: O(1) - 不需要额外空间
 * - 问题: 数据量大时超时
 *
```

\* 方法 2: 归并排序思想 (最优解) ★★★★★

\* - 核心洞察: 利用归并排序过程统计跨越左右两个子数组的逆序对

\* - 先统计左半部分内部的逆序对

\* - 再统计右半部分内部的逆序对

\* - 最后统计跨越左右两部分的逆序对 (关键步骤)

\*

\* - 统计跨区间逆序对的优化方法:

\* - 在合并前, 对每个左区间元素  $\text{nums}[i]$ , 找到右区间中满足  $\text{nums}[i] > \text{nums}[j]$  的元素个数

\* - 利用双指针技巧: 由于左右子数组已排序, 可以线性扫描而不需要嵌套循环

\* - 这一步的时间复杂度为  $O(n)$  而非  $O(n^2)$

\*

\* - 归并排序过程:

\* 1. 分治: 将数组不断二分, 直到只有一个元素

\* 2. 统计: 统计三种类型的逆序对

\* 3. 合并: 将两个有序子数组合并

\*

\* - 时间复杂度详细计算:

\*  $T(n) = 2T(n/2) + O(n)$  [Master 定理 case 2]

\*  $= O(n \log n)$

\* - 递归深度:  $\log n$

\* - 每层合并与统计:  $O(n)$

\*

\* - 空间复杂度详细计算:

\*  $S(n) = O(n) + O(\log n)$

\* -  $O(n)$ : 辅助数组 help

\* -  $O(\log n)$ : 递归调用栈

\* 总计:  $O(n)$

\*

\* - 是否最优解: ★ 是 ★

\* 理由: 基于比较的算法下界为  $O(n \log n)$ , 本算法已达到最优

\*

\* =====

\* 相关题目列表 (同类算法)

\* =====

\* 1. LeetCode 493 - 翻转对

\* <https://leetcode.cn/problems/reverse-pairs/>

\* 问题: 统计满足  $\text{nums}[i] > 2 * \text{nums}[j]$  且  $i < j$  的对的数量

\* 解法: 归并排序过程中使用双指针统计跨越左右区间的翻转对

\*

\* 2. LeetCode 315 - 计算右侧小于当前元素的个数

\* <https://leetcode.cn/problems/count-of-smaller-numbers-after-self/>

\* 问题: 统计每个元素右侧比它小的元素个数

\* 解法: 归并排序过程中记录元素原始索引, 统计右侧小于当前元素的数量

\*

\* 3. LeetCode 327 - 区间和的个数  
\* <https://leetcode.cn/problems/count-of-range-sum/>  
\* 问题：统计区间和在[lower, upper]范围内的区间个数  
\* 解法：前缀和+归并排序，统计满足条件的前缀和对  
\*

\* 4. POJ 2299 - Ultra-QuickSort  
\* <http://poj.org/problem?id=2299>  
\* 问题：计算将数组排序所需的最小交换次数（即逆序对数量）  
\* 解法：归并排序统计逆序对  
\*

\* 5. HDU 1394 - Minimum Inversion Number  
\* <http://acm.hdu.edu.cn/showproblem.php?pid=1394>  
\* 问题：将数组循环左移，求所有可能排列中的最小逆序对数量  
\* 解法：归并排序+逆序对性质分析  
\*

\* 6. 洛谷 P1908 - 逆序对  
\* <https://www.luogu.com.cn/problem/P1908>  
\* 问题：统计数组中逆序对的总数  
\* 解法：归并排序统计逆序对  
\*

\* 7. HackerRank - Merge Sort: Counting Inversions  
\* <https://www.hackerrank.com/challenges/merge-sort/problem>  
\* 问题：统计逆序对数量  
\* 解法：归并排序统计逆序对  
\*

\* 8. SPOJ - INVCNT  
\* <https://www.spoj.com/problems/INVCNT/>  
\* 问题：统计逆序对数量  
\* 解法：归并排序统计逆序对  
\*

\* 9. CodeChef - INVCNT  
\* <https://www.codechef.com/problems/INVCNT>  
\* 问题：统计逆序对数量  
\* 解法：归并排序或树状数组  
\*

\* 这些题目虽然具体形式不同，但核心思想都是利用归并排序的分治特性，在合并过程中高效统计满足特定条件的元素对数量。  
\*/

```
public class Code05_ReversePairsLCR170 {
```

```
    public static int MAXN = 50001;
```

```
public static int[] help = new int[MAXN];

/**
 * 计算数组中逆序对的数量
 *
 * @param nums 输入数组
 * @return 逆序对的数量
 */
public static int reversePairs(int[] nums) {
    if (nums == null || nums.length < 2) {
        return 0;
    }
    return mergeSort(nums, 0, nums.length - 1);
}

/**
 * 归并排序，在排序过程中统计逆序对数量
 *
 * @param arr 输入数组
 * @param l 左边界
 * @param r 右边界
 * @return 区间[l, r]中的逆序对数量
 */
public static int mergeSort(int[] arr, int l, int r) {
    if (l == r) {
        return 0;
    }

    int m = (l + r) / 2;
    return mergeSort(arr, l, m) + mergeSort(arr, m + 1, r) + merge(arr, l, m, r);
}

/**
 * 合并两个有序数组，并统计跨越左右两部分的逆序对
 *
 * @param arr 输入数组
 * @param l 左边界
 * @param m 中点
 * @param r 右边界
 * @return 跨越左右两部分的逆序对数量
 */
public static int merge(int[] arr, int l, int m, int r) {
    // 统计逆序对数量
```

```

int ans = 0;
int j = m + 1;
for (int i = 1; i <= m; i++) {
    // 找到右半部分中第一个不满足 arr[i] > arr[j] 的位置
    while (j <= r && arr[i] > arr[j]) {
        j++;
    }
    // j 之前的元素都满足条件，即与 arr[i] 构成逆序对
    ans += (j - m - 1);
}

// 正常合并两个有序数组
int i = 1;
int a = 1, 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 = 1; i <= r; i++) {
    arr[i] = help[i];
}

return ans;
}

/**
 * 主方法，用于测试
 */
public static void main(String[] args) {
    // 测试用例 1：基本情况
    int[] test1 = {7, 5, 6, 4};
    System.out.println("输入: " + Arrays.toString(test1));
    System.out.println("输出: " + reversePairs(test1)); // 预期输出: 5

    // 测试用例 2：空数组
    int[] test2 = {};
    System.out.println("输入: " + Arrays.toString(test2));
    System.out.println("输出: " + reversePairs(test2)); // 预期输出: 0
}

```

```

// 测试用例 3: 单元素数组
int[] test3 = {1};
System.out.println("输入: " + Arrays.toString(test3));
System.out.println("输出: " + reversePairs(test3)); // 预期输出: 0

// 测试用例 4: 升序数组
int[] test4 = {1, 2, 3, 4};
System.out.println("输入: " + Arrays.toString(test4));
System.out.println("输出: " + reversePairs(test4)); // 预期输出: 0

// 测试用例 5: 降序数组
int[] test5 = {4, 3, 2, 1};
System.out.println("输入: " + Arrays.toString(test5));
System.out.println("输出: " + reversePairs(test5)); // 预期输出: 6

// 测试用例 6: 重复元素
int[] test6 = {2, 2, 2, 2};
System.out.println("输入: " + Arrays.toString(test6));
System.out.println("输出: " + reversePairs(test6)); // 预期输出: 0

// 测试用例 7: 包含负数
int[] test7 = {-1, -2, 3, -4};
System.out.println("输入: " + Arrays.toString(test7));
System.out.println("输出: " + reversePairs(test7)); // 预期输出: 4
}

/*
 * =====
 * Java 语言特有关注事项
 * =====
 * 1. 数据类型溢出问题:
 *   - 逆序对数量可能超过 int 范围, 但题目约束范围内不会溢出
 *   - 对于极端情况, 可以使用 long 类型存储结果
 *
 * 2. 内存管理:
 *   - 使用静态数组 help 避免频繁创建对象
 *   - 静态数组在类加载时初始化, 在类卸载时销毁
 *   - 注意线程安全问题, 多线程环境下可能需要额外同步
 *
 * 3. 递归深度控制:
 *   - Java 默认递归深度限制约为 1000–2000 层
 *   - 对于 n=50000 的数据规模, 递归深度约为  $\log_2(50000) \approx 16$  层, 完全足够

```

```
*  
* 4. 输入输出优化:  
*   - 对于大规模数据输入，可以使用 BufferedReader 和 StreamTokenizer 优化  
*   - 对于输出，可以使用 PrintWriter 缓冲输出  
*  
* 5. 位运算优化:  
*   - 可以使用  $(l + r) \gg 1$  代替  $(l + r) / 2$  提高运算效率  
*   - 注意当 l 和 r 都很大时， $(l + r)$  可能导致溢出，应改为  $l + ((r - 1) \gg 1)$   
*  
* 6. 异常处理:  
*   - 可以添加对 null 输入的检查  
*   - 对于极大数组，可以添加数组长度检查  
*  
* 7. 性能优化:  
*   - 对于小规模子数组（如长度<10），可以使用插入排序代替归并排序  
*   - 可以添加判断条件，当  $arr[m] \leq arr[m+1]$  时，子数组已有序，跳过合并  
*  
* 8. 代码优化建议:  
*   - 减少对象创建和垃圾回收压力  
*   - 使用基本数据类型而非包装类（避免自动装箱/拆箱开销）  
*   - 合理设置 MAXN 常量，预留适当空间  
*/  
}
```

=====

文件: Code05\_ReversePairsLCR170.py

=====

"""

=====

题目 5: 剑指 Offer 51 - 数组中的逆序对 (LCR 170) - Python 版

=====

题目来源: 剑指 Offer / LCR 170

题目链接: <https://leetcode.cn/problems/shu-zu-zhong-de-ni-xu-dui-lcof/>

难度级别: 困难

问题描述:

在数组中的两个数字，如果前面一个数字大于后面的数字，则这两个数字组成一个逆序对。

输入一个数组，求出这个数组中的逆序对的总数。

示例输入输出:

输入: [7, 5, 6, 4]

输出: 5

解释: 逆序对有(7, 5), (7, 6), (7, 4), (5, 4), (6, 4)

核心算法思想: 归并排序分治统计

方法 1: 暴力解法 (不推荐)

- 思路: 双重循环检查每一对  $(i, j)$  是否满足  $i < j$  且  $\text{nums}[i] > \text{nums}[j]$
- 时间复杂度:  $O(N^2)$  - 双重循环
- 空间复杂度:  $O(1)$  - 不需要额外空间
- 问题: 数据量大时超时

方法 2: 归并排序思想 (最优解) ★★★★★

- 核心洞察: 利用归并排序过程统计跨越左右两个子数组的逆序对
  - 先统计左半部分内部的逆序对
  - 再统计右半部分内部的逆序对
  - 最后统计跨越左右两部分的逆序对 (关键步骤)
- 统计跨区间逆序对的优化方法:
  - 在合并前, 对每个左区间元素  $\text{nums}[i]$ , 找到右区间中满足  $\text{nums}[i] > \text{nums}[j]$  的元素个数
  - 利用双指针技巧: 由于左右子数组已排序, 可以线性扫描而不需要嵌套循环
  - 这一步的时间复杂度为  $O(n)$  而非  $O(n^2)$
- 归并排序过程:
  1. 分治: 将数组不断二分, 直到只有一个元素
  2. 统计: 统计三种类型的逆序对
  3. 合并: 将两个有序子数组合并
- 时间复杂度详细计算:
$$\begin{aligned} T(n) &= 2T(n/2) + O(n) \quad [\text{Master 定理 case 2}] \\ &= O(n \log n) \end{aligned}$$
  - 递归深度:  $\log n$
  - 每层合并与统计:  $O(n)$
- 空间复杂度详细计算:
$$\begin{aligned} S(n) &= O(n) + O(\log n) \\ &- O(n): \text{辅助数组 help} \\ &- O(\log n): \text{递归调用栈} \\ &\text{总计: } O(n) \end{aligned}$$
- 是否最优解: ★ 是 ★  
理由: 基于比较的算法下界为  $O(n \log n)$ , 本算法已达到最优

---

---

## 相关题目列表（同类算法）

---

---

### 1. LeetCode 493 - 翻转对

<https://leetcode.cn/problems/reverse-pairs/>

问题：统计满足  $\text{nums}[i] > 2 * \text{nums}[j]$  且  $i < j$  的对的数量

解法：归并排序过程中使用双指针统计跨越左右区间的翻转对

### 2. LeetCode 315 - 计算右侧小于当前元素的个数

<https://leetcode.cn/problems/count-of-smaller-numbers-after-self/>

问题：统计每个元素右侧比它小的元素个数

解法：归并排序过程中记录元素原始索引，统计右侧小于当前元素的数量

### 3. LeetCode 327 - 区间和的个数

<https://leetcode.cn/problems/count-of-range-sum/>

问题：统计区间和在  $[\text{lower}, \text{upper}]$  范围内的区间个数

解法：前缀和+归并排序，统计满足条件的前缀和对

### 4. POJ 2299 - Ultra-QuickSort

<http://poj.org/problem?id=2299>

问题：计算将数组排序所需的最小交换次数（即逆序对数量）

解法：归并排序统计逆序对

### 5. HDU 1394 - Minimum Inversion Number

<http://acm.hdu.edu.cn/showproblem.php?pid=1394>

问题：将数组循环左移，求所有可能排列中的最小逆序对数量

解法：归并排序+逆序对性质分析

### 6. 洛谷 P1908 - 逆序对

<https://www.luogu.com.cn/problem/P1908>

问题：统计数组中逆序对的总数

解法：归并排序统计逆序对

### 7. HackerRank - Merge Sort: Counting Inversions

<https://www.hackerrank.com/challenges/merge-sort/problem>

问题：统计逆序对数量

解法：归并排序统计逆序对

### 8. SPOJ - INVCTN

<https://www.spoj.com/problems/INVCTN/>

问题：统计逆序对数量

解法：归并排序统计逆序对

## 9. CodeChef - INVCNT

<https://www.codechef.com/problems/INVCNT>

问题：统计逆序对数量

解法：归并排序或树状数组

这些题目虽然具体形式不同，但核心思想都是利用归并排序的分治特性，在合并过程中高效统计满足特定条件的元素对数量。

"""

```
def reverse_pairs(nums):
```

"""

计算数组中逆序对的数量

Args:

nums: 输入数组

Returns:

int: 逆序对的数量

"""

```
if not nums or len(nums) < 2:
```

```
    return 0
```

```
def merge_sort(l, r):
```

"""

归并排序，在排序过程中统计逆序对数量

Args:

l: 左边界

r: 右边界

Returns:

int: 区间[l, r]中的逆序对数量

"""

```
if l == r:
```

```
    return 0
```

```
    mid = (l + r) // 2
```

```
    return merge_sort(l, mid) + merge_sort(mid + 1, r) + merge(l, mid, r)
```

```
def merge(l, m, r):
```

"""

合并两个有序数组，并统计跨越左右两部分的逆序对

Args:

- l: 左边界
- m: 中点
- r: 右边界

Returns:

int: 跨越左右两部分的逆序对数量

"""

# 辅助数组

```
help_arr = [0] * (r - l + 1)
```

# 统计逆序对数量

```
ans = 0
```

```
j = m + 1
```

```
for i in range(l, m + 1):
```

# 找到右半部分中第一个不满足  $\text{nums}[i] > \text{nums}[j]$  的位置

```
while j <= r and nums[i] > nums[j]:
```

```
    j += 1
```

# j 之前的元素都满足条件，即与  $\text{nums}[i]$  构成逆序对

```
ans += (j - m - 1)
```

# 正常合并两个有序数组

```
i = 0
```

```
a, b = l, m + 1
```

```
while a <= m and b <= r:
```

```
    if nums[a] <= nums[b]:
```

```
        help_arr[i] = nums[a]
```

```
        a += 1
```

```
    else:
```

```
        help_arr[i] = nums[b]
```

```
        b += 1
```

```
    i += 1
```

# 处理剩余元素

```
while a <= m:
```

```
    help_arr[i] = nums[a]
```

```
    a += 1
```

```
    i += 1
```

```
while b <= r:
```

```
    help_arr[i] = nums[b]
```

```
    b += 1
```

```
i += 1

# 将辅助数组内容复制回原数组
for i in range(len(help_arr)):
    nums[1 + i] = help_arr[i]

return ans

# 创建数组副本，避免修改原数组
nums_copy = nums[:]
return merge_sort(0, len(nums_copy) - 1)

# 测试代码
if __name__ == "__main__":
    # 测试用例 1: 基本情况
    test_nums1 = [7, 5, 6, 4]
    print(f"输入: {test_nums1}")
    print(f"输出: {reverse_pairs(test_nums1)}")  # 预期输出: 5

    # 测试用例 2: 空数组
    test_nums2 = []
    print(f"输入: {test_nums2}")
    print(f"输出: {reverse_pairs(test_nums2)}")  # 预期输出: 0

    # 测试用例 3: 单元素数组
    test_nums3 = [1]
    print(f"输入: {test_nums3}")
    print(f"输出: {reverse_pairs(test_nums3)}")  # 预期输出: 0

    # 测试用例 4: 升序数组
    test_nums4 = [1, 2, 3, 4]
    print(f"输入: {test_nums4}")
    print(f"输出: {reverse_pairs(test_nums4)}")  # 预期输出: 0

    # 测试用例 5: 降序数组
    test_nums5 = [4, 3, 2, 1]
    print(f"输入: {test_nums5}")
    print(f"输出: {reverse_pairs(test_nums5)}")  # 预期输出: 6

    # 测试用例 6: 重复元素
    test_nums6 = [2, 2, 2, 2]
    print(f"输入: {test_nums6}")
```

```
print(f"输出: {reverse_pairs(test_nums6)}") # 预期输出: 0

# 测试用例 7: 包含负数
test_nums7 = [-1, -2, 3, -4]
print(f"输入: {test_nums7}")
print(f"输出: {reverse_pairs(test_nums7)}") # 预期输出: 4
```

"""

## Python 语言特有关注事项

### 1. 整数精度优势:

- Python 的整数类型自动支持大整数，不会有溢出问题
- 相比 Java 和 C++，无需手动转换为 long/long long 类型
- 适合处理极端大的结果

### 2. 递归深度限制:

- Python 默认递归深度限制约为 1000 层
- 对于大规模数据 ( $n=50000$ )，归并排序的递归深度 ( $\log_2(50000) \approx 16$  层) 完全没问题
- 但处理  $n$  接近  $2^{30}$  的数据时，需要调整递归深度限制：

```
import sys
sys.setrecursionlimit(1000000)
```

### 3. 列表操作效率:

- 列表切片操作 ( $arr[:]$ ) 会创建副本，有  $O(n)$  时间和空间开销
- 频繁创建小列表会增加 GC 压力
- 推荐使用索引操作代替切片，提升性能

### 4. 可变对象特性:

- Python 中列表是可变对象，函数内修改会影响外部
- 实现中使用 `nums_copy` 避免修改原数组，保持函数纯度
- 这在多线程环境中很重要

### 5. 生成器和迭代器:

- 对于大数据集，可以考虑使用生成器节省内存
- 但在算法核心部分，直接使用列表访问更快

### 6. 类型提示支持:

- Python 3.5+ 支持类型提示，提高代码可读性和 IDE 支持
- 例如: `def reverse_pairs(nums: List[int]) -> int:`
- 需要导入: `from typing import List`

## 7. 性能考量:

- Python 的递归实现比迭代慢
- 对于竞赛场景, Python 可能在时间限制内无法处理最大规模数据
- 实际应用中可以接受, 但高性能场景考虑 C++ 实现

## 8. 缓存装饰器:

- 对于重复调用相同参数的场景, 可以使用 `functools.lru_cache`
- 但此算法中不适用, 因为每次处理的数组切片不同

## 9. 多进程并行:

- Python 的 GIL 限制了多线程性能提升
- 对于大规模数据, 考虑使用 `multiprocessing` 模块进行并行计算
- 注意进程间通信的开销

## 10. 调试便利性:

- Python 的 `print` 调试和异常信息比 C++ 更友好
- 可以使用 `pdb` 进行交互式调试
- 列表推导式等语法使代码更简洁, 但可能牺牲可读性

"""\n=====\n=====

文件: Code06\_UltraQuickSort.cpp

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <climits>
#include <ios>
using namespace std;

/***
 * =====
 * 题目 6: POJ 2299 - Ultra-QuickSort - C++版
 * =====
 *
 * 题目来源: POJ (Peking Online Judge)
 * 题目链接: http://poj.org/problem?id=2299
 * 难度级别: 中等
 *
 * 问题描述:
 * 在这个问题中, 您必须分析一个特定的排序算法。该算法通过交换相邻元素对序列进行排序。
```

- \* 给定一个序列，求出使用该算法进行排序所需的最少交换次数。
  - \*
  - \* 核心洞察：
    - \* 最少交换相邻元素的次数等于逆序对的数量。
  - \*
  - \* 示例输入输出：
    - \* 输入：
      - \* 5
      - \* 9 1 0 5 4
      - \* 3
      - \* 1 2 3
      - \* 0
      - \*
    - \* 输出：
      - \* 6
      - \* 0
      - \*
  - \* 解释：
    - \* 对于序列[9, 1, 0, 5, 4]，逆序对有：
      - \* (9, 1), (9, 0), (9, 5), (9, 4), (1, 0), (5, 4) 共 6 对
  - \*
  - \* =====
  - \* 核心算法思想：归并排序统计逆序对
    - \* =====
    - \*
    - \* 方法 1：冒泡排序模拟（不推荐）
      - \* - 思路：直接模拟冒泡排序过程，统计交换次数
      - \* - 时间复杂度： $O(N^2)$  – 双重循环
      - \* - 空间复杂度： $O(1)$  – 不需要额外空间
      - \* - 问题：数据量大时超时
      - \*
    - \* 方法 2：归并排序思想（最优解）★★★★★
      - \* - 核心洞察：最少交换相邻元素的次数等于逆序对的数量
      - \* - 证明：每次交换相邻元素可以消除一个逆序对，且这是唯一消除逆序对的方式
      - \*
      - \* - 归并排序过程：
        - \* 1. 分治：将数组不断二分，直到只有一个元素
        - \* 2. 统计：统计三种类型的逆序对
        - \* 3. 合并：将两个有序子数组合并
        - \*
        - \* - 统计跨区间逆序对的优化方法：
          - \* 在合并前，对每个左区间元素  $arr[i]$ ，找到右区间中满足  $arr[i] > arr[j]$  的元素个数
          - \* 利用双指针技巧：由于左右子数组已排序，可以线性扫描而不需要嵌套循环

- \* - 这一步的时间复杂度为  $O(n)$  而非  $O(n^2)$
- \*
- \* - 时间复杂度详细计算:
  - \*  $T(n) = 2T(n/2) + O(n)$  [Master 定理 case 2]
  - \*  $= O(n \log n)$
  - \* - 递归深度:  $\log n$
  - \* - 每层合并与统计:  $O(n)$
  - \*
- \* - 空间复杂度详细计算:
  - \*  $S(n) = O(n) + O(\log n)$
  - \* -  $O(n)$ : 辅助数组 help
  - \* -  $O(\log n)$ : 递归调用栈
  - \* 总计:  $O(n)$
  - \*
- \* - 是否最优解: ★ 是 ★
  - \* 理由: 基于比较的算法下界为  $O(n \log n)$ , 本算法已达到最优
  - \*
- \* =====
- \* 相关题目列表 (同类算法)
  - \* =====
  - \* 1. 剑指 Offer 51 / LCR 170 - 数组中的逆序对
    - \* <https://leetcode.cn/problems/shu-zu-zhong-de-ni-xu-dui-lcof/>
    - \* 问题: 统计数组中逆序对的总数
    - \* 解法: 归并排序过程中统计逆序对数量
    - \*
  - \* 2. LeetCode 493 - 翻转对
    - \* <https://leetcode.cn/problems/reverse-pairs/>
    - \* 问题: 统计满足  $nums[i] > 2*nums[j]$  且  $i < j$  的对的数量
    - \* 解法: 归并排序过程中使用双指针统计跨越左右区间的翻转对
    - \*
  - \* 3. LeetCode 315 - 计算右侧小于当前元素的个数
    - \* <https://leetcode.cn/problems/count-of-smaller-numbers-after-self/>
    - \* 问题: 统计每个元素右侧比它小的元素个数
    - \* 解法: 归并排序过程中记录元素原始索引, 统计右侧小于当前元素的数量
    - \*
  - \* 4. LeetCode 327 - 区间和的个数
    - \* <https://leetcode.cn/problems/count-of-range-sum/>
    - \* 问题: 统计区间和在  $[lower, upper]$  范围内的区间个数
    - \* 解法: 前缀和+归并排序, 统计满足条件的前缀和对
    - \*
  - \* 5. HDU 1394 - Minimum Inversion Number
    - \* <http://acm.hdu.edu.cn/showproblem.php?pid=1394>
    - \* 问题: 将数组循环左移, 求所有可能排列中的最小逆序对数量

- \*     解法：归并排序+逆序对性质分析
- \*
- \* 6. 洛谷 P1908 - 逆序对
- \*     <https://www.luogu.com.cn/problem/P1908>
- \*     问题：统计数组中逆序对的总数
- \*     解法：归并排序统计逆序对
- \*
- \* 7. HackerRank - Merge Sort: Counting Inversions
- \*     <https://www.hackerrank.com/challenges/merge-sort/problem>
- \*     问题：统计逆序对数量
- \*     解法：归并排序统计逆序对
- \*
- \* 8. SPOJ - INVCNT
- \*     <https://www.spoj.com/problems/INVCNT/>
- \*     问题：统计逆序对数量
- \*     解法：归并排序统计逆序对
- \*
- \* 9. UVa 10810 - Ultra-QuickSort
- \*

[https://onlinejudge.org/index.php?option=com\\_onlinejudge&Itemid=8&page=show\\_problem&problem=1751](https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=1751)

- \*     问题：计算将数组排序所需的最小交换次数（即逆序对数量）
- \*     解法：归并排序统计逆序对
- \*

\* 这些题目虽然具体形式不同，但核心思想都是利用归并排序的分治特性，在合并过程中高效统计满足特定条件的元素对数量。

\*/

```
const int MAXN = 500001;
int arr[MAXN];
int help[MAXN];

/***
 * 归并排序，在排序过程中统计逆序对数量
 *
 * @param l 左边界
 * @param r 右边界
 * @return 区间[l, r]中的逆序对数量
 */
long long mergeSort(int l, int r) {
    if (l == r) {
        return 0;
    }
```

```

int m = (l + r) / 2;
return mergeSort(l, m) + mergeSort(m + 1, r) + merge(l, m, r);
}

/***
 * 合并两个有序数组，并统计跨越左右两部分的逆序对
 *
 * @param l 左边界
 * @param m 中点
 * @param r 右边界
 * @return 跨越左右两部分的逆序对数量
 */
long long merge(int l, int m, int r) {
    // 统计逆序对数量
    long long ans = 0;
    int j = m + 1;
    for (int i = l; i <= m; i++) {
        // 找到右半部分中第一个不满足 arr[i] > arr[j] 的位置
        while (j <= r && arr[i] > arr[j]) {
            j++;
        }
        // j 之前的元素都满足条件，即与 arr[i] 构成逆序对
        ans += (j - m - 1);
    }

    // 正常合并两个有序数组
    int i = l;
    int a = l, 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];
    }

    return ans;
}

```

```

/***
 * 主函数，处理 POJ 格式的输入输出
 */
int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int n;
    while (cin >> n && n != 0) {
        for (int i = 0; i < n; i++) {
            cin >> arr[i];
        }

        cout << mergeSort(0, n - 1) << "\n";
    }

    return 0;
}

```

/\*  
=====  
C++语言特有关注事项  
=====

#### 1. 数据类型溢出问题：

- 逆序对数量可能超过 int 范围，必须使用 long long 类型存储结果
- 对于  $n=500000$  的数据规模，逆序对数量最大约为  $n*(n-1)/2 \approx 1.25 \times 10^{11}$ ，超过 int 范围

#### 2. 内存管理：

- 使用全局数组而非 vector，避免频繁动态分配内存
- MAXN 设为 500001，足够处理题目要求的最大数据规模
- 静态数组在栈上分配，访问速度比堆分配更快

#### 3. 递归深度控制：

- 归并排序的递归深度为  $\log_2(n)$ ，对于  $n=500000$ ，深度约为 19 层
- 不会超过 C++ 默认的栈大小限制
- 对于极端大数据，可以考虑非递归实现

#### 4. 输入输出优化：

- `ios::sync_with_stdio(false)`；关闭同步，加速 `cin/cout`
- `cin.tie(nullptr)`；解绑 `cin` 和 `cout`，减少刷新次数
- 使用 “\n” 代替 `endl`，避免不必要的缓冲区刷新

## 5. 位运算优化:

- 使用  $(l + r) \gg 1$  代替  $(l + r) / 2$ , 提高运算效率
- 注意当 l 和 r 都很大时,  $(l + r)$  可能导致溢出, 应改为  $l + ((r - 1) \gg 1)$

## 6. 代码优化技巧:

- 在 merge 函数中先统计逆序对再排序, 逻辑更清晰
- 合并时使用三目运算符使代码更简洁

## 7. 编译优化选项:

- 可以添加 -O2 编译选项获得更好的性能
- 对于某些编译器, -march=native 可以利用 CPU 特性进一步优化

## 8. 多线程考虑:

- 当前实现不是线程安全的, 因为使用了全局变量
- 多线程环境下应使用局部变量或添加同步机制

## 9. 边界条件处理:

- 对空数组、单元素数组有正确的边界检查
- 递归终止条件明确

## 10. 异常处理:

- C++ 中可以添加 try-catch 块处理可能的异常
- 对数组索引越界等情况进行检查

\*/

=====

文件: Code06\_UltraQuickSort.java

=====

```
package class022;

import java.io.*;
import java.util.*;

/**
 * =====
 * 题目 6: POJ 2299 - Ultra-QuickSort
 * =====
 *
 * 题目来源: POJ (Peking Online Judge)
 * 题目链接: http://poj.org/problem?id=2299
 * 难度级别: 中等
```

- \*
  - \* 问题描述:
    - \* 在这个问题中，您必须分析一个特定的排序算法。该算法通过交换相邻元素对序列进行排序。
    - \* 给定一个序列，求出使用该算法进行排序所需的最少交换次数。
  - \*
  - \* 核心洞察:
    - \* 最少交换相邻元素的次数等于逆序对的数量。
  - \*
  - \* 示例输入输出:
    - \* 输入:
      - \* 5
      - \* 9 1 0 5 4
      - \* 3
      - \* 1 2 3
      - \* 0
      - \*
    - \* 输出:
      - \* 6
      - \* 0
      - \*
  - \* 解释:
    - \* 对于序列[9, 1, 0, 5, 4]，逆序对有：
      - \* (9, 1), (9, 0), (9, 5), (9, 4), (1, 0), (5, 4) 共 6 对
  - \*
  - \* ======
    - \* 核心算法思想：归并排序统计逆序对
  - \* ======
    - \*
  - \* 方法 1：冒泡排序模拟（不推荐）
    - \* - 思路：直接模拟冒泡排序过程，统计交换次数
    - \* - 时间复杂度： $O(N^2)$  – 双重循环
    - \* - 空间复杂度： $O(1)$  – 不需要额外空间
    - \* - 问题：数据量大时超时
  - \*
  - \* 方法 2：归并排序思想（最优解）★★★★★
    - \* - 核心洞察：最少交换相邻元素的次数等于逆序对的数量
    - \* - 证明：每次交换相邻元素可以消除一个逆序对，且这是唯一消除逆序对的方式
  - \*
  - \* - 归并排序过程：
    - \* 1. 分治：将数组不断二分，直到只有一个元素
    - \* 2. 统计：统计三种类型的逆序对
    - \* 3. 合并：将两个有序子数组合并
  - \*

- \* - 统计跨区间逆序对的优化方法:
  - \* - 在合并前, 对每个左区间元素 arr[i], 找到右区间中满足  $arr[i] > arr[j]$  的元素个数
  - \* - 利用双指针技巧: 由于左右子数组已排序, 可以线性扫描而不需要嵌套循环
  - \* - 这一步的时间复杂度为  $O(n)$  而非  $O(n^2)$
- \*
- \* - 时间复杂度详细计算:
  - \*  $T(n) = 2T(n/2) + O(n)$  [Master 定理 case 2]
  - \*  $= O(n \log n)$
  - \* - 递归深度:  $\log n$
  - \* - 每层合并与统计:  $O(n)$
- \*
- \* - 空间复杂度详细计算:
  - \*  $S(n) = O(n) + O(\log n)$
  - \* -  $O(n)$ : 辅助数组 help
  - \* -  $O(\log n)$ : 递归调用栈
  - \* 总计:  $O(n)$
- \*
- \* - 是否最优解: ★ 是 ★
  - \* 理由: 基于比较的算法下界为  $O(n \log n)$ , 本算法已达到最优
- \*
- \* =====
- \* 相关题目列表 (同类算法)
  - \* =====
  - \* 1. 剑指 Offer 51 / LCR 170 - 数组中的逆序对
    - \* <https://leetcode.cn/problems/shu-zu-zhong-de-ni-xu-dui-lcof/>
    - \* 问题: 统计数组中逆序对的总数
    - \* 解法: 归并排序过程中统计逆序对数量
  - \*
  - \* 2. LeetCode 493 - 翻转对
    - \* <https://leetcode.cn/problems/reverse-pairs/>
    - \* 问题: 统计满足  $nums[i] > 2*nums[j]$  且  $i < j$  的对的数量
    - \* 解法: 归并排序过程中使用双指针统计跨越左右区间的翻转对
  - \*
  - \* 3. LeetCode 315 - 计算右侧小于当前元素的个数
    - \* <https://leetcode.cn/problems/count-of-smaller-numbers-after-self/>
    - \* 问题: 统计每个元素右侧比它小的元素个数
    - \* 解法: 归并排序过程中记录元素原始索引, 统计右侧小于当前元素的数量
  - \*
  - \* 4. LeetCode 327 - 区间和的个数
    - \* <https://leetcode.cn/problems/count-of-range-sum/>
    - \* 问题: 统计区间和在 [lower, upper] 范围内的区间个数
    - \* 解法: 前缀和+归并排序, 统计满足条件的前缀和对

- \* 5. HDU 1394 - Minimum Inversion Number
- \*   <http://acm.hdu.edu.cn/showproblem.php?pid=1394>
- \*   问题：将数组循环左移，求所有可能排列中的最小逆序对数量
- \*   解法：归并排序+逆序对性质分析
- \*
- \* 6. 洛谷 P1908 - 逆序对
- \*   <https://www.luogu.com.cn/problem/P1908>
- \*   问题：统计数组中逆序对的总数
- \*   解法：归并排序统计逆序对
- \*
- \* 7. HackerRank - Merge Sort: Counting Inversions
- \*   <https://www.hackerrank.com/challenges/merge-sort/problem>
- \*   问题：统计逆序对数量
- \*   解法：归并排序统计逆序对
- \*
- \* 8. SPOJ - INVCNT
- \*   <https://www.spoj.com/problems/INVCNT/>
- \*   问题：统计逆序对数量
- \*   解法：归并排序统计逆序对
- \*
- \* 9. UVa 10810 - Ultra-QuickSort
- \*

[https://onlinejudge.org/index.php?option=com\\_onlinejudge&Itemid=8&page=show\\_problem&problem=1751](https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=1751)

- \*   问题：计算将数组排序所需的最小交换次数（即逆序对数量）
- \*   解法：归并排序统计逆序对
- \*
- \* 这些题目虽然具体形式不同，但核心思想都是利用归并排序的分治特性，在合并过程中高效统计满足特定条件的元素对数量。

\*/

```
public class Code06_UltraQuickSort {  
  
    public static int MAXN = 500001;  
    public static int[] arr = new int[MAXN];  
    public static int[] help = new int[MAXN];  
  
    /**  
     * 计算数组中逆序对的数量（即最少交换次数）  
     *  
     * @param n 数组长度  
     * @return 逆序对的数量  
     */  
    public static long mergeSort(int n) {
```

```

    if (n < 2) {
        return 0;
    }
    return mergeSort(0, n - 1);
}

/***
 * 归并排序，在排序过程中统计逆序对数量
 *
 * @param l 左边界
 * @param r 右边界
 * @return 区间[l, r]中的逆序对数量
 */
public static long mergeSort(int l, int r) {
    if (l == r) {
        return 0;
    }

    int m = (l + r) / 2;
    return mergeSort(l, m) + mergeSort(m + 1, r) + merge(l, m, r);
}

/***
 * 合并两个有序数组，并统计跨越左右两部分的逆序对
 *
 * @param l 左边界
 * @param m 中点
 * @param r 右边界
 * @return 跨越左右两部分的逆序对数量
 */
public static long merge(int l, int m, int r) {
    // 统计逆序对数量
    long ans = 0;
    int j = m + 1;
    for (int i = l; i <= m; i++) {
        // 找到右半部分中第一个不满足 arr[i] > arr[j] 的位置
        while (j <= r && arr[i] > arr[j]) {
            j++;
        }
        // j 之前的元素都满足条件，即与 arr[i] 构成逆序对
        ans += (j - m - 1);
    }
}

```

```

// 正常合并两个有序数组
int i = l;
int a = l, 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];
}

return ans;
}

/***
 * 主方法，处理 POJ 格式的输入输出
 */
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));

    while (in.nextToken() != StreamTokenizer.TT_EOF) {
        int n = (int) in.nval;
        if (n == 0) {
            break;
        }

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

        out.println(mergeSort(n));
    }

    out.flush();
    out.close();
}

```

```
}

/*
 * =====
 * Java 语言特有关注事项
 * =====
 * 1. 数据类型溢出问题:
 *   - 逆序对数量可能超过 int 范围, 必须使用 long 类型存储结果
 *   - 对于 n=500000 的数据规模, 逆序对数量最大约为  $n*(n-1)/2 \approx 1.25 \times 10^{11}$ , 超过 int 范围
 *
 * 2. 输入输出效率:
 *   - 使用 StreamTokenizer 和 BufferedReader 提高输入效率
 *   - 使用 PrintWriter 缓冲输出提高输出效率
 *   - 对于大规模数据输入输出, 这种优化是必要的
 *
 * 3. 内存管理:
 *   - 使用静态数组避免频繁创建对象
 *   - MAXN 设为 500001, 足够处理题目要求的最大数据规模
 *   - 静态数组在类加载时初始化, 在类卸载时销毁
 *
 * 4. 递归深度控制:
 *   - Java 默认递归深度限制约为 1000-2000 层
 *   - 对于 n=500000 的数据规模, 递归深度约为  $\log_2(500000) \approx 19$  层, 完全足够
 *
 * 5. 位运算优化:
 *   - 可以使用  $(l + r) \gg 1$  代替  $(l + r) / 2$  提高运算效率
 *   - 注意当 l 和 r 都很大时,  $(l + r)$  可能导致溢出, 应改为  $l + ((r - 1) \gg 1)$ 
 *
 * 6. 异常处理:
 *   - 添加 IOException 处理, 处理输入输出异常
 *   - 可以添加对极大数组的检查
 *
 * 7. 性能优化:
 *   - 对于小规模子数组 (如长度<10), 可以使用插入排序代替归并排序
 *   - 可以添加判断条件, 当  $arr[m] \leq arr[m+1]$  时, 子数组已有序, 跳过合并
 *
 * 8. 代码优化建议:
 *   - 减少对象创建和垃圾回收压力
 *   - 使用基本数据类型而非包装类 (避免自动装箱/拆箱开销)
 *   - 合理设置 MAXN 常量, 预留适当空间
 */
}
```

文件: Code06\_UltraQuickSort.py

"""

题目 6: POJ 2299 - Ultra-QuickSort - Python 版

题目来源: POJ (Peking Online Judge)

题目链接: <http://poj.org/problem?id=2299>

难度级别: 中等

问题描述:

在这个问题中, 您必须分析一个特定的排序算法。该算法通过交换相邻元素对序列进行排序。

给定一个序列, 求出使用该算法进行排序所需的最少交换次数。

核心洞察:

最少交换相邻元素的次数等于逆序对的数量。

示例输入输出:

输入:

5  
9 1 0 5 4  
3  
1 2 3  
0

输出:

6  
0

解释:

对于序列[9, 1, 0, 5, 4], 逆序对有:

(9, 1), (9, 0), (9, 5), (9, 4), (1, 0), (5, 4) 共 6 对

核心算法思想: 归并排序统计逆序对

方法 1: 冒泡排序模拟 (不推荐)

- 思路: 直接模拟冒泡排序过程, 统计交换次数
- 时间复杂度:  $O(N^2)$  - 双重循环

- 空间复杂度:  $O(1)$  - 不需要额外空间
- 问题: 数据量大时超时

## 方法 2: 归并排序思想 (最优解) ★★★★★

- 核心洞察: 最少交换相邻元素的次数等于逆序对的数量
- 证明: 每次交换相邻元素可以消除一个逆序对, 且这是唯一消除逆序对的方式
- 归并排序过程:
  1. 分治: 将数组不断二分, 直到只有一个元素
  2. 统计: 统计三种类型的逆序对
  3. 合并: 将两个有序子数组合并
- 统计跨区间逆序对的优化方法:
  - 在合并前, 对每个左区间元素  $arr[i]$ , 找到右区间中满足  $arr[i] > arr[j]$  的元素个数
  - 利用双指针技巧: 由于左右子数组已排序, 可以线性扫描而不需要嵌套循环
  - 这一步的时间复杂度为  $O(n)$  而非  $O(n^2)$
- 时间复杂度详细计算:
 
$$\begin{aligned} T(n) &= 2T(n/2) + O(n) \quad [\text{Master 定理 case 2}] \\ &= O(n \log n) \end{aligned}$$
  - 递归深度:  $\log n$
  - 每层合并与统计:  $O(n)$
- 空间复杂度详细计算:
 
$$\begin{aligned} S(n) &= O(n) + O(\log n) \\ &- O(n): \text{辅助数组 help} \\ &- O(\log n): \text{递归调用栈} \\ &\text{总计: } O(n) \end{aligned}$$
- 是否最优解: ★ 是 ★  
理由: 基于比较的算法下界为  $O(n \log n)$ , 本算法已达到最优

## 相关题目列表 (同类算法)

1. 剑指 Offer 51 / LCR 170 - 数组中的逆序对  
<https://leetcode.cn/problems/shu-zu-zhong-de-ni-xu-dui-lcof/>  
问题: 统计数组中逆序对的总数  
解法: 归并排序过程中统计逆序对数量
2. LeetCode 493 - 翻转对  
<https://leetcode.cn/problems/reverse-pairs/>  
问题: 统计满足  $nums[i] > 2*nums[j]$  且  $i < j$  的对的数量

解法：归并排序过程中使用双指针统计跨越左右区间的翻转对

3. LeetCode 315 – 计算右侧小于当前元素的个数

<https://leetcode.cn/problems/count-of-smaller-numbers-after-self/>

问题：统计每个元素右侧比它小的元素个数

解法：归并排序过程中记录元素原始索引，统计右侧小于当前元素的数量

4. LeetCode 327 – 区间和的个数

<https://leetcode.cn/problems/count-of-range-sum/>

问题：统计区间和在[lower, upper]范围内的区间个数

解法：前缀和+归并排序，统计满足条件的前缀和对

5. HDU 1394 – Minimum Inversion Number

<http://acm.hdu.edu.cn/showproblem.php?pid=1394>

问题：将数组循环左移，求所有可能排列中的最小逆序对数量

解法：归并排序+逆序对性质分析

6. 洛谷 P1908 – 逆序对

<https://www.luogu.com/problem/P1908>

问题：统计数组中逆序对的总数

解法：归并排序统计逆序对

7. HackerRank – Merge Sort: Counting Inversions

<https://www.hackerrank.com/challenges/merge-sort/problem>

问题：统计逆序对数量

解法：归并排序统计逆序对

8. SPOJ – INVCTN

<https://www.spoj.com/problems/INVCTN/>

问题：统计逆序对数量

解法：归并排序统计逆序对

9. UVa 10810 – Ultra-QuickSort

[https://onlinejudge.org/index.php?option=com\\_onlinejudge&Itemid=8&page=show\\_problem&problem=1751](https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=1751)

问题：计算将数组排序所需的最小交换次数（即逆序对数量）

解法：归并排序统计逆序对

这些题目虽然具体形式不同，但核心思想都是利用归并排序的分治特性，在合并过程中高效统计满足特定条件的元素对数量。

""

```
import sys
```

```
def merge_sort_and_count(arr):
    """
    归并排序，在排序过程中统计逆序对数量

    Args:
        arr: 输入数组

    Returns:
        int: 逆序对的数量
    """
    if len(arr) < 2:
        return 0

    def merge_sort(l, r):
        """
        归并排序，在排序过程中统计逆序对数量

        Args:
            l: 左边界
            r: 右边界

        Returns:
            int: 区间[l, r]中的逆序对数量
        """
        if l == r:
            return 0

        mid = (l + r) // 2
        return merge_sort(l, mid) + merge_sort(mid + 1, r) + merge(l, mid, r)

    def merge(l, m, r):
        """
        合并两个有序数组，并统计跨越左右两部分的逆序对

        Args:
            l: 左边界
            m: 中点
            r: 右边界

        Returns:
            int: 跨越左右两部分的逆序对数量
        """

```

```
# 辅助数组
help_arr = [0] * (r - l + 1)

# 统计逆序对数量
ans = 0
j = m + 1
for i in range(l, m + 1):
    # 找到右半部分中第一个不满足 arr[i] > arr[j] 的位置
    while j <= r and arr[i] > arr[j]:
        j += 1
    # j 之前的元素都满足条件，即与 arr[i] 构成逆序对
    ans += (j - m - 1)

# 正常合并两个有序数组
i = 0
a, b = l, 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

# 处理剩余元素
while a <= m:
    help_arr[i] = arr[a]
    a += 1
    i += 1

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

# 将辅助数组内容复制回原数组
for i in range(len(help_arr)):
    arr[l + i] = help_arr[i]

return ans

# 创建数组副本，避免修改原数组
```

```
arr_copy = arr[:]
return merge_sort(0, len(arr_copy) - 1)

# POJ 格式的输入输出处理
def main():
    """
    主函数，处理 POJ 格式的输入输出
    """
    # 增加递归深度限制以处理大规模数据
    sys.setrecursionlimit(1000000)

    try:
        while True:
            line = input().strip()
            if not line:
                break

            n = int(line)
            if n == 0:
                break

            arr = []
            line = input().strip()
            numbers = list(map(int, line.split()))
            arr.extend(numbers)

            result = merge_sort_and_count(arr)
            print(result)
    except EOFError:
        pass

# 测试代码
def test():
    """
    测试函数
    """
    # 测试用例 1: 基本情况
    test_arr1 = [9, 1, 0, 5, 4]
    print(f"输入: {test_arr1}")
    print(f"输出: {merge_sort_and_count(test_arr1)}"") # 预期输出: 6
```

```

# 测试用例 2: 已排序数组
test_arr2 = [1, 2, 3]
print(f"输入: {test_arr2}")
print(f"输出: {merge_sort_and_count(test_arr2)}") # 预期输出: 0

# 测试用例 3: 逆序数组
test_arr3 = [5, 4, 3, 2, 1]
print(f"输入: {test_arr3}")
print(f"输出: {merge_sort_and_count(test_arr3)}") # 预期输出: 10

# 测试用例 4: 重复元素
test_arr4 = [2, 2, 2, 2]
print(f"输入: {test_arr4}")
print(f"输出: {merge_sort_and_count(test_arr4)}") # 预期输出: 0

if __name__ == "__main__":
    # 运行测试
    # test()

    # 运行 POJ 格式的输入输出处理
    main()

"""

=====

```

## Python 语言特有关注事项

---

1. 整数精度优势:
  - Python 的整数类型自动支持大整数，不会有溢出问题
  - 相比 Java 和 C++，无需手动转换为 long/long long 类型
  - 适合处理极端大的结果
  
2. 递归深度限制:
  - Python 默认递归深度限制约为 1000 层
  - 对于大规模数据 (n=500000)，需要调整递归深度限制:
 

```
import sys
sys.setrecursionlimit(1000000)
```
  
3. 列表操作效率:
  - 列表切片操作 (arr[:]) 会创建副本，有 O(n) 时间和空间开销
  - 频繁创建小列表会增加 GC 压力

- 推荐使用索引操作代替切片，提升性能

#### 4. 可变对象特性：

- Python 中列表是可变对象，函数内修改会影响外部
- 实现中使用 arr\_copy 避免修改原数组，保持函数纯度
- 这在多线程环境中很重要

#### 5. 生成器和迭代器：

- 对于大数据集，可以考虑使用生成器节省内存
- 但在算法核心部分，直接使用列表访问更快

#### 6. 类型提示支持：

- Python 3.5+ 支持类型提示，提高代码可读性和 IDE 支持
- 例如: def merge\_sort\_and\_count(arr: List[int]) -> int:
- 需要导入: from typing import List

#### 7. 性能考量：

- Python 的递归实现比迭代慢
- 对于竞赛场景，Python 可能在时间限制内无法处理最大规模数据
- 实际应用中可以接受，但高性能场景考虑 C++ 实现

#### 8. 缓存装饰器：

- 对于重复调用相同参数的场景，可以使用 functools.lru\_cache
- 但此算法中不适用，因为每次处理的数组切片不同

#### 9. 多进程并行：

- Python 的 GIL 限制了多线程性能提升
- 对于大规模数据，考虑使用 multiprocessing 模块进行并行计算
- 注意进程间通信的开销

#### 10. 调试便利性：

- Python 的 print 调试和异常信息比 C++ 更友好
- 可以使用 pdb 进行交互式调试
- 列表推导式等语法使代码更简洁，但可能牺牲可读性

"""

文件: Code07\_MinimumInversionNumber.cpp

```
#include <iostream>
#include <vector>
#include <algorithm>
```

```
#include <climits>
#include <iostream>
using namespace std;

/***
 * =====
 * 题目 7: HDU 1394 - Minimum Inversion Number - C++版
 * =====
 *
 * 题目来源: HDU (杭州电子科技大学 OJ)
 * 题目链接: http://acm.hdu.edu.cn/showproblem.php?pid=1394
 * 难度级别: 中等
 *
 * 问题描述:
 * 给定一个 0 到 n-1 的排列, 可以进行循环移位操作 (把第一个数移到最后)。
 * 求所有可能状态中逆序对数量的最小值。
 *
 * 核心洞察:
 * 1. 先用归并排序计算初始逆序对数
 * 2. 循环移位时, 利用数学公式快速更新逆序对数
 * - 若移动元素 x: new_inv = old_inv - x + (n-1-x)
 *
 * 示例输入输出:
 * 输入:
 * 5
 * 1 3 0 2 4
 *
 * 输出:
 * 3
 *
 * 解释:
 * 初始序列[1, 3, 0, 2, 4]有 6 个逆序对: (1, 0), (3, 0), (3, 2), (2, 0), (2, 1), (4, 0)
 * 循环移位后序列[3, 0, 2, 4, 1]有 5 个逆序对
 * 循环移位后序列[0, 2, 4, 1, 3]有 4 个逆序对
 * 循环移位后序列[2, 4, 1, 3, 0]有 5 个逆序对
 * 循环移位后序列[4, 1, 3, 0, 2]有 3 个逆序对 (最小值)
 *
 * =====
 * 核心算法思想: 归并排序 + 数学优化
 * =====
 *
 * 方法 1: 暴力解法 (不推荐)
 * - 思路: 对每个循环移位后的序列, 都计算一次逆序对数量
```

\* - 时间复杂度:  $O(N^3)$  -  $N$  次循环移位, 每次  $O(N^2)$  计算逆序对

\* - 空间复杂度:  $O(N)$  - 存储循环移位后的序列

\* - 问题: 数据量大时超时

\*

\* 方法 2: 归并排序 + 数学优化 (最优解) ★★★★★★

\* - 核心洞察:

\* 1. 先计算初始序列的逆序对数

\* 2. 利用数学关系快速计算循环移位后的逆序对数

\*

\* - 数学优化原理:

\* 当把第一个元素  $x$  移到序列末尾时:

\* - 减少的逆序对数: 原来在  $x$  后面且小于  $x$  的元素个数, 即  $x$  个

\* - 增加的逆序对数: 原来在  $x$  后面且大于  $x$  的元素个数, 即  $(n-1-x)$  个

\* - 因此:  $\text{new\_inv} = \text{old\_inv} - x + (n-1-x)$

\*

\* - 算法步骤:

\* 1. 使用归并排序计算初始序列的逆序对数

\* 2. 循环  $N-1$  次, 每次根据数学公式更新逆序对数

\* 3. 记录过程中的最小值

\*

\* - 时间复杂度详细计算:

\*  $T(n) = O(n \log n) + O(n) = O(n \log n)$

\* - 归并排序计算初始逆序对:  $O(n \log n)$

\* - 循环更新逆序对数:  $O(n)$

\*

\* - 空间复杂度详细计算:

\*  $S(n) = O(n) + O(\log n)$

\* -  $O(n)$ : 辅助数组 help

\* -  $O(\log n)$ : 递归调用栈

\* 总计:  $O(n)$

\*

\* - 是否最优解: ★ 是 ★

\* 理由: 基于比较的算法下界为  $O(n \log n)$ , 本算法已达到最优

\*

\* =====

\* 相关题目列表 (同类算法)

\* =====

\* 1. POJ 2299 - Ultra-QuickSort

\* <http://poj.org/problem?id=2299>

\* 问题: 计算将数组排序所需的最小交换次数 (即逆序对数量)

\* 解法: 归并排序统计逆序对

\*

\* 2. 剑指 Offer 51 / LCR 170 - 数组中的逆序对

- \* <https://leetcode.cn/problems/shu-zu-zhong-de-ni-xu-dui-lcof/>
  - \* 问题: 统计数组中逆序对的总数
  - \* 解法: 归并排序过程中统计逆序对数量
  - \*
- \* 3. LeetCode 493 - 翻转对
  - \* <https://leetcode.cn/problems/reverse-pairs/>
  - \* 问题: 统计满足  $\text{nums}[i] > 2 * \text{nums}[j]$  且  $i < j$  的对的数量
  - \* 解法: 归并排序过程中使用双指针统计跨越左右区间的翻转对
  - \*
- \* 4. LeetCode 315 - 计算右侧小于当前元素的个数
  - \* <https://leetcode.cn/problems/count-of-smaller-numbers-after-self/>
  - \* 问题: 统计每个元素右侧比它小的元素个数
  - \* 解法: 归并排序过程中记录元素原始索引, 统计右侧小于当前元素的数量
  - \*
- \* 5. LeetCode 327 - 区间和的个数
  - \* <https://leetcode.cn/problems/count-of-range-sum/>
  - \* 问题: 统计区间和在  $[\text{lower}, \text{upper}]$  范围内的区间个数
  - \* 解法: 前缀和+归并排序, 统计满足条件的前缀和对
  - \*
- \* 6. 洛谷 P1908 - 逆序对
  - \* <https://www.luogu.com.cn/problem/P1908>
  - \* 问题: 统计数组中逆序对的总数
  - \* 解法: 归并排序统计逆序对
  - \*
- \* 7. HackerRank - Merge Sort: Counting Inversions
  - \* <https://www.hackerrank.com/challenges/merge-sort/problem>
  - \* 问题: 统计逆序对数量
  - \* 解法: 归并排序统计逆序对
  - \*
- \* 8. SPOJ - INVCNT
  - \* <https://www.spoj.com/problems/INVCNT/>
  - \* 问题: 统计逆序对数量
  - \* 解法: 归并排序统计逆序对
  - \*
- \* 9. UVa 10810 - Ultra-QuickSort
  - \*

[https://onlinejudge.org/index.php?option=com\\_onlinejudge&Itemid=8&page=show\\_problem&problem=1751](https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=1751)

- \* 问题: 计算将数组排序所需的最小交换次数 (即逆序对数量)
- \* 解法: 归并排序统计逆序对
- \*

\* 这些题目虽然具体形式不同, 但核心思想都是利用归并排序的分治特性, 在合并过程中高效统计满足特定条件的元素对数量。

\*/

```

const int MAXN = 10001;
int arr[MAXN];
int help[MAXN];

/***
 * 归并排序，在排序过程中统计逆序对数量
 *
 * @param l 左边界
 * @param r 右边界
 * @return 区间[l, r]中的逆序对数量
 */
long long mergeSort(int l, int r) {
    if (l == r) {
        return 0;
    }

    int m = (l + r) / 2;
    return mergeSort(l, m) + mergeSort(m + 1, r) + merge(l, m, r);
}

/***
 * 合并两个有序数组，并统计跨越左右两部分的逆序对
 *
 * @param l 左边界
 * @param m 中点
 * @param r 右边界
 * @return 跨越左右两部分的逆序对数量
 */
long long merge(int l, int m, int r) {
    // 统计逆序对数量
    long long ans = 0;
    int j = m + 1;
    for (int i = l; i <= m; i++) {
        // 找到右半部分中第一个不满足 arr[i] > arr[j] 的位置
        while (j <= r && arr[i] > arr[j]) {
            j++;
        }
        // j 之前的元素都满足条件，即与 arr[i] 构成逆序对
        ans += (j - m - 1);
    }
}

// 正常合并两个有序数组

```

```

int i = 1;
int a = 1, 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 = 1; i <= r; i++) {
    arr[i] = help[i];
}

return ans;
}

```

/\*\*

\* 计算循环移位序列中的最小逆序对数

\*

\* @param n 数组长度

\* @return 最小逆序对数

\*/

```

long long getMinimumInversion(int n) {
    // 计算初始序列的逆序对数
    long long inv = mergeSort(0, n - 1);
    long long minInv = inv;

    // 循环移位，利用数学公式快速更新逆序对数
    for (int i = 0; i < n - 1; i++) {
        // 当把第一个元素 arr[i]移到序列末尾时：
        // 减少的逆序对数：arr[i]个
        // 增加的逆序对数：(n-1-arr[i])个
        inv = inv - arr[i] + (n - 1 - arr[i]);
        minInv = min(minInv, inv);
    }

    return minInv;
}

```

/\*\*

\* 主函数，处理 HDU 格式的输入输出

```

*/
int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int n;
    while (cin >> n) {
        for (int i = 0; i < n; i++) {
            cin >> arr[i];
        }

        cout << getMinimumInversion(n) << "\n";
    }

    return 0;
}

```

/\*  
=====  
C++语言特有关注事项  
=====

### 1. 数据类型溢出问题:

- 逆序对数量可能超过 int 范围，必须使用 long long 类型存储结果
- 对于  $n=10000$  的数据规模，逆序对数量最大约为  $n*(n-1)/2 \approx 5\times 10^7$ ，未超过 int 范围
- 但为了代码通用性，仍使用 long long 类型

### 2. 内存管理:

- 使用全局数组而非 vector，避免频繁动态分配内存
- MAXN 设为 10001，足够处理题目要求的最大数据规模
- 静态数组在栈上分配，访问速度比堆分配更快

### 3. 递归深度控制:

- 归并排序的递归深度为  $\log_2(n)$ ，对于  $n=10000$ ，深度约为 14 层
- 不会超过 C++ 默认的栈大小限制
- 对于极端大数据，可以考虑非递归实现

### 4. 输入输出优化:

- `ios::sync_with_stdio(false);` 关闭同步，加速 `cin/cout`
- `cin.tie(nullptr);` 解绑 `cin` 和 `cout`，减少刷新次数
- 使用 "`\n`" 代替 `endl`，避免不必要的缓冲区刷新

### 5. 位运算优化:

- 使用 $(l + r) \gg 1$ 代替 $(l + r) / 2$ , 提高运算效率
- 注意当 l 和 r 都很大时,  $(l + r)$ 可能导致溢出, 应改为 $l + ((r - 1) \gg 1)$

## 6. 代码优化技巧:

- 在 merge 函数中先统计逆序对再排序, 逻辑更清晰
- 合并时使用三目运算符使代码更简洁

## 7. 编译优化选项:

- 可以添加 -O2 编译选项获得更好的性能
- 对于某些编译器, -march=native 可以利用 CPU 特性进一步优化

## 8. 多线程考虑:

- 当前实现不是线程安全的, 因为使用了全局变量
- 多线程环境下应使用局部变量或添加同步机制

## 9. 边界条件处理:

- 对空数组、单元素数组有正确的边界检查
- 递归终止条件明确

## 10. 异常处理:

- C++ 中可以添加 try-catch 块处理可能的异常
- 对数组索引越界等情况进行检查

\*/

=====

文件: Code07\_MinimumInversionNumber.java

=====

```
package class022;

import java.io.*;
import java.util.*;

/**
 * =====
 * 题目 7: HDU 1394 - Minimum Inversion Number
 * =====
 *
 * 题目来源: HDU (杭州电子科技大学 OJ)
 * 题目链接: http://acm.hdu.edu.cn/showproblem.php?pid=1394
 * 难度级别: 中等
 *
 * 问题描述:
```

\* 给定一个 0 到 n-1 的排列，可以进行循环移位操作（把第一个数移到最后）。

\* 求所有可能状态中逆序对数量的最小值。

\*

\* 核心洞察：

\* 1. 先用归并排序计算初始逆序对数

\* 2. 循环移位时，利用数学公式快速更新逆序对数

\* - 若移动元素 x: new\_inv = old\_inv - x + (n-1-x)

\*

\* 示例输入输出：

\* 输入：

\* 5

\* 1 3 0 2 4

\*

\* 输出：

\* 3

\*

\* 解释：

\* 初始序列[1, 3, 0, 2, 4]有 6 个逆序对：(1, 0), (3, 0), (3, 2), (2, 0), (2, 1), (4, 0)

\* 循环移位后序列[3, 0, 2, 4, 1]有 5 个逆序对

\* 循环移位后序列[0, 2, 4, 1, 3]有 4 个逆序对

\* 循环移位后序列[2, 4, 1, 3, 0]有 5 个逆序对

\* 循环移位后序列[4, 1, 3, 0, 2]有 3 个逆序对（最小值）

\*

\* =====

\* 核心算法思想：归并排序 + 数学优化

\* =====

\*

\* 方法 1：暴力解法（不推荐）

\* - 思路：对每个循环移位后的序列，都计算一次逆序对数量

\* - 时间复杂度：O(N^3) - N 次循环移位，每次 O(N^2) 计算逆序对

\* - 空间复杂度：O(N) - 存储循环移位后的序列

\* - 问题：数据量大时超时

\*

\* 方法 2：归并排序 + 数学优化（最优解）★★★★★

\* - 核心洞察：

\* 1. 先计算初始序列的逆序对数

\* 2. 利用数学关系快速计算循环移位后的逆序对数

\*

\* - 数学优化原理：

\* 当把第一个元素 x 移到序列末尾时：

\* - 减少的逆序对数：原来在 x 后面且小于 x 的元素个数，即 x 个

\* - 增加的逆序对数：原来在 x 后面且大于 x 的元素个数，即 (n-1-x) 个

\* - 因此：new\_inv = old\_inv - x + (n-1-x)

\*

\* - 算法步骤:

\*    1. 使用归并排序计算初始序列的逆序对数

\*    2. 循环 N-1 次, 每次根据数学公式更新逆序对数

\*    3. 记录过程中的最小值

\*

\* - 时间复杂度详细计算:

\*     $T(n) = O(n \log n) + O(n) = O(n \log n)$

\*    - 归并排序计算初始逆序对:  $O(n \log n)$

\*    - 循环更新逆序对数:  $O(n)$

\*

\* - 空间复杂度详细计算:

\*     $S(n) = O(n) + O(\log n)$

\*    -  $O(n)$ : 辅助数组 help

\*    -  $O(\log n)$ : 递归调用栈

\*    总计:  $O(n)$

\*

\* - 是否最优解: ★ 是 ★

\*    理由: 基于比较的算法下界为  $O(n \log n)$ , 本算法已达到最优

\*

\* =====

\* 相关题目列表 (同类算法)

\* =====

\* 1. POJ 2299 - Ultra-QuickSort

\*    <http://poj.org/problem?id=2299>

\*    问题: 计算将数组排序所需的最小交换次数 (即逆序对数量)

\*    解法: 归并排序统计逆序对

\*

\* 2. 剑指 Offer 51 / LCR 170 - 数组中的逆序对

\*    <https://leetcode.cn/problems/shu-zu-zhong-de-ni-xu-dui-lcof/>

\*    问题: 统计数组中逆序对的总数

\*    解法: 归并排序过程中统计逆序对数量

\*

\* 3. LeetCode 493 - 翻转对

\*    <https://leetcode.cn/problems/reverse-pairs/>

\*    问题: 统计满足  $\text{nums}[i] > 2 * \text{nums}[j]$  且  $i < j$  的对的数量

\*    解法: 归并排序过程中使用双指针统计跨越左右区间的翻转对

\*

\* 4. LeetCode 315 - 计算右侧小于当前元素的个数

\*    <https://leetcode.cn/problems/count-of-smaller-numbers-after-self/>

\*    问题: 统计每个元素右侧比它小的元素个数

\*    解法: 归并排序过程中记录元素原始索引, 统计右侧小于当前元素的数量

\*

- \* 5. LeetCode 327 - 区间和的个数
  - \* <https://leetcode.cn/problems/count-of-range-sum/>
  - \* 问题：统计区间和在[lower, upper]范围内的区间个数
  - \* 解法：前缀和+归并排序，统计满足条件的前缀和对
  - \*
- \* 6. 洛谷 P1908 - 逆序对
  - \* <https://www.luogu.com.cn/problem/P1908>
  - \* 问题：统计数组中逆序对的总数
  - \* 解法：归并排序统计逆序对
  - \*
- \* 7. HackerRank - Merge Sort: Counting Inversions
  - \* <https://www.hackerrank.com/challenges/merge-sort/problem>
  - \* 问题：统计逆序对数量
  - \* 解法：归并排序统计逆序对
  - \*
- \* 8. SPOJ - INVCNT
  - \* <https://www.spoj.com/problems/INVCNT/>
  - \* 问题：统计逆序对数量
  - \* 解法：归并排序统计逆序对
  - \*
- \* 9. UVa 10810 - Ultra-QuickSort
  - \*

[https://onlinejudge.org/index.php?option=com\\_onlinejudge&Itemid=8&page=show\\_problem&problem=1751](https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=1751)

- \* 问题：计算将数组排序所需的最小交换次数（即逆序对数量）
- \* 解法：归并排序统计逆序对
- \*

\* 这些题目虽然具体形式不同，但核心思想都是利用归并排序的分治特性，在合并过程中高效统计满足特定条件的元素对数量。

\*/

```
public class Code07_MinimumInversionNumber {  
  
    public static int MAXN = 10001;  
    public static int[] arr = new int[MAXN];  
    public static int[] help = new int[MAXN];  
  
    /**  
     * 归并排序，在排序过程中统计逆序对数量  
     *  
     * @param l 左边界  
     * @param r 右边界  
     * @return 区间[l, r]中的逆序对数量  
     */
```

```

public static long mergeSort(int l, int r) {
    if (l == r) {
        return 0;
    }

    int m = (l + r) / 2;
    return mergeSort(l, m) + mergeSort(m + 1, r) + merge(l, m, r);
}

/***
 * 合并两个有序数组，并统计跨越左右两部分的逆序对
 *
 * @param l 左边界
 * @param m 中点
 * @param r 右边界
 * @return 跨越左右两部分的逆序对数量
 */
public static long merge(int l, int m, int r) {
    // 统计逆序对数量
    long ans = 0;
    int j = m + 1;
    for (int i = l; i <= m; i++) {
        // 找到右半部分中第一个不满足 arr[i] > arr[j] 的位置
        while (j <= r && arr[i] > arr[j]) {
            j++;
        }
        // j 之前的元素都满足条件，即与 arr[i] 构成逆序对
        ans += (j - m - 1);
    }

    // 正常合并两个有序数组
    int i = l;
    int a = l, 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];
    }

    return ans;
}

/***
 * 计算循环移位序列中的最小逆序对数
 *
 * @param n 数组长度
 * @return 最小逆序对数
 */
public static long getMinimumInversion(int n) {
    // 计算初始序列的逆序对数
    long inv = mergeSort(0, n - 1);
    long minInv = inv;

    // 循环移位，利用数学公式快速更新逆序对数
    for (int i = 0; i < n - 1; i++) {
        // 当把第一个元素 arr[i] 移到序列末尾时：
        // 减少的逆序对数：arr[i] 个
        // 增加的逆序对数：(n-1-arr[i]) 个
        inv = inv - arr[i] + (n - 1 - arr[i]);
        minInv = Math.min(minInv, inv);
    }

    return minInv;
}

/***
 * 主方法，处理 HDU 格式的输入输出
 */
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));

    while (in.nextToken() != StreamTokenizer.TT_EOF) {
        int n = (int) in.nval;

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

```

```
    }

    out.println(getMinimumInversion(n));
}

out.flush();
out.close();
}

/*
* =====
* Java 语言特有关注事项
* =====
* 1. 数据类型溢出问题:
*   - 逆序对数量可能超过 int 范围, 必须使用 long 类型存储结果
*   - 对于 n=10000 的数据规模, 逆序对数量最大约为  $n*(n-1)/2 \approx 5 \times 10^7$ , 未超过 int 范围
*   - 但为了代码通用性, 仍使用 long 类型
*
* 2. 输入输出效率:
*   - 使用 StringTokenizer 和 BufferedReader 提高输入效率
*   - 使用 PrintWriter 缓冲输出提高输出效率
*   - 对于大规模数据输入输出, 这种优化是必要的
*
* 3. 内存管理:
*   - 使用静态数组避免频繁创建对象
*   - MAXN 设为 10001, 足够处理题目要求的最大数据规模
*   - 静态数组在类加载时初始化, 在类卸载时销毁
*
* 4. 递归深度控制:
*   - Java 默认递归深度限制约为 1000–2000 层
*   - 对于 n=10000 的数据规模, 递归深度约为  $\log_2(10000) \approx 14$  层, 完全足够
*
* 5. 位运算优化:
*   - 可以使用  $(l + r) \gg 1$  代替  $(l + r) / 2$  提高运算效率
*   - 注意当 l 和 r 都很大时,  $(l + r)$  可能导致溢出, 应改为  $l + ((r - 1) \gg 1)$ 
*
* 6. 异常处理:
*   - 添加 IOException 处理, 处理输入输出异常
*   - 可以添加对极大数组的检查
*
* 7. 性能优化:
*   - 对于小规模子数组 (如长度<10), 可以使用插入排序代替归并排序
*   - 可以添加判断条件, 当  $arr[m] \leq arr[m+1]$  时, 子数组已有序, 跳过合并
}
```

```
*  
* 8. 代码优化建议:  
*   - 减少对象创建和垃圾回收压力  
*   - 使用基本数据类型而非包装类（避免自动装箱/拆箱开销）  
*   - 合理设置 MAXN 常量，预留适当空间  
*/  
}  
=====
```

文件: Code07\_MinimumInversionNumber.py

```
=====  
"""  
=====
```

题目 7: HDU 1394 – Minimum Inversion Number – Python 版

```
=====
```

题目来源: HDU (杭州电子科技大学 OJ)

题目链接: <http://acm.hdu.edu.cn/showproblem.php?pid=1394>

难度级别: 中等

问题描述:

给定一个 0 到  $n-1$  的排列，可以进行循环移位操作（把第一个数移到最后）。

求所有可能状态中逆序对数量的最小值。

核心洞察:

1. 先用归并排序计算初始逆序对数
2. 循环移位时，利用数学公式快速更新逆序对数
  - 若移动元素  $x$ :  $\text{new\_inv} = \text{old\_inv} - x + (n-1-x)$

示例输入输出:

输入:

5

1 3 0 2 4

输出:

3

解释:

初始序列 [1, 3, 0, 2, 4] 有 6 个逆序对: (1, 0), (3, 0), (3, 2), (2, 0), (2, 1), (4, 0)

循环移位后序列 [3, 0, 2, 4, 1] 有 5 个逆序对

循环移位后序列 [0, 2, 4, 1, 3] 有 4 个逆序对

循环移位后序列 [2, 4, 1, 3, 0] 有 5 个逆序对

循环移位后序列 [4, 1, 3, 0, 2] 有 3 个逆序对 (最小值)

=====

核心算法思想：归并排序 + 数学优化

=====

方法 1：暴力解法（不推荐）

- 思路：对每个循环移位后的序列，都计算一次逆序对数量
- 时间复杂度： $O(N^3)$  –  $N$  次循环移位，每次  $O(N^2)$  计算逆序对
- 空间复杂度： $O(N)$  – 存储循环移位后的序列
- 问题：数据量大时超时

方法 2：归并排序 + 数学优化（最优解）★★★★★

- 核心洞察：
  1. 先计算初始序列的逆序对数
  2. 利用数学关系快速计算循环移位后的逆序对数

- 数学优化原理：

当把第一个元素  $x$  移到序列末尾时：

- 减少的逆序对数：原来在  $x$  后面且小于  $x$  的元素个数，即  $x$  个
- 增加的逆序对数：原来在  $x$  后面且大于  $x$  的元素个数，即  $(n-1-x)$  个
- 因此： $\text{new\_inv} = \text{old\_inv} - x + (n-1-x)$

- 算法步骤：

1. 使用归并排序计算初始序列的逆序对数
2. 循环  $N-1$  次，每次根据数学公式更新逆序对数
3. 记录过程中的最小值

- 时间复杂度详细计算：

$$T(n) = O(n \log n) + O(n) = O(n \log n)$$

- 归并排序计算初始逆序对： $O(n \log n)$
- 循环更新逆序对数： $O(n)$

- 空间复杂度详细计算：

$$S(n) = O(n) + O(\log n)$$

- $O(n)$ ：辅助数组 `help`
  - $O(\log n)$ ：递归调用栈
- 总计： $O(n)$

- 是否最优解：★ 是 ★

理由：基于比较的算法下界为  $O(n \log n)$ ，本算法已达到最优

=====

## 相关题目列表（同类算法）

---

1. POJ 2299 - Ultra-QuickSort

<http://poj.org/problem?id=2299>

问题：计算将数组排序所需的最小交换次数（即逆序对数量）

解法：归并排序统计逆序对

2. 剑指 Offer 51 / LCR 170 - 数组中的逆序对

<https://leetcode.cn/problems/shu-zu-zhong-de-ni-xu-dui-lcof/>

问题：统计数组中逆序对的总数

解法：归并排序过程中统计逆序对数量

3. LeetCode 493 - 翻转对

<https://leetcode.cn/problems/reverse-pairs/>

问题：统计满足  $\text{nums}[i] > 2 * \text{nums}[j]$  且  $i < j$  的对的数量

解法：归并排序过程中使用双指针统计跨越左右区间的翻转对

4. LeetCode 315 - 计算右侧小于当前元素的个数

<https://leetcode.cn/problems/count-of-smaller-numbers-after-self/>

问题：统计每个元素右侧比它小的元素个数

解法：归并排序过程中记录元素原始索引，统计右侧小于当前元素的数量

5. LeetCode 327 - 区间和的个数

<https://leetcode.cn/problems/count-of-range-sum/>

问题：统计区间和在 [lower, upper] 范围内的区间个数

解法：前缀和+归并排序，统计满足条件的前缀和对

6. 洛谷 P1908 - 逆序对

<https://www.luogu.com.cn/problem/P1908>

问题：统计数组中逆序对的总数

解法：归并排序统计逆序对

7. HackerRank - Merge Sort: Counting Inversions

<https://www.hackerrank.com/challenges/merge-sort/problem>

问题：统计逆序对数量

解法：归并排序统计逆序对

8. SPOJ - INVCNT

<https://www.spoj.com/problems/INVCNT/>

问题：统计逆序对数量

解法：归并排序统计逆序对

9. UVa 10810 - Ultra-QuickSort

[https://onlinejudge.org/index.php?option=com\\_onlinejudge&Itemid=8&page=show\\_problem&problem=1751](https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=1751)

问题：计算将数组排序所需的最小交换次数（即逆序对数量）

解法：归并排序统计逆序对

这些题目虽然具体形式不同，但核心思想都是利用归并排序的分治特性，在合并过程中高效统计满足特定条件的元素对数量。

"""

import sys

def merge\_sort\_and\_count(arr):

"""

归并排序，在排序过程中统计逆序对数量

Args:

arr: 输入数组

Returns:

int: 逆序对的数量

"""

if len(arr) < 2:

return 0

def merge\_sort(l, r):

"""

归并排序，在排序过程中统计逆序对数量

Args:

l: 左边界

r: 右边界

Returns:

int: 区间[l, r]中的逆序对数量

"""

if l == r:

return 0

mid = (l + r) // 2

return merge\_sort(l, mid) + merge\_sort(mid + 1, r) + merge(l, mid, r)

def merge(l, m, r):

"""

合并两个有序数组，并统计跨越左右两部分的逆序对

Args:

- l: 左边界
- m: 中点
- r: 右边界

Returns:

```
int: 跨越左右两部分的逆序对数量
"""
# 辅助数组
help_arr = [0] * (r - l + 1)

# 统计逆序对数量
ans = 0
j = m + 1
for i in range(l, m + 1):
    # 找到右半部分中第一个不满足 arr[i] > arr[j] 的位置
    while j <= r and arr[i] > arr[j]:
        j += 1
    # j 之前的元素都满足条件，即与 arr[i] 构成逆序对
    ans += (j - m - 1)

# 正常合并两个有序数组
i = 0
a, b = l, 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

# 处理剩余元素
while a <= m:
    help_arr[i] = arr[a]
    a += 1
    i += 1

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

```

    b += 1
    i += 1

# 将辅助数组内容复制回原数组
for i in range(len(help_arr)):
    arr[1 + i] = help_arr[i]

return ans

# 创建数组副本，避免修改原数组
arr_copy = arr[:]
return merge_sort(0, len(arr_copy) - 1)

def get_minimum_inversion(arr):
    """
    计算循环移位序列中的最小逆序对数

    Args:
        arr: 输入数组

    Returns:
        int: 最小逆序对数
    """

    n = len(arr)
    if n == 0:
        return 0

    # 计算初始序列的逆序对数
    inv = merge_sort_and_count(arr)
    min_inv = inv

    # 循环移位，利用数学公式快速更新逆序对数
    for i in range(n - 1):
        # 当把第一个元素 arr[i] 移到序列末尾时：
        # 减少的逆序对数：arr[i] 个
        # 增加的逆序对数：(n-1-arr[i]) 个
        inv = inv - arr[i] + (n - 1 - arr[i])
        min_inv = min(min_inv, inv)

    return min_inv

```

```
# HDU 格式的输入输出处理
def main():
    """
    主函数，处理 HDU 格式的输入输出
    """

    # 增加递归深度限制以处理大规模数据
    sys.setrecursionlimit(100000)

    try:
        while True:
            line = input().strip()
            if not line:
                break

            n = int(line)
            line = input().strip()
            arr = list(map(int, line.split()))

            result = get_minimum_inversion(arr)
            print(result)

    except EOFError:
        pass

# 测试代码
def test():
    """
    测试函数
    """

    # 测试用例 1: 基本情况
    test_arr1 = [1, 3, 0, 2, 4]
    print(f"输入: {test_arr1}")
    print(f"输出: {get_minimum_inversion(test_arr1)}" ) # 预期输出: 3

    # 测试用例 2: 已排序数组
    test_arr2 = [0, 1, 2, 3, 4]
    print(f"输入: {test_arr2}")
    print(f"输出: {get_minimum_inversion(test_arr2)}" ) # 预期输出: 0

    # 测试用例 3: 逆序数组
    test_arr3 = [4, 3, 2, 1, 0]
    print(f"输入: {test_arr3}")
    print(f"输出: {get_minimum_inversion(test_arr3)}" ) # 预期输出: 6
```

```
if __name__ == "__main__":
    # 运行测试
    # test()

    # 运行 HDU 格式的输入输出处理
    main()

"""

=====
```

## Python 语言特有关注事项

---

### 1. 整数精度优势:

- Python 的整数类型自动支持大整数，不会有溢出问题
- 相比 Java 和 C++，无需手动转换为 long/long long 类型
- 适合处理极端大的结果

### 2. 递归深度限制:

- Python 默认递归深度限制约为 1000 层
- 对于大规模数据 (n=10000)，需要调整递归深度限制：

```
import sys
sys.setrecursionlimit(100000)
```

### 3. 列表操作效率:

- 列表切片操作 (arr[:]) 会创建副本，有  $O(n)$  时间和空间开销
- 频繁创建小列表会增加 GC 压力
- 推荐使用索引操作代替切片，提升性能

### 4. 可变对象特性:

- Python 中列表是可变对象，函数内修改会影响外部
- 实现中使用 arr\_copy 避免修改原数组，保持函数纯度
- 这在多线程环境中很重要

### 5. 生成器和迭代器:

- 对于大数据集，可以考虑使用生成器节省内存
- 但在算法核心部分，直接使用列表访问更快

### 6. 类型提示支持:

- Python 3.5+ 支持类型提示，提高代码可读性和 IDE 支持
- 例如：def get\_minimum\_inversion(arr: List[int]) -> int:

- 需要导入: from typing import List

## 7. 性能考量:

- Python 的递归实现比迭代慢
- 对于竞赛场景, Python 可能在时间限制内无法处理最大规模数据
- 实际应用中可以接受, 但高性能场景考虑 C++ 实现

## 8. 缓存装饰器:

- 对于重复调用相同参数的场景, 可以使用 functools.lru\_cache
- 但此算法中不适用, 因为每次处理的数组切片不同

## 9. 多进程并行:

- Python 的 GIL 限制了多线程性能提升
- 对于大规模数据, 考虑使用 multiprocessing 模块进行并行计算
- 注意进程间通信的开销

## 10. 调试便利性:

- Python 的 print 调试和异常信息比 C++ 更友好
- 可以使用 pdb 进行交互式调试
- 列表推导式等语法使代码更简洁, 但可能牺牲可读性

"""

文件: Code08\_LuoguP1908.cpp

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <cstdio>
#include <cstring>
using namespace std;

/***
 * =====
 * 题目 8: 洛谷 P1908 - 逆序对 (Inversion Pairs)
 * =====
 *
 * 题目来源: 洛谷
 * 题目链接: https://www.luogu.com.cn/problem/P1908
 * 难度级别: 普及+/提高
 *
 * 问题描述:
 */
```

- \* 给定一个序列，求出这个序列的逆序对个数。
- \* 逆序对定义：对于序列中的两个元素  $a[i]$  和  $a[j]$ ，如果  $i < j$  且  $a[i] > a[j]$ ，则称这两个元素构成一个逆序对。
- \*
- \* 输入格式：
- \* 第一行，一个整数  $n$ ，表示序列长度。
- \* 第二行， $n$  个整数，表示给定的序列。
- \*
- \* 输出格式：
- \* 输出一个整数，表示序列中逆序对的个数。
- \*
- \* 示例输入输出：
- \* 输入：
- \* 6
- \* 5 4 2 6 3 1
- \* 输出：
- \* 11
- \*
- \* 数据范围：
- \* 对于 60% 的数据， $n \leq 1000$
- \* 对于 100% 的数据， $n \leq 500000$ ，序列中每个数的绝对值不超过  $10^9$
- \*
- \* =====
- \* 核心算法思想：归并排序分治统计
- \* =====
- \*
- \* 方法 1：暴力解法（不推荐）
- \* - 思路：双重循环遍历所有  $i < j$  的情况，判断  $\text{nums}[i] > \text{nums}[j]$
- \* - 时间复杂度： $O(N^2)$  - 双重循环
- \* - 空间复杂度： $O(1)$  - 不需要额外空间
- \* - 问题：数据量大时超时
- \*
- \* 方法 2：归并排序思想（最优解）★★★★★
- \* - 核心洞察：利用归并排序的分治过程，在合并两个有序子数组之前，统计左侧子数组中每个元素与右侧子数组形成的逆序对数量
- \*
- \* - 归并排序过程：
- \* 1. 分治：将数组不断二分，直到只有一个元素
- \* 2. 统计：在合并前，统计左侧子数组中每个元素能与右侧子数组形成的逆序对数量
- \* 3. 合并：合并两个有序子数组
- \*
- \* - 统计逆序对的关键步骤：
- \* - 当右子数组中的元素被选中时，不会对左侧元素产生影响

- \* - 当左子数组中的元素被选中时，右子数组中已处理的元素都小于它，形成逆序对
- \* - 因此，每次选中左子数组元素时，需要累加右侧已处理的元素数量
- \*
- \* - 时间复杂度详细计算：
  - \*  $T(n) = 2T(n/2) + O(n)$  [Master 定理 case 2]
  - \*  $= O(n \log n)$
  - \* - 递归深度： $\log n$
  - \* - 每层统计和合并： $O(n)$
  - \*
- \* - 空间复杂度详细计算：
  - \*  $S(n) = O(n) + O(\log n)$
  - \* -  $O(n)$ ：辅助数组 help
  - \* -  $O(\log n)$ ：递归调用栈
  - \* 总计： $O(n)$
  - \*
- \* - 是否最优解：★ 是 ★
  - \* 理由：基于比较的算法下界为  $O(n \log n)$ ，本算法已达到最优
  - \*
- \* =====
- \* 相关题目列表（基于归并排序的统计问题）
  - \* =====
  - \* 1. LeetCode 315 - 计算右侧小于当前元素的个数
    - \* <https://leetcode.cn/problems/count-of-smaller-numbers-after-self/>
    - \* 问题：统计每个元素右侧比它小的元素个数
    - \* 解法：归并排序过程中记录元素原始索引，统计右侧小于当前元素的数量
    - \*
  - \* 2. LeetCode 493 - 翻转对
    - \* <https://leetcode.cn/problems/reverse-pairs/>
    - \* 问题：统计满足  $nums[i] > 2*nums[j]$  且  $i < j$  的对的数量
    - \* 解法：归并排序过程中使用双指针统计跨越左右区间的翻转对
    - \*
  - \* 3. LeetCode 327 - 区间和的个数
    - \* <https://leetcode.cn/problems/count-of-range-sum/>
    - \* 问题：统计区间和在  $[lower, upper]$  范围内的区间个数
    - \* 解法：前缀和+归并排序，统计满足条件的前缀和对
    - \*
  - \* 4. 剑指 Offer 51 / LCR 170 - 数组中的逆序对
    - \* <https://leetcode.cn/problems/shu-zu-zhong-de-ni-xu-dui-lcof/>
    - \* 问题：统计数组中逆序对的总数
    - \* 解法：归并排序过程中统计逆序对数量
    - \*
  - \* 5. POJ 2299 - Ultra-QuickSort
    - \* <http://poj.org/problem?id=2299>

- \* 问题：计算将数组排序所需的最小交换次数（即逆序对数量）
- \* 解法：归并排序统计逆序对
- \*
- \* 6. HDU 1394 – Minimum Inversion Number
  - \* <http://acm.hdu.edu.cn/showproblem.php?pid=1394>
  - \* 问题：将数组循环左移，求所有可能排列中的最小逆序对数量
  - \* 解法：归并排序+逆序对性质分析
  - \*
- \* 7. HackerRank – Merge Sort: Counting Inversions
  - \* <https://www.hackerrank.com/challenges/merge-sort/problem>
  - \* 问题：统计逆序对数量
  - \* 解法：归并排序统计逆序对
  - \*
- \* 8. SPOJ – INVCNT
  - \* <https://www.spoj.com/problems/INVCNT/>
  - \* 问题：统计逆序对数量
  - \* 解法：归并排序统计逆序对
  - \*
- \* 9. CodeChef – INVCNT
  - \* <https://www.codechef.com/problems/INVCNT>
  - \* 问题：统计逆序对数量
  - \* 解法：归并排序或树状数组
  - \*
- \* 10. UVa 10810 – Ultra-QuickSort
  - \*

[https://onlinejudge.org/index.php?option=com\\_onlinejudge&Itemid=8&page=show\\_problem&problem=1751](https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=1751)

- \* 问题：计算逆序对数量
- \* 解法：归并排序统计逆序对
- \*
- \* 这些题目虽然具体形式不同，但核心思想都是利用归并排序的分治特性，在合并过程中高效统计满足特定条件的元素对数量。

\*/

```
const int MAXN = 500001;
int n;
long long arr[MAXN];
long long help[MAXN];

/***
 * 归并排序并统计逆序对数量
 *
 * @param l 左边界
 * @param r 右边界
 */
```

```

* @return 区间[1, r]中的逆序对数量
*/
long long mergeSort(int l, int r) {
    if (l == r) {
        return 0;
    }

    int m = l + (r - 1) / 2;
    // 分治: 左半部分逆序对 + 右半部分逆序对 + 跨越两部分的逆序对
    return mergeSort(l, m) + mergeSort(m + 1, r) + merge(l, m, r);
}

/**
 * 合并两个有序子数组并统计逆序对数量
 *
 * @param l 左边界
 * @param m 中点
 * @param r 右边界
 * @return 跨越[l, m]和[m+1, r]的逆序对数量
 */
long long merge(int l, int m, int r) {
    long long count = 0;
    int i = l;      // help 数组的当前位置
    int a = l;      // 左侧数组指针
    int b = m + 1; // 右侧数组指针

    // 合并过程, 同时统计逆序对
    while (a <= m && b <= r) {
        if (arr[a] <= arr[b]) {
            // 左侧元素小于等于右侧元素
            // 右侧数组中已处理的元素(b - (m+1))个都小于 arr[a], 构成逆序对
            count += (b - m - 1);
            help[i++] = arr[a++];
        } else {
            // 右侧元素小于左侧元素
            help[i++] = arr[b++];
        }
    }

    // 处理左侧剩余元素
    while (a <= m) {
        // 左侧剩余元素与右侧所有元素都构成逆序对
        count += (b - m - 1);
    }
}

```

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

// 处理右侧剩余元素
while (b <= r) {
    help[i++] = arr[b++];
}

// 将 help 数组拷贝回原数组
for (i = l; i <= r; i++) {
    arr[i] = help[i];
}

return count;
}

/***
 * 主函数 - 处理输入输出
 *
 * 输入处理优化:
 * 使用 scanf 提高输入效率
 * 对于大规模数据(500000 个元素), 这种优化非常必要
 */
int main() {
    // I/O 优化
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    // 读取数组长度
    scanf("%d", &n);

    // 读取数组元素
    for (int i = 0; i < n; i++) {
        scanf("%lld", &arr[i]);
    }

    // 计算并输出逆序对数量
    printf("%lld\n", mergeSort(0, n - 1));

    return 0;
}

/*
```

```
* =====
* C++语言特有关注事项
* =====
* 1. 数据类型溢出问题:
*   - 逆序对数量可能超过 int 范围, 使用 long long 类型存储结果
*   - 当 n=500000 时, 最坏情况下逆序对数量可达  $n*(n-1)/2 \approx 1.25*10^{11}$ , 超出 int 范围
*   - 数组元素范围为  $[-10^9, 10^9]$ , 使用 long long 存储确保不会溢出
*
* 2. 输入输出优化:
*   - 使用 scanf/printf 而非 cin/cout 提高输入输出效率
*   - 使用 ios::sync_with_stdio(false) 关闭 C++ 与 C 的 I/O 同步
*   - 使用 cin.tie(nullptr) 解除 cin 与 cout 的绑定
*
* 3. 内存管理:
*   - 使用全局数组避免频繁内存分配
*   - MAXN 设为 500001, 满足题目要求
*   - 静态数组在栈上分配, 访问速度快
*
* 4. 递归深度:
*   - 归并排序递归深度为  $\log_2(500000) \approx 19$  层, 不会超出 C++ 默认栈限制
*
* 5. 位运算优化:
*   - 可使用  $(l+r) \gg 1$  代替  $(l+r)/2$  提高运算效率
*   - 注意当 l 和 r 都很大时,  $(l+r)$  可能导致溢出, 应改为  $1 + ((r-l) \gg 1)$ 
*
* =====
* 工程化考量
* =====
* 1. 性能优化:
*   - 对于小规模子数组(如 n<10), 可考虑使用插入排序
*   - 可添加判断: 当 arr[m] <= arr[m+1] 时, 子数组已有序, 可跳过合并
*
* 2. 错误处理:
*   - 可添加输入验证, 检查 n 是否在合法范围内
*   - 可添加文件结束检查, 处理输入异常
*
* 3. 可扩展性:
*   - 算法易于扩展到其他统计问题(如翻转对、小和问题)
*   - 可封装为函数库供其他程序调用
*
* 4. 编译优化:
*   - 可使用 -O2 编译选项优化性能
*   - 可使用 -march=native 利用 CPU 特性进一步优化
```

```
*  
* 5. 测试策略:  
*   - 应包含边界测试(空数组、单元素、全相同元素等)  
*   - 应包含性能测试(大规模数据)  
*   - 应包含正确性测试(已知结果的测试用例)  
*/
```

---

文件: Code08\_LuoguP1908.java

```
=====  
=====  
package class022;  
  
/**  
 * =====  
 * 题目 8: 洛谷 P1908 – 逆序对 (Inversion Pairs)  
 * =====  
 *  
 * 题目来源: 洛谷  
 * 题目链接: https://www.luogu.com.cn/problem/P1908  
 * 难度级别: 普及+/提高  
 *  
 * 问题描述:  
 * 给定一个序列, 求出这个序列的逆序对个数。  
 * 逆序对定义: 对于序列中的两个元素  $a[i]$  和  $a[j]$ , 如果  $i < j$  且  $a[i] > a[j]$ , 则称这两个元素构成一个逆序对。  
 *  
 * 输入格式:  
 * 第一行, 一个整数  $n$ , 表示序列长度。  
 * 第二行,  $n$  个整数, 表示给定的序列。  
 *  
 * 输出格式:  
 * 输出一个整数, 表示序列中逆序对的个数。  
 *  
 * 示例输入输出:  
 * 输入:  
 * 6  
 * 5 4 2 6 3 1  
 * 输出:  
 * 11  
 *  
 * 数据范围:  
 * 对于 60% 的数据,  $n \leq 1000$ 
```

\* 对于 100%的数据， $n \leq 500000$ ，序列中每个数的绝对值不超过  $10^9$

\*

\* =====

\* 核心算法思想：归并排序分治统计

\* =====

\*

\* 方法 1：暴力解法（不推荐）

\* - 思路：双重循环遍历所有  $i < j$  的情况，判断  $\text{nums}[i] > \text{nums}[j]$

\* - 时间复杂度： $O(N^2)$  - 双重循环

\* - 空间复杂度： $O(1)$  - 不需要额外空间

\* - 问题：数据量大时超时

\*

\* 方法 2：归并排序思想（最优解）★★★★★

\* - 核心洞察：利用归并排序的分治过程，在合并两个有序子数组之前，统计左侧子数组中每个元素与右侧子数组形成的逆序对数量

\*

\* - 归并排序过程：

\* 1. 分治：将数组不断二分，直到只有一个元素

\* 2. 统计：在合并前，统计左侧子数组中每个元素能与右侧子数组形成的逆序对数量

\* 3. 合并：合并两个有序子数组

\*

\* - 统计逆序对的关键步骤：

\* - 当右子数组中的元素被选中时，不会对左侧元素产生影响

\* - 当左子数组中的元素被选中时，右子数组中已处理的元素都小于它，形成逆序对

\* - 因此，每次选中左子数组元素时，需要累加右侧已处理的元素数量

\*

\* - 时间复杂度详细计算：

\*  $T(n) = 2T(n/2) + O(n)$  [Master 定理 case 2]

\* =  $O(n \log n)$

\* - 递归深度： $\log n$

\* - 每层统计和合并： $O(n)$

\*

\* - 空间复杂度详细计算：

\*  $S(n) = O(n) + O(\log n)$

\* -  $O(n)$ ：辅助数组 help

\* -  $O(\log n)$ ：递归调用栈

\* 总计： $O(n)$

\*

\* - 是否最优解：★ 是 ★

\* 理由：基于比较的算法下界为  $O(n \log n)$ ，本算法已达到最优

\*

\* =====

\* 相关题目列表（基于归并排序的统计问题）

- \* =====
- \* 1. LeetCode 315 - 计算右侧小于当前元素的个数
- \* <https://leetcode.cn/problems/count-of-smaller-numbers-after-self/>
- \* 问题: 统计每个元素右侧比它小的元素个数
- \* 解法: 归并排序过程中记录元素原始索引, 统计右侧小于当前元素的数量
- \*
- \* 2. LeetCode 493 - 翻转对
- \* <https://leetcode.cn/problems/reverse-pairs/>
- \* 问题: 统计满足  $\text{nums}[i] > 2 * \text{nums}[j]$  且  $i < j$  的对的数量
- \* 解法: 归并排序过程中使用双指针统计跨越左右区间的翻转对
- \*
- \* 3. LeetCode 327 - 区间和的个数
- \* <https://leetcode.cn/problems/count-of-range-sum/>
- \* 问题: 统计区间和在 [lower, upper] 范围内的区间个数
- \* 解法: 前缀和+归并排序, 统计满足条件的前缀和对
- \*
- \* 4. 剑指 Offer 51 / LCR 170 - 数组中的逆序对
- \* <https://leetcode.cn/problems/shu-zu-zhong-de-ni-xu-dui-lcof/>
- \* 问题: 统计数组中逆序对的总数
- \* 解法: 归并排序过程中统计逆序对数量
- \*
- \* 5. POJ 2299 - Ultra-QuickSort
- \* <http://poj.org/problem?id=2299>
- \* 问题: 计算将数组排序所需的最小交换次数 (即逆序对数量)
- \* 解法: 归并排序统计逆序对
- \*
- \* 6. HDU 1394 - Minimum Inversion Number
- \* <http://acm.hdu.edu.cn/showproblem.php?pid=1394>
- \* 问题: 将数组循环左移, 求所有可能排列中的最小逆序对数量
- \* 解法: 归并排序+逆序对性质分析
- \*
- \* 7. HackerRank - Merge Sort: Counting Inversions
- \* <https://www.hackerrank.com/challenges/merge-sort/problem>
- \* 问题: 统计逆序对数量
- \* 解法: 归并排序统计逆序对
- \*
- \* 8. SPOJ - INVCNT
- \* <https://www.spoj.com/problems/INVCNT/>
- \* 问题: 统计逆序对数量
- \* 解法: 归并排序统计逆序对
- \*
- \* 9. CodeChef - INVCNT
- \* <https://www.codechef.com/problems/INVCNT>

```

* 问题：统计逆序对数量
* 解法：归并排序或树状数组
*
* 10. UVa 10810 - Ultra-QuickSort
*
https://onlinejudge.org/index.php?option=com\_onlinejudge&Itemid=8&page=show\_problem&problem=1751
* 问题：计算逆序对数量
* 解法：归并排序统计逆序对
*
* 这些题目虽然具体形式不同，但核心思想都是利用归并排序的分治特性，在合并过程中高效统计满足特定条件的元素对数量。
*/
import java.io.*;
import java.util.*;

public class Code08_LuoguP1908 {
    public static int MAXN = 500001;
    public static int[] arr = new int[MAXN];
    public static int[] help = new int[MAXN];

    /**
     * 计算数组中逆序对的数量
     *
     * @param n 数组长度
     * @return 逆序对的数量
     *
     * 算法思路：
     * 使用归并排序的思想，在合并两个有序子数组的过程中统计逆序对数量。
     * 当从左侧子数组选取元素时，右侧子数组中已处理的元素都小于该元素，
     * 因此这些元素与当前元素构成逆序对。
     */
    public static long countInversions(int n) {
        return mergeSort(0, n - 1);
    }

    /**
     * 归并排序并统计逆序对数量
     *
     * @param l 左边界
     * @param r 右边界
     * @return 区间[1, r]中的逆序对数量
     */
    public static long mergeSort(int l, int r) {

```

```

if (l == r) {
    return 0;
}

int m = l + (r - 1) / 2;
// 分治: 左半部分逆序对 + 右半部分逆序对 + 跨越两部分的逆序对
return mergeSort(l, m) + mergeSort(m + 1, r) + merge(l, m, r);
}

/***
 * 合并两个有序子数组并统计逆序对数量
 *
 * @param l 左边界
 * @param m 中点
 * @param r 右边界
 * @return 跨越[l, m]和[m+1, r]的逆序对数量
 */
public static long merge(int l, int m, int r) {
    long count = 0;
    int i = l;      // help 数组的当前位置
    int a = l;      // 左侧数组指针
    int b = m + 1;  // 右侧数组指针

    // 合并过程, 同时统计逆序对
    while (a <= m && b <= r) {
        if (arr[a] <= arr[b]) {
            // 左侧元素小于等于右侧元素
            // 右侧数组中已处理的元素(b - (m+1))个都小于 arr[a], 构成逆序对
            count += (b - m - 1);
            help[i++] = arr[a++];
        } else {
            // 右侧元素小于左侧元素
            help[i++] = arr[b++];
        }
    }

    // 处理左侧剩余元素
    while (a <= m) {
        // 左侧剩余元素与右侧所有元素都构成逆序对
        count += (b - m - 1);
        help[i++] = arr[a++];
    }
}

```

```

// 处理右侧剩余元素
while (b <= r) {
    help[i++] = arr[b++];
}

// 将 help 数组拷贝回原数组
for (i = l; i <= r; i++) {
    arr[i] = help[i];
}

return count;
}

/***
 * 主函数 - 处理输入输出
 *
 * 输入处理优化:
 * 使用 BufferedReader 和 StreamTokenizer 提高输入效率
 * 对于大规模数据(500000 个元素), 这种优化非常必要
 */
public static void main(String[] args) throws IOException {
    // 使用高效 IO 处理
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    StreamTokenizer in = new StreamTokenizer(br);
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

    // 读取数组长度
    in.nextToken();
    int n = (int) in.nval;

    // 读取数组元素
    for (int i = 0; i < n; i++) {
        in.nextToken();
        arr[i] = (int) in.nval;
    }

    // 计算并输出逆序对数量
    out.println(countInversions(n));
    out.flush();
    out.close();
}

/*

```

```
* =====
* Java 语言特有关注事项
* =====
* 1. 数据类型溢出问题:
*   - 逆序对数量可能超过 int 范围, 使用 long 类型存储结果
*   - 当 n=500000 时, 最坏情况下逆序对数量可达  $n*(n-1)/2 \approx 1.25*10^{11}$ , 超出 int 范围
*
* 2. 输入输出优化:
*   - 使用 StreamTokenizer 和 BufferedReader 提高输入效率
*   - 使用 PrintWriter 提高输出效率
*   - 对于大规模数据, Scanner 类效率较低
*
* 3. 内存管理:
*   - 使用静态数组避免频繁内存分配
*   - MAXN 设为 500001, 满足题目要求
*
* 4. 递归深度:
*   - 归并排序递归深度为  $\log_2(500000) \approx 19$  层, 不会超出 Java 默认栈限制
*
* 5. 边界条件处理:
*   - 空数组、单元素数组都有正确的处理
*   - 数组元素可能为负数或 0, 算法仍正确
*
* =====
* 工程化考量
* =====
* 1. 性能优化:
*   - 对于小规模子数组(如 n<10), 可考虑使用插入排序
*   - 可添加判断: 当 arr[m] <= arr[m+1] 时, 子数组已有序, 可跳过合并
*
* 2. 错误处理:
*   - 可添加输入验证, 检查 n 是否在合法范围内
*   - 可添加异常处理, 处理输入格式错误
*
* 3. 可扩展性:
*   - 算法易于扩展到其他统计问题(如翻转对、小和问题)
*   - 可封装为工具类供其他程序调用
*
* 4. 测试策略:
*   - 应包含边界测试(空数组、单元素、全相同元素等)
*   - 应包含性能测试(大规模数据)
*   - 应包含正确性测试(已知结果的测试用例)
*/

```

}

=====

文件: Code08\_LuoguP1908.py

=====

```
# 洛谷 P1908 - 逆序对 (Inversion Pairs)
# 题目来源: 洛谷
# 题目链接: https://www.luogu.com.cn/problem/P1908
# 难度级别: 普及+/提高
```

,,

=====

题目 8: 洛谷 P1908 - 逆序对 (Inversion Pairs)

=====

题目来源: 洛谷

题目链接: <https://www.luogu.com.cn/problem/P1908>

难度级别: 普及+/提高

问题描述:

给定一个序列, 求出这个序列的逆序对个数。

逆序对定义: 对于序列中的两个元素  $a[i]$  和  $a[j]$ , 如果  $i < j$  且  $a[i] > a[j]$ , 则称这两个元素构成一个逆序对。

输入格式:

第一行, 一个整数  $n$ , 表示序列长度。

第二行,  $n$  个整数, 表示给定的序列。

输出格式:

输出一个整数, 表示序列中逆序对的个数。

示例输入输出:

输入:

6

5 4 2 6 3 1

输出:

11

数据范围:

对于 60% 的数据,  $n \leq 1000$

对于 100% 的数据,  $n \leq 500000$ , 序列中每个数的绝对值不超过  $10^9$

=====

方法 1：暴力解法（不推荐）

- 思路：双重循环遍历所有  $i < j$  的情况，判断  $\text{nums}[i] > \text{nums}[j]$
- 时间复杂度： $O(N^2)$  - 双重循环
- 空间复杂度： $O(1)$  - 不需要额外空间
- 问题：数据量大时超时

方法 2：归并排序思想（最优解）★★★★★

- 核心洞察：利用归并排序的分治过程，在合并两个有序子数组之前，统计左侧子数组中每个元素与右侧子数组形成的逆序对数量
- 归并排序过程：
  1. 分治：将数组不断二分，直到只有一个元素
  2. 统计：在合并前，统计左侧子数组中每个元素能与右侧子数组形成的逆序对数量
  3. 合并：合并两个有序子数组
- 统计逆序对的关键步骤：
  - 当右子数组中的元素被选中时，不会对左侧元素产生影响
  - 当左子数组中的元素被选中时，右子数组中已处理的元素都小于它，形成逆序对
  - 因此，每次选中左子数组元素时，需要累加右侧已处理的元素数量
- 时间复杂度详细计算：
$$\begin{aligned} T(n) &= 2T(n/2) + O(n) \quad [\text{Master 定理 case 2}] \\ &= O(n \log n) \end{aligned}$$
  - 递归深度： $\log n$
  - 每层统计和合并： $O(n)$
- 空间复杂度详细计算：
$$S(n) = O(n) + O(\log n)$$
  - $O(n)$ ：辅助数组 help
  - $O(\log n)$ ：递归调用栈总计： $O(n)$
- 是否最优解：★ 是 ★  
理由：基于比较的算法下界为  $O(n \log n)$ ，本算法已达到最优

---

相关题目列表（基于归并排序的统计问题）

---

1. LeetCode 315 – 计算右侧小于当前元素的个数  
<https://leetcode.cn/problems/count-of-smaller-numbers-after-self/>

问题：统计每个元素右侧比它小的元素个数

解法：归并排序过程中记录元素原始索引，统计右侧小于当前元素的数量

2. LeetCode 493 – 翻转对

<https://leetcode.cn/problems/reverse-pairs/>

问题：统计满足  $\text{nums}[i] > 2 * \text{nums}[j]$  且  $i < j$  的对的数量

解法：归并排序过程中使用双指针统计跨越左右区间的翻转对

3. LeetCode 327 – 区间和的个数

<https://leetcode.cn/problems/count-of-range-sum/>

问题：统计区间和在 [lower, upper] 范围内的区间个数

解法：前缀和+归并排序，统计满足条件的前缀和对

4. 剑指 Offer 51 / LCR 170 – 数组中的逆序对

<https://leetcode.cn/problems/shu-zu-zhong-de-ni-xu-dui-lcof/>

问题：统计数组中逆序对的总数

解法：归并排序过程中统计逆序对数量

5. POJ 2299 – Ultra-QuickSort

<http://poj.org/problem?id=2299>

问题：计算将数组排序所需的最小交换次数（即逆序对数量）

解法：归并排序统计逆序对

6. HDU 1394 – Minimum Inversion Number

<http://acm.hdu.edu.cn/showproblem.php?pid=1394>

问题：将数组循环左移，求所有可能排列中的最小逆序对数量

解法：归并排序+逆序对性质分析

7. HackerRank – Merge Sort: Counting Inversions

<https://www.hackerrank.com/challenges/merge-sort/problem>

问题：统计逆序对数量

解法：归并排序统计逆序对

8. SPOJ – INVCTN

<https://www.spoj.com/problems/INVCTN/>

问题：统计逆序对数量

解法：归并排序统计逆序对

9. CodeChef – INVCTN

<https://www.codechef.com/problems/INVCTN>

问题：统计逆序对数量

解法：归并排序或树状数组

## 10. UVa 10810 - Ultra-QuickSort

[https://onlinejudge.org/index.php?option=com\\_onlinejudge&Itemid=8&page=show\\_problem&problem=1751](https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=1751)

问题：计算逆序对数量

解法：归并排序统计逆序对

这些题目虽然具体形式不同，但核心思想都是利用归并排序的分治特性，在合并过程中高效统计满足特定条件的元素对数量。

”，

```
def count_inversions(arr):
    """
    计算数组中逆序对的数量 - Python 实现
```

Args:

arr: 输入数组

Returns:

int: 逆序对的数量

算法思路：

使用归并排序的思想，在合并两个有序子数组的过程中统计逆序对数量。  
当从左侧子数组选取元素时，右侧子数组中已处理的元素都小于该元素，  
因此这些元素与当前元素构成逆序对。

时间复杂度： $O(n \log n)$

空间复杂度： $O(n)$

”，

```
# 创建数组副本，避免修改原数组
```

```
arr_copy = arr[:]
```

```
n = len(arr_copy)
```

```
# 辅助数组
```

```
help_arr = [0] * n
```

```
def merge_sort(l, r):
```

”，

归并排序并统计逆序对数量

Args:

l: 左边界

r: 右边界

Returns:

```
    int: 区间[1, r]中的逆序对数量
"""
if l == r:
    return 0

m = (l + r) // 2
# 分治: 左半部分逆序对 + 右半部分逆序对 + 跨越两部分的逆序对
return merge_sort(l, m) + merge_sort(m + 1, r) + merge(l, m, r)
```

```
def merge(l, m, r):
```

"""

合并两个有序子数组并统计逆序对数量

Args:

l: 左边界

m: 中点

r: 右边界

Returns:

```
    int: 跨越[l, m]和[m+1, r]的逆序对数量
"""
count = 0
i = l      # help_arr 数组的当前位置
a = l      # 左侧数组指针
b = m + 1  # 右侧数组指针
```

# 合并过程, 同时统计逆序对

```
while a <= m and b <= r:
```

```
    if arr_copy[a] <= arr_copy[b]:
```

# 左侧元素小于等于右侧元素

# 右侧数组中已处理的元素(b - (m+1))个都小于 arr\_copy[a], 构成逆序对

```
        count += (b - m - 1)
```

```
        help_arr[i] = arr_copy[a]
```

```
        a += 1
```

```
    else:
```

# 右侧元素小于左侧元素

```
        help_arr[i] = arr_copy[b]
```

```
        b += 1
```

```
        i += 1
```

# 处理左侧剩余元素

```
while a <= m:
```

```
# 左侧剩余元素与右侧所有元素都构成逆序对
count += (b - m - 1)
help_arr[i] = arr_copy[a]
a += 1
i += 1

# 处理右侧剩余元素
while b <= r:
    help_arr[i] = arr_copy[b]
    b += 1
    i += 1

# 将 help_arr 数组拷贝回原数组
for i in range(1, r + 1):
    arr_copy[i] = help_arr[i]

return count

return merge_sort(0, n - 1)

def main():
    """
    主函数 - 处理输入输出
    """
    import sys
    # 调整递归深度限制以处理大规模数据
    sys.setrecursionlimit(1000000)

    # 读取输入
    n = int(input())
    arr = list(map(int, input().split()))

    # 计算并输出逆序对数量
    result = count_inversions(arr)
    print(result)

# =====
# Python 语言特有关注事项
# =====
#
# 1. 递归深度限制:
```

```
#     - Python 默认递归深度限制约为 1000 层
#     - 对于 n=500000 的数据，归并排序递归深度约为  $\log_2(500000) \approx 19$  层，不会超出限制
#     - 但为保险起见，使用 sys.setrecursionlimit(1000000) 设置更大的限制
#
# 2. 整数精度：
#     - Python 的整数类型自动支持大整数，不会有溢出问题
#     - 逆序对数量可能很大，但 Python 能正确处理
#
# 3. 列表操作效率：
#     - 使用索引操作而非切片，避免创建新列表的开销
#     - 预先创建辅助数组，避免在递归中重复创建
#
# 4. 输入输出：
#     - 使用 input() 和 print() 处理标准输入输出
#     - 对于大规模数据，Python 的 I/O 效率可能不如 C++ 和 Java
#
# =====
# 工程化考量
# =====
#
# 1. 性能优化：
#     - 对于小规模子数组（如  $n < 10$ ），可考虑使用插入排序
#     - 可添加判断：当  $arr[m] \leq arr[m+1]$  时，子数组已有序，可跳过合并
#
# 2. 错误处理：
#     - 可添加输入验证，检查  $n$  是否在合法范围内
#     - 可添加异常处理，处理输入格式错误
#
# 3. 可扩展性：
#     - 算法易于扩展到其他统计问题（如翻转对、小和问题）
#     - 可封装为模块供其他程序调用
#
# 4. 测试策略：
#     - 应包含边界测试（空数组、单元素、全相同元素等）
#     - 应包含性能测试（大规模数据）
#     - 应包含正确性测试（已知结果的测试用例）
```

```
if __name__ == "__main__":
    main()
```

```
=====
package class022;

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

/**
 * =====
 * 题目 9: LeetCode 315 - 计算右侧小于当前元素的个数
 * =====
 *
 * 题目来源: LeetCode
 * 题目链接: https://leetcode.cn/problems/count-of-smaller-numbers-after-self/
 * 难度级别: 困难
 *
 * 问题描述:
 * 给你一个整数数组 nums，按要求返回一个新数组 counts。
 * 数组 counts 有该性质: counts[i] 的值是 nums[i] 右侧小于 nums[i] 的元素的数量。
 *
 * 示例输入输出:
 * 输入: nums = [5, 2, 6, 1]
 * 输出: [2, 1, 1, 0]
 *
 * 解释:
 * - 对于 nums[0]=5，右侧小于 5 的元素有 2 和 1，所以 counts[0]=2
 * - 对于 nums[1]=2，右侧小于 2 的元素有 1，所以 counts[1]=1
 * - 对于 nums[2]=6，右侧小于 6 的元素有 1，所以 counts[2]=1
 * - 对于 nums[3]=1，右侧没有元素，所以 counts[3]=0
 *
 * =====
 * 核心算法思想: 归并排序+索引映射
 * =====
 *
 * 方法 1: 暴力解法 (不推荐)
 * 思路: 双重循环检查每个元素右侧有多少元素比它小
 * 时间复杂度: O(N^2) - 双重循环
 * 空间复杂度: O(N) - 结果数组
 * 问题: 数据量大时超时
 *
 * 方法 2: 归并排序思想 (最优解) ★★★★★
 * 核心洞察:
 * 1. 利用归并排序的分治过程统计元素之间的大小关系
 * 2. 关键挑战: 归并排序会改变元素顺序，需要维护原始索引
 * 3. 解决方案: 创建索引数组，对索引进行排序而非对值排序
```

\*

\* - 归并排序过程:

\* 1. 分治: 将数组不断二分, 直到只有一个元素

\* 2. 统计: 在合并过程中统计右侧小于当前元素的数量

\* 3. 合并: 按值的大小合并两个有序子数组

\*

\* - 统计右侧小元素的关键步骤:

\* - 当右子数组中的元素被选中时, 不会对左侧元素产生影响

\* - 当左子数组中的元素被选中时, 右子数组中剩余的所有元素都是比它小的

\* - 因此, 每次选中左子数组元素时, 需要记录右侧已经统计过的元素数量

\*

\* - 时间复杂度详细计算:

\*  $T(n) = 2T(n/2) + O(n)$  [Master 定理 case 2]

\* =  $O(n \log n)$

\* - 递归深度:  $\log n$

\* - 每层合并与统计:  $O(n)$

\*

\* - 空间复杂度详细计算:

\*  $S(n) = O(n) + O(\log n)$

\* -  $O(n)$ : 辅助数组、索引数组、结果数组

\* -  $O(\log n)$ : 递归调用栈

\* 总计:  $O(n)$

\*

\* - 是否最优解: ★ 是 ★

\* 理由: 基于比较的算法下界为  $O(n \log n)$ , 本算法已达到最优

\*

\* =====

\* 相关题目列表 (同类算法)

\* =====

\* 1. LeetCode 493 - 翻转对

\* <https://leetcode.cn/problems/reverse-pairs/>

\* 问题: 统计满足  $nums[i] > 2*nums[j]$  且  $i < j$  的对的数量

\* 解法: 归并排序过程中使用双指针统计跨越左右区间的翻转对

\*

\* 2. LeetCode 327 - 区间和的个数

\* <https://leetcode.cn/problems/count-of-range-sum/>

\* 问题: 统计区间和在  $[lower, upper]$  范围内的区间个数

\* 解法: 前缀和+归并排序, 统计满足条件的前缀和对

\*

\* 3. 剑指 Offer 51 / LCR 170 - 数组中的逆序对

\* <https://leetcode.cn/problems/shu-zu-zhong-de-ni-xu-dui-lcof/>

\* 问题: 统计数组中逆序对的总数

\* 解法: 归并排序过程中统计逆序对数量

\*

\* 4. LeetCode 1365 - 有多少小于当前数字的数字  
\* <https://leetcode.cn/problems/how-many-numbers-are-smaller-than-the-current-number/>  
\* 问题：统计数组中小于当前数字的数字个数（全数组范围）  
\* 解法：排序+哈希表映射  
\*

\* 5. POJ 2299 - Ultra-QuickSort  
\* <http://poj.org/problem?id=2299>  
\* 问题：计算将数组排序所需的最小交换次数（即逆序对数量）  
\* 解法：归并排序统计逆序对  
\*

\* 6. HackerRank - Merge Sort: Counting Inversions  
\* <https://www.hackerrank.com/challenges/merge-sort/problem>  
\* 问题：统计逆序对数量  
\* 解法：归并排序统计逆序对  
\*

\* 7. 牛客网 - 计算右侧小于当前元素的个数  
\* 问题：与 LeetCode 315 相同  
\* 解法：归并排序+索引映射  
\*

\* 8. 杭电 OJ - 1394  
\* <http://acm.hdu.edu.cn/showproblem.php?pid=1394>  
\* 问题：将数组循环左移，求所有可能排列中的最小逆序对数量  
\* 解法：归并排序+逆序对性质分析  
\*

\* 9. 洛谷 P1908 - 逆序对  
\* <https://www.luogu.com.cn/problem/P1908>  
\* 问题：统计数组中逆序对的总数  
\* 解法：归并排序统计逆序对  
\*

\* 10. SPOJ - INVCNT  
\* <https://www.spoj.com/problems/INVCNT/>  
\* 问题：统计逆序对数量  
\* 解法：归并排序统计逆序对  
\*

\* 这些题目虽然具体形式不同，但核心思想都是利用归并排序的分治特性，在合并过程中高效统计满足特定条件的元素对数量。

\*/

```
public class Code09_LeetCode315 {
```

/\*\*

```
 * Pair 类用于在归并排序过程中同时保存元素值和原始索引  
 * 这样可以在排序过程中维护原始数组的位置信息
```

```

*/
static class Pair {
    int val; // 元素值
    int idx; // 元素在原始数组中的索引

    /**
     * 构造函数
     * @param val 元素值
     * @param idx 原始索引
     */
    Pair(int val, int idx) {
        this.val = val;
        this.idx = idx;
    }
}

// 常量定义
public static final int MAXN = 100001;

// 全局数组，避免多次内存分配
public static Pair[] arr = new Pair[MAXN]; // 原数组，保存值和索引
public static Pair[] help = new Pair[MAXN]; // 辅助数组，用于归并过程
public static int[] count = new int[MAXN]; // 结果数组，存储每个元素右侧小于它的元素个数

/**
 * 计算右侧小于当前元素的个数的主方法
 *
 * @param nums 输入整数数组
 * @return 包含每个元素右侧小于它的元素个数的列表
 *
 * Java 语言特性注意事项：
 * 1. 数组是引用类型，作为参数传递时传递的是引用副本
 * 2. 使用全局数组避免了频繁的内存分配和释放
 * 3. 方法使用静态修饰符，可以直接通过类名调用
 */
public static List<Integer> countSmaller(int[] nums) {
    int n = nums.length;
    // 初始化 Pair 数组，将元素值和索引关联起来
    for (int i = 0; i < n; i++) {
        arr[i] = new Pair(nums[i], i);
    }

    // 重置计数数组

```

```
for (int i = 0; i < n; i++) {
    count[i] = 0;
}

// 执行归并排序并统计
mergeSort(0, n - 1);

// 转换结果格式
List<Integer> result = new ArrayList<>(n);
for (int i = 0; i < n; i++) {
    result.add(count[i]);
}
return result;
}

/***
 * 归并排序的核心方法
 *
 * @param l 当前处理区间的左边界
 * @param r 当前处理区间的右边界
 *
 * 算法分析:
 * - 递归实现归并排序
 * - 时间复杂度: O(n log n)
 * - 空间复杂度: O(n)
 */
public static void mergeSort(int l, int r) {
    // 基本情况: 区间只有一个元素时直接返回
    if (l == r) {
        return;
    }

    // 计算中间位置, 使用这种方式避免大整数溢出
    int m = l + (r - 1) / 2;

    // 分治: 递归处理左右子区间
    mergeSort(l, m);
    mergeSort(m + 1, r);

    // 合并两个有序子区间, 同时统计结果
    merge(l, m, r);
}
```

```

/**
 * 合并两个有序子数组，并在合并过程中统计右侧小于当前元素的个数
 *
 * @param l 当前处理区间的左边界
 * @param m 当前处理区间的中点
 * @param r 当前处理区间的右边界
 *
 * 核心统计逻辑：
 * - 当从左侧子数组选取元素时，右侧子数组中已处理的元素都小于该元素
 * - 因此需要将右侧已处理的元素数量累加到该元素的计数中
 */
public static void merge(int l, int m, int r) {
    int i = l; // 辅助数组的指针
    int a = l; // 左侧子数组的指针
    int b = m + 1; // 右侧子数组的指针

    // 合并两个子数组，同时统计右侧小于当前元素的个数
    while (a <= m && b <= r) {
        if (arr[a].val <= arr[b].val) {
            // 当左侧元素小于等于右侧元素时，右侧数组中已经处理的元素都小于当前左侧元素
            // 统计右侧已处理的元素数量：b - (m + 1) = b - m - 1
            count[arr[a].idx] += (b - m - 1);
            help[i++] = arr[a++];
        } else {
            // 当右侧元素小于左侧元素时，将右侧元素放入辅助数组
            // 此时不更新计数，因为左侧元素还未被处理
            help[i++] = arr[b++];
        }
    }

    // 处理左侧剩余元素
    while (a <= m) {
        // 左侧剩余元素的右侧所有元素都小于它
        count[arr[a].idx] += (b - m - 1);
        help[i++] = arr[a++];
    }

    // 处理右侧剩余元素
    while (b <= r) {
        help[i++] = arr[b++];
    }

    // 将辅助数组内容复制回原数组
}

```

```
for (i = 1; i <= r; i++) {
    arr[i] = help[i];
}
}

/***
 * 主方法，用于测试
 */
public static void main(String[] args) {
    // 测试用例 1: 基本情况
    int[] test1 = {5, 2, 6, 1};
    System.out.println("输入: " + java.util.Arrays.toString(test1));
    System.out.println("输出: " + countSmaller(test1)); // 预期输出: [2, 1, 1, 0]

    // 测试用例 2: 空数组
    int[] test2 = {};
    System.out.println("输入: " + java.util.Arrays.toString(test2));
    System.out.println("输出: " + countSmaller(test2)); // 预期输出: []

    // 测试用例 3: 单元素数组
    int[] test3 = {1};
    System.out.println("输入: " + java.util.Arrays.toString(test3));
    System.out.println("输出: " + countSmaller(test3)); // 预期输出: [0]

    // 测试用例 4: 逆序数组
    int[] test4 = {5, 4, 3, 2, 1};
    System.out.println("输入: " + java.util.Arrays.toString(test4));
    System.out.println("输出: " + countSmaller(test4)); // 预期输出: [4, 3, 2, 1, 0]

    // 测试用例 5: 有序数组
    int[] test5 = {1, 2, 3, 4, 5};
    System.out.println("输入: " + java.util.Arrays.toString(test5));
    System.out.println("输出: " + countSmaller(test5)); // 预期输出: [0, 0, 0, 0, 0]

    // 测试用例 6: 重复元素
    int[] test6 = {2, 2, 2};
    System.out.println("输入: " + java.util.Arrays.toString(test6));
    System.out.println("输出: " + countSmaller(test6)); // 预期输出: [0, 0, 0]

    // 测试用例 7: 包含负数
    int[] test7 = {-1, -2, 3, -4, 5};
    System.out.println("输入: " + java.util.Arrays.toString(test7));
    System.out.println("输出: " + countSmaller(test7)); // 预期输出: [2, 1, 1, 0, 0]
```

```
}

/*
 * =====
 * Java 语言特有关注事项
 * =====
 * 1. 静态成员变量的使用:
 *   - 使用静态数组避免了频繁的内存分配和释放
 *   - 但需注意线程安全问题, 多线程环境下可能需要额外的同步措施
 *   - 静态变量在类加载时初始化, 在类卸载时销毁
 *
 * 2. 内存优化:
 *   - 预分配固定大小的数组 (MAXN) 减少了动态扩容的开销
 *   - 使用循环初始化数组, 避免使用 Arrays.fill() 方法
 *   - ArrayList 的初始容量设置为数组大小, 避免扩容开销
 *
 * 3. 整数类型和溢出:
 *   - Java 的 int 类型是 32 位有符号整数, 范围为-2^31 到 2^31-1
 *   - 计算中间点使用  $1 + (r - 1) / 2$  避免整数溢出
 *   - 对于非常大的数组, 索引可能超出 int 范围, 需要考虑使用 long 类型
 *
 * 4. 递归深度控制:
 *   - Java 的默认递归深度限制约为 1000–2000 层 (依赖 JVM 实现)
 *   - 对于大规模数据, 可以通过 JVM 参数-Xss 调整栈大小
 *   - 对于极端情况, 考虑实现迭代版本的归并排序
 *
 * 5. 异常处理:
 *   - Java 中可以使用 try-catch 块处理可能的异常
 *   - 可以添加对 null 输入、极大数组等边界情况的检查
 *
 * 6. 性能优化:
 *   - 对于小规模子数组 (如长度<10), 可以使用插入排序提高效率
 *   - 可以添加判断条件, 当  $arr[m].val \leq arr[m+1].val$  时, 子数组已有序, 跳过合并
 *   - 减少对象创建和垃圾回收压力, 提高性能
 */


```

```
/*
 * =====
 * 工程化考量
 * =====
 * 1. 异常处理:
 *   - 可以添加对 null 输入的检查: if (nums == null) { return new ArrayList<>(); }
 *   - 对于极大数组, 可以添加数组长度检查, 避免超出 MAXN 限制

```

- \*    - 可以抛出 IllegalArgumentException 处理无效输入
- \*
- \* 2. 线程安全:
  - 当前实现使用静态变量，不是线程安全的
  - 改进方案 1：将静态变量改为方法局部变量，每次调用时创建新数组
  - 改进方案 2：使用 ThreadLocal 存储线程局部变量
  - 改进方案 3：添加同步机制（synchronized 或 ReentrantLock）
- \*
- \* 3. 测试与质量保证:
  - 已提供多种测试用例，覆盖常见场景
  - 建议使用 JUnit 框架编写自动化单元测试
  - 可以添加性能测试，验证算法在大规模数据下的表现
- \*
- \* 4. 代码可读性:
  - 使用了清晰的变量命名和方法命名
  - 添加了详细的注释说明算法逻辑
  - 代码结构清晰，易于理解和维护
- \*
- \* 5. 可扩展性:
  - 可以将算法扩展为泛型版本，支持任意 Comparable 类型
  - 可以将核心逻辑抽象为策略模式，方便替换不同的统计策略
  - 可以添加缓存机制，避免重复计算
- \*
- \* 6. 内存管理:
  - 使用预分配的静态数组减少 GC 压力
  - 对于超大规模数据，可以考虑分块处理或使用堆外内存
- \*
- \* 7. 性能调优:
  - 可以使用 JMH（Java Microbenchmark Harness）进行性能基准测试
  - 可以通过 JIT 编译器优化，避免热点路径上的对象创建
  - 考虑使用并行流或 Fork/Join 框架实现并行归并排序
- \*
- \* 8. 代码优化建议:
  - 对于大规模数据，考虑使用更高效的排序算法或数据结构，如二叉搜索树、树状数组等
  - 可以优化合并过程，减少不必要的对象创建
  - 对于特定应用场景，可以考虑启发式优化
- \*
- \* 9. 跨平台兼容性:
  - Java 具有良好的跨平台特性，代码可以在不同的操作系统和 JVM 上运行
  - 但需注意不同 JVM 实现可能在性能和行为上有所差异
- \*
- \* 10. 代码安全性:
  - 避免使用全局变量存储状态，减少副作用

```
*      - 对外部输入进行严格验证，防止注入攻击
*      - 注意数组边界检查，避免越界异常
*/
}
```

文件: Code09\_LeetCode315.py

```
# LeetCode 315 - 计算右侧小于当前元素的个数
# 题目来源: LeetCode
# 题目链接: https://leetcode.cn/problems/count-of-smaller-numbers-after-self/
# 难度级别: 困难
```

, , ,

题目 9: LeetCode 315 - 计算右侧小于当前元素的个数

题目来源: LeetCode

题目链接: https://leetcode.cn/problems/count-of-smaller-numbers-after-self/

难度级别: 困难

问题描述:

给你一个整数数组 `nums`，按要求返回一个新数组 `counts`。

数组 `counts` 有该性质: `counts[i]` 的值是 `nums[i]` 右侧小于 `nums[i]` 的元素的数量。

示例输入输出:

输入: `nums = [5, 2, 6, 1]`

输出: `[2, 1, 1, 0]`

解释:

- 对于 `nums[0]=5`, 右侧小于 5 的元素有 2 和 1, 所以 `counts[0]=2`
- 对于 `nums[1]=2`, 右侧小于 2 的元素有 1, 所以 `counts[1]=1`
- 对于 `nums[2]=6`, 右侧小于 6 的元素有 1, 所以 `counts[2]=1`
- 对于 `nums[3]=1`, 右侧没有元素, 所以 `counts[3]=0`

核心算法思想: 归并排序+索引映射

方法 1: 暴力解法 (不推荐)

- 思路: 双重循环检查每个元素右侧有多少元素比它小
- 时间复杂度:  $O(N^2)$  - 双重循环

- 空间复杂度:  $O(N)$  - 结果数组

- 问题: 数据量大时超时

方法 2: 归并排序思想 (最优解) ★★★★★

- 核心洞察:

1. 利用归并排序的分治过程统计元素之间的大小关系
2. 关键挑战: 归并排序会改变元素顺序, 需要维护原始索引
3. 解决方案: 创建索引数组, 对索引进行排序而非对值排序

- 归并排序过程:

1. 分治: 将数组不断二分, 直到只有一个元素
2. 统计: 在合并过程中统计右侧小于当前元素的数量
3. 合并: 按值的大小合并两个有序子数组

- 统计右侧小元素的关键步骤:

- 当右子数组中的元素被选中时, 不会对左侧元素产生影响
- 当左子数组中的元素被选中时, 右子数组中剩余的所有元素都是比它小的
- 因此, 每次选中左子数组元素时, 需要记录右侧已经统计过的元素数量

- 时间复杂度详细计算:

$$T(n) = 2T(n/2) + O(n) \quad [\text{Master 定理 case 2}]$$

$$= O(n \log n)$$

- 递归深度:  $\log n$

- 每层合并与统计:  $O(n)$

- 空间复杂度详细计算:

$$S(n) = O(n) + O(\log n)$$

-  $O(n)$ : 辅助数组、索引数组、结果数组

-  $O(\log n)$ : 递归调用栈

总计:  $O(n)$

- 是否最优解: ★ 是 ★

理由: 基于比较的算法下界为  $O(n \log n)$ , 本算法已达到最优

---

相关题目列表 (同类算法)

---

1. LeetCode 493 - 翻转对

<https://leetcode.cn/problems/reverse-pairs/>

问题: 统计满足  $\text{nums}[i] > 2*\text{nums}[j]$  且  $i < j$  的对的数量

解法: 归并排序过程中使用双指针统计跨越左右区间的翻转对

2. LeetCode 327 - 区间和的个数

<https://leetcode.cn/problems/count-of-range-sum/>

问题：统计区间和在[lower, upper]范围内的区间个数

解法：前缀和+归并排序，统计满足条件的前缀和对

3. 剑指 Offer 51 / LCR 170 - 数组中的逆序对

<https://leetcode.cn/problems/shu-zu-zhong-de-ni-xu-dui-lcof/>

问题：统计数组中逆序对的总数

解法：归并排序过程中统计逆序对数量

4. LeetCode 1365 - 有多少小于当前数字的数字

<https://leetcode.cn/problems/how-many-numbers-are-smaller-than-the-current-number/>

问题：统计数组中小于当前数字的数字个数（全数组范围）

解法：排序+哈希表映射

5. POJ 2299 - Ultra-QuickSort

<http://poj.org/problem?id=2299>

问题：计算将数组排序所需的最小交换次数（即逆序对数量）

解法：归并排序统计逆序对

6. HackerRank - Merge Sort: Counting Inversions

<https://www.hackerrank.com/challenges/merge-sort/problem>

问题：统计逆序对数量

解法：归并排序统计逆序对

7. 牛客网 - 计算右侧小于当前元素的个数

问题：与 LeetCode 315 相同

解法：归并排序+索引映射

8. 杭电 OJ - 1394

<http://acm.hdu.edu.cn/showproblem.php?pid=1394>

问题：将数组循环左移，求所有可能排列中的最小逆序对数量

解法：归并排序+逆序对性质分析

9. 洛谷 P1908 - 逆序对

<https://www.luogu.com.cn/problem/P1908>

问题：统计数组中逆序对的总数

解法：归并排序统计逆序对

10. SPOJ - INVCNT

<https://www.spoj.com/problems/INVCNT/>

问题：统计逆序对数量

解法：归并排序统计逆序对

这些题目虽然具体形式不同，但核心思想都是利用归并排序的分治特性，在合并过程中高效统计满足特定条件的元素对数量。

,,,

```
def count_smaller(nums):
```

```
    """
```

计算右侧小于当前元素的个数 - Python 实现

Args:

nums: 输入整数数组

Returns:

list: 包含每个元素右侧小于它的元素个数的列表

算法思路:

使用归并排序的思想，在合并两个有序子数组的过程中统计右侧小于当前元素的个数。

关键是维护原始索引信息，以便在排序后仍能正确更新结果数组。

时间复杂度:  $O(n \log n)$

空间复杂度:  $O(n)$

```
"""
```

```
n = len(nums)
```

```
if n == 0:
```

```
    return []
```

```
# 创建索引数组，保存值和原始索引
```

```
indexed_nums = [(nums[i], i) for i in range(n)]
```

```
# 结果数组，存储每个元素右侧小于它的元素个数
```

```
result = [0] * n
```

```
# 辅助数组，用于归并过程
```

```
temp = [None] * n
```

```
def merge_sort(left, right):
```

```
    """
```

归并排序的核心方法

Args:

left: 当前处理区间的左边界

right: 当前处理区间的右边界

```
"""
```

```

# 基本情况：区间只有一个元素时直接返回
if left == right:
    return

# 计算中间位置
mid = left + (right - left) // 2

# 分治：递归处理左右子区间
merge_sort(left, mid)
merge_sort(mid + 1, right)

# 合并两个有序子区间，同时统计结果
merge(left, mid, right)

def merge(left, mid, right):
    """
    合并两个有序子数组，并在合并过程中统计右侧小于当前元素的个数

    Args:
        left: 当前处理区间的左边界
        mid: 当前处理区间的中点
        right: 当前处理区间的右边界
    """

    i = left    # temp 数组的指针
    a = left    # 左侧子数组的指针
    b = mid + 1 # 右侧子数组的指针

    # 合并两个子数组，同时统计右侧小于当前元素的个数
    while a <= mid and b <= right:
        if indexed_nums[a][0] <= indexed_nums[b][0]:
            # 当左侧元素小于等于右侧元素时，右侧数组中已经处理的元素都小于当前左侧元素
            # 统计右侧已处理的元素数量: b - (mid + 1) = b - mid - 1
            result[indexed_nums[a][1]] += (b - mid - 1)
            temp[i] = indexed_nums[a]
            a += 1
        else:
            # 当右侧元素小于左侧元素时，将右侧元素放入辅助数组
            # 此时不更新计数，因为左侧元素还未被处理
            temp[i] = indexed_nums[b]
            b += 1
        i += 1

    # 处理左侧剩余元素

```

```
while a <= mid:
    # 左侧剩余元素的右侧所有元素都小于它
    result[indexed_nums[a][1]] += (b - mid - 1)
    temp[i] = indexed_nums[a]
    a += 1
    i += 1

# 处理右侧剩余元素
while b <= right:
    temp[i] = indexed_nums[b]
    b += 1
    i += 1

# 将辅助数组内容复制回原数组
for i in range(left, right + 1):
    if temp[i] is not None:
        indexed_nums[i] = temp[i]

# 执行归并排序并统计
merge_sort(0, n - 1)

return result

def main():
    """
    主函数，用于测试
    """
    import sys
    # 调整递归深度限制以处理大规模数据
    sys.setrecursionlimit(1000000)

    # 测试用例 1：基本情况
    test1 = [5, 2, 6, 1]
    print("输入:", test1)
    print("输出:", count_smaller(test1)) # 预期输出: [2, 1, 1, 0]

    # 测试用例 2：空数组
    test2 = []
    print("输入:", test2)
    print("输出:", count_smaller(test2)) # 预期输出: []

    # 测试用例 3：单元素数组
```

```
test3 = [1]
print("输入:", test3)
print("输出:", count_smaller(test3)) # 预期输出: [0]

# 测试用例 4: 逆序数组
test4 = [5, 4, 3, 2, 1]
print("输入:", test4)
print("输出:", count_smaller(test4)) # 预期输出: [4, 3, 2, 1, 0]

# 测试用例 5: 有序数组
test5 = [1, 2, 3, 4, 5]
print("输入:", test5)
print("输出:", count_smaller(test5)) # 预期输出: [0, 0, 0, 0, 0]

# 测试用例 6: 重复元素
test6 = [2, 2, 2]
print("输入:", test6)
print("输出:", count_smaller(test6)) # 预期输出: [0, 0, 0]

# 测试用例 7: 包含负数
test7 = [-1, -2, 3, -4, 5]
print("输入:", test7)
print("输出:", count_smaller(test7)) # 预期输出: [2, 1, 1, 0, 0]

# =====
# Python 语言特有关注事项
# =====
#
# 1. 递归深度限制:
#     - Python 默认递归深度限制约为 1000 层
#     - 对于大规模数据, 使用 sys.setrecursionlimit(1000000) 设置更大的限制
#
# 2. 整数精度:
#     - Python 的整数类型自动支持大整数, 不会有溢出问题
#
# 3. 列表操作效率:
#     - 使用索引操作而非切片, 避免创建新列表的开销
#     - 预先创建辅助数组, 避免在递归中重复创建
#
# 4. 内存管理:
#     - Python 自动管理内存, 但频繁创建对象会增加 GC 压力
#     - 使用 None 初始化 temp 数组, 避免不必要的对象创建
```

```
#  
# ======  
# 工程化考量  
# ======  
  
#  
# 1. 性能优化:  
#   - 对于小规模子数组(如 n<10)，可考虑使用插入排序  
#   - 可添加判断: 当 indexed_nums[mid][0] <= indexed_nums[mid+1][0]时，子数组已有序，可跳过合并  
#  
# 2. 错误处理:  
#   - 可添加输入验证，检查输入是否为列表  
#   - 可添加异常处理，处理输入格式错误  
#  
# 3. 可扩展性:  
#   - 算法易于扩展到其他统计问题  
#   - 可封装为模块供其他程序调用  
#  
# 4. 测试策略:  
#   - 已提供多种测试用例，覆盖常见场景  
#   - 可以添加性能测试，验证算法在大规模数据下的表现
```

```
if __name__ == "__main__":  
    main()
```

=====

文件: Code10\_LeetCode493.cpp

=====

```
#include <iostream>  
#include <vector>  
#include <algorithm>  
#include <climits>  
using namespace std;  
  
/**  
 * ======  
 * 题目 10: LeetCode 493 - 翻转对 (Reverse Pairs)  
 * ======  
 *  
 * 题目来源: LeetCode  
 * 题目链接: https://leetcode.cn/problems/reverse-pairs/  
 * 难度级别: 困难  
 */
```

- \* 问题描述:
- \* 给定一个数组 `nums`，如果  $i < j$  且  $nums[i] > 2*nums[j]$ ，我们就将  $(i, j)$  称作一个翻转对。
- \* 你需要返回数组中的翻转对的数量。
- \*
- \* 示例输入输出:
- \* 输入: [1, 3, 2, 3, 1]
- \* 输出: 2
- \* 解释:
- \* (1, 4) -> 3 > 2\*1
- \* (3, 4) -> 3 > 2\*1
- \*
- \* 输入: [2, 4, 3, 5, 1]
- \* 输出: 3
- \* 解释:
- \* (1, 4) -> 4 > 2\*1
- \* (2, 4) -> 3 > 2\*1
- \* (3, 4) -> 5 > 2\*1
- \*
- \* =====
- \* 核心算法思想: 归并排序分治统计
- \* =====
- \*
- \* 方法 1: 暴力解法 (不推荐)
- \* - 思路: 双重循环遍历所有  $i < j$  的情况，判断  $nums[i] > 2*nums[j]$
- \* - 时间复杂度:  $O(N^2)$  - 双重循环
- \* - 空间复杂度:  $O(1)$  - 不需要额外空间
- \* - 问题: 数据量大时超时
- \*
- \* 方法 2: 归并排序思想 (最优解) ★★★★★
- \* - 核心洞察: 利用归并排序的分治过程，在合并两个有序子数组之前，
- \*   统计左侧子数组中满足  $nums[i] > 2*nums[j]$  的元素对数量
- \*
- \* - 归并排序过程:
- \*   1. 分治: 将数组不断二分，直到只有一个元素
- \*   2. 统计: 在合并前，统计左侧子数组中每个元素能与右侧子数组形成的翻转对数量
- \*   3. 合并: 合并两个有序子数组
- \*
- \* - 优化技巧:
- \*   - 由于左右子数组已经各自有序，可以使用双指针技巧高效统计
- \*   - 对于左侧子数组的每个元素  $nums[i]$ ，找到右侧子数组中最大的  $j$ ，使得  $nums[j] < nums[i]/2$
- \*   - 这样，右侧子数组中从  $start$  到  $j$  的元素都可以与  $nums[i]$  形成翻转对
- \*
- \* - 时间复杂度详细计算:

- \*  $T(n) = 2T(n/2) + O(n)$  [Master 定理 case 2]
- \*  $= O(n \log n)$
- \* - 递归深度:  $\log n$
- \* - 每层统计和合并:  $O(n)$
- \*
- \* - 空间复杂度详细计算:
- \*  $S(n) = O(n) + O(\log n)$
- \* -  $O(n)$ : 辅助数组 help
- \* -  $O(\log n)$ : 递归调用栈
- \* 总计:  $O(n)$
- \*
- \* - 是否最优解: ★ 是 ★
- \* 理由: 基于比较的算法下界为  $O(n \log n)$ , 本算法已达到最优
- \*
- \* =====
- \* 相关题目列表 (基于归并排序的统计问题)
- \* =====
- \* 1. LeetCode 315 - 计算右侧小于当前元素的个数
  - \* <https://leetcode.cn/problems/count-of-smaller-numbers-after-self/>
  - \* 问题: 统计每个元素右侧比它小的元素个数
  - \* 解法: 归并排序过程中记录元素原始索引, 统计右侧小于当前元素的数量
  - \*
- \* 2. LeetCode 327 - 区间和的个数
  - \* <https://leetcode.cn/problems/count-of-range-sum/>
  - \* 问题: 统计区间和在  $[lower, upper]$  范围内的区间个数
  - \* 解法: 前缀和+归并排序, 统计满足条件的前缀和对
  - \*
- \* 3. 剑指 Offer 51 / LCR 170 - 数组中的逆序对
  - \* <https://leetcode.cn/problems/shu-zu-zhong-de-ni-xu-dui-lcof/>
  - \* 问题: 统计数组中逆序对的总数
  - \* 解法: 归并排序过程中统计逆序对数量
  - \*
- \* 4. POJ 2299 - Ultra-QuickSort
  - \* <http://poj.org/problem?id=2299>
  - \* 问题: 计算将数组排序所需的最小交换次数 (即逆序对数量)
  - \* 解法: 归并排序统计逆序对
  - \*
- \* 5. HDU 1394 - Minimum Inversion Number
  - \* <http://acm.hdu.edu.cn/showproblem.php?pid=1394>
  - \* 问题: 将数组循环左移, 求所有可能排列中的最小逆序对数量
  - \* 解法: 归并排序+逆序对性质分析
  - \*
- \* 6. 洛谷 P1908 - 逆序对

- \* <https://www.luogu.com.cn/problem/P1908>
- \* 问题：统计数组中逆序对的总数
- \* 解法：归并排序统计逆序对
- \*
- \* 7. HackerRank - Merge Sort: Counting Inversions
  - \* <https://www.hackerrank.com/challenges/merge-sort/problem>
  - \* 问题：统计逆序对数量
  - \* 解法：归并排序统计逆序对
  - \*
- \* 8. SPOJ - INVCNT
  - \* <https://www.spoj.com/problems/INVCNT/>
  - \* 问题：统计逆序对数量
  - \* 解法：归并排序统计逆序对
  - \*
- \* 9. CodeChef - INVCNT
  - \* <https://www.codechef.com/problems/INVCNT>
  - \* 问题：统计逆序对数量
  - \* 解法：归并排序或树状数组
  - \*
- \* 10. UVa 10810 - Ultra-QuickSort
  - \*

[https://onlinejudge.org/index.php?option=com\\_onlinejudge&Itemid=8&page=show\\_problem&problem=1751](https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=1751)

- \* 问题：计算逆序对数量
- \* 解法：归并排序统计逆序对
- \*

\* 这些题目虽然具体形式不同，但核心思想都是利用归并排序的分治特性，在合并过程中高效统计满足特定条件的元素对数量。

\*/

```
const int MAXN = 50001;
int help[MAXN];

/***
 * 归并排序并统计翻转对数量
 *
 * @param arr 数组
 * @param l 左边界
 * @param r 右边界
 * @return 区间[l, r]中的翻转对数量
 */
int mergeSort(vector<int>& arr, int l, int r) {
    if (l == r) {
        return 0;
    }
```

```

}

int m = l + (r - 1) / 2;
// 分治: 左半部分翻转对 + 右半部分翻转对 + 跨越两部分的翻转对
return mergeSort(arr, l, m) + mergeSort(arr, m + 1, r) + merge(arr, l, m, r);
}

/***
 * 合并两个有序子数组并统计翻转对数量
 *
 * @param arr 数组
 * @param l 左边界
 * @param m 中点
 * @param r 右边界
 * @return 跨越[l, m]和[m+1, r]的翻转对数量
 */
int merge(vector<int>& arr, int l, int m, int r) {
    // 统计部分
    int count = 0;
    for (int i = l, j = m + 1; i <= m; i++) {
        // 对于左侧元素 arr[i], 找到右侧子数组中满足 arr[i] > 2*arr[j] 的元素数量
        // 由于右侧子数组已排序, 可以使用双指针技巧
        while (j <= r && (long long)arr[i] > (long long)arr[j] * 2) {
            j++;
        }
        // j 之前的元素都满足条件, 即从 m+1 到 j-1 的元素
        count += j - m - 1;
    }

    // 正常 merge
    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];
    }

    return count;
}

/***
 * 计算数组中翻转对的数量
 *
 * @param nums 输入数组
 * @return 翻转对的数量
 */
int reversePairs(vector<int>& nums) {
    if (nums.empty() || nums.size() < 2) {
        return 0;
    }

    return mergeSort(nums, 0, nums.size() - 1);
}

/***
 * 主函数，用于测试
 */
int main() {
    // I/O 优化
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    // 测试用例 1：基本情况
    vector<int> test1 = {1, 3, 2, 3, 1};
    cout << "输入: [1,3,2,3,1]" << endl;
    cout << "输出: " << reversePairs(test1) << " (预期: 2)" << endl << endl;

    // 测试用例 2：基本情况
    vector<int> test2 = {2, 4, 3, 5, 1};
    cout << "输入: [2,4,3,5,1]" << endl;
    cout << "输出: " << reversePairs(test2) << " (预期: 3)" << endl << endl;

    // 测试用例 3：空数组
    vector<int> test3 = {};
    cout << "输入: []" << endl;
    cout << "输出: " << reversePairs(test3) << " (预期: 0)" << endl << endl;
```

```

// 测试用例 4: 单元素数组
vector<int> test4 = {1};
cout << "输入: [1]" << endl;
cout << "输出: " << reversePairs(test4) << " (预期: 0)" << endl << endl;

// 测试用例 5: 有序数组
vector<int> test5 = {1, 2, 3, 4, 5};
cout << "输入: [1,2,3,4,5]" << endl;
cout << "输出: " << reversePairs(test5) << " (预期: 0)" << endl << endl;

// 测试用例 6: 逆序数组
vector<int> test6 = {5, 4, 3, 2, 1};
cout << "输入: [5,4,3,2,1]" << endl;
cout << "输出: " << reversePairs(test6) << " (预期: 4)" << endl << endl;

// 测试用例 7: 包含负数
vector<int> test7 = {-5, -5, -5};
cout << "输入: [-5,-5,-5]" << endl;
cout << "输出: " << reversePairs(test7) << " (预期: 0)" << endl << endl;

// 测试用例 8: 大数值测试
vector<int> test8 = {INT_MAX, INT_MIN};
cout << "输入: [INT_MAX, INT_MIN]" << endl;
cout << "输出: " << reversePairs(test8) << " (预期: 1)" << endl;

return 0;
}

```

```

/*
* -----
* C++语言特有注意事项
* -----
* 1. 数据类型溢出问题:
*   - 在比较 arr[i] > 2*arr[j] 时, 2*arr[j] 可能会溢出
*   - 使用 long long 类型进行强制转换避免溢出: (long long)arr[i] > (long long)arr[j] * 2
*   - 当数组元素为 INT_MAX 或 INT_MIN 时特别需要注意
*
* 2. 输入输出优化:
*   - 使用 scanf/printf 而非 cin/cout 提高输入输出效率
*   - 使用 ios::sync_with_stdio(false) 关闭 C++与 C 的 I/O 同步
*   - 使用 cin.tie(nullptr) 解除 cin 与 cout 的绑定
*
* 3. 内存管理:

```

```
*      - 使用全局数组避免频繁内存分配
*      - MAXN 设为 50001，满足题目要求
*      - 静态数组在栈上分配，访问速度快
*
* 4. 递归深度：
*      - 归并排序递归深度为  $\log_2(n)$ ，不会超出 C++ 默认栈限制
*
* 5. 位运算优化：
*      - 可使用  $(l+r) \gg 1$  代替  $(l+r)/2$  提高运算效率
*      - 注意当 l 和 r 都很大时， $(l+r)$  可能导致溢出，应改为  $1 + ((r-l) \gg 1)$ 
*
* =====
* 工程化考量
* =====
1. 性能优化：
*      - 对于小规模子数组（如  $n < 10$ ），可考虑使用插入排序
*      - 可添加判断：当  $arr[m] \leq arr[m+1]$  时，子数组已有序，可跳过合并
*
2. 错误处理：
*      - 可添加输入验证，检查数组是否为空
*      - 可添加文件结束检查，处理输入异常
*
3. 可扩展性：
*      - 算法易于扩展到其他统计问题（如逆序对、小和问题）
*      - 可封装为函数库供其他程序调用
*
4. 编译优化：
*      - 可使用 -O2 编译选项优化性能
*      - 可使用 -march=native 利用 CPU 特性进一步优化
*
5. 测试策略：
*      - 已提供多种测试用例，覆盖常见场景
*      - 应包含边界测试（空数组、单元素、全相同元素等）
*      - 应包含性能测试（大规模数据）
*      - 应包含正确性测试（已知结果的测试用例）
*/
=====
```

文件: Code10\_LeetCode493.java

```
package class022;
```

```
/**  
 * ======  
 * 题目 10: LeetCode 493 - 翻转对 (Reverse Pairs)  
 * ======  
 *  
 * 题目来源: LeetCode  
 * 题目链接: https://leetcode.cn/problems/reverse-pairs/  
 * 难度级别: 困难  
 *  
 * 问题描述:  
 * 给定一个数组 nums , 如果  $i < j$  且  $\text{nums}[i] > 2*\text{nums}[j]$  , 我们就将  $(i, j)$  称作一个翻转对。  
 * 你需要返回数组中的翻转对的数量。  
 *  
 * 示例输入输出:  
 * 输入: [1, 3, 2, 3, 1]  
 * 输出: 2  
 * 解释:  
 * (1, 4) -> 3 > 2*1  
 * (3, 4) -> 3 > 2*1  
 *  
 * 输入: [2, 4, 3, 5, 1]  
 * 输出: 3  
 * 解释:  
 * (1, 4) -> 4 > 2*1  
 * (2, 4) -> 3 > 2*1  
 * (3, 4) -> 5 > 2*1  
 *  
 * ======  
 * 核心算法思想: 归并排序分治统计  
 * ======  
 *  
 * 方法 1: 暴力解法 (不推荐)  
 * - 思路: 双重循环遍历所有  $i < j$  的情况, 判断  $\text{nums}[i] > 2*\text{nums}[j]$   
 * - 时间复杂度:  $O(N^2)$  - 双重循环  
 * - 空间复杂度:  $O(1)$  - 不需要额外空间  
 * - 问题: 数据量大时超时  
 *  
 * 方法 2: 归并排序思想 (最优解) ★★★★★★  
 * - 核心洞察: 利用归并排序的分治过程, 在合并两个有序子数组之前,  
 *   统计左侧子数组中满足  $\text{nums}[i] > 2*\text{nums}[j]$  的元素对数量  
 *  
 * - 归并排序过程:  
 *   1. 分治: 将数组不断二分, 直到只有一个元素
```

- \* 2. 统计：在合并前，统计左侧子数组中每个元素能与右侧子数组形成的翻转对数量
- \* 3. 合并：合并两个有序子数组
- \*
- \* - 优化技巧：
  - 由于左右子数组已经各自有序，可以使用双指针技巧高效统计
  - 对于左侧子数组的每个元素  $\text{nums}[i]$ ，找到右侧子数组中最大的  $j$ ，使得  $\text{nums}[j] < \text{nums}[i]/2$
  - 这样，右侧子数组中从  $\text{start}$  到  $j$  的元素都可以与  $\text{nums}[i]$  形成翻转对
- \*
- \* - 时间复杂度详细计算：
  - $T(n) = 2T(n/2) + O(n)$  [Master 定理 case 2]
  - $= O(n \log n)$
  - 递归深度:  $\log n$
  - 每层统计和合并:  $O(n)$
- \*
- \* - 空间复杂度详细计算：
  - $S(n) = O(n) + O(\log n)$
  - $O(n)$ : 辅助数组 `help`
  - $O(\log n)$ : 递归调用栈
  - 总计:  $O(n)$
- \*
- \* - 是否最优解: ★ 是 ★
  - 理由: 基于比较的算法下界为  $O(n \log n)$ ，本算法已达到最优
- \*
- \* =====
- \* 相关题目列表 (基于归并排序的统计问题)
  - \* =====
  - \* 1. LeetCode 315 - 计算右侧小于当前元素的个数
    - <https://leetcode.cn/problems/count-of-smaller-numbers-after-self/>
    - 问题: 统计每个元素右侧比它小的元素个数
    - 解法: 归并排序过程中记录元素原始索引，统计右侧小于当前元素的数量
  - \* 2. LeetCode 327 - 区间和的个数
    - <https://leetcode.cn/problems/count-of-range-sum/>
    - 问题: 统计区间和在  $[lower, upper]$  范围内的区间个数
    - 解法: 前缀和+归并排序，统计满足条件的前缀和对
  - \* 3. 剑指 Offer 51 / LCR 170 - 数组中的逆序对
    - <https://leetcode.cn/problems/shu-zu-zhong-de-ni-xu-dui-lcof/>
    - 问题: 统计数组中逆序对的总数
    - 解法: 归并排序过程中统计逆序对数量
  - \* 4. POJ 2299 - Ultra-QuickSort
    - <http://poj.org/problem?id=2299>

- \* 问题：计算将数组排序所需的最小交换次数（即逆序对数量）
- \* 解法：归并排序统计逆序对
- \*
- \* 5. HDU 1394 – Minimum Inversion Number
  - \* <http://acm.hdu.edu.cn/showproblem.php?pid=1394>
  - \* 问题：将数组循环左移，求所有可能排列中的最小逆序对数量
  - \* 解法：归并排序+逆序对性质分析
  - \*
- \* 6. 洛谷 P1908 – 逆序对
  - \* <https://www.luogu.com/problem/P1908>
  - \* 问题：统计数组中逆序对的总数
  - \* 解法：归并排序统计逆序对
  - \*
- \* 7. HackerRank – Merge Sort: Counting Inversions
  - \* <https://www.hackerrank.com/challenges/merge-sort/problem>
  - \* 问题：统计逆序对数量
  - \* 解法：归并排序统计逆序对
  - \*
- \* 8. SPOJ – INVCNT
  - \* <https://www.spoj.com/problems/INVCNT/>
  - \* 问题：统计逆序对数量
  - \* 解法：归并排序统计逆序对
  - \*
- \* 9. CodeChef – INVCNT
  - \* <https://www.codechef.com/problems/INVCNT>
  - \* 问题：统计逆序对数量
  - \* 解法：归并排序或树状数组
  - \*
- \* 10. UVa 10810 – Ultra-QuickSort
  - \*

[https://onlinejudge.org/index.php?option=com\\_onlinejudge&Itemid=8&page=show\\_problem&problem=1751](https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=1751)

- \* 问题：计算逆序对数量
- \* 解法：归并排序统计逆序对
- \*

\* 这些题目虽然具体形式不同，但核心思想都是利用归并排序的分治特性，在合并过程中高效统计满足特定条件的元素对数量。

\*/

```
public class Code10_LeetCode493 {
```

```
    public static int MAXN = 50001;
    public static int[] help = new int[MAXN];
```

```
    /**
     * 计算逆序对数量
     * @param arr 待排序数组
     * @param l 左端点
     * @param r 右端点
     * @return 逆序对数量
     */
    private static int mergeCount(int[] arr, int l, int r) {
        if (l > r) return 0;
        int mid = (l + r) / 2;
        int count = mergeCount(arr, l, mid) + mergeCount(arr, mid + 1, r);
        int i = l, j = mid + 1, k = l;
        while (i <= mid && j <= r) {
            if (arr[i] > arr[j]) {
                count += mid - i + 1;
                j++;
            } else {
                i++;
            }
        }
        int[] temp = new int[r - l + 1];
        i = l; j = mid + 1; k = l;
        while (i <= mid && j <= r) {
            if (arr[i] < arr[j]) {
                temp[k] = arr[i];
                i++;
            } else {
                temp[k] = arr[j];
                j++;
            }
            k++;
        }
        while (i <= mid) {
            temp[k] = arr[i];
            i++;
            k++;
        }
        while (j <= r) {
            temp[k] = arr[j];
            j++;
            k++;
        }
        for (int m = l; m <= r; m++) {
            arr[m] = temp[m - l];
        }
        return count;
    }
}
```

```

* 计算数组中翻转对的数量
*
* @param arr 输入数组
* @return 翻转对的数量
*
* 算法思路：
* 使用归并排序的思想，在合并两个有序子数组的过程中统计翻转对数量。
* 关键在于统计左侧子数组中满足 nums[i] > 2*nums[j] 的元素对数量。
*/
public static int reversePairs(int[] arr) {
    return counts(arr, 0, arr.length - 1);
}

/***
* 归并排序并统计翻转对数量
*
* @param arr 数组
* @param l 左边界
* @param r 右边界
* @return 区间[l, r]中的翻转对数量
*/
public static int counts(int[] arr, int l, int r) {
    if (l == r) {
        return 0;
    }

    int m = (l + r) / 2;
    // 分治：左半部分翻转对 + 右半部分翻转对 + 跨越两部分的翻转对
    return counts(arr, l, m) + counts(arr, m + 1, r) + merge(arr, l, m, r);
}

/***
* 合并两个有序子数组并统计翻转对数量
*
* @param arr 数组
* @param l 左边界
* @param m 中点
* @param r 右边界
* @return 跨越[l, m]和[m+1, r]的翻转对数量
*/
public static int merge(int[] arr, int l, int m, int r) {
    // 统计部分
    int ans = 0;

```

```

for (int i = 1, j = m + 1; i <= m; i++) {
    // 对于左侧元素 arr[i]，找到右侧子数组中满足 arr[i] > 2*arr[j] 的元素数量
    // 由于右侧子数组已排序，可以使用双指针技巧
    while (j <= r && (long) arr[i] > (long) arr[j] * 2) {
        j++;
    }
    // j 之前的元素都满足条件，即从 m+1 到 j-1 的元素
    ans += j - m - 1;
}

// 正常 merge
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];
}

return ans;
}

/**
 * 主函数，用于测试
 */
public static void main(String[] args) {
    // 测试用例 1：基本情况
    int[] test1 = {1, 3, 2, 3, 1};
    System.out.println("输入：" + java.util.Arrays.toString(test1));
    System.out.println("输出：" + reversePairs(test1)); // 预期输出：2

    // 测试用例 2：基本情况
    int[] test2 = {2, 4, 3, 5, 1};
    System.out.println("输入：" + java.util.Arrays.toString(test2));
    System.out.println("输出：" + reversePairs(test2)); // 预期输出：3
}

```

```

// 测试用例 3: 空数组
int[] test3 = {};
System.out.println("输入: " + java.util.Arrays.toString(test3));
System.out.println("输出: " + reversePairs(test3)); // 预期输出: 0

// 测试用例 4: 单元素数组
int[] test4 = {1};
System.out.println("输入: " + java.util.Arrays.toString(test4));
System.out.println("输出: " + reversePairs(test4)); // 预期输出: 0

// 测试用例 5: 有序数组
int[] test5 = {1, 2, 3, 4, 5};
System.out.println("输入: " + java.util.Arrays.toString(test5));
System.out.println("输出: " + reversePairs(test5)); // 预期输出: 0

// 测试用例 6: 逆序数组
int[] test6 = {5, 4, 3, 2, 1};
System.out.println("输入: " + java.util.Arrays.toString(test6));
System.out.println("输出: " + reversePairs(test6)); // 预期输出: 4

// 测试用例 7: 包含负数
int[] test7 = {-5, -5, -5};
System.out.println("输入: " + java.util.Arrays.toString(test7));
System.out.println("输出: " + reversePairs(test7)); // 预期输出: 0

// 测试用例 8: 大数值测试
int[] test8 = {2147483647, -2147483648};
System.out.println("输入: " + java.util.Arrays.toString(test8));
System.out.println("输出: " + reversePairs(test8)); // 预期输出: 1
}

/*
* =====
* Java 语言特有关注事项
* =====
* 1. 数据类型溢出问题:
*   - 在比较 arr[i] > 2*arr[j] 时, 2*arr[j] 可能会溢出
*   - 使用 long 类型进行强制转换避免溢出: (long) arr[i] > (long) arr[j] * 2
*   - 当数组元素为 Integer.MAX_VALUE 或 Integer.MIN_VALUE 时特别需要注意
*
* 2. 输入输出优化:
*   - 对于大规模数据, 可以使用 StreamTokenizer 和 BufferedReader 提高输入效率

```

```
*      - 使用 PrintWriter 提高输出效率
*
* 3. 内存管理:
*      - 使用静态数组避免频繁内存分配
*      - MAXN 设为 50001, 满足题目要求
*
* 4. 递归深度:
*      - 归并排序递归深度为  $\log_2(n)$ , 不会超出 Java 默认栈限制
*
* 5. 边界条件处理:
*      - 空数组、单元素数组都有正确的处理
*      - 数组元素可能为负数, 算法仍正确
*
* =====
* 工程化考量
* =====
* 1. 性能优化:
*      - 对于小规模子数组(如  $n < 10$ ), 可考虑使用插入排序
*      - 可添加判断: 当  $arr[m] \leq arr[m+1]$  时, 子数组已有序, 可跳过合并
*
* 2. 错误处理:
*      - 可添加输入验证, 检查数组是否为 null
*      - 可添加异常处理, 处理输入异常
*
* 3. 可扩展性:
*      - 算法易于扩展到其他统计问题(如逆序对、小和问题)
*      - 可封装为工具类供其他程序调用
*
* 4. 测试策略:
*      - 已提供多种测试用例, 覆盖常见场景
*      - 应包含边界测试(空数组、单元素、全相同元素等)
*      - 应包含性能测试(大规模数据)
*      - 应包含正确性测试(已知结果的测试用例)
*/
}
```

文件: Code10\_LeetCode493.py

```
# LeetCode 493 - 翻转对 (Reverse Pairs)
# 题目来源: LeetCode
# 题目链接: https://leetcode.cn/problems/reverse-pairs/
```

# 难度级别: 困难

,,,

---

题目 10: LeetCode 493 - 翻转对 (Reverse Pairs)

---

题目来源: LeetCode

题目链接: <https://leetcode.cn/problems/reverse-pairs/>

难度级别: 困难

问题描述:

给定一个数组 `nums`，如果  $i < j$  且  $nums[i] > 2*nums[j]$ ，我们就将  $(i, j)$  称作一个翻转对。  
你需要返回数组中的翻转对的数量。

示例输入输出:

输入: [1, 3, 2, 3, 1]

输出: 2

解释:

$(1, 4) \rightarrow 3 > 2*1$

$(3, 4) \rightarrow 3 > 2*1$

输入: [2, 4, 3, 5, 1]

输出: 3

解释:

$(1, 4) \rightarrow 4 > 2*1$

$(2, 4) \rightarrow 3 > 2*1$

$(3, 4) \rightarrow 5 > 2*1$

---

核心算法思想: 归并排序分治统计

---

方法 1: 暴力解法 (不推荐)

- 思路: 双重循环遍历所有  $i < j$  的情况, 判断  $nums[i] > 2*nums[j]$
- 时间复杂度:  $O(N^2)$  - 双重循环
- 空间复杂度:  $O(1)$  - 不需要额外空间
- 问题: 数据量大时超时

方法 2: 归并排序思想 (最优解) ★★★★★

- 核心洞察: 利用归并排序的分治过程, 在合并两个有序子数组之前,  
统计左侧子数组中满足  $nums[i] > 2*nums[j]$  的元素对数量

- 归并排序过程:
    1. 分治: 将数组不断二分, 直到只有一个元素
    2. 统计: 在合并前, 统计左侧子数组中每个元素能与右侧子数组形成的翻转对数量
    3. 合并: 合并两个有序子数组
  - 优化技巧:
    - 由于左右子数组已经各自有序, 可以使用双指针技巧高效统计
    - 对于左侧子数组的每个元素  $\text{nums}[i]$ , 找到右侧子数组中最大的  $j$ , 使得  $\text{nums}[j] < \text{nums}[i]/2$
    - 这样, 右侧子数组中从  $\text{start}$  到  $j$  的元素都可以与  $\text{nums}[i]$  形成翻转对
  - 时间复杂度详细计算:
 
$$\begin{aligned} T(n) &= 2T(n/2) + O(n) \quad [\text{Master 定理 case 2}] \\ &= O(n \log n) \end{aligned}$$
    - 递归深度:  $\log n$
    - 每层统计和合并:  $O(n)$
  - 空间复杂度详细计算:
 
$$\begin{aligned} S(n) &= O(n) + O(\log n) \\ &- O(n): \text{辅助数组 help} \\ &- O(\log n): \text{递归调用栈} \\ &\text{总计: } O(n) \end{aligned}$$
  - 是否最优解: ★ 是 ★  
 理由: 基于比较的算法下界为  $O(n \log n)$ , 本算法已达到最优
- 

#### 相关题目列表 (基于归并排序的统计问题)

---

1. LeetCode 315 - 计算右侧小于当前元素的个数  
<https://leetcode.cn/problems/count-of-smaller-numbers-after-self/>  
 问题: 统计每个元素右侧比它小的元素个数  
 解法: 归并排序过程中记录元素原始索引, 统计右侧小于当前元素的数量
2. LeetCode 327 - 区间和的个数  
<https://leetcode.cn/problems/count-of-range-sum/>  
 问题: 统计区间和在  $[lower, upper]$  范围内的区间个数  
 解法: 前缀和+归并排序, 统计满足条件的前缀和对
3. 剑指 Offer 51 / LCR 170 - 数组中的逆序对  
<https://leetcode.cn/problems/shu-zu-zhong-de-ni-xu-dui-lcof/>  
 问题: 统计数组中逆序对的总数  
 解法: 归并排序过程中统计逆序对数量

4. POJ 2299 – Ultra-QuickSort

<http://poj.org/problem?id=2299>

问题：计算将数组排序所需的最小交换次数（即逆序对数量）

解法：归并排序统计逆序对

5. HDU 1394 – Minimum Inversion Number

<http://acm.hdu.edu.cn/showproblem.php?pid=1394>

问题：将数组循环左移，求所有可能排列中的最小逆序对数量

解法：归并排序+逆序对性质分析

6. 洛谷 P1908 – 逆序对

<https://www.luogu.com/problem/P1908>

问题：统计数组中逆序对的总数

解法：归并排序统计逆序对

7. HackerRank – Merge Sort: Counting Inversions

<https://www.hackerrank.com/challenges/merge-sort/problem>

问题：统计逆序对数量

解法：归并排序统计逆序对

8. SPOJ – INVCTN

<https://www.spoj.com/problems/INVCTN/>

问题：统计逆序对数量

解法：归并排序统计逆序对

9. CodeChef – INVCTN

<https://www.codechef.com/problems/INVCTN>

问题：统计逆序对数量

解法：归并排序或树状数组

10. UVa 10810 – Ultra-QuickSort

[https://onlinejudge.org/index.php?option=com\\_onlinejudge&Itemid=8&page=show\\_problem&problem=1751](https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=1751)

问题：计算逆序对数量

解法：归并排序统计逆序对

这些题目虽然具体形式不同，但核心思想都是利用归并排序的分治特性，在合并过程中高效统计满足特定条件的元素对数量。

”，

```
def reverse_pairs(nums):
    """
    """
```

## 计算数组中翻转对的数量 - Python 实现

Args:

nums: 输入数组

Returns:

int: 翻转对的数量

算法思路:

使用归并排序的思想，在合并两个有序子数组的过程中统计翻转对数量。

关键在于统计左侧子数组中满足  $\text{nums}[i] > 2 * \text{nums}[j]$  的元素对数量。

时间复杂度:  $O(n \log n)$

空间复杂度:  $O(n)$

"""

```
if not nums or len(nums) < 2:
```

```
    return 0
```

# 创建数组副本，避免修改原数组

```
arr = nums[:]
```

```
n = len(arr)
```

# 辅助数组，用于归并过程

```
help_arr = [0] * n
```

```
def merge_sort(l, r):
```

"""

归并排序并统计翻转对数量

Args:

l: 左边界

r: 右边界

Returns:

int: 区间[l, r]中的翻转对数量

"""

```
if l == r:
```

```
    return 0
```

```
m = (l + r) // 2
```

# 分治: 左半部分翻转对 + 右半部分翻转对 + 跨越两部分的翻转对

```
return merge_sort(l, m) + merge_sort(m + 1, r) + merge(l, m, r)
```

```

def merge(l, m, r):
    """
    合并两个有序子数组并统计翻转对数量
    """

    Args:
        l: 左边界
        m: 中点
        r: 右边界

    Returns:
        int: 跨越[l, m]和[m+1, r]的翻转对数量
    """

    # 统计部分
    count = 0

    # 对于左侧子数组的每个元素，统计右侧子数组中满足条件的元素数量
    for i in range(l, m + 1):
        # 使用双指针技巧找到满足 arr[i] > 2*arr[j] 的元素数量
        j = m + 1
        while j <= r and arr[i] > 2 * arr[j]:
            j += 1
        # j 之前的元素都满足条件，即从 m+1 到 j-1 的元素
        count += j - m - 1

    # 正常 merge
    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

# 将辅助数组内容复制回原数组
for i in range(1, r + 1):
    arr[i] = help_arr[i]

return count

return merge_sort(0, n - 1)

def main():
    """
    主函数，用于测试
    """

    import sys
    # 调整递归深度限制以处理大规模数据
    sys.setrecursionlimit(1000000)

    # 测试用例 1：基本情况
    test1 = [1, 3, 2, 3, 1]
    print("输入:", test1)
    print("输出:", reverse_pairs(test1)) # 预期输出: 2

    # 测试用例 2：基本情况
    test2 = [2, 4, 3, 5, 1]
    print("输入:", test2)
    print("输出:", reverse_pairs(test2)) # 预期输出: 3

    # 测试用例 3：空数组
    test3 = []
    print("输入:", test3)
    print("输出:", reverse_pairs(test3)) # 预期输出: 0

    # 测试用例 4：单元素数组
    test4 = [1]
    print("输入:", test4)
    print("输出:", reverse_pairs(test4)) # 预期输出: 0

    # 测试用例 5：有序数组
    test5 = [1, 2, 3, 4, 5]
    print("输入:", test5)
```

```
print("输出:", reverse_pairs(test5)) # 预期输出: 0

# 测试用例 6: 逆序数组
test6 = [5, 4, 3, 2, 1]
print("输入:", test6)
print("输出:", reverse_pairs(test6)) # 预期输出: 4

# 测试用例 7: 包含负数
test7 = [-5, -5, -5]
print("输入:", test7)
print("输出:", reverse_pairs(test7)) # 预期输出: 0

# 测试用例 8: 大数值测试
test8 = [2147483647, -2147483648]
print("输入:", test8)
print("输出:", reverse_pairs(test8)) # 预期输出: 1

# =====
# Python 语言特有关注事项
# =====
#
# 1. 递归深度限制:
#   - Python 默认递归深度限制约为 1000 层
#   - 对于大规模数据, 使用 sys.setrecursionlimit(1000000) 设置更大的限制
#
# 2. 整数精度:
#   - Python 的整数类型自动支持大整数, 不会有溢出问题
#   - 在计算 2*arr[j] 时不会出现溢出
#
# 3. 列表操作效率:
#   - 使用索引操作而非切片, 避免创建新列表的开销
#   - 预先创建辅助数组, 避免在递归中重复创建
#
# 4. 输入输出:
#   - 使用 input() 和 print() 处理标准输入输出
#
# =====
# 工程化考量
# =====
#
# 1. 性能优化:
#   - 对于小规模子数组(如 n<10), 可考虑使用插入排序
```

```
#     - 可添加判断：当 arr[m] <= arr[m+1] 时，子数组已有序，可跳过合并
#     - 可以优化统计过程，使用更高效的双指针技巧
#
# 2. 错误处理：
#     - 可添加输入验证，检查输入是否为列表
#     - 可添加异常处理，处理输入格式错误
#
# 3. 可扩展性：
#     - 算法易于扩展到其他统计问题（如逆序对、小和问题）
#     - 可封装为模块供其他程序调用
#
# 4. 测试策略：
#     - 已提供多种测试用例，覆盖常见场景
#     - 应包含边界测试（空数组、单元素、全相同元素等）
#     - 应包含性能测试（大规模数据）
#     - 应包含正确性测试（已知结果的测试用例）
```

```
if __name__ == "__main__":
    main()
```

=====

文件: Code11\_P0J2299.cpp

=====

```
#include <cstdio>
using namespace std;

/***
 * =====
 * 题目 11: POJ 2299 - Ultra-QuickSort
 * =====
 *
 * 题目来源: POJ
 * 题目链接: http://poj.org/problem?id=2299
 * 难度级别: 中等
 *
 * 问题描述:
 * 在这个问题中，您必须分析某些特定的排序算法的性能。该算法工作过程如下:
 * 1. 检查输入序列是否已经排序。
 * 2. 如果序列已经排序，则算法终止。
 * 3. 如果序列未排序，则交换输入序列中两个相邻的元素，然后返回步骤 1。
 *
 * 你的任务是计算将输入序列排序所需的最少交换次数。
```

- \*
  - \* 输入格式:
    - \* 输入包含若干测试用例。
    - \* 每个测试用例的第一行包含一个正整数 n,  $n \leq 500000$ , 表示序列的长度。
    - \* 下一行包含 n 个不同的整数  $a_1, \dots, a_n$ , ( $0 \leq a_i \leq 999,999,999$ )。
    - \* 输入以  $n=0$  的一行结束, 该行不应被处理。
  - \*
- \*
  - \* 输出格式:
    - \* 对于每个测试用例, 输出一行包含一个整数, 表示将序列排序所需的最少交换次数。
  - \*
- \*
  - \* 示例输入输出:
    - \* 输入:
      - \* 5
      - \* 9 1 0 5 4
      - \* 3
      - \* 1 2 3
      - \* 0
    - \* 输出:
      - \* 6
      - \* 0
  - \*
- \*
  - \* 解释:
    - \* 对于第一个测试用例 [9, 1, 0, 5, 4], 需要 6 次交换才能排序:
      - \* 1. 交换 9 和 1 得到 [1, 9, 0, 5, 4]
      - \* 2. 交换 9 和 0 得到 [1, 0, 9, 5, 4]
      - \* 3. 交换 9 和 5 得到 [1, 0, 5, 9, 4]
      - \* 4. 交换 9 和 4 得到 [1, 0, 5, 4, 9]
      - \* 5. 交换 5 和 4 得到 [1, 0, 4, 5, 9]
      - \* 6. 交换 1 和 0 得到 [0, 1, 4, 5, 9]
    - \*
  - \* 但实际上, 最少交换次数等于逆序对的数量。
  - \*
- \* ======
  - \* 核心算法思想: 归并排序统计逆序对
  - \* ======
  - \*
- \*
  - \* 关键洞察:
    - \* 将序列排序所需的最少相邻交换次数等于序列中逆序对的数量。
  - \*
- \*
  - \* 证明思路:
    - \* 1. 每次相邻交换只会减少一个逆序对
    - \* 2. 当序列有序时, 逆序对数量为 0
    - \* 3. 因此, 最少交换次数等于初始逆序对数量

\*

\* 算法:

\* 使用归并排序统计逆序对数量。

\*

\* 时间复杂度:  $O(n \log n)$

\* 空间复杂度:  $O(n)$

\*

\* =====

\* 相关题目列表

\* =====

\* 1. LeetCode 315 - 计算右侧小于当前元素的个数

\* <https://leetcode.cn/problems/count-of-smaller-numbers-after-self/>

\* 问题: 统计每个元素右侧比它小的元素个数

\* 解法: 归并排序过程中记录元素原始索引, 统计右侧小于当前元素的数量

\*

\* 2. LeetCode 493 - 翻转对

\* <https://leetcode.cn/problems/reverse-pairs/>

\* 问题: 统计满足  $\text{nums}[i] > 2 * \text{nums}[j]$  且  $i < j$  的对的数量

\* 解法: 归并排序过程中使用双指针统计跨越左右区间的翻转对

\*

\* 3. LeetCode 327 - 区间和的个数

\* <https://leetcode.cn/problems/count-of-range-sum/>

\* 问题: 统计区间和在 [lower, upper] 范围内的区间个数

\* 解法: 前缀和+归并排序, 统计满足条件的前缀和对

\*

\* 4. 剑指 Offer 51 / LCR 170 - 数组中的逆序对

\* <https://leetcode.cn/problems/shu-zu-zhong-de-ni-xu-dui-lcof/>

\* 问题: 统计数组中逆序对的总数

\* 解法: 归并排序过程中统计逆序对数量

\*

\* 5. HDU 1394 - Minimum Inversion Number

\* <http://acm.hdu.edu.cn/showproblem.php?pid=1394>

\* 问题: 将数组循环左移, 求所有可能排列中的最小逆序对数量

\* 解法: 归并排序+逆序对性质分析

\*

\* 6. 洛谷 P1908 - 逆序对

\* <https://www.luogu.com.cn/problem/P1908>

\* 问题: 统计数组中逆序对的总数

\* 解法: 归并排序统计逆序对

\*

\* 7. HackerRank - Merge Sort: Counting Inversions

\* <https://www.hackerrank.com/challenges/merge-sort/problem>

\* 问题: 统计逆序对数量

```

*      解法：归并排序统计逆序对
*
* 8. SPOJ - INVCNT
*      https://www.spoj.com/problems/INVCNT/
*      问题：统计逆序对数量
*      解法：归并排序统计逆序对
*
* 9. CodeChef - INVCNT
*      https://www.codechef.com/problems/INVCNT
*      问题：统计逆序对数量
*      解法：归并排序或树状数组
*
* 10. UVa 10810 - Ultra-QuickSort
*
https://onlinejudge.org/index.php?option=com\_onlinejudge&Itemid=8&page=show\_problem&problem=1751
*      问题：与 POJ 2299 相同
*      解法：归并排序统计逆序对
*/

```

```

const int MAXN = 500001;
int n;
long long arr[MAXN];
long long help[MAXN];

// 函数声明
long long mergeSort(int l, int r);
long long merge(int l, int m, int r);

/***
* 归并排序并统计逆序对数量
*
* @param l 左边界
* @param r 右边界
* @return 区间[l,r]中的逆序对数量
*/
long long mergeSort(int l, int r) {
    if (l == r) {
        return 0;
    }

    int m = l + (r - 1) / 2;
    // 分治：左半部分逆序对 + 右半部分逆序对 + 跨越两部分的逆序对
    return mergeSort(l, m) + mergeSort(m + 1, r) + merge(l, m, r);
}

```

```
}
```

```
/**  
 * 合并两个有序子数组并统计逆序对数量  
 *  
 * @param l 左边界  
 * @param m 中点  
 * @param r 右边界  
 * @return 跨越[l, m]和[m+1, r]的逆序对数量  
 */  
  
long long merge(int l, int m, int r) {  
    long long count = 0;  
    int i = l; // help 数组的当前位置  
    int a = l; // 左侧数组指针  
    int b = m + 1; // 右侧数组指针  
  
    // 合并过程，同时统计逆序对  
    while (a <= m && b <= r) {  
        if (arr[a] <= arr[b]) {  
            // 左侧元素小于等于右侧元素  
            // 右侧数组中已处理的元素(b - (m+1))个都小于 arr[a]，构成逆序对  
            count += (b - m - 1);  
            help[i++] = arr[a++];  
        } else {  
            // 右侧元素小于左侧元素  
            help[i++] = arr[b++];  
        }  
    }  
  
    // 处理左侧剩余元素  
    while (a <= m) {  
        // 左侧剩余元素与右侧所有元素都构成逆序对  
        count += (b - m - 1);  
        help[i++] = arr[a++];  
    }  
  
    // 处理右侧剩余元素  
    while (b <= r) {  
        help[i++] = arr[b++];  
    }  
  
    // 将 help 数组拷贝回原数组  
    for (i = l; i <= r; i++) {
```

```

        arr[i] = help[i];
    }

    return count;
}

/***
 * 主函数 - 处理输入输出
 *
 * 输入处理优化:
 * 使用 scanf 提高输入效率
 * 对于大规模数据(500000 个元素), 这种优化非常必要
 */
int main() {
    while (true) {
        // 读取数组长度
        scanf("%d", &n);

        // 输入以 n=0 结束
        if (n == 0) {
            break;
        }

        // 读取数组元素
        for (int i = 0; i < n; i++) {
            scanf("%lld", &arr[i]);
        }

        // 计算并输出逆序对数量 (即最少交换次数)
        printf("%lld\n", mergeSort(0, n - 1));
    }

    return 0;
}

/*
 * =====
 * C++语言特有注意事项
 * =====
 * 1. 数据类型溢出问题:
 *   - 逆序对数量可能超过 int 范围, 使用 long long 类型存储结果
 *   - 当 n=500000 时, 最坏情况下逆序对数量可达  $n*(n-1)/2 \approx 1.25*10^{11}$ , 超出 int 范围
 *   - 数组元素范围为[0, 999999999], 使用 long long 存储确保不会溢出

```

```
*  
* 2. 输入输出优化:  
*   - 使用 scanf/printf 而非 cin/cout 提高输入输出效率  
*   - 使用 ios::sync_with_stdio(false) 关闭 C++ 与 C 的 IO 同步  
*   - 使用 cin.tie(nullptr) 解除 cin 与 cout 的绑定  
*  
* 3. 内存管理:  
*   - 使用全局数组避免频繁内存分配  
*   - MAXN 设为 500001, 满足题目要求  
*   - 静态数组在栈上分配, 访问速度快  
*  
* 4. 递归深度:  
*   - 归并排序递归深度为  $\log_2(500000) \approx 19$  层, 不会超出 C++ 默认栈限制  
*  
* 5. 位运算优化:  
*   - 可使用  $(l+r) \gg 1$  代替  $(l+r)/2$  提高运算效率  
*   - 注意当 l 和 r 都很大时,  $(l+r)$  可能导致溢出, 应改为  $1 + ((r-l) \gg 1)$   
*  
* ======  
* 工程化考量  
* ======  
* 1. 性能优化:  
*   - 对于小规模子数组(如  $n < 10$ ), 可考虑使用插入排序  
*   - 可添加判断: 当  $arr[m] \leq arr[m+1]$  时, 子数组已有序, 可跳过合并  
*  
* 2. 错误处理:  
*   - 可添加输入验证, 检查 n 是否在合法范围内  
*   - 可添加文件结束检查, 处理输入异常  
*  
* 3. 可扩展性:  
*   - 算法易于扩展到其他统计问题(如翻转对、小和问题)  
*   - 可封装为函数库供其他程序调用  
*  
* 4. 编译优化:  
*   - 可使用 -O2 编译选项优化性能  
*   - 可使用 -march=native 利用 CPU 特性进一步优化  
*  
* 5. 测试策略:  
*   - 应包含边界测试(空数组、单元素、全相同元素等)  
*   - 应包含性能测试(大规模数据)  
*   - 应包含正确性测试(已知结果的测试用例)  
*/
```

=====

文件: Code11\_POJ2299.java

=====

```
package class022;

import java.io.*;

/**
 * =====
 * 题目 11: POJ 2299 - Ultra-QuickSort
 * =====
 *
 * 题目来源: POJ
 * 题目链接: http://poj.org/problem?id=2299
 * 难度级别: 中等
 *
 * 问题描述:
 * 在这个问题中, 您必须分析某些特定的排序算法的性能。该算法工作过程如下:
 * 1. 检查输入序列是否已经排序。
 * 2. 如果序列已经排序, 则算法终止。
 * 3. 如果序列未排序, 则交换输入序列中两个相邻的元素, 然后返回步骤 1。
 *
 * 你的任务是计算将输入序列排序所需的最少交换次数。
 *
 * 输入格式:
 * 输入包含若干测试用例。
 * 每个测试用例的第一行包含一个正整数 n, n<=500000, 表示序列的长度。
 * 下一行包含 n 个不同的整数 a1, ..., an, (0<=ai<=999,999,999)。
 * 输入以 n=0 的一行结束, 该行不应被处理。
 *
 * 输出格式:
 * 对于每个测试用例, 输出一行包含一个整数, 表示将序列排序所需的最少交换次数。
 *
 * 示例输入输出:
 * 输入:
 * 5
 * 9 1 0 5 4
 * 3
 * 1 2 3
 * 0
 * 输出:
 * 6
```

- \* 0
- \*
- \* 解释：
  - \* 对于第一个测试用例[9, 1, 0, 5, 4]，需要 6 次交换才能排序：
    - \* 1. 交换 9 和 1 得到[1, 9, 0, 5, 4]
    - \* 2. 交换 9 和 0 得到[1, 0, 9, 5, 4]
    - \* 3. 交换 9 和 5 得到[1, 0, 5, 9, 4]
    - \* 4. 交换 9 和 4 得到[1, 0, 5, 4, 9]
    - \* 5. 交换 5 和 4 得到[1, 0, 4, 5, 9]
    - \* 6. 交换 1 和 0 得到[0, 1, 4, 5, 9]
  - \*
- \* 但实际上，最少交换次数等于逆序对的数量。
  - \*
  - \* =====
- \* 核心算法思想：归并排序统计逆序对
  - \*
  - \*
- \* 关键洞察：
  - \* 将序列排序所需的最少相邻交换次数等于序列中逆序对的数量。
  - \*
- \* 证明思路：
  - \* 1. 每次相邻交换只会减少一个逆序对
  - \* 2. 当序列有序时，逆序对数量为 0
  - \* 3. 因此，最少交换次数等于初始逆序对数量
  - \*
- \* 算法：
  - \* 使用归并排序统计逆序对数量。
  - \*
  - \* 时间复杂度： $O(n \log n)$
  - \* 空间复杂度： $O(n)$
  - \*
- \* =====
- \* 相关题目列表
  - \*
  - \* =====
  - \* 1. LeetCode 315 – 计算右侧小于当前元素的个数
    - \* <https://leetcode.cn/problems/count-of-smaller-numbers-after-self/>
    - \* 问题：统计每个元素右侧比它小的元素个数
    - \* 解法：归并排序过程中记录元素原始索引，统计右侧小于当前元素的数量
    - \*
  - \* 2. LeetCode 493 – 翻转对
    - \* <https://leetcode.cn/problems/reverse-pairs/>
    - \* 问题：统计满足  $\text{nums}[i] > 2 * \text{nums}[j]$  且  $i < j$  的对的数量
    - \* 解法：归并排序过程中使用双指针统计跨越左右区间的翻转对

```
*  
* 3. LeetCode 327 - 区间和的个数  
*     https://leetcode.cn/problems/count-of-range-sum/  
*     问题: 统计区间和在[lower, upper]范围内的区间个数  
*     解法: 前缀和+归并排序, 统计满足条件的前缀和对  
*  
* 4. 剑指 Offer 51 / LCR 170 - 数组中的逆序对  
*     https://leetcode.cn/problems/shu-zu-zhong-de-ni-xu-dui-lcof/  
*     问题: 统计数组中逆序对的总数  
*     解法: 归并排序过程中统计逆序对数量  
*  
* 5. HDU 1394 - Minimum Inversion Number  
*     http://acm.hdu.edu.cn/showproblem.php?pid=1394  
*     问题: 将数组循环左移, 求所有可能排列中的最小逆序对数量  
*     解法: 归并排序+逆序对性质分析  
*  
* 6. 洛谷 P1908 - 逆序对  
*     https://www.luogu.com.cn/problem/P1908  
*     问题: 统计数组中逆序对的总数  
*     解法: 归并排序统计逆序对  
*  
* 7. HackerRank - Merge Sort: Counting Inversions  
*     https://www.hackerrank.com/challenges/merge-sort/problem  
*     问题: 统计逆序对数量  
*     解法: 归并排序统计逆序对  
*  
* 8. SPOJ - INVCNT  
*     https://www.spoj.com/problems/INVCNT/  
*     问题: 统计逆序对数量  
*     解法: 归并排序统计逆序对  
*  
* 9. CodeChef - INVCNT  
*     https://www.codechef.com/problems/INVCNT  
*     问题: 统计逆序对数量  
*     解法: 归并排序或树状数组  
*  
* 10. UVa 10810 - Ultra-QuickSort  
*  
https://onlinejudge.org/index.php?option=com\_onlinejudge&Itemid=8&page=show\_problem&problem=1751  
*     问题: 与 POJ 2299 相同  
*     解法: 归并排序统计逆序对  
*/  
public class Code11_P0J2299 {
```

```

public static int MAXN = 500001;
public static int[] arr = new int[MAXN];
public static int[] help = new int[MAXN];

/**
 * 计算数组中逆序对的数量
 *
 * @param n 数组长度
 * @return 逆序对的数量
 *
 * 算法思路：
 * 使用归并排序的思想，在合并两个有序子数组的过程中统计逆序对数量。
 * 当从左侧子数组选取元素时，右侧子数组中已处理的元素都小于该元素，
 * 因此这些元素与当前元素构成逆序对。
 */
public static long countInversions(int n) {
    return mergeSort(0, n - 1);
}

/**
 * 归并排序并统计逆序对数量
 *
 * @param l 左边界
 * @param r 右边界
 * @return 区间[l, r]中的逆序对数量
 */
public static long mergeSort(int l, int r) {
    if (l == r) {
        return 0;
    }

    int m = l + (r - 1) / 2;
    // 分治：左半部分逆序对 + 右半部分逆序对 + 跨越两部分的逆序对
    return mergeSort(l, m) + mergeSort(m + 1, r) + merge(l, m, r);
}

/**
 * 合并两个有序子数组并统计逆序对数量
 *
 * @param l 左边界
 * @param m 中点
 * @param r 右边界

```

```

* @return 跨越[1, m]和[m+1, r]的逆序对数量
*/
public static long merge(int l, int m, int r) {
    long count = 0;
    int i = l;          // help 数组的当前位置
    int a = l;          // 左侧数组指针
    int b = m + 1;      // 右侧数组指针

    // 合并过程，同时统计逆序对
    while (a <= m && b <= r) {
        if (arr[a] <= arr[b]) {
            // 左侧元素小于等于右侧元素
            // 右侧数组中已处理的元素(b - (m+1))个都小于 arr[a]，构成逆序对
            count += (b - m - 1);
            help[i++] = arr[a++];
        } else {
            // 右侧元素小于左侧元素
            help[i++] = arr[b++];
        }
    }

    // 处理左侧剩余元素
    while (a <= m) {
        // 左侧剩余元素与右侧所有元素都构成逆序对
        count += (b - m - 1);
        help[i++] = arr[a++];
    }

    // 处理右侧剩余元素
    while (b <= r) {
        help[i++] = arr[b++];
    }

    // 将 help 数组拷贝回原数组
    for (i = l; i <= r; i++) {
        arr[i] = help[i];
    }

    return count;
}

/***
 * 主函数 - 处理输入输出

```

```

/*
 * 输入处理优化:
 * 使用 BufferedReader 和 StreamTokenizer 提高输入效率
 * 对于大规模数据(500000 个元素), 这种优化非常必要
 */
public static void main(String[] args) throws IOException {
    // 使用高效 IO 处理
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    StreamTokenizer in = new StreamTokenizer(br);
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

    while (true) {
        // 读取数组长度
        in.nextToken();
        int n = (int) in.nval;

        // 输入以 n=0 结束
        if (n == 0) {
            break;
        }

        // 读取数组元素
        for (int i = 0; i < n; i++) {
            in.nextToken();
            arr[i] = (int) in.nval;
        }

        // 计算并输出逆序对数量 (即最少交换次数)
        out.println(countInversions(n));
    }

    out.flush();
    out.close();
}

/*
 * =====
 * Java 语言特有关注事项
 * =====
 * 1. 数据类型溢出问题:
 *   - 逆序对数量可能超过 int 范围, 使用 long 类型存储结果
 *   - 当 n=500000 时, 最坏情况下逆序对数量可达  $n*(n-1)/2 \approx 1.25*10^{11}$ , 超出 int 范围
 *

```

```
* 2. 输入输出优化:  
*   - 使用 StreamTokenizer 和 BufferedReader 提高输入效率  
*   - 使用 PrintWriter 提高输出效率  
*   - 对于大规模数据, Scanner 类效率较低  
  
*  
* 3. 内存管理:  
*   - 使用静态数组避免频繁内存分配  
*   - MAXN 设为 500001, 满足题目要求  
  
*  
* 4. 递归深度:  
*   - 归并排序递归深度为  $\log_2(500000) \approx 19$  层, 不会超出 Java 默认栈限制  
  
*  
* 5. 边界条件处理:  
*   - 空数组、单元素数组都有正确的处理  
*   - 数组元素可能为 0, 算法仍正确  
  
*  
* ======  
* 工程化考量  
* ======  
* 1. 性能优化:  
*   - 对于小规模子数组(如 n<10), 可考虑使用插入排序  
*   - 可添加判断: 当 arr[m] <= arr[m+1]时, 子数组已有序, 可跳过合并  
  
*  
* 2. 错误处理:  
*   - 可添加输入验证, 检查 n 是否在合法范围内  
*   - 可添加异常处理, 处理输入格式错误  
  
*  
* 3. 可扩展性:  
*   - 算法易于扩展到其他统计问题(如翻转对、小和问题)  
*   - 可封装为工具类供其他程序调用  
  
*  
* 4. 测试策略:  
*   - 应包含边界测试(空数组、单元素、全相同元素等)  
*   - 应包含性能测试(大规模数据)  
*   - 应包含正确性测试(已知结果的测试用例)  
*/
```

{}

```
=====
```

文件: Code11\_POJ2299.py

```
=====
```

```
# POJ 2299 - Ultra-QuickSort
```

```
# 题目来源: POJ  
# 题目链接: http://poj.org/problem?id=2299  
# 难度级别: 中等
```

,,

---

### 题目 11: POJ 2299 - Ultra-QuickSort

---

题目来源: POJ

题目链接: http://poj.org/problem?id=2299

难度级别: 中等

问题描述:

在这个问题中, 您必须分析某些特定的排序算法的性能。该算法工作过程如下:

1. 检查输入序列是否已经排序。
2. 如果序列已经排序, 则算法终止。
3. 如果序列未排序, 则交换输入序列中两个相邻的元素, 然后返回步骤 1。

你的任务是计算将输入序列排序所需的最少交换次数。

输入格式:

输入包含若干测试用例。

每个测试用例的第一行包含一个正整数 n,  $n \leq 500000$ , 表示序列的长度。

下一行包含 n 个不同的整数  $a_1, \dots, a_n$ , ( $0 \leq a_i \leq 999,999,999$ )。

输入以  $n=0$  的一行结束, 该行不应被处理。

输出格式:

对于每个测试用例, 输出一行包含一个整数, 表示将序列排序所需的最少交换次数。

示例输入输出:

输入:

5

9 1 0 5 4

3

1 2 3

0

输出:

6

0

解释:

对于第一个测试用例 [9, 1, 0, 5, 4], 需要 6 次交换才能排序:

1. 交换 9 和 1 得到 [1, 9, 0, 5, 4]
2. 交换 9 和 0 得到 [1, 0, 9, 5, 4]
3. 交换 9 和 5 得到 [1, 0, 5, 9, 4]
4. 交换 9 和 4 得到 [1, 0, 5, 4, 9]
5. 交换 5 和 4 得到 [1, 0, 4, 5, 9]
6. 交换 1 和 0 得到 [0, 1, 4, 5, 9]

但实际上，最少交换次数等于逆序对的数量。

---

---

核心算法思想：归并排序统计逆序对

---

---

关键洞察：

将序列排序所需的最少相邻交换次数等于序列中逆序对的数量。

证明思路：

1. 每次相邻交换只会减少一个逆序对
2. 当序列有序时，逆序对数量为 0
3. 因此，最少交换次数等于初始逆序对数量

算法：

使用归并排序统计逆序对数量。

时间复杂度： $O(n \log n)$

空间复杂度： $O(n)$

---

---

相关题目列表

---

---

1. LeetCode 315 - 计算右侧小于当前元素的个数

<https://leetcode.cn/problems/count-of-smaller-numbers-after-self/>

问题：统计每个元素右侧比它小的元素个数

解法：归并排序过程中记录元素原始索引，统计右侧小于当前元素的数量

2. LeetCode 493 - 翻转对

<https://leetcode.cn/problems/reverse-pairs/>

问题：统计满足  $\text{nums}[i] > 2 * \text{nums}[j]$  且  $i < j$  的对的数量

解法：归并排序过程中使用双指针统计跨越左右区间的翻转对

3. LeetCode 327 - 区间和的个数

<https://leetcode.cn/problems/count-of-range-sum/>

问题：统计区间和在  $[\text{lower}, \text{upper}]$  范围内的区间个数

解法：前缀和+归并排序，统计满足条件的前缀和对

4. 剑指 Offer 51 / LCR 170 - 数组中的逆序对

<https://leetcode.cn/problems/shu-zu-zhong-de-ni-xu-dui-lcof/>

问题：统计数组中逆序对的总数

解法：归并排序过程中统计逆序对数量

5. HDU 1394 - Minimum Inversion Number

<http://acm.hdu.edu.cn/showproblem.php?pid=1394>

问题：将数组循环左移，求所有可能排列中的最小逆序对数量

解法：归并排序+逆序对性质分析

6. 洛谷 P1908 - 逆序对

<https://www.luogu.com.cn/problem/P1908>

问题：统计数组中逆序对的总数

解法：归并排序统计逆序对

7. HackerRank - Merge Sort: Counting Inversions

<https://www.hackerrank.com/challenges/merge-sort/problem>

问题：统计逆序对数量

解法：归并排序统计逆序对

8. SPOJ - INVCNT

<https://www.spoj.com/problems/INVCNT/>

问题：统计逆序对数量

解法：归并排序统计逆序对

9. CodeChef - INVCNT

<https://www.codechef.com/problems/INVCNT>

问题：统计逆序对数量

解法：归并排序或树状数组

10. UVa 10810 - Ultra-QuickSort

[https://onlinejudge.org/index.php?option=com\\_onlinejudge&Itemid=8&page=show\\_problem&problem=1751](https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=1751)

问题：与 POJ 2299 相同

解法：归并排序统计逆序对

, , ,

```
def count_inversions(arr):
```

```
    """
```

计算数组中逆序对的数量 - Python 实现

Args:

arr: 输入数组

Returns:

int: 逆序对的数量

算法思路:

使用归并排序的思想，在合并两个有序子数组的过程中统计逆序对数量。

当从左侧子数组选取元素时，右侧子数组中已处理的元素都小于该元素，因此这些元素与当前元素构成逆序对。

时间复杂度:  $O(n \log n)$

空间复杂度:  $O(n)$

"""

# 创建数组副本，避免修改原数组

arr\_copy = arr[:]

n = len(arr\_copy)

# 辅助数组

help\_arr = [0] \* n

def merge\_sort(l, r):

"""

归并排序并统计逆序对数量

Args:

l: 左边界

r: 右边界

Returns:

int: 区间[l, r]中的逆序对数量

"""

if l == r:

    return 0

m = (l + r) // 2

# 分治: 左半部分逆序对 + 右半部分逆序对 + 跨越两部分的逆序对

return merge\_sort(l, m) + merge\_sort(m + 1, r) + merge(l, m, r)

def merge(l, m, r):

"""

合并两个有序子数组并统计逆序对数量

Args:

- 1: 左边界
- m: 中点
- r: 右边界

Returns:

```
int: 跨越[1, m]和[m+1, r]的逆序对数量
"""
count = 0
i = 1      # help_arr 数组的当前位置
a = 1      # 左侧数组指针
b = m + 1  # 右侧数组指针

# 合并过程，同时统计逆序对
while a <= m and b <= r:
    if arr_copy[a] <= arr_copy[b]:
        # 左侧元素小于等于右侧元素
        # 右侧数组中已处理的元素(b - (m+1))个都小于 arr_copy[a]，构成逆序对
        count += (b - m - 1)
        help_arr[i] = arr_copy[a]
        a += 1
    else:
        # 右侧元素小于左侧元素
        help_arr[i] = arr_copy[b]
        b += 1
    i += 1

# 处理左侧剩余元素
while a <= m:
    # 左侧剩余元素与右侧所有元素都构成逆序对
    count += (b - m - 1)
    help_arr[i] = arr_copy[a]
    a += 1
    i += 1

# 处理右侧剩余元素
while b <= r:
    help_arr[i] = arr_copy[b]
    b += 1
    i += 1

# 将 help_arr 数组拷贝回原数组
```

```
for i in range(1, r + 1):
    arr_copy[i] = help_arr[i]

return count

return merge_sort(0, n - 1)

def main():
    """
    主函数 - 处理输入输出
    """
    import sys
    # 调整递归深度限制以处理大规模数据
    sys.setrecursionlimit(1000000)

    while True:
        # 读取数组长度
        try:
            n = int(input())
        except:
            break

        # 输入以 n=0 结束
        if n == 0:
            break

        # 读取数组元素
        arr = list(map(int, input().split()))

        # 计算并输出逆序对数量（即最少交换次数）
        result = count_inversions(arr)
        print(result)

# =====
# Python 语言特有注意事项
# =====
#
# 1. 递归深度限制:
#     - Python 默认递归深度限制约为 1000 层
#     - 对于 n=500000 的数据，归并排序递归深度约为  $\log_2(500000) \approx 19$  层，不会超出限制
#     - 但为保险起见，使用 sys.setrecursionlimit(1000000) 设置更大的限制
```

```
#  
# 2. 整数精度:  
#     - Python 的整数类型自动支持大整数，不会有溢出问题  
#     - 逆序对数量可能很大，但 Python 能正确处理  
  
#  
# 3. 列表操作效率:  
#     - 使用索引操作而非切片，避免创建新列表的开销  
#     - 预先创建辅助数组，避免在递归中重复创建  
  
#  
# 4. 输入输出:  
#     - 使用 input() 和 print() 处理标准输入输出  
#     - 对于大规模数据，Python 的 I/O 效率可能不如 C++ 和 Java  
  
# ======  
  
# 工程化考量  
# ======  
  
#  
# 1. 性能优化:  
#     - 对于小规模子数组（如 n<10），可考虑使用插入排序  
#     - 可添加判断：当 arr[m] <= arr[m+1] 时，子数组已有序，可跳过合并  
  
#  
# 2. 错误处理:  
#     - 可添加输入验证，检查 n 是否在合法范围内  
#     - 可添加异常处理，处理输入格式错误  
  
#  
# 3. 可扩展性:  
#     - 算法易于扩展到其他统计问题（如翻转对、小和问题）  
#     - 可封装为模块供其他程序调用  
  
#  
# 4. 测试策略:  
#     - 应包含边界测试（空数组、单元素、全相同元素等）  
#     - 应包含性能测试（大规模数据）  
#     - 应包含正确性测试（已知结果的测试用例）  
  
if __name__ == "__main__":  
    main()  
  
=====
```