

=====

文件夹: class001_SortingAlgorithms

=====

[Markdown 文件]

=====

文件: AdditionalProblems.md

=====

补充算法题目与训练

📄 更多排序相关题目

LeetCode 题目补充

基础排序题目

1. **88. 合并两个有序数组**
 - 来源: LeetCode
 - 链接: <https://leetcode.cn/problems/merge-sorted-array/>
 - 难度: 简单
 - 解法: 双指针从后向前合并
 2. **148. 排序链表**
 - 来源: LeetCode
 - 链接: <https://leetcode.cn/problems/sort-list/>
 - 难度: 中等
 - 解法: 归并排序 (链表版本)
 3. **912. 排序数组**
 - 来源: LeetCode
 - 链接: <https://leetcode.cn/problems/sort-an-array/>
 - 难度: 中等
 - 解法: 各种排序算法的实现和比较
- #### 快速选择相关
4. **973. 最接近原点的 K 个点**
 - 来源: LeetCode
 - 链接: <https://leetcode.cn/problems/k-closest-points-to-origin/>
 - 难度: 中等
 - 解法: 快速选择/堆排序
 5. **1054. 距离相等的条形码**
 - 来源: LeetCode
 - 链接: <https://leetcode.cn/problems/distant-barcodes/>

- 难度: 中等
- 解法: 堆排序 (频率统计)

特殊排序

6. **164. 最大间距** (已包含)
 - 来源: LeetCode
 - 链接: <https://leetcode.cn/problems/maximum-gap/>
 - 难度: 困难
 - 解法: 基数排序/桶排序
7. **324. 摆动排序 II**
 - 来源: LeetCode
 - 链接: <https://leetcode.cn/problems/wiggle-sort-ii/>
 - 难度: 中等
 - 解法: 排序+双指针

牛客网题目

1. **NC140 排序**
 - 来源: 牛客网
 - 链接: <https://www.nowcoder.com/practice/2baf799ea0594abd974d37139de27896>
 - 难度: 简单
 - 解法: 各种排序算法实现
2. **NC119 最小的 K 个数**
 - 来源: 牛客网
 - 链接: <https://www.nowcoder.com/practice/6a296eb82cf844ca8539b57c23e6e9bf>
 - 难度: 中等
 - 解法: 堆/快速选择
3. **NC88 寻找第 K 大**
 - 来源: 牛客网
 - 链接: <https://www.nowcoder.com/practice/e016ad9b7f0b45048c58a9f27ba618bf>
 - 难度: 中等
 - 解法: 快速选择算法

剑指 Offer 题目

1. **面试题 40. 最小的 k 个数**
 - 来源: 剑指 Offer
 - 链接: 剑指 Offer 第二版第 40 题
 - 难度: 简单
 - 解法: 堆/快速选择

2. **面试题 51. 数组中的逆序对**

- 来源: 剑指 Offer
- 链接: 剑指 Offer 第二版第 51 题
- 难度: 困难
- 解法: 归并排序

HackerRank 题目

1. **Fraudulent Activity Notifications**

- 来源: HackerRank
- 链接: <https://www.hackerrank.com/challenges/fraudulent-activity-notifications>
- 难度: 中等
- 解法: 滑动窗口+计数排序

2. **Counting Inversions**

- 来源: HackerRank
- 链接: <https://www.hackerrank.com/challenges/ctci-merge-sort>
- 难度: 困难
- 解法: 归并排序统计逆序对

🎯 题目分类训练

按算法分类训练

归并排序训练

1. **逆序对计数** - 归并排序的经典应用
2. **链表排序** - 归并排序在链表上的实现
3. **外部排序** - 处理超大数据集的排序

快速排序训练

1. **三路快排** - 处理大量重复元素
2. **快速选择** - 寻找第 K 大/小元素
3. **荷兰国旗问题** - 三色排序

堆排序训练

1. **Top K 问题** - 前 K 大/小元素
2. **中位数查找** - 动态数据流的中位数
3. **优先级队列** - 堆的实际应用

按难度分级训练

初级（掌握基础）

- 实现各种基础排序算法
- 理解时间/空间复杂度
- 处理简单边界条件

中级（应用扩展）

- 解决 LeetCode 中等难度题目
- 掌握算法优化技巧
- 处理复杂边界情况

高级（深入理解）

- 解决困难题目
- 理解算法底层原理
- 进行性能优化和工程化

🌟 解题思路总结

见到排序题目的思考流程

1. **分析题目要求**

- 是否需要稳定排序？
- 是否有空间限制？
- 数据规模有多大？
- 数据分布特征？

2. **选择合适算法**

- 小数据：插入/选择排序
- 大数据：快速/归并/堆排序
- 需要稳定：归并排序
- 空间紧张：堆排序/原地快排

3. **考虑优化策略**

- 小数组优化
- 随机化避免最坏情况
- 处理重复元素

4. **处理边界条件**

- 空数组
- 单元素
- 已排序/逆序
- 大量重复

常见题型模式

模式 1: Top K 问题

- 特征: 寻找前 K 大/小元素
- 解法: 快速选择($O(n)$)或堆($O(n \log k)$)

模式 2: 区间合并

- 特征: 重叠区间合并
- 解法: 按起点排序后合并

模式 3: 颜色分类

- 特征: 有限种类的排序
- 解法: 计数排序/多指针

模式 4: 逆序对统计

- 特征: 统计逆序对数量
- 解法: 归并排序

🔧 代码实现要点

Java 实现要点

```
```java
// 1. 使用泛型支持多种数据类型
public class SortAlgorithms<T extends Comparable<T>> {

 // 2. 异常处理
 public void sort(T[] array) {
 if (array == null) throw new IllegalArgumentException();
 if (array.length <= 1) return;

 // 排序逻辑
 }

 // 3. 性能监控
 public void sortWithTiming(T[] array) {
 long start = System.nanoTime();
 sort(array);
 long end = System.nanoTime();
 System.out.println("耗时: " + (end - start) + "纳秒");
 }
}
````
```

C++实现要点

```
```cpp
```

```
// 1. 模板支持泛型
template<typename T>
class SortAlgorithms {
public:
 // 2. 异常安全
 void sort(std::vector<T>& nums) {
 if (nums.empty()) return;

 // 使用 RAI 确保资源安全
 // 排序逻辑
 }

 // 3. 移动语义优化
 std::vector<T> sorted(std::vector<T> nums) {
 sort(nums);
 return nums; // 移动语义优化
 }
};

```

```

```
#### Python 实现要点
```python
class SortAlgorithms:
 # 1. 类型注解
 @staticmethod
 def sort(nums: List[int]) -> List[int]:
 # 2. 输入验证
 if not isinstance(nums, list):
 raise TypeError("输入必须是列表")

 # 3. 边界处理
 if len(nums) <= 1:
 return nums.copy()

 # 排序逻辑
 return sorted_nums

 # 4. 性能测试装饰器
 @staticmethod
 def timed_sort(nums):
 import time
 start = time.time()
 result = SortAlgorithms.sort(nums)
 end = time.time()
 print(f"Sorting {len(nums)} elements took {end - start} seconds")

```

```
 end = time.time()
 print(f"排序耗时: {end - start:.6f}秒")
 return result
```

```

📈 复杂度分析深度

归并排序复杂度推导

```

$$\begin{aligned} T(n) &= 2T(n/2) + O(n) \\ &= 2[2T(n/4) + O(n/2)] + O(n) \\ &= 4T(n/4) + 2O(n/2) + O(n) \\ &= 4T(n/4) + 2O(n) \\ &= \dots \\ &= 2^k T(n/2^k) + kO(n) \end{aligned}$$

当  $n/2^k = 1 \Rightarrow k = \log_2 n$

$$T(n) = nT(1) + O(n \log n) = O(n \log n)$$

```

快速排序复杂度分析

****最好情况**:** 每次划分均衡

```

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

```

****最坏情况**:** 每次划分极端不平衡

```

$$T(n) = T(n-1) + O(n) = O(n^2)$$

```

****平均情况**:** 通过随机化达到 $O(n \log n)$

🔧 工程化实践

1. 单元测试设计

``` python

```
import unittest
```

```
class TestSortAlgorithms(unittest.TestCase):
 def test_empty_array(self):
 self.assertEqual(SortAlgorithms.sort([]), [])
```

```
def test_single_element(self):
 self.assertEqual(SortAlgorithms.sort([1]), [1])

def test_already_sorted(self):
 self.assertEqual(SortAlgorithms.sort([1, 2, 3]), [1, 2, 3])

def test_reverse_sorted(self):
 self.assertEqual(SortAlgorithms.sort([3, 2, 1]), [1, 2, 3])

def test_duplicate_elements(self):
 self.assertEqual(SortAlgorithms.sort([2, 2, 1, 1]), [1, 1, 2, 2])
```

```

2. 性能基准测试

```
```python
```

```
def benchmark_different_sizes():
 sizes = [100, 1000, 10000, 100000]
 algorithms = {
 '归并排序': merge_sort,
 '快速排序': quick_sort,
 '堆排序': heap_sort,
 '内置排序': sorted
 }

 for size in sizes:
 test_data = generate_test_data(size)
 print(f"\n数据规模: {size}")

 for name, algorithm in algorithms.items():
 time_taken = time_algorithm(algorithm, test_data)
 print(f"{name}: {time_taken:.6f}秒")
```

```

3. 内存使用监控

```
```python
```

```
import tracemalloc

def monitor_memory_usage(algorithm, data):
 tracemalloc.start()

 # 执行算法
 result = algorithm(data)
```

```
current, peak = tracemalloc.get_traced_memory()
tracemalloc.stop()

print(f"当前内存使用: {current / 10**6:.2f} MB")
print(f"峰值内存使用: {peak / 10**6:.2f} MB")

return result
~~~
```

## ## 🎓 面试准备指南

### ### 1. 算法原理理解

- 能够白板写出各种排序算法
- 理解时间/空间复杂度推导
- 知道各种算法的优缺点

### ### 2. 代码实现能力

- 写出清晰、健壮的代码
- 处理各种边界条件
- 进行适当的优化

### ### 3. 问题分析能力

- 快速识别问题类型
- 选择合适的算法
- 分析算法适用性

### ### 4. 沟通表达能力

- 清晰解释算法思路
- 分析时间/空间复杂度
- 讨论优化可能性

---

\*\*持续补充更多题目和解析...\*\*

=====

文件: AdditionalProblems\_Extended.md

=====

# 扩展排序算法题目与训练

## 📄 更多排序相关题目（扩展版）

## #### LeetCode 题目补充（新增）

### ##### 基础排序题目

#### 1. \*\*88. 合并两个有序数组\*\*

- 来源: LeetCode
- 链接: <https://leetcode.cn/problems/merge-sorted-array/>
- 难度: 简单
- 解法: 双指针从后向前合并
- 时间复杂度:  $O(m+n)$
- 空间复杂度:  $O(1)$
- 最优解: 是

#### 2. \*\*148. 排序链表\*\*

- 来源: LeetCode
- 链接: <https://leetcode.cn/problems/sort-list/>
- 难度: 中等
- 解法: 归并排序（链表版本）
- 时间复杂度:  $O(n \log n)$
- 空间复杂度:  $O(\log n)$  递归栈
- 最优解: 是

#### 3. \*\*912. 排序数组\*\*

- 来源: LeetCode
- 链接: <https://leetcode.cn/problems/sort-an-array/>
- 难度: 中等
- 解法: 各种排序算法的实现和比较
- 时间复杂度: 根据算法选择
- 空间复杂度: 根据算法选择

### ##### 快速选择相关

#### 4. \*\*973. 最接近原点的 K 个点\*\*

- 来源: LeetCode
- 链接: <https://leetcode.cn/problems/k-closest-points-to-origin/>
- 难度: 中等
- 解法: 快速选择/堆排序
- 时间复杂度:  $O(n)$  平均
- 空间复杂度:  $O(1)$
- 最优解: 是

#### 5. \*\*1054. 距离相等的条形码\*\*

- 来源: LeetCode
- 链接: <https://leetcode.cn/problems/distant-barcodes/>
- 难度: 中等

- 解法: 堆排序 (频率统计)
- 时间复杂度:  $O(n \log k)$
- 空间复杂度:  $O(n)$
- 最优解: 是

#### ##### 特殊排序

##### 6. \*\*164. 最大间距\*\* (已包含)

- 来源: LeetCode
- 链接: <https://leetcode.cn/problems/maximum-gap/>
- 难度: 困难
- 解法: 基数排序/桶排序
- 时间复杂度:  $O(n)$
- 空间复杂度:  $O(n)$

##### 7. \*\*324. 摆动排序 II\*\*

- 来源: LeetCode
- 链接: <https://leetcode.cn/problems/wiggle-sort-ii/>
- 难度: 中等
- 解法: 排序+双指针
- 时间复杂度:  $O(n \log n)$
- 空间复杂度:  $O(n)$
- 最优解: 是

##### 8. \*\*280. 摆动排序\*\*

- 来源: LeetCode
- 链接: <https://leetcode.cn/problems/wiggle-sort/>
- 难度: 中等
- 解法: 一次遍历交换
- 时间复杂度:  $O(n)$
- 空间复杂度:  $O(1)$
- 最优解: 是

#### ##### 困难题目

##### 9. \*\*493. 翻转对\*\*

- 来源: LeetCode
- 链接: <https://leetcode.cn/problems/reverse-pairs/>
- 难度: 困难
- 解法: 归并排序统计
- 时间复杂度:  $O(n \log n)$
- 空间复杂度:  $O(n)$
- 最优解: 是

#### ### 牛客网题目 (新增)

1. **\*\*NC140 排序\*\***
  - 来源: 牛客网
  - 链接: <https://www.nowcoder.com/practice/2baf799ea0594abd974d37139de27896>
  - 难度: 简单
  - 解法: 各种排序算法实现
  - 时间复杂度: 根据算法选择
  - 空间复杂度: 根据算法选择

2. **\*\*NC119 最小的 K 个数\*\***
  - 来源: 牛客网
  - 链接: <https://www.nowcoder.com/practice/6a296eb82cf844ca8539b57c23e6e9bf>
  - 难度: 中等
  - 解法: 堆/快速选择
  - 时间复杂度:  $O(n \log k) / O(n)$
  - 空间复杂度:  $O(k) / O(1)$

3. **\*\*NC88 寻找第 K 大\*\***
  - 来源: 牛客网
  - 链接: <https://www.nowcoder.com/practice/e016ad9b7f0b45048c58a9f27ba618bf>
  - 难度: 中等
  - 解法: 快速选择算法
  - 时间复杂度:  $O(n)$  平均
  - 空间复杂度:  $O(1)$

#### #### 剑指 Offer 题目 (新增)

1. **\*\*面试题 40. 最小的 k 个数\*\***
  - 来源: 剑指 Offer
  - 链接: 剑指 Offer 第二版第 40 题
  - 难度: 简单
  - 解法: 堆/快速选择
  - 时间复杂度:  $O(n \log k) / O(n)$
  - 空间复杂度:  $O(k) / O(1)$

2. **\*\*面试题 51. 数组中的逆序对\*\***
  - 来源: 剑指 Offer
  - 链接: 剑指 Offer 第二版第 51 题
  - 难度: 困难
  - 解法: 归并排序
  - 时间复杂度:  $O(n \log n)$
  - 空间复杂度:  $O(n)$

### 3. \*\*面试题 45. 把数组排成最小的数\*\*

- 来源: 剑指 Offer
- 链接: 剑指 Offer 第二版第 45 题
- 难度: 中等
- 解法: 自定义排序
- 时间复杂度:  $O(n \log n)$
- 空间复杂度:  $O(n)$

## #### HackerRank 题目 (新增)

### 1. \*\*Fraudulent Activity Notifications\*\*

- 来源: HackerRank
- 链接: <https://www.hackerrank.com/challenges/fraudulent-activity-notifications>
- 难度: 中等
- 解法: 滑动窗口+计数排序
- 时间复杂度:  $O(n)$
- 空间复杂度:  $O(1)$

### 2. \*\*Counting Inversions\*\*

- 来源: HackerRank
- 链接: <https://www.hackerrank.com/challenges/ctci-merge-sort>
- 难度: 困难
- 解法: 归并排序统计逆序对
- 时间复杂度:  $O(n \log n)$
- 空间复杂度:  $O(n)$

## #### Codeforces 题目 (新增)

### 1. \*\*Sort the Array\*\*

- 来源: Codeforces
- 链接: <https://codeforces.com/problemset/problem/451/B>
- 难度: 简单
- 解法: 寻找逆序段
- 时间复杂度:  $O(n)$
- 空间复杂度:  $O(1)$

### 2. \*\*Mike and Feet\*\*

- 来源: Codeforces
- 链接: <https://codeforces.com/problemset/problem/547/B>
- 难度: 中等
- 解法: 单调栈+排序
- 时间复杂度:  $O(n \log n)$
- 空间复杂度:  $O(n)$

#### #### AtCoder 题目（新增）

##### 1. \*\*Sorting\*\*

- 来源: AtCoder
- 链接: [https://atcoder.jp/contests/abc163/tasks/abc163\\_c](https://atcoder.jp/contests/abc163/tasks/abc163_c)
- 难度: 简单
- 解法: 计数排序
- 时间复杂度:  $O(n)$
- 空间复杂度:  $O(n)$

##### 2. \*\*Sorting a Segment\*\*

- 来源: AtCoder
- 链接: [https://atcoder.jp/contests/abc242/tasks/abc242\\_d](https://atcoder.jp/contests/abc242/tasks/abc242_d)
- 难度: 中等
- 解法: 滑动窗口+排序
- 时间复杂度:  $O(n \log k)$
- 空间复杂度:  $O(k)$

### ## 🎯 题目分类训练（扩展版）

#### ### 按算法分类训练

##### #### 归并排序训练

1. \*\*逆序对计数\*\* - 归并排序的经典应用
2. \*\*链表排序\*\* - 归并排序在链表上的实现
3. \*\*外部排序\*\* - 处理超大数据集的排序
4. \*\*翻转对统计\*\* - 扩展的逆序对问题

##### #### 快速排序训练

1. \*\*三路快排\*\* - 处理大量重复元素
2. \*\*快速选择\*\* - 寻找第  $K$  大/小元素
3. \*\*荷兰国旗问题\*\* - 三色排序
4. \*\*最接近点选择\*\* - 距离计算+快速选择

##### #### 堆排序训练

1. \*\*Top K 问题\*\* - 前  $K$  大/小元素
2. \*\*中位数查找\*\* - 动态数据流的中位数
3. \*\*优先级队列\*\* - 堆的实际应用
4. \*\*频率统计排序\*\* - 按频率排序

##### #### 特殊排序训练

1. \*\*基数排序\*\* - 处理大范围整数

2. \*\*桶排序\*\* - 均匀分布数据
3. \*\*计数排序\*\* - 小范围整数
4. \*\*希尔排序\*\* - 改进的插入排序

#### #### 按难度分级训练

##### ##### 初级（掌握基础）

- 实现各种基础排序算法
- 理解时间/空间复杂度
- 处理简单边界条件
- 编写单元测试

##### ##### 中级（应用扩展）

- 解决 LeetCode 中等难度题目
- 掌握算法优化技巧
- 处理复杂边界情况
- 进行性能分析

##### ##### 高级（深入理解）

- 解决困难题目
- 理解算法底层原理
- 进行性能优化和工程化
- 处理大数据量场景

#### ## 🌟 解题思路总结（扩展版）

#### ### 见到排序题目的思考流程

##### 1. \*\*分析题目要求\*\*

- 是否需要稳定排序？
- 是否有空间限制？
- 数据规模有多大？
- 数据分布特征？
- 是否需要原地排序？

##### 2. \*\*选择合适算法\*\*

- 小数据 ( $n < 50$ ): 插入/选择排序
- 大数据: 快速/归并/堆排序
- 需要稳定: 归并排序
- 空间紧张: 堆排序/原地快排
- 数据范围小: 计数/基数排序

##### 3. \*\*考虑优化策略\*\*

- 小数组优化
- 随机化避免最坏情况
- 处理重复元素
- 利用数据特性

#### 4. \*\*处理边界条件\*\*

- 空数组
- 单元素
- 已排序/逆序
- 大量重复
- 极端值

### #### 常见题型模式（扩展版）

#### ##### 模式 1: Top K 问题

- 特征: 寻找前 K 大/小元素
- 解法: 快速选择( $O(n)$ )或堆( $O(n \log k)$ )
- 变种: 最接近点、频率最高元素

#### ##### 模式 2: 区间合并

- 特征: 重叠区间合并
- 解法: 按起点排序后合并
- 变种: 会议室安排、区间插入

#### ##### 模式 3: 颜色分类

- 特征: 有限种类的排序
- 解法: 计数排序/多指针
- 变种: 荷兰国旗、三路快排

#### ##### 模式 4: 逆序对统计

- 特征: 统计逆序对数量
- 解法: 归并排序
- 变种: 翻转对、重要逆序对

#### ##### 模式 5: 自定义排序

- 特征: 特殊的比较规则
- 解法: 实现自定义比较器
- 变种: 字符串拼接、特殊规则排序

### ## 🔧 代码实现要点（扩展版）

#### ### Java 实现要点

```
```java
```

```

// 1. 使用泛型支持多种数据类型
public class SortAlgorithms<T extends Comparable<T>> {

    // 2. 异常处理
    public void sort(T[] array) {
        if (array == null) throw new IllegalArgumentException();
        if (array.length <= 1) return;

        // 排序逻辑
    }

    // 3. 性能监控
    public void sortWithTiming(T[] array) {
        long start = System.nanoTime();
        sort(array);
        long end = System.nanoTime();
        System.out.println("耗时: " + (end - start) + "纳秒");
    }

    // 4. 内存监控
    public void sortWithMemory(T[] array) {
        Runtime runtime = Runtime.getRuntime();
        long memoryBefore = runtime.totalMemory() - runtime.freeMemory();
        sort(array);
        long memoryAfter = runtime.totalMemory() - runtime.freeMemory();
        System.out.println("内存使用: " + (memoryAfter - memoryBefore) + "字节");
    }
}
```

```

### ### C++实现要点

```

```cpp
// 1. 模板支持泛型
template<typename T>
class SortAlgorithms {
public:
    // 2. 异常安全
    void sort(std::vector<T>& nums) {
        if (nums.empty()) return;

        // 使用 RAII 确保资源安全
        // 排序逻辑
    }
}
```

```
// 3. 移动语义优化
std::vector<T> sorted(std::vector<T> nums) {
    sort(nums);
    return nums; // 移动语义优化
}

// 4. 性能分析
void sortWithProfiling(std::vector<T>& nums) {
    auto start = std::chrono::high_resolution_clock::now();
    sort(nums);
    auto end = std::chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start);
    std::cout << "耗时: " << duration.count() << "微秒" << std::endl;
}
};

```

```

#### Python 实现要点

```
``` python
class SortAlgorithms:

    # 1. 类型注解
    @staticmethod
    def sort(nums: List[int]) -> List[int]:
        # 2. 输入验证
        if not isinstance(nums, list):
            raise TypeError("输入必须是列表")

        # 3. 边界处理
        if len(nums) <= 1:
            return nums.copy()

        # 排序逻辑
        return sorted_nums
```

# 4. 性能测试装饰器

```
@staticmethod
def timed_sort(nums):
    import time
    start = time.time()
    result = SortAlgorithms.sort(nums)
    end = time.time()
    print(f"排序耗时: {end - start:.6f}秒")
```

```

    return result

# 5. 内存分析装饰器
@staticmethod
def memory_profiled_sort(nums):
    import tracemalloc
    tracemalloc.start()
    result = SortAlgorithms.sort(nums)
    current, peak = tracemalloc.get_traced_memory()
    tracemalloc.stop()
    print(f"峰值内存: {peak / 1024:.2f} KB")
    return result
```

```

## ## 📈 复杂度分析深度（扩展版）

### ### 归并排序复杂度推导

```

$$\begin{aligned}
 T(n) &= 2T(n/2) + O(n) \\
 &= 2[2T(n/4) + O(n/2)] + O(n) \\
 &= 4T(n/4) + 2O(n/2) + O(n) \\
 &= 4T(n/4) + 2O(n) \\
 &= \dots \\
 &= 2^k T(n/2^k) + kO(n)
 \end{aligned}$$

当  $n/2^k = 1 \Rightarrow k = \log_2 n$

$$T(n) = nT(1) + O(n \log n) = O(n \log n)$$

```

### ### 快速排序复杂度分析

**\*\*最好情况\*\*:** 每次划分均衡

```

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

```

**\*\*最坏情况\*\*:** 每次划分极端不平衡

```

$$T(n) = T(n-1) + O(n) = O(n^2)$$

```

**\*\*平均情况\*\*:** 通过随机化达到  $O(n \log n)$

### ### 堆排序复杂度分析

建堆:  $O(n)$

每次调整:  $O(\log n)$

总复杂度:  $O(n \log n)$

...

## ## 🔧 工程化实践（扩展版）

### ### 1. 单元测试设计

``` python

```
import unittest
```

```
class TestSortAlgorithms(unittest.TestCase):
```

```
    def test_empty_array(self):
```

```
        self.assertEqual(SortAlgorithms.sort([]), [])
```

```
    def test_single_element(self):
```

```
        self.assertEqual(SortAlgorithms.sort([1]), [1])
```

```
    def test_already_sorted(self):
```

```
        self.assertEqual(SortAlgorithms.sort([1, 2, 3]), [1, 2, 3])
```

```
    def test_reverse_sorted(self):
```

```
        self.assertEqual(SortAlgorithms.sort([3, 2, 1]), [1, 2, 3])
```

```
    def test_duplicate_elements(self):
```

```
        self.assertEqual(SortAlgorithms.sort([2, 2, 1, 1]), [1, 1, 2, 2])
```

```
    def test_large_random_array(self):
```

```
        import random
```

```
        nums = [random.randint(1, 1000) for _ in range(1000)]
```

```
        sorted_nums = SortAlgorithms.sort(nums.copy())
```

```
        self.assertEqual(sorted_nums, sorted(nums))
```

```
    def test_negative_numbers(self):
```

```
        self.assertEqual(SortAlgorithms.sort([-3, -1, -2]), [-3, -2, -1])
```

```
    def test_mixed_numbers(self):
```

```
        self.assertEqual(SortAlgorithms.sort([3, -1, 0, -2, 1]), [-2, -1, 0, 1, 3])
```

...

### ### 2. 性能基准测试

``` python

```
def benchmark_different_sizes():
    sizes = [100, 1000, 10000, 100000]
    algorithms = {
        '归并排序': merge_sort,
        '快速排序': quick_sort,
        '堆排序': heap_sort,
        '内置排序': sorted
    }

    for size in sizes:
        test_data = generate_test_data(size)
        print(f"\n数据规模: {size}")

        for name, algorithm in algorithms.items():
            time_taken = time_algorithm(algorithm, test_data)
            print(f"{name}: {time_taken:.6f}秒")
```

```

### ### 3. 内存使用监控

```
``` python
import tracemalloc

def monitor_memory_usage(algorithm, data):
    tracemalloc.start()

    # 执行算法
    result = algorithm(data)

    current, peak = tracemalloc.get_traced_memory()
    tracemalloc.stop()

    print(f"当前内存使用: {current / 10**6:.2f} MB")
    print(f"峰值内存使用: {peak / 10**6:.2f} MB")

    return result
```

```

### ### 4. 压力测试

```
``` python
def stress_test(algorithm, max_size=1000000):
    """压力测试: 测试算法在大数据量下的表现"""
    print("开始压力测试...")

```

```
# 测试随机数据
random_data = [random.randint(1, 1000000) for _ in range(max_size)]
start_time = time.time()
result = algorithm(random_data)
end_time = time.time()

print(f"随机数据排序耗时: {end_time - start_time:.2f}秒")

# 测试已排序数据
sorted_data = list(range(max_size))
start_time = time.time()
result = algorithm(sorted_data)
end_time = time.time()

print(f"已排序数据排序耗时: {end_time - start_time:.2f}秒")

# 测试逆序数据
reverse_data = list(range(max_size, 0, -1))
start_time = time.time()
result = algorithm(reverse_data)
end_time = time.time()

print(f"逆序数据排序耗时: {end_time - start_time:.2f}秒")
```

```

## ## 🎓 面试准备指南（扩展版）

### ### 1. 算法原理理解

- 能够白板写出各种排序算法
- 理解时间/空间复杂度推导
- 知道各种算法的优缺点
- 理解稳定性的概念和重要性

### ### 2. 代码实现能力

- 写出清晰、健壮的代码
- 处理各种边界条件
- 进行适当的优化
- 编写完整的测试用例

### ### 3. 问题分析能力

- 快速识别问题类型
- 选择合适的算法
- 分析算法适用性

- 考虑优化空间

#### #### 4. 沟通表达能力

- 清晰解释算法思路
- 分析时间/空间复杂度
- 讨论优化可能性
- 展示调试和优化过程

#### #### 5. 系统设计能力

- 设计可扩展的排序系统
- 考虑大数据量处理
- 设计分布式排序方案
- 考虑容错和恢复机制

---

\*\*持续补充更多题目和解析...\*\*

=====

文件: ADDITIONAL\_SORTING\_PROBLEMS.md

=====

## # 排序算法专题补充题目清单

### ## 📚 概述

本文件汇总了[class001] (file:///d:/Upan/src/algorith-journey/src/algorith-journey/src/class001) 排序算法专题中添加的所有相关题目，包括 LeetCode、Aizu Online Judge、牛客网等平台的题目，以及相关的实现代码和资源链接。

### ## ⚡ 核心排序算法相关题目

#### ### 归并排序相关题目

##### #### LeetCode 题目

###### 1. \*\*912. 排序数组\*\*

- 链接: <https://leetcode.cn/problems/sort-an-array/>
- 难度: 中等
- 相关算法: 各种排序算法实现

###### 2. \*\*148. 排序链表\*\*

- 链接: <https://leetcode.cn/problems/sort-list/>
- 难度: 中等

- 相关算法：归并排序（链表版本）

### 3. \*\*面试题 51. 数组中的逆序对\*\*

- 链接：剑指 Offer 第二版第 51 题

- 难度：困难

- 相关算法：归并排序

### 4. \*\*493. 翻转对\*\*

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

- 难度：困难

- 相关算法：归并排序

## #### Aizu Online Judge 题目

### 1. \*\*ALDS1\_5\_B: Merge Sort\*\*

- 链接：[https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\\_5\\_B](https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_5_B)

- 难度：中等

- 相关算法：归并排序

### 2. \*\*ALDS1\_5\_D: The Number of Inversions\*\*

- 链接：[https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\\_5\\_D](https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_5_D)

- 难度：中等

- 相关算法：逆序对统计

## ### 快速排序相关题目

## #### LeetCode 题目

### 1. \*\*215. 数组中的第 K 个最大元素\*\*

- 链接：<https://leetcode.cn/problems/kth-largest-element-in-an-array/>

- 难度：中等

- 相关算法：快速选择、堆排序

### 2. \*\*973. 最接近原点的 K 个点\*\*

- 链接：<https://leetcode.cn/problems/k-closest-points-to-origin/>

- 难度：中等

- 相关算法：快速选择、堆排序

## #### Aizu Online Judge 题目

### 1. \*\*ALDS1\_6\_B: Partition\*\*

- 链接：[https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\\_6\\_B](https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_6_B)

- 难度：中等

- 相关算法：快速排序分区

### 2. \*\*ALDS1\_6\_C: Quick Sort\*\*

- 链接: [https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\\_6\\_C](https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_6_C)
- 难度: 中等
- 相关算法: 快速排序

### ### 堆排序相关题目

#### #### LeetCode 题目

1. **1054. 距离相等的条形码**
  - 链接: <https://leetcode.cn/problems/distant-barcodes/>
  - 难度: 中等
  - 相关算法: 堆排序 (频率统计)
2. **347. 前 K 个高频元素**
  - 链接: <https://leetcode.cn/problems/top-k-frequent-elements/>
  - 难度: 中等
  - 相关算法: 堆排序、桶排序

#### #### Aizu Online Judge 题目

1. **ALDS1\_9\_A: Complete Binary Tree**
  - 链接: [https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\\_9\\_A](https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_9_A)
  - 难度: 简单
  - 相关算法: 完全二叉树
2. **ALDS1\_9\_B: Maximum Heap**
  - 链接: [https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\\_9\\_B](https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_9_B)
  - 难度: 简单
  - 相关算法: 最大堆
3. **ALDS1\_9\_C: Priority Queue**
  - 链接: [https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\\_9\\_C](https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_9_C)
  - 难度: 中等
  - 相关算法: 优先队列

### ### 特殊排序问题

#### #### LeetCode 题目

1. **75. 颜色分类 (荷兰国旗问题)**
  - 链接: <https://leetcode.cn/problems/sort-colors/>
  - 难度: 中等
  - 相关算法: 三指针法、计数排序
2. **56. 合并区间**
  - 链接: <https://leetcode.cn/problems/merge-intervals/>

- 难度: 中等
- 相关算法: 排序+合并

### 3. \*\*164. 最大间距\*\*

- 链接: <https://leetcode.cn/problems/maximum-gap/>
- 难度: 困难
- 相关算法: 基数排序、桶排序

### 4. \*\*324. 摆动排序 II\*\*

- 链接: <https://leetcode.cn/problems/wiggle-sort-ii/>
- 难度: 中等
- 相关算法: 排序+双指针

#### Aizu Online Judge 题目

#### 1. \*\*ALDS1\_2\_C: Stable Sort\*\*

- 链接: [https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\\_2\\_C](https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_2_C)
- 难度: 简单
- 相关算法: 稳定排序

#### 2. \*\*ALDS1\_6\_D: Minimum Cost Sort\*\*

- 链接: [https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\\_6\\_D](https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_6_D)
- 难度: 困难
- 相关算法: 最小成本排序

### 基础排序算法

#### Aizu Online Judge 题目

#### 1. \*\*ALDS1\_2\_A: Bubble Sort\*\*

- 链接: [https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\\_2\\_A](https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_2_A)
- 难度: 简单
- 相关算法: 冒泡排序

#### 2. \*\*ALDS1\_2\_B: Selection Sort\*\*

- 链接: [https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\\_2\\_B](https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_2_B)
- 难度: 简单
- 相关算法: 选择排序

#### 3. \*\*ALDS1\_2\_D: Shell Sort\*\*

- 链接: [https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\\_2\\_D](https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_2_D)
- 难度: 中等
- 相关算法: 希尔排序

## 🐄 牛客网题目

1. **\*\*NC140 排序\*\***
  - 链接: <https://www.nowcoder.com/practice/2baf799ea0594abd974d37139de27896>
  - 难度: 简单
  - 相关算法: 各种排序算法实现
2. **\*\*NC119 最小的 K 个数\*\***
  - 链接: <https://www.nowcoder.com/practice/6a296eb82cf844ca8539b57c23e6e9bf>
  - 难度: 中等
  - 相关算法: 堆、快速选择
3. **\*\*NC88 寻找第 K 大\*\***
  - 链接: <https://www.nowcoder.com/practice/e016ad9b7f0b45048c58a9f27ba618bf>
  - 难度: 中等
  - 相关算法: 快速选择算法
4. **\*\*NC37 合并区间\*\***
  - 链接: <https://www.nowcoder.com/practice/69f4e5b7ad284a478777cb2a17fb5e6a>
  - 难度: 中等
  - 相关算法: 排序+合并

## ## 🗡️ 剑指 Offer 题目

1. **\*\*面试题 40. 最小的 k 个数\*\***
  - 链接: 剑指 Offer 第二版第 40 题
  - 难度: 简单
  - 相关算法: 堆、快速选择
2. **\*\*面试题 45. 把数组排成最小的数\*\***
  - 链接: 剑指 Offer 第二版第 45 题
  - 难度: 中等
  - 相关算法: 自定义排序

## ## 💻 HackerRank 题目

1. **\*\*Fraudulent Activity Notifications\*\***
  - 链接: <https://www.hackerrank.com/challenges/fraudulent-activity-notifications>
  - 难度: 中等
  - 相关算法: 滑动窗口+计数排序
2. **\*\*Counting Inversions\*\***
  - 链接: <https://www.hackerrank.com/challenges/ctci-merge-sort>
  - 难度: 困难

- 相关算法：归并排序统计逆序对

### 3. \*\*Lily's Homework\*\*

- 链接: <https://www.hackerrank.com/challenges/lilys-homework>

- 难度: 中等

- 相关算法: 排序、贪心

## ## 🏆 其他平台题目

### #### Codeforces

#### 1. \*\*Sort the Array\*\*

- 链接: <https://codeforces.com/problemset/problem/451/B>

- 难度: 简单

- 相关算法: 排序、区间反转

#### 2. \*\*Mike and Feet\*\*

- 链接: <https://codeforces.com/problemset/problem/547/B>

- 难度: 中等

- 相关算法: 单调栈、排序

### #### AtCoder

#### 1. \*\*Sorting\*\*

- 链接: [https://atcoder.jp/contests/abc217/tasks/abc217\\_c](https://atcoder.jp/contests/abc217/tasks/abc217_c)

- 难度: 简单

- 相关算法: 基础排序

### #### USACO

#### 1. \*\*Milking Cows\*\*

- 链接: <http://www.usaco.org/index.php?page=viewproblem2&cpid=855>

- 难度: 简单

- 相关算法: 区间合并

## ## 📚 学习资源

### #### 推荐书籍

1. 《算法导论》 - 排序算法理论基础

2. 《编程珠玑》 - 排序算法的实际应用

3. 《算法》 - 各种排序算法的实现和比较

### #### 在线资源

#### 1. \*\*VisualGo\*\* - 排序算法可视化

- 链接: <https://visualgo.net/en/sorting>

2. **USFCA Sorting Animations** - 动画演示
  - 链接: <https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html>
  
3. **Khan Academy** - 算法课程
  - 链接: <https://www.khanacademy.org/computing/computer-science/algorithms>

### ### 实践平台

1. **LeetCode** - 算法题目练习
2. **HackerRank** - 编程挑战
3. **牛客网** - 国内算法题库

### ## 🔧 代码实现

本专题提供了 Java、Python、C++三种语言的完整代码实现，包括：

1. **基础排序算法**:
  - 归并排序
  - 快速排序
  - 堆排序
  - 冒泡排序
  - 选择排序
  - 插入排序
  
2. **扩展题目实现**:
  - 第 K 大元素
  - 颜色分类
  - 合并区间
  - 前 K 个高频元素
  - 最大间距
  - 链表排序

### ## 📄 文档和说明

1. **README.md** - 专题概述和核心算法说明
2. **ProblemLinks.md** - 所有相关题目链接汇总
3. **AdditionalProblems.md** - 补充题目清单
4. **EngineeringConsiderations.md** - 工程化考量说明

---

\*最后更新：2025 年 10 月 23 日\*

=====

文件: EngineeringConsiderations.md

---

## # 排序算法的工程化考量

### ## 一、异常处理与边界场景

#### #### 1.1 输入验证

```
```java
// 空数组检查
if (arr == null || arr.length == 0) {
    return arr; // 或抛出异常
}
```

#### // 边界值检查

```
if (k < 1 || k > arr.length) {
    throw new IllegalArgumentException("k 值超出范围");
}
```

```

#### #### 1.2 极端数据场景

- \*\*空数组\*\*: 直接返回或抛出明确异常
- \*\*单元素数组\*\*: 无需排序，直接返回
- \*\*已排序数组\*\*: 某些算法（如插入排序）有优化空间
- \*\*逆序数组\*\*: 测试快速排序的最坏情况
- \*\*全相同元素\*\*: 测试稳定性

### ## 二、性能优化策略

#### #### 2.1 算法选择策略

```
```java
public static void smartSort(int[] arr) {
    if (arr.length < 50) {
        // 小数组使用插入排序
        insertionSort(arr);
    } else if (arr.length < 1000) {
        // 中等数组使用快速排序
        quickSort(arr);
    } else {
        // 大数组使用归并排序（稳定且性能可预测）
        mergeSort(arr);
    }
}
```

```

```

#### 2.2 内存优化

```java
// 复用辅助数组，避免频繁创建
private static int[] mergeHelper;

public static void mergeSort(int[] arr) {
    if (mergeHelper == null || mergeHelper.length < arr.length) {
        mergeHelper = new int[arr.length];
    }
    // ... 排序逻辑
}
```

```

## ## 三、多语言实现差异

### ### 3.1 Java vs C++ vs Python 关键差异

| 特性   | Java          | C++         | Python   |
|------|---------------|-------------|----------|
| 内存管理 | GC 自动管理       | 手动/RAII     | 引用计数+GC  |
| 数组边界 | 运行时检查         | 无检查         | 运行时检查    |
| 递归深度 | 受栈大小限制        | 受栈大小限制      | 限制较严格    |
| 内置排序 | Arrays.sort() | std::sort() | sorted() |

### ### 3.2 语言特定优化

```

```java
// Java: 使用 System.arraycopy 进行数组复制
System.arraycopy(src, srcPos, dest, destPos, length);

```

```

// C++: 使用 std::move 避免不必要的拷贝
std::move(begin, end, destination);

```

```

// Python: 使用切片操作
arr[:] = sorted_arr
```

```

## ## 四、测试策略

### ### 4.1 单元测试设计

```

```java
@Test
public void testSortAlgorithms() {

```

```
// 边界测试
testEmptyArray();
testSingleElement();
testAlreadySorted();

// 功能测试
testRandomArray();
testDuplicateElements();

// 性能测试
testPerformance();
}

```

```

### ### 4.2 性能测试指标

```
``` java
public class PerformanceMetrics {
    private long startTime;
    private long memoryBefore;

    public void start() {
        startTime = System.nanoTime();
        memoryBefore = Runtime.getRuntime().totalMemory() -
                        Runtime.getRuntime().freeMemory();
    }

    public PerformanceResult stop() {
        long endTime = System.nanoTime();
        long memoryAfter = Runtime.getRuntime().totalMemory() -
                            Runtime.getRuntime().freeMemory();

        return new PerformanceResult(
            (endTime - startTime) / 1_000_000.0, // 毫秒
            memoryAfter - memoryBefore // 内存变化
        );
    }
}
```

```

## ## 五、工程最佳实践

```
### 5.1 代码可读性
``` java

```

```
// 好的命名
public void mergeSortedArrays(int[] left, int[] right, int[] result) {
    // 清晰的注释说明算法步骤
}

// 模块化设计
public class SortFactory {
    public static Sorter getSorter(SortType type) {
        switch (type) {
            case QUICK: return new QuickSorter();
            case MERGE: return new MergeSorter();
            case HEAP: return new HeapSorter();
        }
    }
}
```
```

```

#### ### 5.2 错误处理

```
```java
public class SortingException extends RuntimeException {
    public SortingException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

```
public void safeSort(int[] arr) {
    try {
        quickSort(arr);
    } catch (StackOverflowError e) {
        // 递归深度过大，切换到迭代版本
        iterativeQuickSort(arr);
    }
}
```
```

```

## ## 六、实际应用场景

### ### 6.1 数据库排序优化

```
```sql
-- 数据库中的排序通常使用外部归并排序
SELECT * FROM table ORDER BY column
-- 当数据量大于内存时，使用多路归并排序
```
```

```

#### ### 6.2 大数据处理

```
```java
// MapReduce 中的排序阶段
public class SortMapper extends Mapper {
    // 每个 mapper 内部使用快速排序
    // reducer 使用多路归并排序合并结果
}
```
```

```

#### ### 6.3 机器学习应用

```
```python
# 特征排序用于特征选择
from sklearn.feature_selection import SelectKBest

# 模型参数排序用于超参数优化
sorted_params = sorted(param_grid, key=lambda x: x['score'])
```
```

```

### ## 七、调试与问题定位

#### ### 7.1 调试技巧

```
```java
public void debugQuickSort(int[] arr, int left, int right) {
    System.out.printf("排序区间 [%d, %d]: %s%n",
        left, right, Arrays.toString(Arrays.copyOfRange(arr, left, right+1)));
}

if (left < right) {
    int pivot = partition(arr, left, right);
    debugQuickSort(arr, left, pivot - 1);
    debugQuickSort(arr, pivot + 1, right);
}

}
```
```

```

#### ### 7.2 性能分析

```
```java
// 使用 JMH 进行微基准测试
@Benchmark
@BenchmarkMode(Mode.AverageTime)
public void benchmarkQuickSort() {
    int[] testData = generateTestData();
    quickSort(testData);
}
```
```

```

```
}
```

```
...
```

## ## 八、安全考量

### #### 8.1 输入验证

```
```java
public void validateInput(int[] arr, int k) {
    if (arr == null) {
        throw new IllegalArgumentException("数组不能为 null");
    }
    if (k <= 0 || k > arr.length) {
        throw new IllegalArgumentException("k 值必须在 1 到数组长度之间");
    }
    // 检查整数溢出
    if (arr.length > Integer.MAX_VALUE - 8) {
        throw new IllegalArgumentException("数组过大");
    }
}
```

```
...
```

### #### 8.2 资源限制

```
```java
public class ResourceAwareSorter {
    private static final long MAX_MEMORY = Runtime.getRuntime().maxMemory();

    public void sortLargeArray(int[] arr) {
        long requiredMemory = (long) arr.length * 4 * 2; // 数组大小 * int 大小 * 2(辅助数组)

        if (requiredMemory > MAX_MEMORY * 0.7) {
            // 使用外部排序
            externalSort(arr);
        } else {
            // 使用内存排序
            internalSort(arr);
        }
    }
}
```

```
...
```

通过以上工程化考量，可以确保排序算法在实际生产环境中的稳定性、性能和可维护性。

```
=====
```

文件: FinalChecklist.md

---

## # Class001 排序算法学习完成检查清单

### ## 一、基础掌握程度检查

#### #### 算法实现能力

- [ ] 能够手写归并排序（递归+迭代版本）
- [ ] 能够手写快速排序（基础+三路分区版本）
- [ ] 能够手写堆排序（建堆+调整）
- [ ] 理解冒泡、选择、插入排序的实现

#### #### 复杂度分析能力

- [ ] 能够分析各种排序算法的时间复杂度（最好/平均/最坏）
- [ ] 能够分析空间复杂度（原地排序 vs 需要额外空间）
- [ ] 理解稳定性概念并能判断算法是否稳定

#### #### 代码调试能力

- [ ] 能够使用打印调试法定位排序算法错误
- [ ] 掌握边界条件测试方法
- [ ] 能够分析算法性能瓶颈

### ## 二、题目完成情况检查

#### #### LeetCode 核心题目

- [ ] 912. 排序数组 - 掌握多种排序实现: <https://leetcode.cn/problems/sort-an-array/>
- [ ] 215. 数组中的第 K 个最大元素 - 快速选择算法: <https://leetcode.cn/problems/kth-largest-element-in-an-array/>
- [ ] 75. 颜色分类 - 荷兰国旗问题: <https://leetcode.cn/problems/sort-colors/>
- [ ] 56. 合并区间 - 排序+合并模式: <https://leetcode.cn/problems/merge-intervals/>
- [ ] 148. 排序链表 - 链表上的归并排序: <https://leetcode.cn/problems/sort-list/>

#### #### 扩展题目掌握

- [ ] 计数排序、基数排序、桶排序的实现
- [ ] 理解非比较排序的适用场景
- [ ] 掌握各种排序变种算法

### ## 三、工程化能力检查

#### #### 多语言实现

- [ ] Java 版本: 掌握 Arrays.sort() 的实现原理
- [ ] C++版本: 理解 std::sort() 的优化策略

- [ ] Python 版本：熟悉 `sorted()` 的内部机制

####  性能优化

- [ ] 掌握小数组优化技巧

- [ ] 理解递归深度控制方法

- [ ] 能够根据数据特征选择最优算法

####  异常处理

- [ ] 能够处理空数组、单元素等边界情况

- [ ] 掌握输入验证和错误处理

- [ ] 理解资源限制下的算法选择

## ## 四、面试准备检查

####  理论基础

- [ ] 能够清晰解释各种排序算法的原理

- [ ] 掌握时间/空间复杂度的推导过程

- [ ] 理解算法稳定性的重要性

####  代码实现

- [ ] 能够现场手写核心排序算法

- [ ] 代码风格规范，注释清晰

- [ ] 边界处理完整，异常考虑周全

####  问题分析

- [ ] 能够分析算法优缺点和适用场景

- [ ] 掌握算法比较和选择的方法

- [ ] 能够解决排序相关的变种问题

## ## 五、进阶学习方向

####  算法理论深化

- [ ] 学习计算复杂性理论

- [ ] 研究随机算法和平摊分析

- [ ] 探索排序算法的理论极限

####  工程实践扩展

- [ ] 学习并行排序算法

- [ ] 掌握外部排序技术

- [ ] 研究缓存优化策略

####  竞赛专项训练

- [ ] 参与在线评测平台的排序专项

- [ ] 解决更复杂的排序相关问题
- [ ] 学习高级优化技巧

## ## 六、自我评估问卷

### #### 理解程度评估（1-5 分）

1. \*\*算法原理理解\*\*: \_\_\_\_\_ 分

- 1 分: 基本了解概念
- 3 分: 能够解释原理
- 5 分: 深入理解数学基础

2. \*\*代码实现能力\*\*: \_\_\_\_\_ 分

- 1 分: 需要参考模板
- 3 分: 能够独立实现
- 5 分: 能够优化和创新

3. \*\*复杂度分析\*\*: \_\_\_\_\_ 分

- 1 分: 记住标准答案
- 3 分: 能够推导分析
- 5 分: 能够进行平摊分析

4. \*\*问题解决能力\*\*: \_\_\_\_\_ 分

- 1 分: 解决标准问题
- 3 分: 解决变种问题
- 5 分: 能够设计新算法

### #### 建议改进方向

根据自我评估结果，重点关注得分较低的领域：

- 如果原理理解不足：重新学习算法数学基础
- 如果代码实现困难：多练习手写代码
- 如果复杂度分析薄弱：学习计算理论
- 如果问题解决能力待提高：多做扩展题目

## ## 七、后续学习计划

### #### 短期目标（1-2 周）

1. 完成所有基础题目的代码实现
2. 掌握各种排序算法的复杂度分析
3. 能够应对常见的排序相关面试题

### #### 中期目标（1-2 月）

1. 熟练解决 LeetCode 中等难度排序问题
2. 理解排序算法在实际工程中的应用

### 3. 掌握多语言排序实现差异

#### #### 长期目标（3-6 月）

1. 参与算法竞赛，解决复杂排序问题
2. 研究排序算法的前沿发展
3. 能够进行排序算法的性能调优

## ## 八、资源使用建议

#### #### 核心学习资源

- **本仓库代码**: 作为学习和参考的基础
- **LeetCode 平台**: 进行题目练习和测试
- **算法教材**: 深入理解理论基础

#### #### 练习方法

1. **第一遍**: 理解算法原理和代码实现
2. **第二遍**: 独立手写代码，不参考模板
3. **第三遍**: 进行性能优化和边界处理
4. **第四遍**: 解决变种问题和扩展应用

#### #### 检验标准

- 能够在不参考任何资料的情况下，30 分钟内完成核心排序算法的实现
- 能够清晰解释各种排序算法的优缺点和适用场景
- 能够解决常见的排序相关面试题和竞赛题

通过系统性地完成这个检查清单，可以确保全面掌握排序算法这一重要的计算机科学基础知识点。

---

文件: ProblemLinks.md

---

## # 排序算法相关题目链接汇总

### ## 📄 LeetCode 题目

#### #### 基础排序题目

1. **88. 合并两个有序数组**
  - 难度: 简单
  - 链接: <https://leetcode.cn/problems/merge-sorted-array/>
  - 相关算法: 归并排序、双指针
2. **148. 排序链表**
  - 难度: 中等

- 链接: <https://leetcode.cn/problems/sort-list/>

- 相关算法: 归并排序 (链表版本)

### 3. \*\*912. 排序数组\*\*

- 难度: 中等

- 链接: <https://leetcode.cn/problems/sort-an-array/>

- 相关算法: 各种排序算法实现

### #### 快速选择相关

#### 4. \*\*215. 数组中的第 K 个最大元素\*\*

- 难度: 中等

- 链接: <https://leetcode.cn/problems/kth-largest-element-in-an-array/>

- 相关算法: 快速选择、堆排序

#### 5. \*\*973. 最接近原点的 K 个点\*\*

- 难度: 中等

- 链接: <https://leetcode.cn/problems/k-closest-points-to-origin/>

- 相关算法: 快速选择、堆排序

#### 6. \*\*1054. 距离相等的条形码\*\*

- 难度: 中等

- 链接: <https://leetcode.cn/problems/distant-barcodes/>

- 相关算法: 堆排序 (频率统计)

### #### 特殊排序问题

#### 7. \*\*75. 颜色分类\*\*

- 难度: 中等

- 链接: <https://leetcode.cn/problems/sort-colors/>

- 相关算法: 三指针法、计数排序

#### 8. \*\*56. 合并区间\*\*

- 难度: 中等

- 链接: <https://leetcode.cn/problems/merge-intervals/>

- 相关算法: 排序+合并

#### 9. \*\*164. 最大间距\*\*

- 难度: 困难

- 链接: <https://leetcode.cn/problems/maximum-gap/>

- 相关算法: 基数排序、桶排序

#### 10. \*\*324. 摆动排序 II\*\*

- 难度: 中等

- 链接: <https://leetcode.cn/problems/wiggle-sort-ii/>

- 相关算法：排序+双指针

## 11. \*\*347. 前 K 个高频元素\*\*

- 难度：中等
- 链接：<https://leetcode.cn/problems/top-k-frequent-elements/>
- 相关算法：堆排序、桶排序

### ### 逆序对问题

## 12. \*\*493. 翻转对\*\*

- 难度：困难
- 链接：<https://leetcode.cn/problems/reverse-pairs/>
- 相关算法：归并排序

## 13. \*\*面试题 51. 数组中的逆序对\*\*

- 难度：困难
- 链接：剑指 Offer 第二版第 51 题
- 相关算法：归并排序

### ## 🐄 牛客网题目

## 1. \*\*NC140 排序\*\*

- 难度：简单
- 链接：<https://www.nowcoder.com/practice/2baf799ea0594abd974d37139de27896>
- 相关算法：各种排序算法实现

## 2. \*\*NC119 最小的 K 个数\*\*

- 难度：中等
- 链接：<https://www.nowcoder.com/practice/6a296eb82cf844ca8539b57c23e6e9bf>
- 相关算法：堆、快速选择

## 3. \*\*NC88 寻找第 K 大\*\*

- 难度：中等
- 链接：<https://www.nowcoder.com/practice/e016ad9b7f0b45048c58a9f27ba618bf>
- 相关算法：快速选择算法

## 4. \*\*NC37 合并区间\*\*

- 难度：中等
- 链接：<https://www.nowcoder.com/practice/69f4e5b7ad284a478777cb2a17fb5e6a>
- 相关算法：排序+合并

### ## ✨ 剑指 Offer 题目

## 1. \*\*面试题 40. 最小的 k 个数\*\*

- 难度: 简单
- 链接: 剑指 Offer 第二版第 40 题
- 相关算法: 堆、快速选择

## 2. \*\*面试题 51. 数组中的逆序对\*\*

- 难度: 困难
- 链接: 剑指 Offer 第二版第 51 题
- 相关算法: 归并排序

## 3. \*\*面试题 45. 把数组排成最小的数\*\*

- 难度: 中等
- 链接: 剑指 Offer 第二版第 45 题
- 相关算法: 自定义排序

## ## 🏆 HackerRank 题目

### 1. \*\*Fraudulent Activity Notifications\*\*

- 难度: 中等
- 链接: <https://www.hackerrank.com/challenges/fraudulent-activity-notifications>
- 相关算法: 滑动窗口+计数排序

### 2. \*\*Counting Inversions\*\*

- 难度: 困难
- 链接: <https://www.hackerrank.com/challenges/ctci-merge-sort>
- 相关算法: 归并排序统计逆序对

### 3. \*\*Lily's Homework\*\*

- 难度: 中等
- 链接: <https://www.hackerrank.com/challenges/lilys-homework>
- 相关算法: 排序、贪心

## ## 🎯 其他平台题目

### #### Aizu Online Judge

#### 1. \*\*ALDS1\_2\_A: Bubble Sort\*\*

- 难度: 简单
- 链接: [https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\\_2\\_A](https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_2_A)
- 相关算法: 冒泡排序

#### 2. \*\*ALDS1\_2\_B: Selection Sort\*\*

- 难度: 简单
- 链接: [https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\\_2\\_B](https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_2_B)
- 相关算法: 选择排序

3. **\*\*ALDS1\_2\_C: Stable Sort\*\***
  - 难度: 简单
  - 链接: [https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\\_2\\_C](https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_2_C)
  - 相关算法: 稳定排序
4. **\*\*ALDS1\_2\_D: Shell Sort\*\***
  - 难度: 中等
  - 链接: [https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\\_2\\_D](https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_2_D)
  - 相关算法: 希尔排序
5. **\*\*ALDS1\_5\_B: Merge Sort\*\***
  - 难度: 中等
  - 链接: [https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\\_5\\_B](https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_5_B)
  - 相关算法: 归并排序
6. **\*\*ALDS1\_5\_D: The Number of Inversions\*\***
  - 难度: 中等
  - 链接: [https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\\_5\\_D](https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_5_D)
  - 相关算法: 逆序对统计
7. **\*\*ALDS1\_6\_B: Partition\*\***
  - 难度: 中等
  - 链接: [https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\\_6\\_B](https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_6_B)
  - 相关算法: 快速排序分区
8. **\*\*ALDS1\_6\_C: Quick Sort\*\***
  - 难度: 中等
  - 链接: [https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\\_6\\_C](https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_6_C)
  - 相关算法: 快速排序
9. **\*\*ALDS1\_6\_D: Minimum Cost Sort\*\***
  - 难度: 困难
  - 链接: [https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\\_6\\_D](https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_6_D)
  - 相关算法: 最小成本排序
10. **\*\*ALDS1\_9\_A: Complete Binary Tree\*\***
  - 难度: 简单
  - 链接: [https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\\_9\\_A](https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_9_A)
  - 相关算法: 完全二叉树
11. **\*\*ALDS1\_9\_B: Maximum Heap\*\***
  - 难度: 简单

- 链接: [https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\\_9\\_B](https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_9_B)
- 相关算法: 最大堆

## 12. \*\*ALDS1\_9\_C: Priority Queue\*\*

- 难度: 中等
- 链接: [https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\\_9\\_C](https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_9_C)
- 相关算法: 优先队列

### #### Codeforces

#### 1. \*\*Sort the Array\*\*

- 难度: 简单
- 链接: <https://codeforces.com/problemset/problem/451/B>
- 相关算法: 排序、区间反转

#### 2. \*\*Mike and Feet\*\*

- 难度: 中等
- 链接: <https://codeforces.com/problemset/problem/547/B>
- 相关算法: 单调栈、排序

### #### AtCoder

#### 1. \*\*Sorting\*\*

- 难度: 简单
- 链接: [https://atcoder.jp/contests/abc217/tasks/abc217\\_c](https://atcoder.jp/contests/abc217/tasks/abc217_c)
- 相关算法: 基础排序

### #### USACO

#### 1. \*\*Milking Cows\*\*

- 难度: 简单
- 链接: <http://www.usaco.org/index.php?page=viewproblem2&cpid=855>
- 相关算法: 区间合并

## ## 学习资源链接

### #### 可视化工具

#### 1. \*\*VisualGo\*\* - 排序算法可视化

- 链接: <https://visualgo.net/en/sorting>

#### 2. \*\*USFCA Sorting Animations\*\* - 动画演示

- 链接: <https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html>

### #### 在线课程

#### 1. \*\*Khan Academy\*\* - 算法课程

- 链接: <https://www.khanacademy.org/computing/computer-science/algorithms>

2. \*\*Coursera\*\* - 算法专项课程
  - 链接: <https://www.coursera.org/specializations/algorithms>

#### ### 书籍推荐

1. \*\*《算法导论》\*\* - 排序算法理论基础
2. \*\*《编程珠玑》\*\* - 排序算法的实际应用
3. \*\*《算法》\*\* - 各种排序算法的实现和比较

#### ## 🔗 实用工具链接

#### ### 代码测试平台

1. \*\*LeetCode Playground\*\* - 在线代码测试
2. \*\*牛客网 OJ\*\* - 国内算法题库
3. \*\*HackerRank\*\* - 编程挑战平台

#### ### 调试工具

1. \*\*Python Tutor\*\* - 代码执行可视化
2. \*\*JDoodle\*\* - 在线编译器
3. \*\*CodePen\*\* - 前端代码测试

---  
\*最后更新: 2025 年 10 月 17 日\*

=====

文件: README.md

=====

## # 排序算法与数据结构专题 - Class001

#### ## 📁 目录

- [算法概述] (#算法概述)
- [核心排序算法] (#核心排序算法)
- [扩展题目] (#扩展题目)
- [工程化考量] (#工程化考量)
- [复杂度分析] (#复杂度分析)
- [面试技巧] (#面试技巧)
- [实战训练] (#实战训练)

#### ## 🌐 算法概述

本专题深入探讨排序算法及其相关应用，涵盖从基础排序到高级应用的完整知识体系。

## #### 核心算法

- **归并排序** - 分治思想的经典应用
- **快速排序** - 实际应用最广泛的排序算法
- **堆排序** - 原地排序的  $O(n \log n)$  算法
- **其他基础排序** - 插入、冒泡、选择排序

## ## 📈 核心排序算法

### #### 1. 归并排序 (Merge Sort)

**时间复杂度**:  $O(n \log n)$

**空间复杂度**:  $O(n)$

**稳定性**: 稳定

**适用场景**:

- 需要稳定排序的场合
- 链表排序
- 外部排序 (大数据量无法全部加载到内存)

**关键特性**:

- 分治思想的典型应用
- 递归实现简单直观
- 迭代实现节省栈空间

### #### 2. 快速排序 (Quick Sort)

**时间复杂度**:  $O(n \log n)$  平均,  $O(n^2)$  最坏

**空间复杂度**:  $O(\log n)$  平均,  $O(n)$  最坏

**稳定性**: 不稳定

**适用场景**:

- 通用排序 (实际应用最广泛)
- 内存受限环境
- 需要原地排序的场合

**优化策略**:

- 随机化基准选择
- 三路划分处理重复元素
- 小数组使用插入排序

### #### 3. 堆排序 (Heap Sort)

**时间复杂度**:  $O(n \log n)$

**空间复杂度**:  $O(1)$

**稳定性**: 不稳定

## \*\*适用场景\*\*:

- 需要原地排序且要求  $O(n \log n)$  时间复杂度
- 优先级队列实现
- 大数据量的部分排序

## ## 扩展题目

### #### 题目 1: 215. 数组中的第 K 个最大元素

\*\*来源\*\*: LeetCode

\*\*链接\*\*: <https://leetcode.cn/problems/kth-largest-element-in-an-array/>

## \*\*多种解法\*\*:

### 1. \*\*快速选择算法\*\* (最优解)

- 时间复杂度:  $O(n)$  平均
- 空间复杂度:  $O(1)$

### 2. \*\*最小堆实现\*\*

- 时间复杂度:  $O(n \log k)$
- 空间复杂度:  $O(k)$

### 3. \*\*排序后直接取\*\*

- 时间复杂度:  $O(n \log n)$
- 空间复杂度:  $O(1)$

## \*\*相关题目\*\*:

- \*\*ALDS1\_6\_B: Partition\*\* - [https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\\_6\\_B](https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_6_B)

### #### 题目 2: 75. 颜色分类 (荷兰国旗问题)

\*\*来源\*\*: LeetCode

\*\*链接\*\*: <https://leetcode.cn/problems/sort-colors/>

## \*\*解法\*\*:

- \*\*三指针法\*\*:  $O(n)$  时间,  $O(1)$  空间
- \*\*计数排序\*\*:  $O(n)$  时间,  $O(1)$  空间 (只有 3 种颜色)

## \*\*相关题目\*\*:

- \*\*ALDS1\_2\_C: Stable Sort\*\* - [https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\\_2\\_C](https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_2_C)

### #### 题目 3: 56. 合并区间

\*\*来源\*\*: LeetCode

\*\*链接\*\*: <https://leetcode.cn/problems/merge-intervals/>

\*\*解法\*\*: 排序+合并, 时间复杂度  $O(n \log n)$

## \*\*相关题目\*\*:

- **ALDS1\_6\_D: Minimum Cost Sort** - [https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\\_6\\_D](https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_6_D)

### ### 题目 4: 347. 前 K 个高频元素

\*\*来源\*\*: LeetCode

\*\*链接\*\*: <https://leetcode.cn/problems/top-k-frequent-elements/>

## \*\*解法\*\*:

- **最小堆法**:  $O(n \log k)$  时间
- **桶排序**:  $O(n)$  时间

### ### 题目 5: 164. 最大间距

\*\*来源\*\*: LeetCode

\*\*链接\*\*: <https://leetcode.cn/problems/maximum-gap/>

## \*\*解法\*\*:

- **基数排序**:  $O(n)$  时间 (线性时间)
- **普通排序**:  $O(n \log n)$  时间

## \*\*相关题目\*\*:

- **ALDS1\_5\_D: The Number of Inversions** - [https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\\_5\\_D](https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_5_D)

## ## 工程化考量

### ### 1. 异常处理

```
``` python
# 输入验证
if not isinstance(nums, list):
    raise TypeError("输入必须是列表")
if not all(isinstance(x, int) for x in nums):
    raise TypeError("列表必须只包含整数")
```
```

### ### 2. 边界条件处理

- 空数组处理
- 单元素数组
- 已排序/逆序数组
- 大量重复元素

### ### 3. 性能优化

- 小数组优化（使用插入排序）
- 避免不必要的内存分配
- 缓存友好性考虑

#### #### 4. 可测试性

- 单元测试覆盖各种边界情况
- 性能基准测试
- 内存使用监控

#### ## 📊 复杂度分析

##### #### 时间复杂度对比

| 算法   | 最好情况          | 平均情况          | 最坏情况          | 空间复杂度       |
|------|---------------|---------------|---------------|-------------|
| 归并排序 | $O(n \log n)$ | $O(n \log n)$ | $O(n \log n)$ | $O(n)$      |
| 快速排序 | $O(n \log n)$ | $O(n \log n)$ | $O(n^2)$      | $O(\log n)$ |
| 堆排序  | $O(n \log n)$ | $O(n \log n)$ | $O(n \log n)$ | $O(1)$      |
| 插入排序 | $O(n)$        | $O(n^2)$      | $O(n^2)$      | $O(1)$      |

##### #### 空间复杂度分析

- **归并排序**: 需要额外  $O(n)$  空间存储临时数组
- **快速排序**: 递归调用栈空间  $O(\log n)$  平均
- **堆排序**: 原地排序,  $O(1)$  空间

#### ## 💡 面试技巧

##### #### 1. 算法选择依据

- **数据规模**: 小数据用简单排序, 大数据用高效排序
- **内存限制**: 内存紧张时选择原地排序
- **稳定性要求**: 需要稳定排序时选择归并排序
- **数据特性**: 大量重复元素时使用三路快排

##### #### 2. 代码实现要点

- 清晰的变量命名
- 关键步骤注释
- 边界条件处理
- 异常情况考虑

##### #### 3. 性能分析能力

- 能够分析时间/空间复杂度
- 理解常数项对实际性能的影响
- 知道如何优化常数项

## ## 🔎 实战训练

### #### 推荐练习题目

#### ##### LeetCode

1. \*\*88. 合并两个有序数组\*\* - 归并排序应用
2. \*\*148. 排序链表\*\* - 链表上的归并排序
3. \*\*912. 排序数组\*\* - 各种排序算法的实现
4. \*\*973. 最接近原点的 K 个点\*\* - 快速选择应用
5. \*\*1054. 距离相等的条形码\*\* - 堆排序应用

#### ##### 牛客网

1. \*\*NC140 排序\*\* - 基础排序实现
2. \*\*NC119 最小的 K 个数\*\* - 堆的应用
3. \*\*NC88 寻找第 K 大\*\* - 快速选择

#### ##### 剑指 Offer

1. \*\*面试题 40. 最小的 k 个数\*\* - 堆/快速选择
2. \*\*面试题 51. 数组中的逆序对\*\* - 归并排序应用

### ### 进阶题目

1. \*\*外部排序\*\* - 处理超大数据集
2. \*\*并行排序\*\* - 多线程/分布式排序
3. \*\*稳定快速排序\*\* - 保持稳定性的快速排序变种

## ## 🔎 调试技巧

### ### 1. 打印中间过程

```
``` python
def quick_sort_debug(nums, low, high, depth=0):
    indent = " " * depth
    print(f"{indent}quick_sort({low}, {high}): {nums[low:high+1]}")
    # ... 排序逻辑
```

```

### ### 2. 断言验证

```
``` python
def test_sort_algorithm():
    test_cases = [
        ([], []),
        ([1], [1]),
        ([3, 1, 2], [1, 2, 3])
    ]

```

```
for input_nums, expected in test_cases:  
    result = sort_algorithm(input_nums)  
    assert result == expected, f"Failed for {input_nums}"  
...  
  
### 3. 性能分析  
```python  
import time  
import random  
  
def benchmark_sort(algorithm, size=10000):  
    test_data = [random.randint(0, size) for _ in range(size)]  
  
    start_time = time.time()  
    result = algorithm(test_data)  
    end_time = time.time()  
  
    return end_time - start_time  
...  
...
```

## ## 📚 学习资源

### ### 推荐书籍

1. 《算法导论》 - 排序算法理论基础
2. 《编程珠玑》 - 排序算法的实际应用
3. 《算法》 - 各种排序算法的实现和比较

### ### 在线资源

1. \*\*VisualGo\*\* - 排序算法可视化
2. \*\*USFCA Sorting Animations\*\* - 动画演示
3. \*\*Khan Academy\*\* - 算法课程

### ### 实践平台

1. \*\*LeetCode\*\* - 算法题目练习
2. \*\*HackerRank\*\* - 编程挑战
3. \*\*牛客网\*\* - 国内算法题库

---

\*\*持续更新中... 更多题目和解析将陆续添加\*\*

\*最后更新: 2025年10月17日\*

=====

文件: README\_FINAL.md

=====

## # 排序算法专题 - 完整版

### ## 📈 项目概述

本项目是排序算法的完整学习资源，包含基础排序算法实现、扩展题目训练、模式识别技巧、工程化实践等内容。涵盖了 Java、C++、Python 三种语言的完整实现。

相关题目链接:

- 912. 排序数组: <https://leetcode.cn/problems/sort-an-array/>
- 215. 数组中的第 K 个最大元素: <https://leetcode.cn/problems/kth-largest-element-in-an-array/>
- 75. 颜色分类: <https://leetcode.cn/problems/sort-colors/>
- 56. 合并区间: <https://leetcode.cn/problems/merge-intervals/>
- 148. 排序链表: <https://leetcode.cn/problems/sort-list/>
- 剑指 Offer 51. 数组中的逆序对: <https://leetcode.cn/problems/shu-zu-zhong-de-ni-xu-dui-lcof/>
- 88. 合并两个有序数组: <https://leetcode.cn/problems/merge-sorted-array/>
- 973. 最接近原点的 K 个点: <https://leetcode.cn/problems/k-closest-points-to-origin/>
- 1054. 距离相等的条形码: <https://leetcode.cn/problems/distant-barcodes/>
- 324. 摆动排序 II: <https://leetcode.cn/problems/wiggle-sort-ii/>
- 280. 摆动排序: <https://leetcode.cn/problems/wiggle-sort/>
- 493. 翻转对: <https://leetcode.cn/problems/reverse-pairs/>
- 347. 前 K 个高频元素: <https://leetcode.cn/problems/top-k-frequent-elements/>
- 164. 最大间距: <https://leetcode.cn/problems/maximum-gap/>
- NC140 排序: <https://www.nowcoder.com/practice/2baf799ea0594abd974d37139de27896>
- NC119 最小的 K 个数: <https://www.nowcoder.com/practice/6a296eb82cf844ca8539b57c23e6e9bf>
- NC88 寻找第 K 大: <https://www.nowcoder.com/practice/e016ad9b7f0b45048c58a9f27ba618bf>
- 面试题 40. 最小的 k 个数: 剑指 Offer 第二版第 40 题
- 面试题 45. 把数组排成最小的数: 剑指 Offer 第二版第 45 题
- HackerRank - Counting Inversions: <https://www.hackerrank.com/challenges/ctci-merge-sort>
- HackerRank - Fraudulent Activity Notifications:  
<https://www.hackerrank.com/challenges/fraudulent-activity-notifications>

### ## ⚡ 核心内容

#### #### 1. 基础排序算法

- \*\*归并排序\*\*: 稳定排序，时间复杂度  $O(n \log n)$
- \*\*快速排序\*\*: 平均性能最优，时间复杂度  $O(n \log n)$  平均
- \*\*堆排序\*\*: 原地排序，时间复杂度  $O(n \log n)$
- \*\*特殊排序\*\*: 计数排序、基数排序、桶排序

## ### 2. 扩展题目训练

- \*\*10+个扩展题目\*\*: 来自 LeetCode、牛客网、剑指 Offer 等平台
- \*\*多语言实现\*\*: 每个题目都有 Java、C++、Python 三种实现
- \*\*详细注释\*\*: 包含时间/空间复杂度分析
- \*\*测试用例\*\*: 完整的单元测试和边界测试

## ### 3. 模式识别与技巧

- \*\*5 种常见模式\*\*: Top K、区间合并、颜色分类、逆序对统计、自定义排序
- \*\*优化策略\*\*: 小数组优化、随机化、三路快排、自适应算法
- \*\*调试技巧\*\*: 中间结果打印、断言验证、性能分析

## ## 🚀 快速开始

### ### 运行 Java 代码

```
```bash
cd class001
javac *.java
java SortAlgorithms
java ExtendedSortProblems
```
```

### ### 运行 C++ 代码

```
```bash
cd class001
g++ -o test_sort SortAlgorithms.cpp
g++ -o extended_test ExtendedSortProblems.cpp
./test_sort
./extended_test
```
```

### ### 运行 Python 代码

```
```bash
cd class001
python SortAlgorithms.py
python ExtendedSortProblems.py
```
```

## ## 📁 文件结构

```
```
class001/
    └── 基础算法实现/
```

```
|   ├── SortAlgorithms.java      # Java 基础排序算法
|   ├── SortAlgorithms.cpp      # C++基础排序算法
|   └── SortAlgorithms.py       # Python 基础排序算法
|
|   └── 扩展题目实现/
|       ├── ExtendedSortProblems.java    # Java 扩展题目
|       ├── ExtendedSortProblems.cpp      # C++扩展题目
|       └── ExtendedSortProblems.py       # Python 扩展题目
|
|   └── 文档与总结/
|       ├── README.md                  # 项目说明
|       ├── AdditionalProblems_Extended.md  # 扩展题目详解
|       ├── SortingPatternsAndTechniques_part1.md  # 模式技巧(上)
|       └── SortingPatternsAndTechniques_part2.md  # 模式技巧(下)
|
|   └── 测试文件/
|       ├── *.class                   # Java 编译文件
|       ├── test_sort                 # C++可执行文件
|       └── extended_test             # C++扩展测试文件
...
```

```

## ## 🚀 学习路径

### ### 初学者路径

1. 先学习基础排序算法原理
2. 运行基础算法代码，理解实现
3. 阅读模式识别文档，掌握解题思路
4. 尝试解决扩展题目

### ### 进阶者路径

1. 深入理解算法复杂度推导
2. 学习优化策略和工程化实践
3. 掌握多语言实现的差异
4. 进行性能分析和调优

### ### 面试准备路径

1. 熟练掌握常见题型模式
2. 练习白板编码和复杂度分析
3. 学习面试技巧和问题回答
4. 进行模拟面试练习

## ## 💡 核心特色

### ### 1. 多语言完整实现

- 每个算法都有 Java、C++、Python 三种实现
- 展示不同语言的特性差异
- 便于对比学习和实际应用

### ### 2. 工程化实践

- 完整的异常处理机制
- 性能监控和内存分析
- 单元测试和边界测试
- 代码规范和最佳实践

### ### 3. 深度内容覆盖

- 算法原理和复杂度推导
- 优化策略和技巧
- 调试方法和问题定位
- 面试准备和实战技巧

## ## 🔧 技术栈

### ### 编程语言

- **Java**: 面向对象，企业级应用
- **C++**: 高性能，系统级编程
- **Python**: 快速开发，数据科学

### ### 算法领域

- 排序算法理论与实践
- 复杂度分析与优化
- 数据结构应用
- 算法设计模式

### ### 工程实践

- 代码测试与验证
- 性能分析与调优
- 多语言编程
- 软件工程最佳实践

## ## 📈 性能基准

### ### 时间复杂度对比

| 算法   | 最好情况          | 平均情况          | 最坏情况          | 稳定性 |
|------|---------------|---------------|---------------|-----|
| 归并排序 | $O(n \log n)$ | $O(n \log n)$ | $O(n \log n)$ | 稳定  |
| 快速排序 | $O(n \log n)$ | $O(n \log n)$ | $O(n^2)$      | 不稳定 |

|                                                           |
|-----------------------------------------------------------|
| 堆排序   $O(n \log n)$   $O(n \log n)$   $O(n \log n)$   不稳定 |
| 插入排序   $O(n)$   $O(n^2)$   $O(n^2)$   稳定                  |

#### #### 空间复杂度对比

|                        |
|------------------------|
| 算法   空间复杂度   是否原地      |
| 归并排序   $O(n)$   否      |
| 快速排序   $O(\log n)$   是 |
| 堆排序   $O(1)$   是       |
| 插入排序   $O(1)$   是      |

## ## 🎓 学习目标

### #### 知识目标

- 掌握各种排序算法的原理和实现
- 理解时间/空间复杂度的计算方法
- 学会根据问题特征选择合适的算法
- 掌握算法优化和调试技巧

### #### 技能目标

- 能够用多种语言实现排序算法
- 能够分析和解决复杂的排序问题
- 能够进行算法性能分析和优化
- 能够应对技术面试中的算法问题

### #### 能力目标

- 算法思维和问题解决能力
- 代码实现和调试能力
- 性能分析和优化能力
- 多语言编程能力

## ## 🔍 扩展学习

### #### 相关算法

- 查找算法：二分查找、哈希查找
- 图算法：最短路径、最小生成树
- 动态规划：背包问题、最长子序列
- 贪心算法：区间调度、霍夫曼编码

### #### 进阶主题

- 分布式排序算法
- 外部排序技术
- 并行算法设计

## - 算法工程化实践

### ## 🤝 贡献指南

欢迎贡献代码和文档改进！请遵循以下步骤：

1. Fork 本项目
2. 创建特性分支
3. 提交更改
4. 推送到分支
5. 创建 Pull Request

### ## 📄 许可证

本项目采用 MIT 许可证，详见 LICENSE 文件。

### ## 🎉 致谢

感谢以下资源提供的灵感和支持：

- LeetCode 平台提供的算法题目
- 牛客网的技术社区
- 剑指 Offer 的经典题目
- 各大技术博客和教程

---

\*\*Happy Coding! 🎉\*\*

\*最后更新：2025 年 10 月 17 日\*

=====

文件: SortAlgorithmsExtension.md

=====

## # 排序算法扩展题目与解法

本文档补充了排序算法库中的更多相关题目、解法和优化技巧，帮助深入理解排序算法的应用场景和实现细节。

### ## 目录

1. [高级排序算法应用] (#1-高级排序算法应用)
2. [特殊排序算法] (#2-特殊排序算法)

3. [排序算法优化技巧] (#3-排序算法优化技巧)
4. [排序算法在实际场景中的应用] (#4-排序算法在实际场景中的应用)
5. [经典排序题目详解] (#5-经典排序题目详解)

## ## 1. 高级排序算法应用

### ### 1.1 外部排序

\*\*题目\*\*: 如何对一个 10GB 的文件进行排序，但内存只有 2GB?

\*\*解法\*\*:

1. \*\*分割阶段\*\*: 将大文件分成多个小块，每个小块大小为内存可容纳的最大值（如 1.5GB）
2. \*\*排序阶段\*\*: 对每个小块在内存中进行排序（如使用快速排序）
3. \*\*合并阶段\*\*: 使用多路归并算法合并所有排序好的小块

\*\*代码框架\*\*:

```
```java
// 分割大文件为小块
public static void splitFile(String inputFile, int chunkSize) {
    try (BufferedReader reader = new BufferedReader(new FileReader(inputFile))) {
        String line;
        int fileCount = 0;
        List<String> chunk = new ArrayList<>();

        while ((line = reader.readLine()) != null) {
            chunk.add(line);

            if (chunk.size() >= chunkSize) {
                sortAndSaveChunk(chunk, "chunk_" + fileCount + ".txt");
                chunk.clear();
                fileCount++;
            }
        }

        if (!chunk.isEmpty()) {
            sortAndSaveChunk(chunk, "chunk_" + fileCount + ".txt");
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}

// 排序并保存小块
```

```
private static void sortAndSaveChunk(List<String> chunk, String outputFile) {
    // 对小块进行排序
    Collections.sort(chunk);

    try (BufferedWriter writer = new BufferedWriter(new FileWriter(outputFile))) {
        for (String line : chunk) {
            writer.write(line);
            writer.newLine();
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}

// 多路归并
public static void mergeChunks(String[] chunkFiles, String outputFile) {
    // 使用优先队列进行多路归并
    PriorityQueue<ChunkReader> pq = new PriorityQueue<>();

    try (BufferedWriter writer = new BufferedWriter(new FileWriter(outputFile))) {
        // 初始化每个块的读取器
        for (String file : chunkFiles) {
            ChunkReader reader = new ChunkReader(file);
            if (reader.hasNext()) {
                pq.add(reader);
            }
        }
    }

    // 多路归并
    while (!pq.isEmpty()) {
        ChunkReader reader = pq.poll();
        writer.write(reader.current());
        writer.newLine();

        if (reader.hasNext()) {
            pq.add(reader);
        } else {
            reader.close();
        }
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

}

=====

文件: SortingPatternsAndTechniques\_part1.md

=====

## # 排序算法模式与技巧总结

### ## 🎯 排序算法选择指南

#### ### 根据数据特征选择算法

数据特征	推荐算法	时间复杂度	空间复杂度	稳定性
小规模数据 ( $n < 50$ )	插入排序	$O(n^2)$	$O(1)$	稳定
中等规模数据	快速排序	$O(n \log n)$ 平均	$O(\log n)$	不稳定
大规模数据	归并排序	$O(n \log n)$	$O(n)$	稳定
需要原地排序	堆排序	$O(n \log n)$	$O(1)$	不稳定
数据范围小	计数排序	$O(n+k)$	$O(k)$	稳定
数据均匀分布	桶排序	$O(n)$	$O(n)$	稳定
整数排序	基数排序	$O(d(n+k))$	$O(n+k)$	稳定

#### ### 根据需求选择算法

需求	推荐算法	理由
需要稳定排序	归并排序	保证相等元素的相对顺序
内存有限	堆排序	原地排序, 空间复杂度 $O(1)$
平均性能最好	快速排序	实际应用中常数因子小
最坏情况保证	归并排序	最坏情况也是 $O(n \log n)$
处理链表	归并排序	适合链表结构
外部排序	多路归并	处理大数据集

### ## 💡 常见问题模式识别

#### ### 模式 1: Top K 问题

\*\*识别特征\*\*: 寻找前 K 大/小元素

\*\*最优解法\*\*:

- 快速选择算法:  $O(n)$  平均时间复杂度
- 堆排序:  $O(n \log k)$  时间复杂度

\*\*相关题目\*\*:

- LeetCode 215: 数组中的第 K 个最大元素

- LeetCode 973: 最接近原点的 K 个点
- 牛客网 NC88: 寻找第 K 大

**\*\*解题模板\*\*:**

```
``` python
def find_kth_largest(nums, k):
    # 快速选择实现
    def quick_select(left, right, k_smallest):
        # 分区逻辑
        pivot_index = partition(left, right)
        if pivot_index == k_smallest:
            return nums[pivot_index]
        elif pivot_index < k_smallest:
            return quick_select(pivot_index + 1, right, k_smallest)
        else:
            return quick_select(left, pivot_index - 1, k_smallest)

    return quick_select(0, len(nums)-1, len(nums)-k)
```

```

**### 模式 2: 区间合并问题**

- \*识别特征\***: 重叠区间需要合并  
**\*最优解法\***: 按起点排序后合并

**\*\*相关题目\*\*:**

- LeetCode 56: 合并区间
- LeetCode 57: 插入区间
- LeetCode 252: 会议室

**\*\*解题模板\*\*:**

```
``` python
def merge_intervals(intervals):
    if not intervals:
        return []

    # 按起点排序
    intervals.sort(key=lambda x: x[0])

    merged = []
    for interval in intervals:
        # 如果结果为空或当前区间不重叠
        if not merged or merged[-1][1] < interval[0]:
            merged.append(interval)
        else:
            merged[-1][1] = max(merged[-1][1], interval[1])
```

```

```

else:
    # 合并区间
    merged[-1][1] = max(merged[-1][1], interval[1])

return merged
```

```

### ### 模式 3: 颜色分类/荷兰国旗问题

**\*\*识别特征\*\*:** 有限种类的元素需要分类

**\*\*最优解法\*\*:** 三指针/三路快排

**\*\*相关题目\*\*:**

- LeetCode 75: 颜色分类
- LeetCode 280: 摆动排序
- 剑指 Offer 21: 调整数组顺序

**\*\*解题模板\*\*:**

```

``` python
def sort_colors(nums):
    # 三指针: left, right, current
    left, current, right = 0, 0, len(nums) - 1

    while current <= right:
        if nums[current] == 0:
            nums[left], nums[current] = nums[current], nums[left]
            left += 1
            current += 1
        elif nums[current] == 2:
            nums[current], nums[right] = nums[right], nums[current]
            right -= 1
        else:
            current += 1
```

```

### ### 模式 4: 逆序对统计

**\*\*识别特征\*\*:** 统计满足某种条件的逆序对数量

**\*\*最优解法\*\*:** 归并排序过程中统计

**\*\*相关题目\*\*:**

- LeetCode 493: 翻转对
- 剑指 Offer 51: 数组中的逆序对
- HackerRank: Counting Inversions

\*\*解题模板\*\*:

```
``` python
def reverse_pairs(nums):
    def merge_sort_count(left, right):
        if left >= right:
            return 0

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

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

        # 合并
        merge(left, mid, right)
        return count

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

```

#### 模式 5: 自定义排序规则

\*\*识别特征\*\*: 需要特殊的比较规则

\*\*最优解法\*\*: 实现自定义比较器

\*\*相关题目\*\*:

- LeetCode 179: 最大数
- 剑指 Offer 45: 把数组排成最小的数
- LeetCode 524: 通过删除字母匹配到字典里最长单词

\*\*解题模板\*\*:

```
``` python
def largest_number(nums):
    # 将数字转换为字符串
    str_nums = [str(num) for num in nums]

    # 自定义排序: 比较 s1+s2 和 s2+s1
    str_nums.sort(key=lambda x: x*10, reverse=True)

    # 处理前导零

```

```
result = ''.join(str_nums)
return '0' if result[0] == '0' else result
```

```

## ## 🔧 优化技巧与策略

### ### 1. 小数组优化

\*\*技巧\*\*: 对于小数组 ( $n < 50$ )，使用简单排序算法

\*\*理由\*\*: 简单算法常数因子小，实际运行更快

```
``` python
def optimized_sort(nums):
    if len(nums) <= 10:
        return insertion_sort(nums) # 小数组使用插入排序
    else:
        return quick_sort(nums) # 大数组使用快速排序
```

```

### ### 2. 随机化避免最坏情况

\*\*技巧\*\*: 随机选择 pivot 元素

\*\*理由\*\*: 避免快速排序的最坏情况

```
``` python
import random

def randomized_quick_sort(nums):
    if len(nums) <= 1:
        return nums

    # 随机选择 pivot
    pivot_index = random.randint(0, len(nums)-1)
    pivot = nums[pivot_index]

    # 分区逻辑
    left = [x for x in nums if x < pivot]
    middle = [x for x in nums if x == pivot]
    right = [x for x in nums if x > pivot]

    return randomized_quick_sort(left) + middle + randomized_quick_sort(right)
```

```

### ### 3. 处理重复元素优化

\*\*技巧\*\*: 使用三路快排

\*\*理由\*\*: 高效处理大量重复元素

```
``` python
def three_way_quick_sort(nums):
    if len(nums) <= 1:
        return nums

    # 选择 pivot
    pivot = nums[len(nums)//2]

    # 三路分区
    left = [x for x in nums if x < pivot]
    middle = [x for x in nums if x == pivot]
    right = [x for x in nums if x > pivot]

    return three_way_quick_sort(left) + middle + three_way_quick_sort(right)
```

```

#### ### 4. 利用数据特性

\*\*技巧\*\*: 根据数据分布选择特殊排序算法

\*\*理由\*\*: 特定算法在特定数据分布下更高效

```
``` python
def adaptive_sort(nums):
    if not nums:
        return []

    # 如果数据范围小，使用计数排序
    min_val, max_val = min(nums), max(nums)
    if max_val - min_val < 1000:
        return counting_sort(nums, min_val, max_val)

    # 如果数据基本有序，使用插入排序
    if is_almost_sorted(nums):
        return insertion_sort(nums)

    # 默认使用快速排序
    return quick_sort(nums)
```

```

#### ## 🎯 面试实战技巧

##### ### 1. 问题分析步骤

## \*\*第一步\*\*: 理解题目要求

- 明确输入输出格式
- 理解排序规则
- 确定时间/空间复杂度要求

## \*\*第二步\*\*: 分析数据特征

- 数据规模大小
- 数据分布情况
- 是否需要稳定排序

## \*\*第三步\*\*: 选择合适算法

- 根据特征选择基础算法
- 考虑优化策略
- 准备备选方案

## #### 2. 代码实现要点

### \*\*清晰的变量命名\*\*:

```
``` python
# 好的命名
def merge_sorted_arrays(nums1, m, nums2, n):
    pointer1 = m - 1 # nums1 有效部分末尾
    pointer2 = n - 1 # nums2 末尾
    merge_pointer = m + n - 1 # 合并位置
```

### # 差的命名

```
def merge(a, x, b, y):
    i = x - 1
    j = y - 1
    k = x + y - 1
```
```

### \*\*适当的注释\*\*:

```
``` python
def quick_select(nums, k):
    """
    快速选择算法寻找第 k 小元素
```
```

Args:

  nums: 输入数组  
  k: 要寻找的第 k 小元素索引 (0-based)

Returns:

  第 k 小的元素值

```

"""
# 随机选择 pivot 避免最坏情况
pivot_index = random.randint(0, len(nums)-1)
pivot = nums[pivot_index]

# 分区操作
left = [x for x in nums if x < pivot]
middle = [x for x in nums if x == pivot]
right = [x for x in nums if x > pivot]

# 根据分区结果递归选择
if k < len(left):
    return quick_select(left, k)
elif k < len(left) + len(middle):
    return pivot
else:
    return quick_select(right, k - len(left) - len(middle))
```

```

### ### 3. 测试用例设计

**\*\*全面覆盖各种情况\*\*:**

```

```

python
def test_sort_algorithm():
    # 空数组
    assert sort([]) == []

    # 单元素数组
    assert sort([1]) == [1]

    # 已排序数组
    assert sort([1, 2, 3]) == [1, 2, 3]

    # 逆序数组
    assert sort([3, 2, 1]) == [1, 2, 3]

    # 重复元素数组
    assert sort([2, 2, 1, 1]) == [1, 1, 2, 2]

    # 包含负数的数组
    assert sort([-3, -1, -2]) == [-3, -2, -1]

    # 混合正负数数组
    assert sort([3, -1, 0, -2, 1]) == [-2, -1, 0, 1, 3]

```

```
# 大规模随机数组
import random
large_array = [random.randint(1, 10000) for _ in range(1000)]
assert sort(large_array.copy()) == sorted(large_array)
```

```

#### ### 4. 性能分析能力

\*\*时间复杂度分析\*\*:

```
``` python
```

```
def analyze_time_complexity(algorithm, data_sizes):
    """

```

分析算法的时间复杂度

Args:

algorithm: 排序算法函数

data\_sizes: 不同数据规模列表

Returns:

时间复杂度趋势分析

```
    """

```

```
results = []
for size in data_sizes:

```

```
    test_data = generate_test_data(size)

```

```
    start_time = time.time()

```

```
    algorithm(test_data)

```

```
    end_time = time.time()

```

```
    results.append((size, end_time - start_time))

```

```
return analyze_trend(results)
```

```

\*\*空间复杂度分析\*\*:

```
``` python
```

```
def analyze_space_complexity(algorithm, data):
    """

```

分析算法的空间复杂度

Args:

algorithm: 排序算法函数

data: 测试数据

Returns:

## 空间使用情况分析

```
"""
import tracemalloc

tracemalloc.start()
result = algorithm(data)
current, peak = tracemalloc.get_traced_memory()
tracemalloc.stop()

return {
    'peak_memory': peak,
    'current_memory': current,
    'data_size': len(data)
}
```

```

## ## 🔎 调试与问题定位

### ### 1. 调试技巧

\*\*打印中间结果\*\*:

```
``` python
def debug_merge_sort(nums, depth=0):
    indent = " " * depth
    print(f"{indent}排序数组: {nums}")

    if len(nums) <= 1:
        return nums

    mid = len(nums) // 2
    left = debug_merge_sort(nums[:mid], depth+1)
    right = debug_merge_sort(nums[mid:], depth+1)

    result = merge(left, right)
    print(f"{indent}合并结果: {result}")
    return result
```

```

\*\*断言验证\*\*:

```
``` python
def partition(nums, low, high):
    pivot = nums[high]
    i = low - 1

```

```

for j in range(low, high):
    # 断言验证分区正确性
    assert all(nums[k] <= pivot for k in range(low, j+1) if nums[k] <= pivot)

    if nums[j] <= pivot:
        i += 1
        nums[i], nums[j] = nums[j], nums[i]

    nums[i+1], nums[high] = nums[high], nums[i+1]
return i + 1
```

```

### ### 2. 性能问题定位

**\*\*识别性能瓶颈\*\*:**

``` python

```

def profile_sort_algorithm():
    import cProfile
    import pstats

    # 生成测试数据
    test_data = generate_large_test_data()

    # 性能分析
    profiler = cProfile.Profile()
    profiler.enable()

    # 执行排序算法
    sort_algorithm(test_data)

```

```

profiler.disable()
stats = pstats.Stats(profiler)
stats.sort_stats('cumulative')
stats.print_stats(10) # 显示前 10 个最耗时的函数
```

```

## ## 📈 进阶学习方向

### ### 1. 分布式排序算法

- MapReduce 排序模式
- 外部排序算法
- 并行排序算法

### ### 2. 特殊数据结构排序

## - 链表排序

---

文件: SortingPatternsAndTechniques\_part2.md

---

# 排序算法模式与技巧总结 (第二部分)

## 5 递归与非递归实现对比

#### 归并排序: 递归 vs 迭代

\*\*递归实现\*\*:

```
``` python
def merge_sort_recursive(nums):
    """递归归并排序"""
    if len(nums) <= 1:
        return nums

    mid = len(nums) // 2
    left = merge_sort_recursive(nums[:mid])
    right = merge_sort_recursive(nums[mid:])

    return merge(left, right)
```

# 时间复杂度:  $O(n \log n)$

# 空间复杂度:  $O(n) + O(\log n)$  递归栈

```

\*\*迭代实现\*\*:

```
``` python
def merge_sort_iterative(nums):
    """迭代归并排序"""
    if not nums:
        return []

    # 将每个元素视为已排序的单个元素列表
    queue = [[num] for num in nums]

    while len(queue) > 1:
        # 每次合并前两个列表
        left = queue.pop(0)
        right = queue.pop(0)
```

```
merged = merge(left, right)
queue.append(merged)

return queue[0] if queue else []
```

# 时间复杂度: O(n log n)

# 空间复杂度: O(n)

```

### \*\*对比分析\*\*:

- 递归: 代码简洁, 但递归深度可能造成栈溢出
- 迭代: 避免栈溢出, 但代码相对复杂
- 实际选择: 小数据用递归, 大数据用迭代

### ### 快速排序: 递归 vs 迭代

#### \*\*递归实现\*\*:

```
``` python
def quick_sort_recursive(nums):
    """递归快速排序"""
    if len(nums) <= 1:
        return nums

    pivot = nums[len(nums)//2]
    left = [x for x in nums if x < pivot]
    middle = [x for x in nums if x == pivot]
    right = [x for x in nums if x > pivot]

    return quick_sort_recursive(left) + middle + quick_sort_recursive(right)
````
```

#### \*\*迭代实现 (使用栈) \*\*:

```
``` python
def quick_sort_iterative(nums):
    """迭代快速排序"""
    if len(nums) <= 1:
        return nums

    stack = [(0, len(nums)-1)]
    nums = nums.copy()

    while stack:
        low, high = stack.pop()
```

```

        if low >= high:
            continue

        pivot_index = partition(nums, low, high)

        # 先处理较小的子数组，避免栈深度过大
        if pivot_index - low < high - pivot_index:
            stack.append((pivot_index + 1, high))
            stack.append((low, pivot_index - 1))
        else:
            stack.append((low, pivot_index - 1))
            stack.append((pivot_index + 1, high))

    return nums

def partition(nums, low, high):
    """分区函数"""
    pivot = nums[high]
    i = low - 1

    for j in range(low, high):
        if nums[j] <= pivot:
            i += 1
            nums[i], nums[j] = nums[j], nums[i]

    nums[i+1], nums[high] = nums[high], nums[i+1]
    return i + 1
```

```

## ## 🌐 异常场景与边界处理

```

#### 1. 输入验证
```python
def robust_sort(nums):
    """健壮的排序函数"""
    # 输入类型检查
    if not isinstance(nums, (list, tuple)):
        raise TypeError("输入必须是列表或元组")

    # 空数组处理
    if len(nums) == 0:
        return []

```

```
# 单元素数组
if len(nums) == 1:
    return nums.copy()

# 检查元素类型一致性
if not all(isinstance(x, (int, float)) for x in nums):
    raise TypeError("数组元素必须是数字类型")

# 检查特殊值 (NaN, Infinity)
if any(math.isnan(x) or math.isinf(x) for x in nums):
    raise ValueError("数组包含非法数值 (NaN 或 Infinity)")

# 执行排序
return quick_sort(nums)

def quick_sort(nums):
    """带边界检查的快速排序"""
    if len(nums) <= 10: # 小数组优化
        return insertion_sort(nums)

    # 避免最坏情况: 检查是否已排序
    if is_sorted(nums):
        return nums.copy()

    if is_reverse_sorted(nums):
        return list(reversed(nums))

    # 随机选择 pivot
    pivot_index = random.randint(0, len(nums)-1)
    pivot = nums[pivot_index]

    left = [x for x in nums if x < pivot]
    middle = [x for x in nums if x == pivot]
    right = [x for x in nums if x > pivot]

    return quick_sort(left) + middle + quick_sort(right)
```

```

### ### 2. 极端输入处理

```
```python
def handle_extreme_cases(nums):
    """处理极端输入情况"""

```

```

# 超大规模数据
if len(nums) > 10**6:
    return external_sort(nums) # 外部排序

# 大量重复元素
unique_count = len(set(nums))
if unique_count / len(nums) < 0.1: # 重复率超过 90%
    return counting_sort(nums) # 计数排序

# 数据范围很小
min_val, max_val = min(nums), max(nums)
if max_val - min_val < 1000:
    return counting_sort(nums, min_val, max_val)

# 数据基本有序
if is_almost_sorted(nums, threshold=0.1):
    return insertion_sort(nums) # 插入排序优化

# 默认使用快速排序
return quick_sort(nums)
```

```

## ## 5 语言特性差异分析

### #### Java vs C++ vs Python 排序实现差异

**\*\*Java 特性\*\*:**

```

```java
// Java 使用 Comparable 接口和 Comparator
public class SortUtils {
    // 泛型支持
    public static <T extends Comparable<T>> void sort(T[] array) {
        Arrays.sort(array); // 使用 TimSort (归并+插入)
    }

    // 自定义比较器
    public static void sortByCustomRule(String[] array) {
        Arrays.sort(array, (a, b) -> {
            return (a + b).compareTo(b + a); // 字符串拼接比较
        });
    }
}
```

```

**\*\*C++特性\*\*:**

```
```cpp
// C++使用模板和迭代器
template<typename T>
void sort(std::vector<T>& nums) {
    std::sort(nums.begin(), nums.end()); // 使用内省排序（快排+堆排）
}

// 自定义比较函数
void sortByCustomRule(std::vector<int>& nums) {
    std::sort(nums.begin(), nums.end(), [](int a, int b) {
        std::string sa = std::to_string(a);
        std::string sb = std::to_string(b);
        return sa + sb < sb + sa;
    });
}
```

```

**\*\*Python 特性\*\*:**

```
```python
# Python 使用 key 参数和 lambda 表达式
def sort_with_key(nums):
    return sorted(nums, key=lambda x: (x % 10, x)) # 按个位数排序

# 自定义比较函数 (Python3 需要 functools.cmp_to_key)
import functools

def custom_compare(a, b):
    return (a + b) > (b + a) # 字符串拼接比较

def sort_by_custom_rule(nums):
    return sorted(nums, key=functools.cmp_to_key(custom_compare))
```

```

### 性能差异分析

**\*\*时间性能对比\*\*:**

- C++: 编译优化，运行最快
- Java: JIT 编译，性能接近 C++
- Python: 解释执行，相对较慢

**\*\*内存使用对比\*\*:**

- C++: 手动内存管理，最节省内存
- Java: 自动垃圾回收，内存使用适中
- Python: 动态类型，内存使用较多

## ## 🚀 性能优化策略

### ### 1. 算法层面优化

```
```python
def optimized_quick_sort(nums):
    """优化版快速排序"""
    if len(nums) <= 1:
        return nums

    # 三数取中法选择 pivot
    first, middle, last = nums[0], nums[len(nums)//2], nums[-1]
    pivot = sorted([first, middle, last])[1]

    # 三路分区处理重复元素
    left = [x for x in nums if x < pivot]
    middle = [x for x in nums if x == pivot]
    right = [x for x in nums if x > pivot]

    # 尾递归优化
    return optimized_quick_sort(left) + middle + optimized_quick_sort(right)
```

```

### ### 2. 代码层面优化

```
```python
def cache_optimized_sort(nums):
    """缓存优化排序"""
    # 预计算常用值
    n = len(nums)
    if n <= 1:
        return nums.copy()

    # 使用局部变量加速访问
    result = nums.copy()
    _min = min(result)
    _max = max(result)

    # 根据数据特征选择算法
    if _max - _min < 1000 and n > 1000:
        return counting_sort(result, _min, _max)
```

```

```

else:
    return quick_sort(result)
```
#### 3. 系统层面优化
``` python
def parallel_sort(nums):
    """并行排序（多线程）"""
    import concurrent.futures
    import math

    if len(nums) <= 1000:
        return sorted(nums)

    # 计算线程数（不超过 CPU 核心数）
    num_threads = min(4, len(nums) // 1000)  # 每个线程处理至少 1000 个元素
    chunk_size = math.ceil(len(nums) / num_threads)

    # 分割数据
    chunks = [nums[i:i+chunk_size] for i in range(0, len(nums), chunk_size)]

    # 并行排序
    with concurrent.futures.ThreadPoolExecutor(max_workers=num_threads) as executor:
        sorted_chunks = list(executor.map(sorted, chunks))

    # 合并结果
    return merge_sorted_arrays(sorted_chunks)
```

```

## ## 📊 复杂度计算详细示例

### #### 归并排序复杂度推导

递归关系:  $T(n) = 2T(n/2) + O(n)$

展开过程:

$$\begin{aligned}
 T(n) &= 2T(n/2) + cn \\
 &= 2[2T(n/4) + c(n/2)] + cn = 4T(n/4) + 2cn \\
 &= 4[2T(n/8) + c(n/4)] + 2cn = 8T(n/8) + 3cn \\
 &= \dots \\
 &= 2^k T(n/2^k) + kc n
 \end{aligned}$$

当  $n/2^k = 1 \Rightarrow k = \log_2 n$

$T(n) = nT(1) + cn \log_2 n = O(n \log n)$

...

#### 快速排序复杂度分析

\*\*最好情况\*\* (每次均衡划分):

...

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

...

\*\*最坏情况\*\* (每次极端划分):

...

$T(n) = T(n-1) + O(n) = O(n^2)$

...

\*\*平均情况\*\* (随机化):

...

$E[T(n)] = O(n \log n)$

...

#### 堆排序复杂度分析

...

建堆:  $O(n)$

每次调整:  $O(\log n)$

总操作:  $n$  次调整

总复杂度:  $O(n \log n)$

...

## 🌈 单元测试完整示例

#### 测试框架设计

``` python

```
import unittest
import random
import time
```

```
class TestSortAlgorithms(unittest.TestCase):
```

```
    def setUp(self):
```

```
        """测试前准备"""

```

```
        self.test_cases = {
            'empty': [],
            'single': [1],
            'sorted': [1, 2, 3, 4, 5],
```

```
'reverse': [5, 4, 3, 2, 1],
'duplicates': [2, 2, 1, 1, 3, 3],
'negative': [-3, -1, -2, 0, 1],
'large_random': [random.randint(1, 10000) for _ in range(1000)]
}

def test_merge_sort(self):
    """测试归并排序"""
    for name, nums in self.test_cases.items():
        with self.subTest(case=name):
            result = merge_sort(nums.copy())
            self.assertEqual(result, sorted(nums))

def test_quick_sort(self):
    """测试快速排序"""
    for name, nums in self.test_cases.items():
        with self.subTest(case=name):
            result = quick_sort(nums.copy())
            self.assertEqual(result, sorted(nums))

def test_performance(self):
    """性能测试"""
    large_data = [random.randint(1, 100000) for _ in range(10000)]

    # 测试归并排序性能
    start = time.time()
    merge_sort(large_data.copy())
    merge_time = time.time() - start

    # 测试快速排序性能
    start = time.time()
    quick_sort(large_data.copy())
    quick_time = time.time() - start

    print(f"归并排序耗时: {merge_time:.4f} s")
    print(f"快速排序耗时: {quick_time:.4f} s")

    # 快速排序应该比归并排序快
    self.assertLess(quick_time, merge_time * 1.5)

def test_stability(self):
    """测试稳定性"""
    # 创建包含重复元素的复杂数据
```

```

data = [
    (3, 'a'), (1, 'b'), (2, 'c'), (1, 'd'), (3, 'e')
]

# 稳定排序应该保持相等元素的相对顺序
stable_result = stable_sort(data, key=lambda x: x[0])

# 检查稳定性
positions = {}
for i, (val, char) in enumerate(stable_result):
    if val not in positions:
        positions[val] = []
    positions[val].append((i, char))

# 对于每个值，字符应该保持原始相对顺序
for val in positions:
    chars = [char for _, char in positions[val]]
    original_chars = [char for v, char in data if v == val]
    self.assertEqual(chars, original_chars)

if __name__ == '__main__':
    unittest.main()
```
#### 边界条件测试
```python
class EdgeCaseTests(unittest.TestCase):

    def test_very_large_numbers(self):
        """测试极大数字"""
        nums = [10**18, 10**18-1, 10**18+1]
        result = quick_sort(nums.copy())
        self.assertEqual(result, sorted(nums))

    def test_float_precision(self):
        """测试浮点数精度"""
        nums = [0.1 + 0.2, 0.3, 0.1, 0.2]
        result = quick_sort(nums.copy())

# 浮点数比较需要容差
expected = sorted(nums)
for r, e in zip(result, expected):
    self.assertAlmostEqual(r, e, places=10)

```

```

def test_mixed_types(self):
    """测试混合类型（应该抛出异常）"""
    nums = [1, '2', 3.0]
    with self.assertRaises(TypeError):
        quick_sort(nums)

def test_nan_values(self):
    """测试NaN值处理"""
    import math
    nums = [1, 2, float('nan'), 3]
    with self.assertRaises(ValueError):
        quick_sort(nums)

# 运行特定测试
def run_comprehensive_tests():
    """运行全面的测试套件"""
    # 创建测试加载器
    loader = unittest.TestLoader()

    # 添加所有测试用例
    suite = loader.loadTestsFromTestCase(TestSortAlgorithms)
    suite.addTests(loader.loadTestsFromTestCase(EdgeCaseTests))

    # 运行测试
    runner = unittest.TextTestRunner(verbosity=2)
    result = runner.run(suite)

    return result.wasSuccessful()
```

```

## ## 🎯 面试深度问题准备

### ### 1. 算法原理深度问题

**\*\*问题\*\*:** “为什么快速排序在实际应用中比归并排序更快？”

**\*\*回答要点\*\*:**

- 常数因子: 快速排序的常数因子更小
- 缓存友好: 快速排序对缓存更友好
- 原地排序: 快速排序是原地排序, 减少内存分配
- 实际数据: 实际数据很少出现最坏情况

**\*\*示例回答\*\*:**

“快速排序在实际应用中通常比归并排序更快，主要有以下几个原因：首先，快速排序的常数因子更小，每次分区操作的开销相对较低。其次，快速排序对 CPU 缓存更友好，因为它的内存访问模式是连续的。另外，快速排序是原地排序算法，不需要额外的内存分配，这在处理大数据时非常重要。虽然快速排序的最坏时间复杂度是  $O(n^2)$ ，但通过随机化选择 pivot，实际应用中很少遇到最坏情况。”

### ### 2. 工程实践问题

**\*\*问题\*\*：**“在大数据场景下，你会如何设计排序系统？”

**\*\*回答要点\*\*：**

- 外部排序：使用多路归并排序
- 分布式处理：MapReduce 模式
- 内存管理：分批处理，避免内存溢出
- 容错机制：处理节点故障

**\*\*示例回答\*\*：**

“在大数据场景下，我会采用外部排序结合分布式处理的方案。首先，将大数据集分割成适合内存的小块，对每个块进行内部排序。然后使用多路归并算法将排序好的块合并。如果数据量特别大，我会使用分布式框架如 MapReduce，让多个节点并行处理不同的数据块。同时需要考虑容错机制，确保单个节点故障不会影响整体排序任务。内存管理方面，我会设计流式处理，避免一次性加载所有数据到内存。”

### ### 3. 优化策略问题

**\*\*问题\*\*：**“如何优化排序算法处理大量重复元素的情况？”

**\*\*回答要点\*\*：**

- 三路快排：专门处理重复元素
- 计数排序：适合小范围整数
- 提前终止：检测特殊情况
- 自适应算法：根据数据特征选择算法

**\*\*示例回答\*\*：**

“对于大量重复元素的情况，我会优先考虑三路快速排序，它能够将数组分成小于、等于、大于 pivot 的三部分，高效处理重复元素。如果数据范围较小，计数排序是更好的选择，时间复杂度可以达到  $O(n+k)$ 。另外，我会在排序前检测数据的重复率，如果重复率超过某个阈值，直接选择更适合的算法。还可以实现自适应算法，根据运行时数据特征动态调整排序策略。”

---

**\*\*持续补充更多高级内容和实战技巧...\*\***

文件：SummaryAndPatterns.md

## # 排序算法总结与模式识别

### ## 🔍 算法选择指南

#### ### 根据数据特性选择算法

| 数据特性                  | 推荐算法 | 理由                     |
|-----------------------|------|------------------------|
| 小规模数据 ( $n \leq 50$ ) | 插入排序 | 常数项小，实际效率高             |
| 中等规模数据                | 快速排序 | 平均性能最优                 |
| 大规模数据                 | 归并排序 | 稳定且性能可靠                |
| 内存受限                  | 堆排序  | 原地排序，空间复杂度 $O(1)$      |
| 需要稳定排序                | 归并排序 | 唯一稳定的 $O(n \log n)$ 算法 |
| 大量重复元素                | 三路快排 | 专门优化重复元素               |
| 数据基本有序                | 插入排序 | 接近 $O(n)$ 时间复杂度        |
| 数据范围有限                | 计数排序 | $O(n + k)$ 线性时间        |
| 数位数固定                 | 基数排序 | $O(dn)$ 线性时间           |

#### ### 根据应用场景选择算法

##### \*\*面试场景\*\*:

- 快速选择算法 (Top K 问题)
- 归并排序 (链表排序、逆序对)
- 堆排序 (优先级队列)

##### \*\*工程应用\*\*:

- 快速排序 (通用排序)
- 计数排序 (小范围整数)
- 基数排序 (大整数排序)

##### \*\*特殊需求\*\*:

- 稳定排序: 归并排序
- 原地排序: 堆排序、快速排序
- 外部排序: 归并排序

### ##💡 解题模式识别

#### ### 模式 1: Top K 问题

##### \*\*特征\*\*: 寻找前 K 大/小元素

##### \*\*解法选择\*\*:

- K 较小: 最小堆  $O(n \log k)$
- K 接近 n: 快速选择  $O(n)$
- 需要稳定: 排序后取  $O(n \log n)$

## \*\*相关题目\*\*:

- 215. 数组中的第 K 个最大元素: <https://leetcode.cn/problems/kth-largest-element-in-an-array/>
- 973. 最接近原点的 K 个点: <https://leetcode.cn/problems/k-closest-points-to-origin/>
- 347. 前 K 个高频元素: <https://leetcode.cn/problems/top-k-frequent-elements/>

## #### 模式 2: 区间合并

\*\*特征\*\*: 重叠区间合并

\*\*解法\*\*: 按起点排序后合并  $O(n \log n)$

## \*\*相关题目\*\*:

- 56. 合并区间: <https://leetcode.cn/problems/merge-intervals/>
- 986. 区间列表的交集: <https://leetcode.cn/problems/interval-list-intersections/>

## #### 模式 3: 颜色分类

\*\*特征\*\*: 有限种类排序

\*\*解法\*\*: 多指针法  $O(n)$

## \*\*相关题目\*\*:

- 75. 颜色分类: <https://leetcode.cn/problems/sort-colors/>
- 280. 摆动排序: <https://leetcode.cn/problems/wiggle-sort/>

## #### 模式 4: 逆序对统计

\*\*特征\*\*: 统计逆序对数量

\*\*解法\*\*: 归并排序  $O(n \log n)$

## \*\*相关题目\*\*:

- 493. 翻转对: <https://leetcode.cn/problems/reverse-pairs/>
- 面试题 51. 数组中的逆序对: <https://leetcode.cn/problems/shu-zu-zhong-de-ni-xu-dui-lcof/>

## ## 🔧 工程化考量

### #### 异常处理策略

```
```java
public void sort(int[] nums) {
    // 输入验证
    if (nums == null) throw new IllegalArgumentException("数组不能为 null");
    if (nums.length <= 1) return;

    // 边界检查
    if (nums.length > MAX_SIZE) {
        throw new IllegalArgumentException("数据规模过大");
    }
}
```

```
// 实际排序逻辑  
// ...  
}  
~~~
```

### ### 性能优化技巧

#### \*\*快速排序优化\*\*:

1. 随机化基准选择
2. 三路划分处理重复元素
3. 小数组使用插入排序
4. 尾递归优化

#### \*\*归并排序优化\*\*:

1. 迭代实现避免递归深度
2. 小数组使用插入排序
3. 原地归并减少空间使用

#### \*\*堆排序优化\*\*:

1. 减少交换次数
2. 使用 siftDown 优化建堆

### ### 内存使用优化

#### \*\*空间优化策略\*\*:

- 优先选择原地排序算法
- 避免不必要的数组拷贝
- 及时释放临时内存

#### \*\*缓存友好性\*\*:

- 顺序访问模式
- 减少随机内存访问
- 利用局部性原理

## ## 📈 复杂度分析深度

### ### 归并排序复杂度推导

~~~

$$\begin{aligned}T(n) &= 2T(n/2) + O(n) \\&= 2[2T(n/4) + O(n/2)] + O(n) \\&= 4T(n/4) + 2O(n/2) + O(n) \\&= 4T(n/4) + 2O(n)\end{aligned}$$

$$\begin{aligned} &= \dots \\ &= 2^k T(n/2^k) + kO(n) \end{aligned}$$

当  $n/2^k = 1 \Rightarrow k = \log_2 n$

$$T(n) = nT(1) + O(n \log n) = O(n \log n)$$

...

### ### 快速排序复杂度分析

\*\*最好情况\*\* (每次划分均衡) :

...

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

...

\*\*最坏情况\*\* (每次划分极端不平衡) :

...

$$T(n) = T(n-1) + O(n) = O(n^2)$$

...

\*\*平均情况\*\* (随机化达到  $O(n \log n)$ ) :

...

$$E[T(n)] = O(n \log n)$$

...

### ### 堆排序复杂度分析

\*\*建堆过程\*\*:

...

$$T(n) = O(n) \quad // \text{线性时间建堆}$$

...

\*\*排序过程\*\*:

...

$$T(n) = O(n \log n) \quad // \text{每次调整 } O(\log n), \text{ 共 } n \text{ 次}$$

...

## ## 🎓 面试技巧总结

### ### 算法原理理解

1. \*\*能够白板写出算法\*\*

- 归并排序的分治思想
- 快速排序的划分过程
- 堆排序的建堆和调整

2. \*\*复杂度分析能力\*\*

- 推导时间复杂度
- 分析空间复杂度
- 理解常数项影响

### 3. \*\*算法比较能力\*\*

- 各种排序算法的优缺点
- 适用场景分析
- 性能对比

## #### 代码实现要点

### 1. \*\*清晰的代码结构\*\*

- 模块化设计
- 清晰的变量命名
- 适当的注释

### 2. \*\*边界条件处理\*\*

- 空数组
- 单元素
- 已排序/逆序
- 重复元素

### 3. \*\*异常情况考虑\*\*

- 输入验证
- 内存限制
- 性能退化

## #### 问题分析能力

### 1. \*\*快速识别问题类型\*\*

- Top K 问题 → 快速选择/堆排序
- 区间合并 → 排序+合并
- 颜色分类 → 多指针法

### 2. \*\*选择合适的算法\*\*

- 根据数据规模选择
- 根据内存限制选择
- 根据稳定性要求选择

### 3. \*\*优化策略思考\*\*

- 常数项优化
- 空间优化
- 缓存友好性

#### #### 算法理论深入

1. \*\*排序网络理论\*\*
2. \*\*比较排序的下界\*\*
3. \*\*线性时间排序的局限性\*\*

#### #### 工程应用扩展

##### 1. \*\*并行排序算法\*\*

- 多线程归并排序
- MapReduce 排序

##### 2. \*\*外部排序\*\*

- 多路归并
- 败者树优化

##### 3. \*\*数据库排序优化\*\*

- 索引排序
- 多字段排序

#### #### 特殊数据类型排序

##### 1. \*\*字符串排序\*\*

- 字典序排序
- 后缀数组

##### 2. \*\*对象排序\*\*

- 多关键字排序
- 自定义比较器

##### 3. \*\*流数据排序\*\*

- 在线算法
- 近似排序

----

\*\*持续学习和实践是掌握排序算法的关键！\*\*

文件：补充说明.md

# Class001 项目补充说明

## 📁 项目文件结构说明

### ### 核心算法文件

- `SortAlgorithms.java` - Java 版本排序算法实现
- `SortAlgorithms.cpp` - C++版本排序算法实现
- `SortAlgorithms\_part1.py` - Python 版本排序算法第一部分
- `SortAlgorithms\_part2.py` - Python 版本排序算法第二部分

### ### 扩展问题实现

- `ExtendedProblems.java` - Java 版本扩展题目
- `ExtendedProblems.cpp` - C++版本扩展题目
- `ExtendedProblems.py` - Python 版本扩展题目

### ### 算法变种与优化

- `AlgorithmVariants\_part1.java` - 快速排序和归并排序变种
- `AlgorithmVariants\_part2.java` - 堆排序、计数排序、基数排序变种

### ### 测试与验证

- `ComprehensiveTest.java` - 综合测试类
- 包含边界条件测试、性能测试、内存监控等

### ### 文档资料

- `README.md` - 项目主文档
- `AdditionalProblems.md` - 补充题目列表
- `ProblemLinks.md` - 题目链接汇总
- `SummaryAndPatterns.md` - 算法总结与模式识别
- `补充说明.md` - 本项目说明文件

## ## 🔧 编译与运行说明

### ### Java 版本

```
```bash
# 编译所有 Java 文件
javac class001/*.java
```

运行测试

```
java class001.ComprehensiveTest
java class001.ExtendedProblems
```
```

### ### C++版本

```
```bash
# 编译 C++ 文件
g++ -std=c++11 class001/SortAlgorithms.cpp -o sort_algorithms
```

```
g++ -std=c++11 class001/ExtendedProblems.cpp -o extended_problems
```

```
# 运行程序  
../sort_algorithms  
../extended_problems  
```
```

```
Python 版本
```bash  
# 运行 Python 文件  
python class001/SortAlgorithms_part1.py  
python class001/SortAlgorithms_part2.py  
python class001/ExtendedProblems.py  
```
```

## ## 🔧 已知问题与修复

### #### C++编译问题

当前 C++文件存在编译错误，主要问题：

1. 头文件包含不完整
2. 标准库使用问题

**\*\*修复方案\*\*:**

```
```cpp  
// 添加必要的头文件  
#include <iostream>  
#include <vector>  
#include <algorithm>  
#include <queue>  
#include <unordered_map>  
#include <random>  
#include <chrono>  
#include <string>  
#include <sstream>  
#include <stdexcept>  
#include <functional>  
#include <memory>  
```
```

### #### Python 导入问题

Python 文件存在模块导入问题，需要确保文件在同一目录下。

## ## 📈 项目特色与亮点

#### #### 1. 多语言实现

- 提供 Java、C++、Python 三种语言的完整实现
- 每种语言都遵循其最佳实践和编码规范

#### #### 2. 算法变种全面

- 包含各种排序算法的变种实现
- 每种变种都有详细的时间/空间复杂度分析

#### #### 3. 工程化考量

- 完整的异常处理机制
- 边界条件全面测试
- 性能监控和内存使用分析

#### #### 4. 题目覆盖广泛

- LeetCode、牛客网、剑指 Offer 等平台题目
- 每种题目提供多种解法
- 包含最优解分析和复杂度比较

#### #### 5. 学习资源丰富

- 算法原理深度解析
- 解题模式识别指南
- 面试技巧总结
- 进阶学习方向

### ## 🚀 使用建议

#### #### 学习路径

1. \*\*初学者\*\*: 先从`SortAlgorithms`文件开始，理解基础算法
2. \*\*进阶学习\*\*: 学习`AlgorithmVariants`中的算法变种
3. \*\*实战练习\*\*: 使用`ExtendedProblems`进行题目练习
4. \*\*综合测试\*\*: 运行`ComprehensiveTest`验证掌握程度

#### #### 面试准备

1. \*\*基础算法\*\*: 熟练掌握归并、快速、堆排序
2. \*\*扩展题目\*\*: 重点练习 Top K、区间合并等经典问题
3. \*\*代码实现\*\*: 能够白板写出无 bug 的代码
4. \*\*复杂度分析\*\*: 能够准确分析时间/空间复杂度

#### #### 工程应用

1. \*\*算法选择\*\*: 根据实际场景选择合适的排序算法
2. \*\*性能优化\*\*: 应用算法变种和优化技巧
3. \*\*异常处理\*\*: 确保代码的健壮性和可靠性

## ## 📈 更新计划

### #### 短期更新

- [ ] 修复 C++ 编译问题
- [ ] 添加更多单元测试用例
- [ ] 完善性能基准测试

### #### 长期规划

- [ ] 添加并行排序算法实现
- [ ] 扩展外部排序相关内容
- [ ] 添加更多平台的题目解析

## ## 🤝 贡献指南

欢迎对本项目进行改进和扩展：

1. **\*\*问题反馈\*\***: 发现 bug 或问题请提交 Issue
2. **\*\*代码贡献\*\***: 欢迎提交 Pull Request
3. **\*\*文档完善\*\***: 帮助完善文档和注释
4. **\*\*题目补充\*\***: 添加新的算法题目和解析

## ## 📞 联系方式

如有问题或建议，欢迎通过以下方式联系：

- 项目 Issue: 提交 GitHub Issue
- 邮件联系: [您的邮箱]
- 技术讨论: [相关技术社区]

---

**\*\*祝您学习愉快，算法精进！\*\***

\*最后更新: 2025 年 10 月 17 日\*

=====

[代码文件]

=====

文件: AlgorithmVariants.java

=====

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

```
/**
 * 排序算法变种和优化版本 - Java 版本
 * 包含各种排序算法的变种实现和特殊场景优化
 *
 * 算法变种列表：
 * 1. 三路快速排序变种 - 针对大量重复元素的优化版本
 * 2. 自底向上的归并排序（迭代版本）- 避免递归调用，节省栈空间
 * 3. 原地归并排序 - 空间复杂度优化版本
 * 4. 计数排序（非比较排序）- 适用于元素范围有限的整数排序
 * 5. 基数排序（非比较排序）- 适用于整数或字符串排序
 * 6. 桶排序 - 适用于均匀分布的浮点数排序
 * 7. 睡眠排序（搞笑算法，实际不可用）- 每个元素启动一个线程，睡眠对应时间后输出
 * 8. 猴子排序（Bogo Sort，随机算法）- 不断随机排列数组，直到有序
 * 9. 鸡尾酒排序（双向冒泡排序） - 冒泡排序的优化版本，双向进行
 * 10. 梳排序（Comb Sort） - 冒泡排序的改进，使用较大的间隔
 */
```

```
public class AlgorithmVariants {

 /**
 * 1. 三路快速排序变种
 * 针对大量重复元素的优化版本
 * 时间复杂度：O(n log n) 平均
 * 空间复杂度：O(log n)
 */

 public static class ThreeWayQuickSort {

 public static void sort(int[] arr) {
 if (arr == null || arr.length <= 1) return;
 threeWayQuickSort(arr, 0, arr.length - 1);
 }

 private static void threeWayQuickSort(int[] arr, int low, int high) {
 if (low >= high) return;

 // 三路分区
 int[] partition = threeWayPartition(arr, low, high);
 int lt = partition[0]; // 小于区域的右边界
 int gt = partition[1]; // 大于区域的左边界

 // 递归排序小于区域和大于区域
 threeWayQuickSort(arr, low, lt - 1);
 threeWayQuickSort(arr, gt + 1, high);
 }
 }
}
```

```

// 等于区域已经有序，不需要排序
}

private static int[] threeWayPartition(int[] arr, int low, int high) {
 int pivot = arr[low]; // 选择第一个元素作为基准
 int lt = low; // 小于区域的右边界
 int gt = high; // 大于区域的左边界
 int i = low + 1; // 当前指针

 while (i <= gt) {
 if (arr[i] < pivot) {
 swap(arr, lt++, i++);
 } else if (arr[i] > pivot) {
 swap(arr, i, gt--);
 } else {
 i++;
 }
 }

 return new int[]{lt, gt};
}

public static void test() {
 System.out.println("== 三路快速排序测试 ==");
 int[] arr = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5};
 System.out.println("原始数组: " + Arrays.toString(arr));

 sort(arr);
 System.out.println("排序后: " + Arrays.toString(arr));
}
}

/**
 * 2. 自底向上的归并排序（迭代版本）
 * 避免递归调用，节省栈空间
 * 时间复杂度: O(n log n)
 * 空间复杂度: O(n)
 */
public static class IterativeMergeSort {

 public static void sort(int[] arr) {
 if (arr == null || arr.length <= 1) return;

```

```

int n = arr.length;
int[] aux = new int[n];

// 从 1 开始，每次倍增
for (int size = 1; size < n; size *= 2) {
 for (int left = 0; left < n - size; left += 2 * size) {
 int mid = left + size - 1;
 int right = Math.min(left + 2 * size - 1, n - 1);
 merge(arr, aux, left, mid, right);
 }
}

private static void merge(int[] arr, int[] aux, int left, int mid, int right) {
 // 复制到辅助数组
 for (int k = left; k <= right; k++) {
 aux[k] = arr[k];
 }

 int i = left, j = mid + 1;
 for (int k = left; k <= right; k++) {
 if (i > mid) {
 arr[k] = aux[j++];
 } else if (j > right) {
 arr[k] = aux[i++];
 } else if (aux[i] <= aux[j]) {
 arr[k] = aux[i++];
 } else {
 arr[k] = aux[j++];
 }
 }
}

public static void test() {
 System.out.println("\n==== 迭代归并排序测试 ===");
 int[] arr = {64, 34, 25, 12, 22, 11, 90};
 System.out.println("原始数组: " + Arrays.toString(arr));

 sort(arr);
 System.out.println("排序后: " + Arrays.toString(arr));
}
}

```

```
/**
 * 3. 原地归并排序
 * 空间复杂度优化版本，但时间复杂度稍差
 * 时间复杂度: O(n log2 n)
 * 空间复杂度: O(1)
 */

public static class InPlaceMergeSort {

 public static void sort(int[] arr) {
 if (arr == null || arr.length <= 1) return;
 inPlaceMergeSort(arr, 0, arr.length - 1);
 }

 private static void inPlaceMergeSort(int[] arr, int left, int right) {
 if (left >= right) return;

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

 private static void inPlaceMerge(int[] arr, int left, int mid, int right) {
 int i = left;
 int j = mid + 1;

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

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

 // 更新指针
 i++;
 }
 }
 }
}
```

```

 mid++;
 j++;
 }
}

public static void test() {
 System.out.println("\n==== 原地归并排序测试 ===");
 int[] arr = {5, 2, 8, 1, 9, 3};
 System.out.println("原始数组: " + Arrays.toString(arr));

 sort(arr);
 System.out.println("排序后: " + Arrays.toString(arr));
}

/***
 * 4. 计数排序（非比较排序）
 * 适用于元素范围有限的整数排序
 * 时间复杂度: O(n + k), k 为元素范围
 * 空间复杂度: O(k)
 */
public static class CountingSort {

 public static void sort(int[] arr) {
 if (arr == null || arr.length <= 1) return;

 // 找到最大值和最小值
 int max = Arrays.stream(arr).max().getAsInt();
 int min = Arrays.stream(arr).min().getAsInt();
 int range = max - min + 1;

 // 创建计数数组
 int[] count = new int[range];
 int[] output = new int[arr.length];

 // 统计每个元素的出现次数
 for (int num : arr) {
 count[num - min]++;
 }

 // 计算累积计数
 for (int i = 1; i < range; i++) {

```

```

 count[i] += count[i - 1];
 }

 // 构建输出数组（保持稳定性）
 for (int i = arr.length - 1; i >= 0; i--) {
 output[count[arr[i] - min] - 1] = arr[i];
 count[arr[i] - min]--;
 }

 // 复制回原数组
 System.arraycopy(output, 0, arr, 0, arr.length);
}

public static void test() {
 System.out.println("\n==== 计数排序测试 ====");
 int[] arr = {4, 2, 2, 8, 3, 3, 1};
 System.out.println("原始数组: " + Arrays.toString(arr));

 sort(arr);
 System.out.println("排序后: " + Arrays.toString(arr));
}

/**
 * 5. 基数排序（非比较排序）
 * 适用于整数或字符串排序
 * 时间复杂度: O(d * (n + k)), d 为最大数位数
 * 空间复杂度: O(n + k)
 */
public static class RadixSort {

 public static void sort(int[] arr) {
 if (arr == null || arr.length <= 1) return;

 // 找到最大值，确定最大位数
 int max = Arrays.stream(arr).max().getAsInt();

 // 从最低位到最高位进行计数排序
 for (int exp = 1; max / exp > 0; exp *= 10) {
 countingSortByDigit(arr, exp);
 }
 }
}

```

```

private static void countingSortByDigit(int[] arr, int exp) {
 int n = arr.length;
 int[] output = new int[n];
 int[] count = new int[10];

 // 统计当前位的数字出现次数
 for (int num : arr) {
 count[(num / exp) % 10]++;
 }

 // 计算累积计数
 for (int i = 1; i < 10; i++) {
 count[i] += count[i - 1];
 }

 // 构建输出数组（从后往前保持稳定性）
 for (int i = n - 1; i >= 0; i--) {
 int digit = (arr[i] / exp) % 10;
 output[count[digit] - 1] = arr[i];
 count[digit]--;
 }

 // 复制回原数组
 System.arraycopy(output, 0, arr, 0, n);
}

public static void test() {
 System.out.println("\n==== 基数排序测试 ====");
 int[] arr = {170, 45, 75, 90, 2, 802, 24, 66};
 System.out.println("原始数组: " + Arrays.toString(arr));

 sort(arr);
 System.out.println("排序后: " + Arrays.toString(arr));
}

/**
 * 6. 桶排序
 * 适用于均匀分布的浮点数排序
 * 时间复杂度: O(n + k) 平均, O(n2) 最坏
 * 空间复杂度: O(n + k)
 */
public static class BucketSort {

```

```
public static void sort(double[] arr) {
 if (arr == null || arr.length <= 1) return;

 int n = arr.length;

 // 创建桶
 List<Double>[] buckets = new ArrayList[n];
 for (int i = 0; i < n; i++) {
 buckets[i] = new ArrayList<>();
 }

 // 将元素分配到桶中
 for (double num : arr) {
 int bucketIndex = (int) (num * n);
 buckets[bucketIndex].add(num);
 }

 // 对每个桶进行排序
 for (List<Double> bucket : buckets) {
 Collections.sort(bucket);
 }

 // 合并桶
 int index = 0;
 for (List<Double> bucket : buckets) {
 for (double num : bucket) {
 arr[index++] = num;
 }
 }
}

public static void test() {
 System.out.println("\n==== 桶排序测试 ====");
 double[] arr = {0.42, 0.32, 0.33, 0.52, 0.37, 0.47, 0.51};
 System.out.println("原始数组: " + Arrays.toString(arr));

 sort(arr);
 System.out.println("排序后: " + Arrays.toString(arr));
}

/**
```

```
* 7. 睡眠排序（搞笑算法，实际不可用）
* 每个元素启动一个线程，睡眠对应时间后输出
* 时间复杂度：O(max(arr)) 实际不可用
*/
public static class SleepSort {

 public static void sort(int[] arr) throws InterruptedException {
 if (arr == null || arr.length == 0) return;

 List<Integer> result = Collections.synchronizedList(new ArrayList<>());
 List<Thread> threads = new ArrayList<>();

 for (int num : arr) {
 Thread thread = new Thread(() -> {
 try {
 Thread.sleep(num * 10L); // 睡眠时间与数值成正比
 result.add(num);
 } catch (InterruptedException e) {
 Thread.currentThread().interrupt();
 }
 });
 threads.add(thread);
 thread.start();
 }

 // 等待所有线程完成
 for (Thread thread : threads) {
 thread.join();
 }

 // 将结果复制回原数组（仅用于演示）
 for (int i = 0; i < arr.length; i++) {
 arr[i] = result.get(i);
 }
 }

 public static void test() throws InterruptedException {
 System.out.println("\n==== 睡眠排序测试（仅演示） ====");
 int[] arr = {3, 1, 4, 1, 5};
 System.out.println("原始数组: " + Arrays.toString(arr));

 sort(arr);
 System.out.println("排序后: " + Arrays.toString(arr));
 }
}
```

```
 System.out.println("注意：睡眠排序仅用于演示，实际不可用！");
 }
}

/**
 * 8. 猴子排序 (Bogo Sort, 随机算法)
 * 不断随机排列数组，直到有序
 * 时间复杂度: O(∞) 平均, O(n!) 最坏
 */
public static class BogoSort {

 public static void sort(int[] arr) {
 if (arr == null || arr.length <= 1) return;

 Random random = new Random();
 int attempts = 0;
 int maxAttempts = 1000000; // 防止无限循环

 while (!isSorted(arr) && attempts < maxAttempts) {
 shuffle(arr, random);
 attempts++;
 }

 if (attempts >= maxAttempts) {
 System.out.println("猴子排序失败，尝试次数：" + attempts);
 } else {
 System.out.println("猴子排序成功，尝试次数：" + attempts);
 }
 }

 private static void shuffle(int[] arr, Random random) {
 for (int i = arr.length - 1; i > 0; i--) {
 int j = random.nextInt(i + 1);
 swap(arr, i, j);
 }
 }

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

```

 return true;
 }

public static void test() {
 System.out.println("\n==== 猴子排序测试（仅小数组演示） ====");
 int[] arr = {3, 1, 2};
 System.out.println("原始数组: " + Arrays.toString(arr));

 sort(arr);
 System.out.println("排序后: " + Arrays.toString(arr));
 System.out.println("注意: 猴子排序仅用于演示, 实际不可用!");
}
}

```

```

/**
 * 9. 鸡尾酒排序（双向冒泡排序）
 * 冒泡排序的优化版本，双向进行
 * 时间复杂度: O(n2)
 * 空间复杂度: O(1)
 */

```

```
public static class CocktailSort {
```

```

 public static void sort(int[] arr) {
 if (arr == null || arr.length <= 1) return;

 boolean swapped = true;
 int start = 0;
 int end = arr.length - 1;

 while (swapped) {
 swapped = false;

 // 从左到右
 for (int i = start; i < end; i++) {
 if (arr[i] > arr[i + 1]) {
 swap(arr, i, i + 1);
 swapped = true;
 }
 }

 if (!swapped) break;

 swapped = false;

 // 从右到左
 for (int i = end - 1; i > start; i--) {
 if (arr[i] < arr[i - 1]) {
 swap(arr, i, i - 1);
 swapped = true;
 }
 }
 }
 }
}
```

```

 end--;

 // 从右到左
 for (int i = end - 1; i >= start; i--) {
 if (arr[i] > arr[i + 1]) {
 swap(arr, i, i + 1);
 swapped = true;
 }
 }

 start++;
 }
}

public static void test() {
 System.out.println("\n==== 鸡尾酒排序测试 ===");
 int[] arr = {5, 1, 4, 2, 8, 0, 2};
 System.out.println("原始数组: " + Arrays.toString(arr));

 sort(arr);
 System.out.println("排序后: " + Arrays.toString(arr));
}
}

/**
 * 10. 梳排序 (Comb Sort)
 * 冒泡排序的改进，使用较大的间隔
 * 时间复杂度: O(n2) 最坏, O(n log n) 平均
 */
public static class CombSort {

 public static void sort(int[] arr) {
 if (arr == null || arr.length <= 1) return;

 int n = arr.length;
 int gap = n;
 boolean swapped = true;
 double shrink = 1.3; // 收缩因子

 while (gap > 1 || swapped) {
 gap = Math.max(1, (int)(gap / shrink));
 swapped = false;

 for (int i = 0; i < n - gap; i++) {
 if (arr[i] > arr[i + gap]) {
 swap(arr, i, i + gap);
 swapped = true;
 }
 }
 }
 }
}

```

```

 for (int i = 0; i + gap < n; i++) {
 if (arr[i] > arr[i + gap]) {
 swap(arr, i, i + gap);
 swapped = true;
 }
 }
 }

public static void test() {
 System.out.println("\n==== 梳排序测试 ====");
 int[] arr = {8, 4, 1, 56, 3, -44, 23, -6, 28, 0};
 System.out.println("原始数组: " + Arrays.toString(arr));

 sort(arr);
 System.out.println("排序后: " + Arrays.toString(arr));
}

private static void swap(int[] arr, int i, int j) {
 int temp = arr[i];
 arr[i] = arr[j];
 arr[j] = temp;
}

/**
 * 运行所有变种算法测试
 */
public static void runAllTests() {
 ThreeWayQuickSort.test();
 IterativeMergeSort.test();
 InPlaceMergeSort.test();
 CountingSort.test();
 RadixSort.test();
 BucketSort.test();
 CocktailSort.test();
 CombSort.test();

 // 注意: 以下算法仅用于演示, 实际不可用
 try {
 SleepSort.test();
 } catch (InterruptedException e) {
 System.out.println("睡眠排序被中断");
 }
}

```

```
 }
 BogoSort. test();
}

public static void main(String[] args) {
 runAllTests();
}
}
```

=====

文件: AlgorithmVariants\_part1.java

```
=====
/***
 * 排序算法变种与优化 - Java 版本 (第一部分)
 * 包含快速排序的各种变种实现
 */
```

```
import java.util.*;
import java.util.concurrent.ThreadLocalRandom;
```

```
public class AlgorithmVariants_part1 {
```

```
 /**
 * 快速排序的各种变种
 */
 public static class QuickSortVariants {
```

```
 /**
 * 基础快速排序
 * 时间复杂度: O(n log n) 平均, O(n2) 最坏
 * 空间复杂度: O(log n) 平均
 */

```

```
 public static void quickSortBasic(int[] nums) {
 if (nums == null || nums.length <= 1) return;
 quickSortBasic(nums, 0, nums.length - 1);
 }
```

```
 private static void quickSortBasic(int[] nums, int left, int right) {
 if (left >= right) return;
```

```
 int pivotIndex = partitionBasic(nums, left, right);
 quickSortBasic(nums, left, pivotIndex - 1);
 }
```

```

 quickSortBasic(nums, pivotIndex + 1, right);
 }

private static int partitionBasic(int[] nums, int left, int right) {
 int pivot = nums[right];
 int i = left;

 for (int j = left; j < right; j++) {
 if (nums[j] <= pivot) {
 swap(nums, i, j);
 i++;
 }
 }
 swap(nums, i, right);
 return i;
}

/***
 * 三路快速排序 - 处理大量重复元素
 * 时间复杂度: O(n log n) 平均
 * 空间复杂度: O(log n)
 */
public static void quickSortThreeWay(int[] nums) {
 if (nums == null || nums.length <= 1) return;
 quickSortThreeWay(nums, 0, nums.length - 1);
}

private static void quickSortThreeWay(int[] nums, int low, int high) {
 if (low >= high) return;

 // 三路划分
 int[] bounds = partitionThreeWay(nums, low, high);
 int lt = bounds[0]; // 小于 pivot 的右边界
 int gt = bounds[1]; // 大于 pivot 的左边界

 quickSortThreeWay(nums, low, lt - 1);
 quickSortThreeWay(nums, gt + 1, high);
}

private static int[] partitionThreeWay(int[] nums, int low, int high) {
 int pivot = nums[low];
 int lt = low; // nums[low..lt-1] < pivot
 int gt = high; // nums[gt+1..high] > pivot

```

```

int i = low + 1; // nums[lt..i-1] == pivot

while (i <= gt) {
 if (nums[i] < pivot) {
 swap(nums, lt++, i++);
 } else if (nums[i] > pivot) {
 swap(nums, i, gt--);
 } else {
 i++;
 }
}

return new int[] {lt, gt};
}

/***
 * 随机化快速排序 - 避免最坏情况
 * 时间复杂度: O(n log n) 期望
 * 空间复杂度: O(log n)
 */
public static void quickSortRandomized(int[] nums) {
 if (nums == null || nums.length <= 1) return;
 quickSortRandomized(nums, 0, nums.length - 1);
}

private static void quickSortRandomized(int[] nums, int left, int right) {
 if (left >= right) return;

 // 随机选择基准
 int randomIndex = ThreadLocalRandom.current().nextInt(left, right + 1);
 swap(nums, randomIndex, right);

 int pivotIndex = partitionBasic(nums, left, right);
 quickSortRandomized(nums, left, pivotIndex - 1);
 quickSortRandomized(nums, pivotIndex + 1, right);
}

/***
 * 尾递归优化快速排序 - 减少递归深度
 * 时间复杂度: O(n log n) 平均
 * 空间复杂度: O(log n) 最坏
 */
public static void quickSortTailRecursive(int[] nums) {

```

```

 if (nums == null || nums.length <= 1) return;
 quickSortTailRecursive(nums, 0, nums.length - 1);
}

private static void quickSortTailRecursive(int[] nums, int left, int right) {
 while (left < right) {
 int pivotIndex = partitionBasic(nums, left, right);

 // 递归处理较小的部分，迭代处理较大的部分
 if (pivotIndex - left < right - pivotIndex) {
 quickSortTailRecursive(nums, left, pivotIndex - 1);
 left = pivotIndex + 1;
 } else {
 quickSortTailRecursive(nums, pivotIndex + 1, right);
 right = pivotIndex - 1;
 }
 }
}

/***
 * 插入排序优化 - 小数组使用插入排序
 * 时间复杂度: O(n log n) 平均
 * 空间复杂度: O(log n)
 */
public static void quickSortWithInsertion(int[] nums) {
 if (nums == null || nums.length <= 1) return;
 quickSortWithInsertion(nums, 0, nums.length - 1);
}

private static void quickSortWithInsertion(int[] nums, int left, int right) {
 // 小数组使用插入排序
 if (right - left + 1 <= 16) {
 insertionSort(nums, left, right);
 return;
 }

 int pivotIndex = partitionBasic(nums, left, right);
 quickSortWithInsertion(nums, left, pivotIndex - 1);
 quickSortWithInsertion(nums, pivotIndex + 1, right);
}

private static void insertionSort(int[] nums, int left, int right) {
 for (int i = left + 1; i <= right; i++) {

```

```
int key = nums[i];
int j = i - 1;
while (j >= left && nums[j] > key) {
 nums[j + 1] = nums[j];
 j--;
}
nums[j + 1] = key;
}

private static void swap(int[] nums, int i, int j) {
 int temp = nums[i];
 nums[i] = nums[j];
 nums[j] = temp;
}

/**
 * 测试各种快速排序变种
 */
public static void testAllVariants() {
 System.out.println("== 快速排序变种测试 ==");

 int[] testData = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5};
 System.out.println("原始数据: " + Arrays.toString(testData));

 // 测试基础快速排序
 int[] data1 = testData.clone();
 quickSortBasic(data1);
 System.out.println("基础快速排序: " + Arrays.toString(data1));

 // 测试三路快速排序
 int[] data2 = testData.clone();
 quickSortThreeWay(data2);
 System.out.println("三路快速排序: " + Arrays.toString(data2));

 // 测试随机化快速排序
 int[] data3 = testData.clone();
 quickSortRandomized(data3);
 System.out.println("随机化快速排序: " + Arrays.toString(data3));

 // 测试尾递归优化
 int[] data4 = testData.clone();
 quickSortTailRecursive(data4);
}
```

```
 System.out.println("尾递归优化: " + Arrays.toString(data4));\n\n // 测试插入排序优化\n int[] data5 = testData.clone();\n quickSortWithInsertion(data5);\n System.out.println("插入排序优化: " + Arrays.toString(data5));\n }\n}\n\n/**\n * 归并排序的各种变种\n */\npublic static class MergeSortVariants {\n\n /**\n * 递归归并排序\n * 时间复杂度: O(n log n)\n * 空间复杂度: O(n)\n */\n public static void mergeSortRecursive(int[] nums) {\n if (nums == null || nums.length <= 1) return;\n mergeSortRecursive(nums, 0, nums.length - 1);\n }\n\n private static void mergeSortRecursive(int[] nums, int left, int right) {\n if (left >= right) return;\n\n int mid = left + (right - left) / 2;\n mergeSortRecursive(nums, left, mid);\n mergeSortRecursive(nums, mid + 1, right);\n merge(nums, left, mid, right);\n }\n\n /**\n * 迭代归并排序 (自底向上)\n * 时间复杂度: O(n log n)\n * 空间复杂度: O(n)\n */\n public static void mergeSortIterative(int[] nums) {\n if (nums == null || nums.length <= 1) return;\n\n int n = nums.length;\n int[] temp = new int[n];
```

```

for (int size = 1; size < n; size *= 2) {
 for (int left = 0; left < n - size; left += 2 * size) {
 int mid = left + size - 1;
 int right = Math.min(left + 2 * size - 1, n - 1);
 merge(nums, left, mid, right);
 }
}
}

/***
 * 原地归并排序 (减少空间使用)
 * 时间复杂度: O(n log n)
 * 空间复杂度: O(1) 额外空间
 */
public static void mergeSortInPlace(int[] nums) {
 if (nums == null || nums.length <= 1) return;
 mergeSortInPlace(nums, 0, nums.length - 1);
}

private static void mergeSortInPlace(int[] nums, int left, int right) {
 if (left >= right) return;

 int mid = left + (right - left) / 2;
 mergeSortInPlace(nums, left, mid);
 mergeSortInPlace(nums, mid + 1, right);
 mergeInPlace(nums, left, mid, right);
}

private static void mergeInPlace(int[] nums, int left, int mid, int right) {
 int i = left, j = mid + 1;

 while (i <= mid && j <= right) {
 if (nums[i] <= nums[j]) {
 i++;
 } else {
 int value = nums[j];
 int index = j;

 // 向右移动元素
 while (index != i) {
 nums[index] = nums[index - 1];
 index--;
 }
 }
 }
}

```

```

 }

 nums[i] = value;

 i++;
 mid++;
 j++;
 }
}

}

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

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

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

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

/***
 * 测试各种归并排序变种
 */
public static void testAllVariants() {
 System.out.println("\n==== 归并排序变种测试 ===");

 int[] testData = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5};
 System.out.println("原始数据: " + Arrays.toString(testData));

 // 测试递归归并排序
 int[] data1 = testData.clone();
 mergeSortRecursive(data1);
 System.out.println("递归归并排序: " + Arrays.toString(data1));

 // 测试迭代归并排序
}

```

```

 int[] data2 = testData.clone();
 mergeSortIterative(data2);
 System.out.println("迭代归并排序: " + Arrays.toString(data2));

 // 测试原地归并排序
 int[] data3 = testData.clone();
 mergeSortInPlace(data3);
 System.out.println("原地归并排序: " + Arrays.toString(data3));
 }

}

// 主函数
public static void main(String[] args) {
 QuickSortVariants.testAllVariants();
 MergeSortVariants.testAllVariants();
}
}
=====

文件: AlgorithmVariants_part2.java
=====

/***
 * 排序算法变种与优化 - Java 版本 (第二部分)
 * 包含堆排序、计数排序、基数排序的变种实现
 */

```

```

import java.util.*;
import java.util.concurrent.ThreadLocalRandom;

public class AlgorithmVariants_part2 {

 /**
 * 堆排序的各种变种
 */
 public static class HeapSortVariants {

 /**
 * 基础堆排序 (最大堆)
 * 时间复杂度: O(n log n)
 * 空间复杂度: O(1)
 */
 public static void heapSortBasic(int[] nums) {

```

```
if (nums == null || nums.length <= 1) return;

int n = nums.length;

// 构建最大堆
for (int i = n / 2 - 1; i >= 0; i--) {
 heapify(nums, n, i);
}

// 逐个提取最大元素
for (int i = n - 1; i > 0; i--) {
 swap(nums, 0, i);
 heapify(nums, i, 0);
}

/***
 * 最小堆排序 (降序排序)
 * 时间复杂度: O(n log n)
 * 空间复杂度: O(1)
 */
public static void heapSortMinHeap(int[] nums) {
 if (nums == null || nums.length <= 1) return;

 int n = nums.length;

 // 构建最小堆
 for (int i = n / 2 - 1; i >= 0; i--) {
 heapifyMin(nums, n, i);
 }

 // 逐个提取最小元素
 for (int i = n - 1; i > 0; i--) {
 swap(nums, 0, i);
 heapifyMin(nums, i, 0);
 }
}

/***
 * 原地堆排序 (优化空间使用)
 * 时间复杂度: O(n log n)
 * 空间复杂度: O(1)
 */

```

```
public static void heapSortInPlace(int[] nums) {
 if (nums == null || nums.length <= 1) return;
 heapSortBasic(nums); // 基础堆排序已经是原地排序
}

/**
 * 堆排序优化 - 减少交换次数
 * 时间复杂度: O(n log n)
 * 空间复杂度: O(1)
 */
public static void heapSortOptimized(int[] nums) {
 if (nums == null || nums.length <= 1) return;

 int n = nums.length;

 // 构建堆的优化版本
 for (int i = (n - 2) / 2; i >= 0; i--) {
 siftDown(nums, i, n);
 }

 // 排序优化版本
 for (int i = n - 1; i > 0; i--) {
 swap(nums, 0, i);
 siftDown(nums, 0, i);
 }
}

private static void heapify(int[] nums, int n, int i) {
 int largest = i;
 int left = 2 * i + 1;
 int right = 2 * i + 2;

 if (left < n && nums[left] > nums[largest]) {
 largest = left;
 }

 if (right < n && nums[right] > nums[largest]) {
 largest = right;
 }

 if (largest != i) {
 swap(nums, i, largest);
 heapify(nums, n, largest);
 }
}
```

```
 }
 }

private static void heapifyMin(int[] nums, int n, int i) {
 int smallest = i;
 int left = 2 * i + 1;
 int right = 2 * i + 2;

 if (left < n && nums[left] < nums[smallest]) {
 smallest = left;
 }

 if (right < n && nums[right] < nums[smallest]) {
 smallest = right;
 }

 if (smallest != i) {
 swap(nums, i, smallest);
 heapifyMin(nums, n, smallest);
 }
}

private static void siftDown(int[] nums, int i, int n) {
 int value = nums[i];

 while (2 * i + 1 < n) {
 int child = 2 * i + 1;

 // 选择较大的子节点
 if (child + 1 < n && nums[child + 1] > nums[child]) {
 child++;
 }

 if (value >= nums[child]) {
 break;
 }

 nums[i] = nums[child];
 i = child;
 }

 nums[i] = value;
}
```

```
private static void swap(int[] nums, int i, int j) {
 int temp = nums[i];
 nums[i] = nums[j];
 nums[j] = temp;
}

/**
 * 测试各种堆排序变种
 */
public static void testAllVariants() {
 System.out.println("== 堆排序变种测试 ==");

 int[] testData = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5};
 System.out.println("原始数据: " + Arrays.toString(testData));

 // 测试基础堆排序
 int[] data1 = testData.clone();
 heapSortBasic(data1);
 System.out.println("基础堆排序: " + Arrays.toString(data1));

 // 测试最小堆排序
 int[] data2 = testData.clone();
 heapSortMinHeap(data2);
 System.out.println("最小堆排序: " + Arrays.toString(data2));

 // 测试优化堆排序
 int[] data3 = testData.clone();
 heapSortOptimized(data3);
 System.out.println("优化堆排序: " + Arrays.toString(data3));
}

}

/**
 * 计数排序的各种变种
 */
public static class CountingSortVariants {

 /**
 * 基础计数排序 (非负整数)
 * 时间复杂度: O(n + k)
 * 空间复杂度: O(k)
 */
}
```

```
public static void countingSortBasic(int[] nums) {
 if (nums == null || nums.length <= 1) return;

 // 找出最大值
 int max = Arrays.stream(nums).max().getAsInt();

 // 创建计数数组
 int[] count = new int[max + 1];

 // 统计每个数字出现的次数
 for (int num : nums) {
 count[num]++;
 }

 // 重新填充数组
 int index = 0;
 for (int i = 0; i <= max; i++) {
 while (count[i] > 0) {
 nums[index++] = i;
 count[i]--;
 }
 }
}

/**
 * 稳定计数排序（保持相同元素的相对顺序）
 * 时间复杂度: O(n + k)
 * 空间复杂度: O(n + k)
 */
public static void countingSortStable(int[] nums) {
 if (nums == null || nums.length <= 1) return;

 int max = Arrays.stream(nums).max().getAsInt();
 int min = Arrays.stream(nums).min().getAsInt();
 int range = max - min + 1;

 int[] count = new int[range];
 int[] output = new int[nums.length];

 // 统计频率
 for (int num : nums) {
 count[num - min]++;
 }
```

```

// 计算累积频率
for (int i = 1; i < range; i++) {
 count[i] += count[i - 1];
}

// 从后向前遍历，保证稳定性
for (int i = nums.length - 1; i >= 0; i--) {
 output[count[nums[i] - min] - 1] = nums[i];
 count[nums[i] - min]--;
}

// 复制回原数组
System.arraycopy(output, 0, nums, 0, nums.length);
}

/**
 * 计数排序优化 - 处理负数
 * 时间复杂度: O(n + k)
 * 空间复杂度: O(k)
 */
public static void countingSortWithNegative(int[] nums) {
 if (nums == null || nums.length <= 1) return;

 int max = Arrays.stream(nums).max().getAsInt();
 int min = Arrays.stream(nums).min().getAsInt();
 int range = max - min + 1;

 int[] count = new int[range];

 // 统计频率
 for (int num : nums) {
 count[num - min]++;
 }

 // 重新填充数组
 int index = 0;
 for (int i = 0; i < range; i++) {
 while (count[i] > 0) {
 nums[index++] = i + min;
 count[i]--;
 }
 }
}

```

```

}

/**
 * 测试各种计数排序变种
 */
public static void testAllVariants() {
 System.out.println("\n==== 计数排序变种测试 ===");

 // 测试非负整数
 int[] testData1 = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5};
 System.out.println("原始数据(非负): " + Arrays.toString(testData1));

 int[] data1 = testData1.clone();
 countingSortBasic(data1);
 System.out.println("基础计数排序: " + Arrays.toString(data1));

 // 测试包含负数
 int[] testData2 = {3, -1, 4, 1, -5, 9, 2, -6, 5, 3, 5};
 System.out.println("原始数据(含负数): " + Arrays.toString(testData2));

 int[] data2 = testData2.clone();
 countingSortWithNegative(data2);
 System.out.println("负数计数排序: " + Arrays.toString(data2));

 // 测试稳定排序
 int[] testData3 = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5};
 System.out.println("原始数据(稳定性测试): " + Arrays.toString(testData3));

 int[] data3 = testData3.clone();
 countingSortStable(data3);
 System.out.println("稳定计数排序: " + Arrays.toString(data3));
}

/**
 * 基数排序的各种变种
 */
public static class RadixSortVariants {

 /**
 * LSD 基数排序 (最低位优先)
 * 时间复杂度: O(d * (n + k))
 * 空间复杂度: O(n + k)
 */
}

```

```

*/
public static void radixSortLSD(int[] nums) {
 if (nums == null || nums.length <= 1) return;

 int max = Arrays.stream(nums).max().getAsInt();

 // 按每一位进行计数排序
 for (int exp = 1; max / exp > 0; exp *= 10) {
 countingSortByDigit(nums, exp);
 }
}

/**
 * MSD 基数排序 (最高位优先)
 * 时间复杂度: O(d * (n + k))
 * 空间复杂度: O(n + k)
 */
public static void radixSortMSD(int[] nums) {
 if (nums == null || nums.length <= 1) return;
 radixSortMSD(nums, 0, nums.length - 1, getMaxDigits(nums));
}

private static void radixSortMSD(int[] nums, int low, int high, int digit) {
 if (digit <= 0 || low >= high) return;

 // 使用计数排序按当前位排序
 int[] count = new int[10];
 int[] output = new int[high - low + 1];
 int exp = (int) Math.pow(10, digit - 1);

 // 统计频率
 for (int i = low; i <= high; i++) {
 int digitValue = (nums[i] / exp) % 10;
 count[digitValue]++;
 }

 // 计算累积频率
 for (int i = 1; i < 10; i++) {
 count[i] += count[i - 1];
 }

 // 从后向前遍历
 for (int i = high; i >= low; i--) {

```

```

 int digitValue = (nums[i] / exp) % 10;
 output[count[digitValue] - 1] = nums[i];
 count[digitValue]--;
 }

 // 复制回原数组
 System.arraycopy(output, 0, nums, low, output.length);

 // 递归处理每个桶
 int start = low;
 for (int i = 0; i < 10; i++) {
 int bucketSize = (i == 0) ? count[0] : count[i] - count[i - 1];
 if (bucketSize > 1) {
 radixSortMSD(nums, start, start + bucketSize - 1, digit - 1);
 }
 start += bucketSize;
 }
}

/**
 * 基数排序优化 - 处理负数
 * 时间复杂度: O(d * (n + k))
 * 空间复杂度: O(n + k)
 */
public static void radixSortWithNegative(int[] nums) {
 if (nums == null || nums.length <= 1) return;

 // 分离正负数
 List<Integer> positives = new ArrayList<>();
 List<Integer> negatives = new ArrayList<>();

 for (int num : nums) {
 if (num >= 0) {
 positives.add(num);
 } else {
 negatives.add(-num); // 转换为正数处理
 }
 }

 // 分别排序
 int[] posArr = positives.stream().mapToInt(i -> i).toArray();
 int[] negArr = negatives.stream().mapToInt(i -> i).toArray();
}

```

```

if (posArr.length > 0) radixSortLSD(posArr);
if (negArr.length > 0) radixSortLSD(negArr);

// 合并结果 (负数需要反转)
int index = 0;
for (int i = negArr.length - 1; i >= 0; i--) {
 nums[index++] = -negArr[i];
}
for (int num : posArr) {
 nums[index++] = num;
}
}

private static void countingSortByDigit(int[] nums, int exp) {
 int n = nums.length;
 int[] output = new int[n];
 int[] count = new int[10];

 // 统计频率
 for (int i = 0; i < n; i++) {
 int digit = (nums[i] / exp) % 10;
 count[digit]++;
 }

 // 计算累积频率
 for (int i = 1; i < 10; i++) {
 count[i] += count[i - 1];
 }

 // 从后向前遍历
 for (int i = n - 1; i >= 0; i--) {
 int digit = (nums[i] / exp) % 10;
 output[count[digit] - 1] = nums[i];
 count[digit]--;
 }

 // 复制回原数组
 System.arraycopy(output, 0, nums, 0, n);
}

private static int getMaxDigits(int[] nums) {
 int max = Arrays.stream(nums).max().getAsInt();
 return (int) Math.log10(max) + 1;
}

```

```

}

/**
 * 测试各种基数排序变种
 */
public static void testAllVariants() {
 System.out.println("\n==== 基数排序变种测试 ===");

 int[] testData = {170, 45, 75, 90, 2, 802, 24, 66};
 System.out.println("原始数据: " + Arrays.toString(testData));

 int[] data1 = testData.clone();
 radixSortLSD(data1);
 System.out.println("LSD 基数排序: " + Arrays.toString(data1));

 int[] data2 = testData.clone();
 radixSortMSD(data2);
 System.out.println("MSD 基数排序: " + Arrays.toString(data2));

 // 测试负数处理
 int[] testData2 = {170, -45, 75, -90, 2, 802, -24, 66};
 System.out.println("原始数据(含负数): " + Arrays.toString(testData2));

 int[] data3 = testData2.clone();
 radixSortWithNegative(data3);
 System.out.println("负数基数排序: " + Arrays.toString(data3));
}

}

// 主函数
public static void main(String[] args) {
 HeapSortVariants.testAllVariants();
 CountingSortVariants.testAllVariants();
 RadixSortVariants.testAllVariants();
}
}

```

=====

文件: ComprehensiveTest.java

=====

```

/**
 * 综合测试类 - 测试所有排序算法和扩展题目

```

```
* 包含完整的单元测试、性能测试和边界测试
*
* 时间复杂度分析：各种测试场景的时间复杂度
* 空间复杂度分析：内存使用情况监控
* 最优解验证：确保算法实现是最优解
*
* @author Algorithm Specialist
* @version 1.0
* @date 2025-10-17
*
* 相关题目链接：
* - 基础排序算法测试
* - 扩展题目测试
* - 性能测试
* - 边界测试
*/

```

```
import java.util.*;
import java.util.concurrent.TimeUnit;

// 导入排序算法类
// 直接使用类名调用方法

public class ComprehensiveTest {

 /**
 * 主测试方法 - 运行所有测试
 */
 public static void main(String[] args) {
 System.out.println("==> 排序算法综合测试开始 ==>\n");

 // 运行基础算法测试
 testBasicAlgorithms();

 // 运行扩展题目测试
 testExtendedProblems();

 // 运行性能测试
 testPerformance();

 // 运行边界测试
 testEdgeCases();
 }
}
```

```
System.out.println("\n==> 排序算法综合测试完成 ==>");
}

/**
 * 测试基础排序算法
 * 时间复杂度: O(n2) 到 O(n log n) 取决于算法
 * 空间复杂度: O(1) 到 O(n) 取决于算法
 */
public static void testBasicAlgorithms() {
 System.out.println("💡 测试基础排序算法");

 // 测试数据
 int[] testArray = {64, 34, 25, 12, 22, 11, 90};
 int[] expected = {11, 12, 22, 25, 34, 64, 90};

 // 测试归并排序
 int[] mergeResult = Arrays.copyOf(testArray, testArray.length);
 SortAlgorithms.mergeSort(mergeResult);
 assertArrayEquals("归并排序", expected, mergeResult);

 // 测试快速排序
 int[] quickResult = Arrays.copyOf(testArray, testArray.length);
 SortAlgorithms.quickSort(quickResult);
 assertArrayEquals("快速排序", expected, quickResult);

 // 测试堆排序
 int[] heapResult = Arrays.copyOf(testArray, testArray.length);
 SortAlgorithms.heapSort(heapResult);
 assertArrayEquals("堆排序", expected, heapResult);

 System.out.println("✅ 基础排序算法测试通过\n");
}

/**
 * 测试扩展题目
 * 验证各种排序相关问题的解决方案
 */
public static void testExtendedProblems() {
 System.out.println("💡 测试扩展题目");

 // 测试合并有序数组
 testMergeSortedArrays();
}
```

```
// 测试最接近点选择
testKClosestPoints();

// 测试条形码重排
testRearrangeBarcodes();

// 测试摆动排序
testWiggleSort();

// 测试翻转对统计
testReversePairs();

// 测试最小数字排列
testMinNumber();

// 测试逆序对计数
testReversePairsCount();

System.out.println("✅ 扩展题目测试通过\n");
}
```

```
/***
 * 性能测试 - 测试算法在不同数据规模下的表现
 * 时间复杂度分析：验证算法的时间复杂度
 * 空间复杂度监控：检测内存使用情况
 */
```

```
public static void testPerformance() {
 System.out.println("📊 性能测试");

 // 测试不同规模的数据
 int[] sizes = {100, 1000, 5000, 10000};

 for (int size : sizes) {
 System.out.println("\n测试数据规模: " + size);

 // 生成测试数据
 int[] data = generateRandomArray(size);
 int[] dataCopy1 = Arrays.copyOf(data, data.length);
 int[] dataCopy2 = Arrays.copyOf(data, data.length);
 int[] dataCopy3 = Arrays.copyOf(data, data.length);

 // 测试归并排序性能
 long startTime = System.nanoTime();
```

```
SortAlgorithms.mergeSort(data);
long mergeTime = System.nanoTime() - startTime;

// 测试快速排序性能
startTime = System.nanoTime();
SortAlgorithms.quickSort(dataCopy1);
long quickTime = System.nanoTime() - startTime;

// 测试堆排序性能
startTime = System.nanoTime();
SortAlgorithms.heapSort(dataCopy2);
long heapTime = System.nanoTime() - startTime;

// 测试内置排序性能
startTime = System.nanoTime();
Arrays.sort(dataCopy3);
long builtinTime = System.nanoTime() - startTime;

System.out.printf("归并排序: %10d ns%n", mergeTime);
System.out.printf("快速排序: %10d ns%n", quickTime);
System.out.printf("堆排序: %10d ns%n", heapTime);
System.out.printf("内置排序: %10d ns%n", builtinTime);

// 验证排序结果正确性
assert isSorted(data) : "归并排序结果错误";
assert isSorted(dataCopy1) : "快速排序结果错误";
assert isSorted(dataCopy2) : "堆排序结果错误";
}

System.out.println("✅ 性能测试完成\n");
}

/***
 * 边界测试 - 测试各种极端情况
 * 确保算法在各种边界条件下的鲁棒性
 */
public static void testEdgeCases() {
 System.out.println("⚠️ 边界测试");

 // 测试空数组
 testEmptyArray();

 // 测试单元素数组
}
```

```
testSingleElement();

// 测试已排序数组
testSortedArray();

// 测试逆序数组
testReverseSortedArray();

// 测试重复元素数组
testDuplicateElements();

// 测试包含负数的数组
testNegativeNumbers();

// 测试大规模重复数据
testLargeDuplicateData();

System.out.println("✅ 边界测试通过\n");
}
```

// ===== 扩展题目具体测试方法 =====

```
private static void testMergeSortedArrays() {
 int[] nums1 = {1, 2, 3, 0, 0, 0};
 int[] nums2 = {2, 5, 6};
 ExtendedSortProblems.mergeSortedArrays(nums1, 3, nums2, 3);
 int[] expected = {1, 2, 2, 3, 5, 6};
 assertArrayEquals("合并有序数组", expected, nums1);
}
```

```
private static void testKClosestPoints() {
 int[][] points = {{1, 3}, {-2, 2}, {0, 1}};
 int k = 2;
 int[][] result = ExtendedSortProblems.kClosest(points, k);
 System.out.println("最接近点测试通过");
}
```

```
private static void testRearrangeBarcodes() {
 int[] barcodes = {1, 1, 1, 2, 2, 2};
 int[] result = ExtendedSortProblems.rearrangeBarcodes(barcodes);
 // 验证相邻元素不重复
 for (int i = 1; i < result.length; i++) {
 assert result[i] != result[i-1] : "条形码重排错误";
 }
}
```

```

 }

 System.out.println("条形码重排测试通过");
}

private static void testWiggleSort() {
 int[] nums = {1, 5, 1, 1, 6, 4};
 ExtendedSortProblems.wiggleSort(nums);
 // 验证摆动排序条件
 for (int i = 1; i < nums.length - 1; i += 2) {
 assert nums[i] >= nums[i-1] && nums[i] >= nums[i+1] : "摆动排序错误";
 }
 System.out.println("摆动排序测试通过");
}

private static void testReversePairs() {
 int[] nums = {1, 3, 2, 3, 1};
 int result = ExtendedSortProblems.reversePairs493(nums);
 assert result == 2 : "翻转对统计错误";
 System.out.println("翻转对统计测试通过");
}

private static void testMinNumber() {
 int[] nums = {10, 2};
 String result = ExtendedSortProblems.minNumber(nums);
 assert "102".equals(result) : "最小数字排列错误";
 System.out.println("最小数字排列测试通过");
}

private static void testReversePairsCount() {
 int[] nums = {7, 5, 6, 4};
 long result = ExtendedSortProblems.countInversions(nums);
 assert result == 5 : "逆序对计数错误";
 System.out.println("逆序对计数测试通过");
}

// ====== 边界测试具体方法 ======

private static void testEmptyArray() {
 int[] empty = {};
 SortAlgorithms.mergeSort(empty); // 应该不报错
 System.out.println("空数组测试通过");
}

```

```
private static void testSingleElement() {
 int[] single = {42};
 SortAlgorithms.quickSort(single);
 assert single[0] == 42 : "单元素数组测试失败";
 System.out.println("单元素数组测试通过");
}

private static void testSortedArray() {
 int[] sorted = {1, 2, 3, 4, 5};
 SortAlgorithms.heapSort(sorted);
 assert isSorted(sorted) : "已排序数组测试失败";
 System.out.println("已排序数组测试通过");
}

private static void testReverseSortedArray() {
 int[] reverse = {5, 4, 3, 2, 1};
 SortAlgorithms.mergeSort(reverse);
 assert isSorted(reverse) : "逆序数组测试失败";
 System.out.println("逆序数组测试通过");
}

private static void testDuplicateElements() {
 int[] duplicates = {2, 2, 1, 1, 3, 3};
 SortAlgorithms.quickSort(duplicates);
 assert isSorted(duplicates) : "重复元素数组测试失败";
 System.out.println("重复元素数组测试通过");
}

private static void testNegativeNumbers() {
 int[] negatives = {-3, -1, -2, 0, 1};
 SortAlgorithms.heapSort(negatives);
 assert isSorted(negatives) : "负数数组测试失败";
 System.out.println("负数数组测试通过");
}

private static void testLargeDuplicateData() {
 int[] largeData = new int[1000];
 Arrays.fill(largeData, 42); // 所有元素相同
 Arrays.fill(largeData, 500, 1000, 24); // 部分元素不同

 SortAlgorithms.mergeSort(largeData);
 assert isSorted(largeData) : "大规模重复数据测试失败";
 System.out.println("大规模重复数据测试通过");
}
```

```
}

// ===== 工具方法 =====

/***
 * 生成随机数组用于测试
 */
private static int[] generateRandomArray(int size) {
 Random random = new Random();
 int[] array = new int[size];
 for (int i = 0; i < size; i++) {
 array[i] = random.nextInt(size * 10);
 }
 return array;
}

/***
 * 验证数组是否已排序
 */
private static boolean isSorted(int[] array) {
 for (int i = 1; i < array.length; i++) {
 if (array[i] < array[i - 1]) {
 return false;
 }
 }
 return true;
}

/***
 * 断言两个数组相等
 */
private static void assertArrayEquals(String testName, int[] expected, int[] actual) {
 if (expected.length != actual.length) {
 throw new AssertionError(testName + "失败: 数组长度不匹配");
 }
 for (int i = 0; i < expected.length; i++) {
 if (expected[i] != actual[i]) {
 throw new AssertionError(testName + "失败: 索引 " + i + " 处值不匹配");
 }
 }
 System.out.println("✓ " + testName + " 测试通过");
}
```

```
/**
 * 内存使用监控（简化版）
 */

private static void monitorMemoryUsage() {
 Runtime runtime = Runtime.getRuntime();
 long usedMemory = runtime.totalMemory() - runtime.freeMemory();
 System.out.println("当前内存使用: " + usedMemory + " bytes");
}

/**
 * 时间复杂度趋势分析
 */

private static void analyzeTimeComplexityTrend(int[] sizes, long[] times) {
 System.out.println("\n🕒 时间复杂度趋势分析:");
 for (int i = 0; i < sizes.length; i++) {
 double ratio = (double) times[i] / sizes[i];
 System.out.printf("规模 %d: 时间 %d ns, 比例: %.2f ns/element%n",
 sizes[i], times[i], ratio);
 }
}
}

/**
 * 性能监控工具类
 * 提供更详细的内存和时间监控功能
 */

class PerformanceMonitor {
 private long startTime;
 private long startMemory;

 public void start() {
 startTime = System.nanoTime();
 Runtime runtime = Runtime.getRuntime();
 startMemory = runtime.totalMemory() - runtime.freeMemory();
 }

 public PerformanceResult stop() {
 long endTime = System.nanoTime();
 Runtime runtime = Runtime.getRuntime();
 long endMemory = runtime.totalMemory() - runtime.freeMemory();

 return new PerformanceResult(
 endTime - startTime,
```

```
 endMemory - startMemory
);
}

public static class PerformanceResult {
 public final long timeNanos;
 public final long memoryBytes;

 public PerformanceResult(long timeNanos, long memoryBytes) {
 this.timeNanos = timeNanos;
 this.memoryBytes = memoryBytes;
 }

 @Override
 public String toString() {
 return String.format("时间: %d ns, 内存: %d bytes", timeNanos, memoryBytes);
 }
}

/**
 * 测试数据生成器
 * 生成各种类型的测试数据
 */
class TestDataGenerator {

 /**
 * 生成基本有序数组 (90%有序)
 */
 public static int[] generateMostlySortedArray(int size) {
 int[] array = new int[size];
 for (int i = 0; i < size; i++) {
 array[i] = i;
 }

 // 随机交换 10%的元素
 Random random = new Random();
 int swaps = size / 10;
 for (int i = 0; i < swaps; i++) {
 int idx1 = random.nextInt(size);
 int idx2 = random.nextInt(size);
 int temp = array[idx1];
 array[idx1] = array[idx2];
 array[idx2] = temp;
 }
 }
}
```

```
 array[idx2] = temp;
 }

 return array;
}

/***
 * 生成高斯分布数据
 */
public static int[] generateGaussianData(int size, double mean, double stdDev) {
 int[] array = new int[size];
 Random random = new Random();

 for (int i = 0; i < size; i++) {
 double value = mean + stdDev * random.nextGaussian();
 array[i] = (int) Math.round(value);
 }

 return array;
}

/***
 * 生成 Zipf 分布数据（常见于真实世界数据）
 */
public static int[] generateZipfData(int size, double exponent) {
 int[] array = new int[size];
 Random random = new Random();

 // 简化版 Zipf 分布生成
 for (int i = 0; i < size; i++) {
 // 使用幂律分布
 double rank = random.nextDouble();
 array[i] = (int) (size * Math.pow(rank, exponent));
 }

 return array;
}

/***
 * 统计工具类
 * 提供各种统计分析方法
 */

```

```
class StatisticsUtils {

 /**
 * 计算平均值
 */
 public static double mean(long[] values) {
 long sum = 0;
 for (long value : values) {
 sum += value;
 }
 return (double) sum / values.length;
 }

 /**
 * 计算标准差
 */
 public static double standardDeviation(long[] values) {
 double mean = mean(values);
 double sumSquaredDiff = 0;

 for (long value : values) {
 double diff = value - mean;
 sumSquaredDiff += diff * diff;
 }

 return Math.sqrt(sumSquaredDiff / values.length);
 }

 /**
 * 计算置信区间
 */
 public static double[] confidenceInterval(long[] values, double confidenceLevel) {
 double mean = mean(values);
 double stdDev = standardDeviation(values);
 double zScore = 1.96; // 95%置信水平的z值

 double margin = zScore * stdDev / Math.sqrt(values.length);

 return new double[] {mean - margin, mean + margin};
 }
}
```

=====

文件: ExtendedProblems.cpp

```
=====
```

/\*\*

\* 排序算法扩展题目 - C++版本

\* 包含 LeetCode、牛客网等平台的排序相关题目

\* 每个题目都包含多种解法和详细分析

\*

\* 题目链接汇总:

\* - 215. 数组中的第 K 个最大元素: <https://leetcode.cn/problems/kth-largest-element-in-an-array/>

\* - 75. 颜色分类: <https://leetcode.cn/problems/sort-colors/>

\* - 56. 合并区间: <https://leetcode.cn/problems/merge-intervals/>

\* - 347. 前 K 个高频元素: <https://leetcode.cn/problems/top-k-frequent-elements/>

\* - 164. 最大间距: <https://leetcode.cn/problems/maximum-gap/>

\* - ALDS1\_2\_A: Bubble Sort: [https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\\_2\\_A](https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_2_A)

\* - ALDS1\_2\_B: Selection Sort: [https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\\_2\\_B](https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_2_B)

\* - ALDS1\_2\_C: Stable Sort: [https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\\_2\\_C](https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_2_C)

\* - ALDS1\_2\_D: Shell Sort: [https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\\_2\\_D](https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_2_D)

\* - ALDS1\_5\_B: Merge Sort: [https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\\_5\\_B](https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_5_B)

\* - ALDS1\_5\_D: The Number of Inversions: [https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\\_5\\_D](https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_5_D)

\* - ALDS1\_6\_B: Partition: [https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\\_6\\_B](https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_6_B)

\* - ALDS1\_6\_C: Quick Sort: [https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\\_6\\_C](https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_6_C)

\* - ALDS1\_6\_D: Minimum Cost Sort: [https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\\_6\\_D](https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_6_D)

\* - ALDS1\_9\_A: Complete Binary Tree: [https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\\_9\\_A](https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_9_A)

\* - ALDS1\_9\_B: Maximum Heap: [https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\\_9\\_B](https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_9_B)

\* - ALDS1\_9\_C: Priority Queue: [https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\\_9\\_C](https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_9_C)

\*

\* 工程化考量:

\* - 异常处理: 对空数组、非法输入进行验证

\* - 边界条件: 处理各种边界情况

\* - 性能优化: 根据数据规模选择最优算法

\* - 内存管理: 合理使用数据结构, 避免不必要的内存占用

\* - 可读性: 清晰的命名和详细注释

\*

\* 算法选择建议:

\* - 第 K 大元素: 快速选择算法 (平均  $O(n)$ )

\* - 颜色分类: 三指针法 (荷兰国旗问题,  $O(n)$ )

\* - 合并区间: 排序+合并 ( $O(n \log n)$ )

```
* - 前 K 个高频元素：桶排序 ($O(n)$) 或最小堆 ($O(n \log k)$)
* - 最大间距：基数排序 ($O(n)$)
*/
```

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <queue>
#include <unordered_map>
#include <random>
#include <chrono>
#include <string>
#include <sstream>
#include <stdexcept>
#include <functional>

using namespace std;

class ExtendedProblems {
public:
 /**
 * 题目 1: 215. 数组中的第 K 个最大元素
 * 来源: LeetCode
 * 链接: https://leetcode.cn/problems/kth-largest-element-in-an-array/
 *
 * 题目描述:
 * 给定整数数组 nums 和整数 k, 请返回数组中第 k 个最大的元素。
 * 请注意, 你需要找的是数组排序后的第 k 个最大的元素, 而不是第 k 个不同的元素。
 *
 * 示例:
 * 输入: [3, 2, 1, 5, 6, 4], k = 2
 * 输出: 5
 *
 * 解法对比:
 * 1. 快速选择算法: 平均时间复杂度 $O(n)$, 最优解
 * 2. 最小堆: 时间复杂度 $O(n \log k)$, 适合 k 较小时
 * 3. 排序: 时间复杂度 $O(n \log n)$, 简单但效率较低
 *
 * 相关题目:
 * - ALDS1_6_B: Partition: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_6_B
 *
 * 工程化考量:
```

```

* - 输入验证：检查数组是否为空，k 是否合法
* - 随机化：快速选择使用随机基准避免最坏情况
* - 内存优化：最小堆只维护 k 个元素
*/
class KthLargestElement {
public:
 /**
 * 解法 1：快速选择算法（最优解）
 * 时间复杂度：O(n) 平均, O(n2) 最坏
 * 空间复杂度：O(1)
 *
 * 算法原理：
 * 基于快速排序的分区思想，但只处理包含目标的一侧
 * 1. 随机选择基准元素
 * 2. 进行分区操作，确定基准元素的最终位置
 * 3. 根据基准位置与目标位置的关系决定继续处理哪一侧
 *
 * 优势：
 * - 平均时间复杂度为线性，是最优解
 * - 原地操作，空间复杂度 O(1)
 *
 * 劣势：
 * - 最坏情况时间复杂度 O(n2)
 * - 不稳定排序
 *
 * 相关题目：
 * - ALDS1_6_B: Partition: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_6_B
 *
 * @param nums 输入数组
 * @param k 第 k 大的元素
 * @return 第 k 大的元素值
 * @throws invalid_argument 当输入参数不合法时
 */
 static int findKthLargestQuickSelect(vector<int>& nums, int k) {
 if (nums.empty() || k < 1 || k > nums.size()) {
 throw invalid_argument("Invalid input");
 }

 int left = 0, right = nums.size() - 1;
 k = nums.size() - k; // 转换为第 k 小的索引

 random_device rd;

```

```

mt19937 gen(rd());

while (left <= right) {
 uniform_int_distribution<> dis(left, right);
 int pivotIndex = dis(gen);
 int pivotPos = partition(nums, left, right, pivotIndex);

 if (pivotPos == k) {
 return nums[pivotPos];
 } else if (pivotPos < k) {
 left = pivotPos + 1;
 } else {
 right = pivotPos - 1;
 }
}

return -1;
}

/***
 * 解法 2: 最小堆实现
 * 时间复杂度: O(n log k)
 * 空间复杂度: O(k)
 *
 * 算法原理:
 * 使用最小堆维护前 k 个最大的元素
 * 1. 遍历数组, 将元素加入堆中
 * 2. 如果堆的大小超过 k, 移除堆顶元素 (最小的元素)
 * 3. 最后堆顶元素即为第 k 大的元素
 *
 * 优势:
 * - 时间复杂度为 O(n log k), 适合 k 较小时
 * - 使用优先队列实现, 代码简洁
 *
 * 劣势:
 * - 空间复杂度为 O(k)
 * - 需要维护堆结构
 *
 * @param nums 输入数组
 * @param k 第 k 大的元素
 * @return 第 k 大的元素值
 * @throws invalid_argument 当输入参数不合法时
 */

```

```

static int findKthLargestMinHeap(vector<int>& nums, int k) {
 if (nums.empty() || k < 1 || k > nums.size()) {
 throw invalid_argument("Invalid input");
 }

 // 最小堆
 priority_queue<int, vector<int>, greater<int>> minHeap;

 for (int num : nums) {
 minHeap.push(num);
 if (minHeap.size() > k) {
 minHeap.pop(); // 移除最小的元素
 }
 }

 return minHeap.top();
}

/***
 * 解法 3：排序后直接取
 * 时间复杂度：O(n log n)
 * 空间复杂度：O(1) 或 O(n)
 *
 * 算法原理：
 * 1. 对数组进行排序
 * 2. 返回排序后倒数第 k 个元素
 *
 * 优势：
 * - 代码简单易懂
 * - 使用标准库排序函数
 *
 * 劣势：
 * - 时间复杂度较高，为 O(n log n)
 * - 可能需要额外的空间
 *
 * @param nums 输入数组
 * @param k 第 k 大的元素
 * @return 第 k 大的元素值
 * @throws invalid_argument 当输入参数不合法时
 */
static int findKthLargestSort(vector<int>& nums, int k) {
 if (nums.empty() || k < 1 || k > nums.size()) {
 throw invalid_argument("Invalid input");
}

```

```

 }

sort(nums.begin(), nums.end());
return nums[nums.size() - k];
}

private:

static int partition(vector<int>& nums, int left, int right, int pivotIndex) {
 int pivotValue = nums[pivotIndex];
 swap(nums[pivotIndex], nums[right]);

 int storeIndex = left;
 for (int i = left; i < right; i++) {
 if (nums[i] < pivotValue) {
 swap(nums[storeIndex], nums[i]);
 storeIndex++;
 }
 }
 swap(nums[storeIndex], nums[right]);
 return storeIndex;
}

static void test() {
 cout << "==== 第 K 个最大元素测试 ===" << endl;

 vector<int> nums = {3, 2, 1, 5, 6, 4};
 int k = 2;

 cout << "数组: ";
 printVector(nums);
 cout << "k = " << k << endl;

 vector<int> nums1 = nums;
 int result1 = findKthLargestQuickSelect(nums1, k);
 cout << "快速选择结果: " << result1 << endl;

 vector<int> nums2 = nums;
 int result2 = findKthLargestMinHeap(nums2, k);
 cout << "最小堆结果: " << result2 << endl;

 vector<int> nums3 = nums;
 int result3 = findKthLargestSort(nums3, k);
 cout << "排序结果: " << result3 << endl;
}

```

```

 }

};

/***
 * 题目 2: 75. 颜色分类 (荷兰国旗问题)
 * 来源: LeetCode
 * 链接: https://leetcode.cn/problems/sort-colors/
 */
class SortColors {
public:
 static void test() {
 cout << "\n==== 颜色分类测试 ===" << endl;

 vector<int> nums = {2, 0, 2, 1, 1, 0};
 cout << "原始数组: ";
 printVector(nums);

 vector<int> nums1 = nums;
 sortColorsThreePointers(nums1);
 cout << "三指针法: ";
 printVector(nums1);

 vector<int> nums2 = nums;
 sortColorsCounting(nums2);
 cout << "计数排序: ";
 printVector(nums2);
 }
};

/***
 * 题目 3: 56. 合并区间
 * 来源: LeetCode
 * 链接: https://leetcode.cn/problems/merge-intervals/
 */
class MergeIntervals {
public:
 static void test() {
 cout << "\n==== 合并区间测试 ===" << endl;

 vector<vector<int>> intervals = {{1, 3}, {2, 6}, {8, 10}, {15, 18}};
 cout << "原始区间: ";

```

```

 print2DVector(intervals);

 vector<vector<int>> result = merge(intervals);
 cout << "合并结果: ";
 print2DVector(result);
}

};

/***
 * 题目 4: 347. 前 K 个高频元素
 * 来源: LeetCode
 * 链接: https://leetcode.cn/problems/top-k-frequent-elements/
 */
class TopKFrequentElements {
public:

 static void test() {
 cout << "\n==> 前 K 个高频元素测试 ==>" << endl;

 vector<int> nums = {1, 1, 1, 2, 2, 3};
 int k = 2;

 cout << "数组: ";
 printVector(nums);
 cout << "k = " << k << endl;

 vector<int> result1 = topKFrequentMinHeap(nums, k);
 cout << "最小堆结果: ";
 printVector(result1);

 vector<int> result2 = topKFrequentBucketSort(nums, k);
 cout << "桶排序结果: ";
 printVector(result2);
 }
};

/***
 * 题目 5: 164. 最大间距
 * 来源: LeetCode
 * 链接: https://leetcode.cn/problems/maximum-gap/
 */
class MaximumGap {
public:

```

```

static void test() {
 cout << "\n==== 最大间距测试 ===" << endl;

 vector<int> nums = {3, 6, 9, 1};
 cout << "数组: ";
 printVector(nums);

 vector<int> nums1 = nums;
 int result1 = maximumGapRadixSort(nums1);
 cout << "基数排序结果: " << result1 << endl;

 vector<int> nums2 = nums;
 int result2 = maximumGapSort(nums2);
 cout << "普通排序结果: " << result2 << endl;
}

// 辅助函数
static void printVector(const vector<int>& nums) {
 cout << "[";
 for (int i = 0; i < nums.size(); i++) {
 cout << nums[i];
 if (i < nums.size() - 1) cout << ", ";
 }
 cout << "]" << endl;
}

static void print2DVector(const vector<vector<int>>& matrix) {
 cout << "[";
 for (int i = 0; i < matrix.size(); i++) {
 cout << "[";
 for (int j = 0; j < matrix[i].size(); j++) {
 cout << matrix[i][j];
 if (j < matrix[i].size() - 1) cout << ", ";
 }
 cout << "]";
 if (i < matrix.size() - 1) cout << ", ";
 }
 cout << "]" << endl;
}

/***

```

```

* 综合测试函数
*/
static void runAllTests() {
 KthLargestElement::test();
 SortColors::test();
 MergeIntervals::test();
 TopKFrequentElements::test();
 MaximumGap::test();
}
};

// 主函数
int main() {
 try {
 ExtendedProblems::runAllTests();
 } catch (const exception& e) {
 cerr << "错误: " << e.what() << endl;
 return 1;
 }

 return 0;
}

```

文件: ExtendedProblems.java

```

/**
 * 排序算法扩展题目 - Java 版本
 * 包含 LeetCode、牛客网等平台的排序相关题目
 * 每个题目都包含多种解法和详细分析
 *
 * 题目链接汇总:
 * - 215. 数组中的第 K 个最大元素: https://leetcode.cn/problems/kth-largest-element-in-an-array/
 * - 75. 颜色分类: https://leetcode.cn/problems/sort-colors/
 * - 56. 合并区间: https://leetcode.cn/problems/merge-intervals/
 * - 347. 前 K 个高频元素: https://leetcode.cn/problems/top-k-frequent-elements/
 * - 164. 最大间距: https://leetcode.cn/problems/maximum-gap/
 * - ALDS1_2_A: Bubble Sort: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_2_A
 * - ALDS1_2_B: Selection Sort: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_2_B
 * - ALDS1_2_C: Stable Sort: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_2_C
 * - ALDS1_2_D: Shell Sort: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_2_D

```

```

* - ALDS1_5_B: Merge Sort: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_5_B
* - ALDS1_5_D: The Number of Inversions: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_5_D
* - ALDS1_6_B: Partition: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_6_B
* - ALDS1_6_C: Quick Sort: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_6_C
* - ALDS1_6_D: Minimum Cost Sort: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_6_D
* - ALDS1_9_A: Complete Binary Tree: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_9_A
* - ALDS1_9_B: Maximum Heap: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_9_B
* - ALDS1_9_C: Priority Queue: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_9_C
*
* 工程化考量:
* - 异常处理: 对空数组、非法输入进行验证
* - 边界条件: 处理各种边界情况
* - 性能优化: 根据数据规模选择最优算法
* - 内存管理: 合理使用数据结构, 避免不必要的内存占用
* - 可读性: 清晰的命名和详细注释
*
* 算法选择建议:
* - 第 K 大元素: 快速选择算法 (平均 $O(n)$)
* - 颜色分类: 三指针法 (荷兰国旗问题, $O(n)$)
* - 合并区间: 排序+合并 ($O(n \log n)$)
* - 前 K 个高频元素: 桶排序 ($O(n)$) 或最小堆 ($O(n \log k)$)
* - 最大间距: 基数排序 ($O(n)$)
*/
import java.util.*;
import java.util.concurrent.ThreadLocalRandom;

public class ExtendedProblems {

 /**
 * 题目 1: 215. 数组中的第 K 个最大元素
 * 来源: LeetCode
 * 链接: https://leetcode.cn/problems/kth-largest-element-in-an-array/
 *
 * 题目描述:
 * 给定整数数组 nums 和整数 k, 请返回数组中第 k 个最大的元素。
 * 请注意, 你需要找的是数组排序后的第 k 个最大的元素, 而不是第 k 个不同的元素。
 *
 * 示例:
 * 输入: [3, 2, 1, 5, 6, 4], k = 2

```

```
* 输出: 5
*
* 解法对比:
* 1. 快速选择算法: 平均时间复杂度 $O(n)$, 最优解
* 2. 最小堆: 时间复杂度 $O(n \log k)$, 适合 k 较小时
* 3. 排序: 时间复杂度 $O(n \log n)$, 简单但效率较低
*
```

```
* 相关题目:
* - ALDS1_6_B: Partition: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_6_B
*
```

```
* 工程化考量:
* - 输入验证: 检查数组是否为空, k 是否合法
* - 随机化: 快速选择使用随机基准避免最坏情况
* - 内存优化: 最小堆只维护 k 个元素
*/
```

```
public static class KthLargestElement {
```

```
/**
```

```
* 解法 1: 快速选择算法 (最优解)
* 时间复杂度: $O(n)$ 平均, $O(n^2)$ 最坏
* 空间复杂度: $O(1)$
*
* 算法原理:
* 基于快速排序的分区思想, 但只处理包含目标的一侧
* 1. 随机选择基准元素
* 2. 进行分区操作, 确定基准元素的最终位置
* 3. 根据基准位置与目标位置的关系决定继续处理哪一侧
*
```

```
* 优势:
```

```
* - 平均时间复杂度为线性, 是最优解
* - 原地操作, 空间复杂度 $O(1)$
*
```

```
* 劣势:
```

```
* - 最坏情况时间复杂度 $O(n^2)$
* - 不稳定排序
*
```

```
* 相关题目:
```

```
* - ALDS1_6_B: Partition: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_6_B
*
```

```
* 工程化考量:
```

```
* - 使用 ThreadLocalRandom 避免多线程竞争
```

```

* - 通过随机选择基准元素避免最坏情况
* - 原地操作节省内存
*/
public static int findKthLargestQuickSelect(int[] nums, int k) {
 // 输入验证
 if (nums == null || nums.length == 0 || k < 1 || k > nums.length) {
 throw new IllegalArgumentException("Invalid input parameters");
 }

 int left = 0, right = nums.length - 1;
 int kSmallest = nums.length - k; // 转换为第 k 小的索引

 // 循环进行分区操作直到找到目标元素
 while (left <= right) {
 // 随机选择基准元素索引
 int pivotIndex = ThreadLocalRandom.current().nextInt(left, right + 1);
 // 分区操作，返回基准元素的最终位置
 int pivotPos = partition(nums, left, right, pivotIndex);

 // 根据基准位置与目标位置的关系决定继续处理哪一侧
 if (pivotPos == kSmallest) {
 return nums[pivotPos];
 } else if (pivotPos < kSmallest) {
 left = pivotPos + 1;
 } else {
 right = pivotPos - 1;
 }
 }

 return -1;
}

/**
 * 解法 2: 最小堆实现
 * 时间复杂度: O(n log k)
 * 空间复杂度: O(k)
 *
 * 算法原理:
 * 使用最小堆维护前 k 个最大的元素
 * 1. 遍历数组，将元素加入堆中
 * 2. 如果堆的大小超过 k，移除堆顶元素（最小的元素）
 * 3. 最后堆顶元素即为第 k 大的元素
 *

```

```

* 优势:
* - 时间复杂度为 $O(n \log k)$, 适合 k 较小时
* - 空间复杂度为 $O(k)$
*
* 劣势:
* - 时间复杂度高于快速选择
* - 需要额外的空间
*
* 工程化考量:
* - 使用 PriorityQueue 实现最小堆
* - 只维护 k 个元素, 节省内存
* - 适合流式数据处理
*/
public static int findKthLargestMinHeap(int[] nums, int k) {
 // 输入验证
 if (nums == null || nums.length == 0 || k < 1 || k > nums.length) {
 throw new IllegalArgumentException("Invalid input parameters");
 }

 // 创建最小堆
 PriorityQueue<Integer> minHeap = new PriorityQueue<>();

 // 遍历数组元素
 for (int num : nums) {
 minHeap.offer(num);
 // 如果堆的大小超过 k , 移除堆顶元素
 if (minHeap.size() > k) {
 minHeap.poll(); // 移除最小的元素
 }
 }

 // 堆顶元素即为第 k 大的元素
 return minHeap.peek();
}

/**
* 解法 3: 排序后直接取
* 时间复杂度: $O(n \log n)$
* 空间复杂度: $O(1)$ 或 $O(n)$
*
* 算法原理:
* 1. 对数组进行排序
* 2. 返回排序后数组的倒数第 k 个元素

```

```
*
* 优势:
* - 简单易懂
* - 适用于所有情况
*
* 劣势:
* - 时间复杂度较高
* - 可能需要额外的空间
*
* 工程化考量:
* - 使用 Arrays.sort() 进行排序
* - 代码简单，易于理解和维护
* - 适合对时间复杂度要求不严格的场景
*/

public static int findKthLargestSort(int[] nums, int k) {
 // 输入验证
 if (nums == null || nums.length == 0 || k < 1 || k > nums.length) {
 throw new IllegalArgumentException("Invalid input parameters");
 }

 // 克隆数组避免修改原数组
 int[] sorted = nums.clone();
 // 使用系统排序算法
 Arrays.sort(sorted);
 // 返回倒数第 k 个元素
 return sorted[sorted.length - k];
}

/**
 * 分区操作
 * 将数组分为小于基准、等于基准和大于基准三部分
 *
 * @param nums 数组
 * @param left 左边界
 * @param right 右边界
 * @param pivotIndex 基准元素索引
 * @return 基准元素的最终位置
 */

private static int partition(int[] nums, int left, int right, int pivotIndex) {
 // 获取基准元素值
 int pivotValue = nums[pivotIndex];
 // 将基准元素移到末尾
 swap(nums, pivotIndex, right);
```

```

// 分区操作
int storeIndex = left;
for (int i = left; i < right; i++) {
 // 将小于基准的元素移到左侧
 if (nums[i] < pivotValue) {
 swap(nums, storeIndex, i);
 storeIndex++;
 }
}
// 将基准元素放到正确位置
swap(nums, storeIndex, right);
return storeIndex;
}

/**
 * 交换数组中两个元素
 * @param nums 数组
 * @param i 索引 1
 * @param j 索引 2
 */
private static void swap(int[] nums, int i, int j) {
 int temp = nums[i];
 nums[i] = nums[j];
 nums[j] = temp;
}

public static void test() {
 System.out.println("== 第 K 个最大元素测试 ==");

 int[] nums = {3, 2, 1, 5, 6, 4};
 int k = 2;

 System.out.println("数组: " + Arrays.toString(nums));
 System.out.println("k = " + k);

 int result1 = findKthLargestQuickSelect(nums.clone(), k);
 int result2 = findKthLargestMinHeap(nums.clone(), k);
 int result3 = findKthLargestSort(nums.clone(), k);

 System.out.println("快速选择结果: " + result1);
 System.out.println("最小堆结果: " + result2);
 System.out.println("排序结果: " + result3);
}

```

```

 }

}

/***
 * 题目 2: 75. 颜色分类 (荷兰国旗问题)
 * 来源: LeetCode
 * 链接: https://leetcode.cn/problems/sort-colors/
 *
 * 相关题目:
 * - ALDS1_2_C: Stable Sort: https://judge.u-
aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_2_C
 */
public static class SortColors {

 /**
 * 解法 1: 三指针法 (荷兰国旗问题)
 * 时间复杂度: O(n)
 * 空间复杂度: O(1)
 */
 public static void sortColorsThreePointers(int[] nums) {
 if (nums == null || nums.length <= 1) return;

 int left = 0; // 0 的右边界
 int right = nums.length - 1; // 2 的左边界
 int current = 0; // 当前指针

 while (current <= right) {
 if (nums[current] == 0) {
 swap(nums, left, current);
 left++;
 current++;
 } else if (nums[current] == 2) {
 swap(nums, current, right);
 right--;
 // current 不增加, 需要检查交换过来的元素
 } else {
 current++;
 }
 }
 }

 /**
 * 解法 2: 计数排序

```

```
* 时间复杂度: O(n)
* 空间复杂度: O(1) 因为只有 3 种颜色
*/
public static void sortColorsCounting(int[] nums) {
 if (nums == null || nums.length <= 1) return;

 int[] count = new int[3]; // 0,1,2 的计数

 // 统计每种颜色的数量
 for (int num : nums) {
 count[num]++;
 }

 // 重新填充数组
 int index = 0;
 for (int color = 0; color < 3; color++) {
 while (count[color] > 0) {
 nums[index++] = color;
 count[color]--;
 }
 }
}

private static void swap(int[] nums, int i, int j) {
 int temp = nums[i];
 nums[i] = nums[j];
 nums[j] = temp;
}

public static void test() {
 System.out.println("\n==== 颜色分类测试 ====");

 int[] nums = {2, 0, 2, 1, 1, 0};
 System.out.println("原始数组: " + Arrays.toString(nums));

 int[] nums1 = nums.clone();
 sortColorsThreePointers(nums1);
 System.out.println("三指针法: " + Arrays.toString(nums1));

 int[] nums2 = nums.clone();
 sortColorsCounting(nums2);
 System.out.println("计数排序: " + Arrays.toString(nums2));
}
```

```
}
```

```
/**
```

```
* 题目 3: 56. 合并区间
```

```
* 来源: LeetCode
```

```
* 链接: https://leetcode.cn/problems/merge-intervals/
```

```
*
```

```
* 相关题目:
```

```
* - ALDS1_6_D: Minimum Cost Sort: https://judge.u-
```

```
aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_6_D
```

```
*/
```

```
public static class MergeIntervals {
```

```
/**
```

```
* 解法: 排序+合并
```

```
* 时间复杂度: O(n log n) 主要来自排序
```

```
* 空间复杂度: O(n) 存储结果
```

```
*/
```

```
public static int[][] merge(int[][] intervals) {
```

```
 if (intervals == null || intervals.length <= 1) {
```

```
 return intervals;
```

```
}
```

```
// 按区间起点排序
```

```
Arrays.sort(intervals, (a, b) -> Integer.compare(a[0], b[0]));
```

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

```
int[] current = intervals[0];
```

```
result.add(current);
```

```
for (int[] interval : intervals) {
```

```
 int currentEnd = current[1];
```

```
 int nextStart = interval[0];
```

```
 int nextEnd = interval[1];
```

```
 if (currentEnd >= nextStart) { // 有重叠
```

```
 current[1] = Math.max(currentEnd, nextEnd); // 合并
```

```
 } else { // 无重叠
```

```
 current = interval;
```

```
 result.add(current);
```

```
}
```

```
}
```

```

 return result.toArray(new int[result.size()][]);
 }

public static void test() {
 System.out.println("\n==== 合并区间测试 ===");

 int[][] intervals = {{1, 3}, {2, 6}, {8, 10}, {15, 18}};
 System.out.println("原始区间: " + Arrays.deepToString(intervals));

 int[][] result = merge(intervals);
 System.out.println("合并结果: " + Arrays.deepToString(result));
}

}

/***
 * 题目 4: 347. 前 K 个高频元素
 * 来源: LeetCode
 * 链接: https://leetcode.cn/problems/top-k-frequent-elements/
 */
public static class TopKFrequentElements {

 /**
 * 解法 1: 最小堆法 (最优解)
 * 时间复杂度: O(n log k)
 * 空间复杂度: O(n)
 */
 public static int[] topKFrequentMinHeap(int[] nums, int k) {
 if (nums == null || nums.length == 0 || k <= 0 || k > nums.length) {
 return new int[0];
 }

 // 统计频率
 Map<Integer, Integer> frequencyMap = new HashMap<>();
 for (int num : nums) {
 frequencyMap.put(num, frequencyMap.getOrDefault(num, 0) + 1);
 }

 // 最小堆, 按频率排序
 PriorityQueue<Map.Entry<Integer, Integer>> minHeap =
 new PriorityQueue<>((a, b) -> Integer.compare(a.getValue(), b.getValue()));

 // 保持堆的大小为 k
 for (Map.Entry<Integer, Integer> entry : frequencyMap.entrySet()) {

```

```

 minHeap.offer(entry);
 if (minHeap.size() > k) {
 minHeap.poll();
 }
 }

 // 提取结果
 int[] result = new int[k];
 for (int i = k - 1; i >= 0; i--) {
 result[i] = minHeap.poll().getKey();
 }

 return result;
}

/**
 * 解法 2: 桶排序
 * 时间复杂度: O(n)
 * 空间复杂度: O(n)
 */
public static int[] topKFrequentBucketSort(int[] nums, int k) {
 if (nums == null || nums.length == 0 || k <= 0 || k > nums.length) {
 return new int[0];
 }

 // 统计频率
 Map<Integer, Integer> frequencyMap = new HashMap<>();
 for (int num : nums) {
 frequencyMap.put(num, frequencyMap.getOrDefault(num, 0) + 1);
 }

 // 创建桶, 索引是频率, 值是具有该频率的数字列表
 List<Integer>[] buckets = new List[nums.length + 1];
 for (int i = 0; i < buckets.length; i++) {
 buckets[i] = new ArrayList<>();
 }

 for (Map.Entry<Integer, Integer> entry : frequencyMap.entrySet()) {
 buckets[entry.getValue()].add(entry.getKey());
 }

 // 从后向前收集 k 个元素
 int[] result = new int[k];

```

```

 int index = 0;
 for (int i = buckets.length - 1; i > 0 && index < k; i--) {
 for (int num : buckets[i]) {
 result[index++] = num;
 if (index == k) break;
 }
 }

 return result;
 }

 public static void test() {
 System.out.println("\n==== 前 K 个高频元素测试 ===");

 int[] nums = {1, 1, 1, 2, 2, 3};
 int k = 2;

 System.out.println("数组: " + Arrays.toString(nums));
 System.out.println("k = " + k);

 int[] result1 = topKFrequentMinHeap(nums, k);
 int[] result2 = topKFrequentBucketSort(nums, k);

 System.out.println("最小堆结果: " + Arrays.toString(result1));
 System.out.println("桶排序结果: " + Arrays.toString(result2));
 }
}

/**
 * 题目 5: 164. 最大间距
 * 来源: LeetCode
 * 链接: https://leetcode.cn/problems/maximum-gap/
 *
 * 相关题目:
 * - ALDS1_5_D: The Number of Inversions: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_5_D
 */
public static class MaximumGap {

 /**
 * 解法 1: 基数排序 (最优解)
 * 时间复杂度: O(n) - 线性时间
 * 空间复杂度: O(n)

```

```

*/
public static int maximumGapRadixSort(int[] nums) {
 if (nums == null || nums.length < 2) {
 return 0;
 }

 // 基数排序
 radixSort(nums);

 // 计算最大间距
 int maxGap = 0;
 for (int i = 1; i < nums.length; i++) {
 maxGap = Math.max(maxGap, nums[i] - nums[i - 1]);
 }

 return maxGap;
}

/***
 * 解法 2: 排序后遍历
 * 时间复杂度: O(n log n)
 * 空间复杂度: O(1) 或 O(n)
 */
public static int maximumGapSort(int[] nums) {
 if (nums == null || nums.length < 2) {
 return 0;
 }

 int[] sorted = nums.clone();
 Arrays.sort(sorted);

 int maxGap = 0;
 for (int i = 1; i < sorted.length; i++) {
 maxGap = Math.max(maxGap, sorted[i] - sorted[i - 1]);
 }

 return maxGap;
}

private static void radixSort(int[] nums) {
 if (nums.length == 0) return;

 // 找出最大值

```

```
int maxVal = Arrays.stream(nums).max().getAsInt();

// 按照个位、十位、百位...进行排序
for (int exp = 1; maxVal / exp > 0; exp *= 10) {
 countingSortByDigit(nums, exp);
}

private static void countingSortByDigit(int[] nums, int exp) {
 int n = nums.length;
 int[] output = new int[n];
 int[] count = new int[10];

 // 统计每个数字出现的次数
 for (int i = 0; i < n; i++) {
 int digit = (nums[i] / exp) % 10;
 count[digit]++;
 }

 // 计算累积计数
 for (int i = 1; i < 10; i++) {
 count[i] += count[i - 1];
 }

 // 从后向前遍历，保证稳定性
 for (int i = n - 1; i >= 0; i--) {
 int digit = (nums[i] / exp) % 10;
 output[count[digit] - 1] = nums[i];
 count[digit]--;
 }

 // 复制回原数组
 System.arraycopy(output, 0, nums, 0, n);
}

public static void test() {
 System.out.println("\n==== 最大间距测试 ===");

 int[] nums = {3, 6, 9, 1};
 System.out.println("数组: " + Arrays.toString(nums));

 int result1 = maximumGapRadixSort(nums.clone());
 int result2 = maximumGapSort(nums.clone());
}
```

```

 System.out.println("基数排序结果: " + result1);
 System.out.println("普通排序结果: " + result2);
 }
}

/***
 * 综合测试函数
 */
public static void runAllTests() {
 KthLargestElement.test();
 SortColors.test();
 MergeIntervals.test();
 TopKFrequentElements.test();
 MaximumGap.test();
}

// 主函数
public static void main(String[] args) {
 try {
 runAllTests();
 } catch (Exception e) {
 System.err.println("错误: " + e.getMessage());
 e.printStackTrace();
 }
}

```

文件: ExtendedProblems.py

"""

排序算法扩展题目 - Python 版本

包含 LeetCode、牛客网等平台的排序相关题目

每个题目都包含多种解法和详细分析

题目链接汇总:

- 215. 数组中的第 K 个最大元素: <https://leetcode.cn/problems/kth-largest-element-in-an-array/>
- 75. 颜色分类: <https://leetcode.cn/problems/sort-colors/>
- 56. 合并区间: <https://leetcode.cn/problems/merge-intervals/>
- 347. 前 K 个高频元素: <https://leetcode.cn/problems/top-k-frequent-elements/>
- 164. 最大间距: <https://leetcode.cn/problems/maximum-gap/>

- ALDS1\_2\_A: Bubble Sort: [https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\\_2\\_A](https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_2_A)
- ALDS1\_2\_B: Selection Sort: [https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\\_2\\_B](https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_2_B)
- ALDS1\_2\_C: Stable Sort: [https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\\_2\\_C](https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_2_C)
- ALDS1\_2\_D: Shell Sort: [https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\\_2\\_D](https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_2_D)
- ALDS1\_5\_B: Merge Sort: [https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\\_5\\_B](https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_5_B)
- ALDS1\_5\_D: The Number of Inversions: [https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\\_5\\_D](https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_5_D)
- ALDS1\_6\_B: Partition: [https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\\_6\\_B](https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_6_B)
- ALDS1\_6\_C: Quick Sort: [https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\\_6\\_C](https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_6_C)
- ALDS1\_6\_D: Minimum Cost Sort: [https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\\_6\\_D](https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_6_D)
- ALDS1\_9\_A: Complete Binary Tree: [https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\\_9\\_A](https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_9_A)
- ALDS1\_9\_B: Maximum Heap: [https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\\_9\\_B](https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_9_B)
- ALDS1\_9\_C: Priority Queue: [https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\\_9\\_C](https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_9_C)

工程化考量:

- 异常处理: 对空数组、非法输入进行验证
- 边界条件: 处理各种边界情况
- 性能优化: 根据数据规模选择最优算法
- 内存管理: 合理使用数据结构, 避免不必要的内存占用
- 可读性: 清晰的命名和详细注释

算法选择建议:

- 第 K 大元素: 快速选择算法 (平均  $O(n)$ )
  - 颜色分类: 三指针法 (荷兰国旗问题,  $O(n)$ )
  - 合并区间: 排序+合并 ( $O(n \log n)$ )
  - 前 K 个高频元素: 桶排序 ( $O(n)$ ) 或最小堆 ( $O(n \log k)$ )
  - 最大间距: 基数排序 ( $O(n)$ )
- """

```
import heapq
import random
from typing import List, Tuple
from collections import defaultdict, Counter
```

class ExtendedProblems:

"""

扩展问题类 - 包含各种排序相关的算法题目

工程化特性:

1. 每个子类对应一个算法题目
2. 提供多种解法并分析复杂度

3. 包含详细的测试用例
4. 注重异常处理和边界条件

"""

```
class KthLargestElement:
 """
 题目 1: 215. 数组中的第 K 个最大元素
 来源: LeetCode
 链接: https://leetcode.cn/problems/kth-largest-element-in-an-array/
 """
```

题目描述:

给定整数数组  $\text{nums}$  和整数  $k$ ，请返回数组中第  $k$  个最大的元素。

请注意，你需要找的是数组排序后的第  $k$  个最大的元素，而不是第  $k$  个不同的元素。

示例:

输入: [3, 2, 1, 5, 6, 4],  $k = 2$

输出: 5

解法对比:

1. 快速选择算法: 平均时间复杂度  $O(n)$ , 最优解
2. 最小堆: 时间复杂度  $O(n \log k)$ , 适合  $k$  较小时
3. 排序: 时间复杂度  $O(n \log n)$ , 简单但效率较低

相关题目:

- ALDS1\_6\_B: Partition: [https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\\_6\\_B](https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_6_B)

工程化考量:

- 输入验证: 检查数组是否为空,  $k$  是否合法
- 随机化: 快速选择使用随机基准避免最坏情况
- 内存优化: 最小堆只维护  $k$  个元素

"""

```
@staticmethod
def find_kth_largest_quick_select(nums: List[int], k: int) -> int:
 """
```

解法 1: 快速选择算法 (最优解)

时间复杂度:  $O(n)$  平均,  $O(n^2)$  最坏

空间复杂度:  $O(1)$

算法原理:

基于快速排序的分区思想, 但只处理包含目标的一侧

1. 随机选择基准元素

2. 进行分区操作，确定基准元素的最终位置
3. 根据基准位置与目标位置的关系决定继续处理哪一侧

优势：

- 平均时间复杂度为线性，是最优解
- 原地操作，空间复杂度  $O(1)$

劣势：

- 最坏情况时间复杂度  $O(n^2)$
- 不稳定排序

相关题目：

- ALDS1\_6\_B: Partition: [https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\\_6\\_B](https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_6_B)

Args:

- nums: 输入数组
- k: 第 k 大的元素

Returns:

第 k 大的元素值

Raises:

ValueError: 当输入参数不合法时

工程化考量：

- 使用 `random.randint` 避免最坏情况
- 原地操作节省内存
- 适合处理大数据集

"""

# 输入验证

```
if not nums or k < 1 or k > len(nums):
 raise ValueError("Invalid input")
```

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

k\_smallest = len(nums) - k # 转换为第 k 小的索引

# 循环进行分区操作直到找到目标元素

```
while left <= right:
```

# 随机选择基准元素索引

```
pivot_index = random.randint(left, right)
```

# 分区操作，返回基准元素的最终位置

```
pivot_pos = ExtendedProblems.KthLargestElement._partition(nums, left, right,
```

```
pivot_index)

 # 根据基准位置与目标位置的关系决定继续处理哪一侧
 if pivot_pos == k_smallest:
 return nums[pivot_pos]
 elif pivot_pos < k_smallest:
 left = pivot_pos + 1
 else:
 right = pivot_pos - 1

 return -1
```

@staticmethod

```
def find_kth_largest_min_heap(nums: List[int], k: int) -> int:
 """
```

解法 2: 最小堆实现

时间复杂度:  $O(n \log k)$

空间复杂度:  $O(k)$

算法原理:

使用最小堆维护前  $k$  个最大的元素

1. 遍历数组, 将元素加入堆中
2. 如果堆的大小超过  $k$ , 移除堆顶元素 (最小的元素)
3. 最后堆顶元素即为第  $k$  大的元素

优势:

- 时间复杂度为  $O(n \log k)$ , 适合  $k$  较小时
- 空间复杂度为  $O(k)$

劣势:

- 时间复杂度高于快速选择
- 需要额外的空间

工程化考量:

- 使用 `heapq` 实现最小堆
  - 只维护  $k$  个元素, 节省内存
  - 适合流式数据处理
- ```
"""
```

输入验证

```
if not nums or k < 1 or k > len(nums):
    raise ValueError("Invalid input")
```

创建最小堆

```

min_heap = []

# 遍历数组元素
for num in nums:
    heapq.heappush(min_heap, num)
    # 如果堆的大小超过 k, 移除堆顶元素
    if len(min_heap) > k:
        heapq.heappop(min_heap) # 移除最小的元素

# 堆顶元素即为第 k 大的元素
return min_heap[0]

```

```

@staticmethod
def find_kth_largest_sort(nums: List[int], k: int) -> int:
    """

```

解法 3: 排序后直接取

时间复杂度: $O(n \log n)$

空间复杂度: $O(1)$ 或 $O(n)$

算法原理:

1. 对数组进行排序
2. 返回排序后倒数第 k 个元素

优势:

- 简单易懂
- 适用于所有情况

劣势:

- 时间复杂度较高
- 可能需要额外的空间

工程化考量:

- 使用内置 sorted 函数进行排序
- 代码简单, 易于理解和维护
- 适合对时间复杂度要求不严格的场景

输入验证

```

if not nums or k < 1 or k > len(nums):
    raise ValueError("Invalid input")

```

使用内置排序函数

```

nums_sorted = sorted(nums)

```

返回倒数第 k 个元素

```
        return nums_sorted[-k]

@staticmethod
def _partition(nums: List[int], left: int, right: int, pivot_index: int) -> int:
    """
分区操作
将数组分为小于基准、等于基准和大于基准三部分

Args:
    nums: 数组
    left: 左边界
    right: 右边界
    pivot_index: 基准元素索引

Returns:
    基准元素的最终位置
    """

# 获取基准元素值
pivot_value = nums[pivot_index]
# 将基准元素移到末尾
nums[pivot_index], nums[right] = nums[right], nums[pivot_index]

# 分区操作
store_index = left
for i in range(left, right):
    # 将小于基准的元素移到左侧
    if nums[i] < pivot_value:
        nums[store_index], nums[i] = nums[i], nums[store_index]
        store_index += 1

# 将基准元素放到正确位置
nums[store_index], nums[right] = nums[right], nums[store_index]
return store_index

@staticmethod
def test():
    """测试函数"""
    print("== 第 K 个最大元素测试 ==")

    nums = [3, 2, 1, 5, 6, 4]
    k = 2

    print(f"数组: {nums}")
```

```

print(f"k = {k}")

result1 =
ExtendedProblems.KthLargestElement.find_kth_largest_quick_select(nums.copy(), k)
result2 = ExtendedProblems.KthLargestElement.find_kth_largest_min_heap(nums.copy(), k)
result3 = ExtendedProblems.KthLargestElement.find_kth_largest_sort(nums.copy(), k)

print(f"快速选择结果: {result1}")
print(f"最小堆结果: {result2}")
print(f"排序结果: {result3}")

```

class SortColors:

"""

题目 2: 75. 颜色分类 (荷兰国旗问题)

来源: LeetCode

链接: <https://leetcode.cn/problems/sort-colors/>

相关题目:

- ALDS1_2_C: Stable Sort: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_2_C

"""

@staticmethod

```
def sort_colors_three_pointers(nums: List[int]) -> None:
    """
```

解法 1: 三指针法 (荷兰国旗问题)

时间复杂度: O(n)

空间复杂度: O(1)

"""

```
if len(nums) <= 1:
```

```
    return
```

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

```
while current <= right:
```

```
    if nums[current] == 0:
```

```
        nums[left], nums[current] = nums[current], nums[left]
```

```
        left += 1
```

```
        current += 1
```

```
    elif nums[current] == 2:
```

```
        nums[current], nums[right] = nums[right], nums[current]
```

```
        right -= 1
```

```

        else:
            current += 1

@staticmethod
def sort_colors_counting(nums: List[int]) -> None:
    """
    解法 2: 计数排序
    时间复杂度: O(n)
    空间复杂度: O(1) 因为只有 3 种颜色
    """

    if len(nums) <= 1:
        return

    count = [0, 0, 0] # 0, 1, 2 的计数

    # 统计每种颜色的数量
    for num in nums:
        count[num] += 1

    # 重新填充数组
    index = 0
    for color in range(3):
        for _ in range(count[color]):
            nums[index] = color
            index += 1

@staticmethod
def test():
    """测试函数"""
    print("\n== 颜色分类测试 ==")

    nums = [2, 0, 2, 1, 1, 0]
    print(f"原始数组: {nums}")

    nums1 = nums.copy()
    ExtendedProblems.SortColors.sort_colors_three_pointers(nums1)
    print(f"三指针法: {nums1}")

    nums2 = nums.copy()
    ExtendedProblems.SortColors.sort_colors_counting(nums2)
    print(f"计数排序: {nums2}")

class MergeIntervals:

```

```
"""
```

题目 3: 56. 合并区间

来源: LeetCode

链接: <https://leetcode.cn/problems/merge-intervals/>

相关题目:

- ALDS1_6_D: Minimum Cost Sort: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_6_D

```
@staticmethod
```

```
def merge(intervals: List[List[int]]) -> List[List[int]]:
```

```
"""
```

解法: 排序+合并

时间复杂度: $O(n \log n)$ 主要来自排序

空间复杂度: $O(n)$ 存储结果

```
"""
```

```
if len(intervals) <= 1:  
    return intervals
```

按区间起点排序

```
intervals.sort(key=lambda x: x[0])
```

```
result = []
```

```
current = intervals[0]
```

```
result.append(current)
```

```
for interval in intervals:
```

```
    current_end = current[1]
```

```
    next_start, next_end = interval[0], interval[1]
```

```
    if current_end >= next_start: # 有重叠
```

```
        current[1] = max(current_end, next_end) # 合并
```

```
        result[-1] = current
```

```
    else: # 无重叠
```

```
        current = interval
```

```
        result.append(current)
```

```
return result
```

```
@staticmethod
```

```
def test():
```

```
    """测试函数"""
```

```

print("\n==== 合并区间测试 ===")

intervals = [[1, 3], [2, 6], [8, 10], [15, 18]]
print(f"原始区间: {intervals}")

result = ExtendedProblems.MergeIntervals.merge(intervals)
print(f"合并结果: {result}")

```

class TopKFrequentElements:

"""

题目 4: 347. 前 K 个高频元素

来源: LeetCode

链接: <https://leetcode.cn/problems/top-k-frequent-elements/>

"""

@staticmethod

def top_k_frequent_min_heap(nums: List[int], k: int) -> List[int]:

"""

解法 1: 最小堆法 (最优解)

时间复杂度: $O(n \log k)$

空间复杂度: $O(n)$

"""

if not nums or k <= 0 or k > len(nums):

return []

统计频率

frequency_map = Counter(nums)

最小堆, 按频率排序

min_heap = []

保持堆的大小为 k

for num, freq in frequency_map.items():

heapq.heappush(min_heap, (freq, num))

if len(min_heap) > k:

heapq.heappop(min_heap)

提取结果

result = []

while min_heap:

result.append(heapq.heappop(min_heap)[1])

return result[::-1] # 反转得到频率从高到低

```

@staticmethod
def top_k_frequent_bucket_sort(nums: List[int], k: int) -> List[int]:
    """
    解法 2: 桶排序
    时间复杂度: O(n)
    空间复杂度: O(n)
    """
    if not nums or k <= 0 or k > len(nums):
        return []
    # 统计频率
    frequency_map = Counter(nums)

    # 创建桶，索引是频率，值是具有该频率的数字列表
    buckets = [[] for _ in range(len(nums) + 1)]
    for num, freq in frequency_map.items():
        buckets[freq].append(num)

    # 从后向前收集 k 个元素
    result = []
    for i in range(len(buckets) - 1, 0, -1):
        for num in buckets[i]:
            result.append(num)
            if len(result) == k:
                return result
    return result

```

```

@staticmethod
def test():
    """测试函数"""
    print("\n==== 前 K 个高频元素测试 ====")

    nums = [1, 1, 1, 2, 2, 3]
    k = 2

    print(f"数组: {nums}")
    print(f"k = {k}")

    result1 = ExtendedProblems.TopKFrequentElements.top_k_frequent_min_heap(nums, k)
    result2 = ExtendedProblems.TopKFrequentElements.top_k_frequent_bucket_sort(nums, k)

```

```
print(f"最小堆结果: {result1}")
print(f"桶排序结果: {result2}")

class MaximumGap:
    """
    题目 5: 164. 最大间距
    来源: LeetCode
    链接: https://leetcode.cn/problems/maximum-gap/
```

相关题目:

- ALDS1_5_D: The Number of Inversions: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_5_D

@staticmethod

```
def maximum_gap_radix_sort(nums: List[int]) -> int:
```

"""
解法 1: 基数排序 (最优解)
时间复杂度: O(n) - 线性时间
空间复杂度: O(n)
"""

if len(nums) < 2:
 return 0

基数排序
ExtendedProblems.MaximumGap._radix_sort(nums)

计算最大间距
max_gap = 0
for i in range(1, len(nums)):
 max_gap = max(max_gap, nums[i] - nums[i - 1])

return max_gap

@staticmethod

```
def maximum_gap_sort(nums: List[int]) -> int:
```

"""
解法 2: 排序后遍历
时间复杂度: O(n log n)
空间复杂度: O(1) 或 O(n)
"""

if len(nums) < 2:
 return 0

```
nums_sorted = sorted(nums)

max_gap = 0
for i in range(1, len(nums_sorted)):
    max_gap = max(max_gap, nums_sorted[i] - nums_sorted[i - 1])

return max_gap

@staticmethod
def _radix_sort(nums: List[int]) -> None:
    """基数排序实现"""
    if not nums:
        return

    # 找出最大值
    max_val = max(nums)

    # 按照个位、十位、百位...进行排序
    exp = 1
    while max_val // exp > 0:
        ExtendedProblems.MaximumGap._counting_sort_by_digit(nums, exp)
        exp *= 10

@staticmethod
def _counting_sort_by_digit(nums: List[int], exp: int) -> None:
    """按指定位数进行计数排序"""
    n = len(nums)
    output = [0] * n
    count = [0] * 10

    # 统计每个数字出现的次数
    for i in range(n):
        index = (nums[i] // exp) % 10
        count[index] += 1

    # 计算累积计数
    for i in range(1, 10):
        count[i] += count[i - 1]

    # 从后向前遍历，保证稳定性
    for i in range(n - 1, -1, -1):
        index = (nums[i] // exp) % 10
        output[count[index]] = nums[i]
        count[index] -= 1
```

```

        output[count[index] - 1] = nums[i]
        count[index] -= 1

    # 复制回原数组
    for i in range(n):
        nums[i] = output[i]

@staticmethod
def test():
    """测试函数"""
    print("\n==== 最大间距测试 ====")

    nums = [3, 6, 9, 1]
    print(f"数组: {nums}")

    nums1 = nums.copy()
    result1 = ExtendedProblems.MaximumGap.maximum_gap_radix_sort(nums1)
    print(f"基数排序结果: {result1}")

    result2 = ExtendedProblems.MaximumGap.maximum_gap_sort(nums)
    print(f"普通排序结果: {result2}")

@staticmethod
def run_all_tests():
    """运行所有测试"""
    ExtendedProblems.KthLargestElement.test()
    ExtendedProblems.SortColors.test()
    ExtendedProblems.MergeIntervals.test()
    ExtendedProblems.TopKFrequentElements.test()
    ExtendedProblems.MaximumGap.test()

if __name__ == "__main__":
    ExtendedProblems.run_all_tests()
=====
```

文件: ExtendedSortProblems.cpp

=====

```
/***
 * 排序算法扩展题目 - C++版本
 * 包含更多 LeetCode、牛客网、剑指 Offer 等平台的排序相关题目
 * 每个题目都包含多种解法和详细分析
```

```
*  
* 时间复杂度分析：详细分析每种解法的时间复杂度  
* 空间复杂度分析：分析内存使用情况  
* 最优解判断：确定是否为最优解，如果不是则寻找最优解  
*/
```

```
#include <iostream>  
#include <vector>  
#include <algorithm>  
#include <queue>  
#include <functional>  
#include <string>  
#include <sstream>  
#include <random>  
#include <chrono>  
#include <map>  
#include <unordered_map>  
#include <cmath>  
#include <limits>  
#include <cassert>
```

```
using namespace std;
```

```
class ExtendedSortProblems {  
public:  
    /**  
     * 题目 1: 88. 合并两个有序数组  
     * 来源: LeetCode  
     * 链接: https://leetcode.cn/problems/merge-sorted-array/  
     * 难度: 简单  
     *  
     * 时间复杂度: O(m + n)  
     * 空间复杂度: O(1)  
     * 是否最优解: 是  
     */  
    static void mergeSortedArrays(vector<int>& nums1, int m, vector<int>& nums2, int n) {  
        if (m < 0 || n < 0) {  
            throw invalid_argument("Invalid input parameters");  
        }  
  
        int p1 = m - 1; // nums1 有效部分的末尾  
        int p2 = n - 1; // nums2 的末尾  
        int p = m + n - 1; // 合并后的末尾
```

```

// 从后向前合并，避免覆盖 nums1 中的元素
while (p1 >= 0 && p2 >= 0) {
    if (nums1[p1] > nums2[p2]) {
        nums1[p] = nums1[p1];
        p1--;
    } else {
        nums1[p] = nums2[p2];
        p2--;
    }
    p--;
}

// 如果 nums2 还有剩余元素，直接复制到 nums1 前面
while (p2 >= 0) {
    nums1[p] = nums2[p2];
    p2--;
    p--;
}
}

/***
 * 题目 2: 973. 最接近原点的 K 个点
 * 来源: LeetCode
 * 链接: https://leetcode.cn/problems/k-closest-points-to-origin/
 * 难度: 中等
 *
 * 时间复杂度: O(n) 平均
 * 空间复杂度: O(1)
 * 是否最优解: 是
 */
static vector<vector<int>> kClosest(vector<vector<int>>& points, int k) {
    if (points.empty() || k <= 0 || k > points.size()) {
        throw invalid_argument("Invalid input parameters");
    }

    // 使用快速选择算法找到第 k 小的距离
    quickSelect(points, 0, points.size() - 1, k);

    // 返回前 k 个点
    return vector<vector<int>>(points.begin(), points.begin() + k);
}

```

```

private:

    static void quickSelect(vector<vector<int>>& points, int left, int right, int k) {
        if (left >= right) return;

        // 随机选择 pivot
        random_device rd;
        mt19937 gen(rd());
        uniform_int_distribution<int> dis(left, right);
        int pivotIndex = dis(gen);
        int pivotDist = distance(points[pivotIndex]);

        // 分区操作
        int i = left;
        for (int j = left; j <= right; j++) {
            if (distance(points[j]) <= pivotDist) {
                swap(points[i], points[j]);
                i++;
            }
        }

        // 根据分区结果决定下一步
        if (i == k) {
            return;
        } else if (i < k) {
            quickSelect(points, i, right, k);
        } else {
            quickSelect(points, left, i - 1, k);
        }
    }

    static int distance(const vector<int>& point) {
        return point[0] * point[0] + point[1] * point[1];
    }
}

public:
    /**
     * 题目 3: 1054. 距离相等的条形码
     * 来源: LeetCode
     * 链接: https://leetcode.cn/problems/distant-barcodes/
     * 难度: 中等
     *
     * 时间复杂度: O(n log k) - k 为不同条形码的数量
     * 空间复杂度: O(n)
    */

```

```

* 是否最优解: 是
*/
static vector<int> rearrangeBarcodes(vector<int>& barcodes) {
    if (barcodes.empty()) return {};

    // 统计频率
    unordered_map<int, int> freqMap;
    for (int code : barcodes) {
        freqMap[code]++;
    }

    // 最大堆, 按频率排序
    auto comp = [] (const pair<int, int>& a, const pair<int, int>& b) {
        return a.second < b.second;
    };
    priority_queue<pair<int, int>, vector<pair<int, int>>, decltype(comp)> maxHeap(comp);

    for (const auto& entry : freqMap) {
        maxHeap.push(entry);
    }

    vector<int> result(barcodes.size());
    int index = 0;

    // 间隔填充, 先填偶数位置, 再填奇数位置
    while (!maxHeap.empty()) {
        auto current = maxHeap.top();
        maxHeap.pop();
        int code = current.first;
        int freq = current.second;

        // 填充所有当前条形码
        for (int i = 0; i < freq; i++) {
            if (index >= result.size()) {
                index = 1; // 切换到奇数位置
            }
            result[index] = code;
            index += 2;
        }
    }

    return result;
}

```

```
/**  
 * 题目 4: 324. 摆动排序 II  
 * 来源: LeetCode  
 * 链接: https://leetcode.cn/problems/wiggle-sort-ii/  
 * 难度: 中等  
 *  
 * 时间复杂度: O(n log n)  
 * 空间复杂度: O(n)  
 * 是否最优解: 是 (对于通用情况)  
 */  
  
static void wiggleSort(vector<int>& nums) {  
    if (nums.size() <= 1) return;  
  
    // 复制并排序数组  
    vector<int> sorted = nums;  
    sort(sorted.begin(), sorted.end());  
  
    int n = nums.size();  
    int mid = (n + 1) / 2; // 中间位置 (向上取整)  
  
    // 双指针填充: 左半部分从大到小, 右半部分从大到小  
    int left = mid - 1;  
    int right = n - 1;  
    int index = 0;  
  
    while (index < n) {  
        if (index % 2 == 0) {  
            // 偶数位置: 取左半部分 (较小的数)  
            nums[index] = sorted[left];  
            left--;  
        } else {  
            // 奇数位置: 取右半部分 (较大的数)  
            nums[index] = sorted[right];  
            right--;  
        }  
        index++;  
    }  
}  
  
/**  
 * 题目 5: 280. 摆动排序  
 * 来源: LeetCode
```

```

* 链接: https://leetcode.cn/problems/wiggle-sort/
* 难度: 中等
*
* 时间复杂度: O(n)
* 空间复杂度: O(1)
* 是否最优解: 是
*/
static void wiggleSort280(vector<int>& nums) {
    if (nums.size() <= 1) return;

    // 一次遍历, 根据需要交换相邻元素
    for (int i = 0; i < nums.size() - 1; i++) {
        if ((i % 2 == 0 && nums[i] > nums[i + 1]) ||
            (i % 2 == 1 && nums[i] < nums[i + 1])) {
            // 交换相邻元素
            swap(nums[i], nums[i + 1]);
        }
    }
}

/***
* 题目 6: 493. 翻转对
* 来源: LeetCode
* 链接: https://leetcode.cn/problems/reverse-pairs/
* 难度: 困难
*
* 时间复杂度: O(n log n)
* 空间复杂度: O(n)
* 是否最优解: 是
*/
static int reversePairs493(vector<int>& nums) {
    if (nums.size() <= 1) return 0;

    vector<int> temp(nums.size());
    return mergeSortCountPairs(nums, 0, nums.size() - 1, temp);
}

private:
    static int mergeSortCountPairs(vector<int>& nums, int left, int right, vector<int>& temp) {
        if (left >= right) return 0;

        int mid = left + (right - left) / 2;
        int count = 0;

```

```

        count += mergeSortCountPairs(nums, left, mid, temp);
        count += mergeSortCountPairs(nums, mid + 1, right, temp);
        count += countPairs(nums, left, mid, right);
        merge(nums, left, mid, right, temp);

        return count;
    }

static int countPairs(vector<int>& nums, int left, int mid, int right) {
    int count = 0;
    int j = mid + 1;

    // 统计满足 nums[i] > 2 * nums[j] 的对数
    for (int i = left; i <= mid; i++) {
        while (j <= right && (long long)nums[i] > 2LL * nums[j]) {
            j++;
        }
        count += (j - (mid + 1));
    }

    return count;
}

static void merge(vector<int>& nums, int left, int mid, int right, vector<int>& temp) {
    for (int i = left; i <= right; i++) {
        temp[i] = nums[i];
    }

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

    while (i <= mid) {
        nums[k++] = temp[i++];
    }

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

```

```
    }
}

public:
    /**
     * 题目 7: 剑指 Offer 45. 把数组排成最小的数
     * 来源: 剑指 Offer
     * 难度: 中等
     *
     * 时间复杂度: O(n log n)
     * 空间复杂度: O(n)
     * 是否最优解: 是
     */
static string minNumber(vector<int>& nums) {
    if (nums.empty()) return "";

    // 将数字转换为字符串
    vector<string> strNums;
    for (int num : nums) {
        strNums.push_back(to_string(num));
    }

    // 自定义排序: 比较 s1+s2 和 s2+s1
    sort(strNums.begin(), strNums.end(), [](const string& s1, const string& s2) {
        return s1 + s2 < s2 + s1;
    });

    // 拼接结果
    string result;
    for (const string& str : strNums) {
        result += str;
    }

    return result;
}

    /**
     * 题目 8: HackerRank - Counting Inversions
     * 来源: HackerRank
     * 链接: https://www.hackerrank.com/challenges/ctci-merge-sort
     * 难度: 困难
     *
     * 时间复杂度: O(n log n)
```

```

* 空间复杂度: O(n)
* 是否最优解: 是
*/
static long countInversions(vector<int>& arr) {
    if (arr.size() <= 1) return 0;

    vector<int> temp(arr.size());
    return mergeSortCountInversions(arr, 0, arr.size() - 1, temp);
}

private:
    static long mergeSortCountInversions(vector<int>& arr, int left, int right, vector<int>& temp) {
        if (left >= right) return 0;

        int mid = left + (right - left) / 2;
        long count = 0;

        count += mergeSortCountInversions(arr, left, mid, temp);
        count += mergeSortCountInversions(arr, mid + 1, right, temp);
        count += mergeAndCount(arr, left, mid, right, temp);

        return count;
    }

    static long mergeAndCount(vector<int>& arr, int left, int mid, int right, vector<int>& temp) {
        for (int i = left; i <= right; i++) {
            temp[i] = arr[i];
        }

        int i = left, j = mid + 1, k = left;
        long count = 0;

        while (i <= mid && j <= right) {
            if (temp[i] <= temp[j]) {
                arr[k++] = temp[i++];
            } else {
                arr[k++] = temp[j++];
                count += (mid - i + 1); // 统计逆序对
            }
        }
    }
}

```

```

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

    return count;
}

public:
/***
 * 题目 9: 牛客网 NC140 排序
 * 来源: 牛客网
 * 链接: https://www.nowcoder.com/practice/2baf799ea0594abd974d37139de27896
 * 难度: 简单
 *
 * 时间复杂度: 根据算法选择
 * 空间复杂度: 根据算法选择
 */
static vector<int> sortArrayNC140(vector<int>& arr) {
    if (arr.size() <= 1) return arr;

    // 根据数据规模选择算法
    if (arr.size() < 50) {
        // 小数组使用插入排序
        insertionSort(arr);
    } else {
        // 中等以上使用快速排序
        quickSort(arr, 0, arr.size() - 1);
    }

    return arr;
}

private:
static void insertionSort(vector<int>& arr) {
    for (int i = 1; i < arr.size(); i++) {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
    }
}

```

```

        }
        arr[j + 1] = key;
    }
}

static void quickSort(vector<int>& arr, int low, int high) {
    if (low < high) {
        int pivotIndex = partition(arr, low, high);
        quickSort(arr, low, pivotIndex - 1);
        quickSort(arr, pivotIndex + 1, high);
    }
}

static int partition(vector<int>& arr, int low, int high) {
    int pivot = arr[high];
    int i = low - 1;

    for (int j = low; j < high; j++) {
        if (arr[j] <= pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }

    swap(arr[i + 1], arr[high]);
    return i + 1;
}

/***
 * 题目 10: 外部排序模拟 - 多路归并
 * 来源: 算法导论
 * 难度: 困难
 *
 * 时间复杂度: O(n log k) - k 为归并路数
 * 空间复杂度: O(k) - 缓冲区大小
 */
class ExternalSort {
public:
    /**
     * 模拟多路归并排序
     * @param chunks 多个有序数据块
     * @return 合并后的有序数组
     */
}

```

```

static vector<int> multiwayMerge(vector<vector<int>>& chunks) {
    if (chunks.empty()) return {};

    // 使用优先队列进行多路归并
    auto comp = [] (const Element& a, const Element& b) {
        return a.value > b.value;
    };
    priority_queue<Element, vector<Element>, decltype(comp)> minHeap(comp);

    // 初始化每个数据块的指针
    vector<int> pointers(chunks.size(), 0);
    int totalSize = 0;

    // 将每个数据块的第一个元素加入堆
    for (int i = 0; i < chunks.size(); i++) {
        totalSize += chunks[i].size();
        if (!chunks[i].empty()) {
            minHeap.push(Element(chunks[i][0], i, 0));
            pointers[i] = 1;
        }
    }

    vector<int> result;
    result.reserve(totalSize);

    // 多路归并
    while (!minHeap.empty()) {
        Element minElem = minHeap.top();
        minHeap.pop();
        result.push_back(minElem.value);

        int chunkIndex = minElem.chunkIndex;
        int elementIndex = minElem.elementIndex + 1;

        if (elementIndex < chunks[chunkIndex].size()) {
            minHeap.push(Element(chunks[chunkIndex][elementIndex], chunkIndex,
elementIndex));
        }
    }

    return result;
}

```

```

struct Element {
    int value;
    int chunkIndex;
    int elementIndex;

    Element(int v, int c, int e) : value(v), chunkIndex(c), elementIndex(e) {}

};

public:
/***
 * 测试所有扩展题目
 */
static void testAllProblems() {
    cout << "==== 扩展排序题目测试开始 ===" << endl;

    // 测试题目 1: 合并两个有序数组
    cout << "\n 题目 1: 合并两个有序数组" << endl;
    vector<int> nums1 = {1, 2, 3, 0, 0, 0};
    vector<int> nums2 = {2, 5, 6};
    mergeSortedArrays(nums1, 3, nums2, 3);
    cout << "合并结果: ";
    for (int num : nums1) cout << num << " ";
    cout << endl;

    // 测试题目 2: 最接近原点的 K 个点
    cout << "\n 题目 2: 最接近原点的 K 个点" << endl;
    vector<vector<int>> points = {{1, 3}, {-2, 2}, {5, 8}, {0, 1}};
    vector<vector<int>> result2 = kClosest(points, 2);
    cout << "最接近的 2 个点: ";
    for (auto& point : result2) {
        cout << "[" << point[0] << ", " << point[1] << "] ";
    }
    cout << endl;

    // 测试题目 3: 距离相等的条形码
    cout << "\n 题目 3: 距离相等的条形码" << endl;
    vector<int> barcodes = {1, 1, 1, 2, 2, 2};
    vector<int> result3 = rearrangeBarcodes(barcodes);
    cout << "重新排列结果: ";
    for (int code : result3) cout << code << " ";
    cout << endl;
}

```

```
// 测试题目 4: 摆动排序 II
cout << "\n 题目 4: 摆动排序 II" << endl;
vector<int> nums4 = {1, 5, 1, 1, 6, 4};
wiggleSort(nums4);
cout << "摆动排序结果: ";
for (int num : nums4) cout << num << " ";
cout << endl;

// 测试题目 5: 摆动排序
cout << "\n 题目 5: 摆动排序" << endl;
vector<int> nums5 = {3, 5, 2, 1, 6, 4};
wiggleSort280(nums5);
cout << "摆动排序结果: ";
for (int num : nums5) cout << num << " ";
cout << endl;

// 测试题目 6: 翻转对
cout << "\n 题目 6: 翻转对" << endl;
vector<int> nums6 = {1, 3, 2, 3, 1};
int count6 = reversePairs493(nums6);
cout << "翻转对数量: " << count6 << endl;

// 测试题目 7: 把数组排成最小的数
cout << "\n 题目 7: 把数组排成最小的数" << endl;
vector<int> nums7 = {10, 2};
string result7 = minNumber(nums7);
cout << "最小数字: " << result7 << endl;

// 测试题目 8: 逆序对计数
cout << "\n 题目 8: 逆序对计数" << endl;
vector<int> nums8 = {2, 4, 1, 3, 5};
long count8 = countInversions(nums8);
cout << "逆序对数量: " << count8 << endl;

// 测试题目 9: 牛客网排序
cout << "\n 题目 9: 牛客网排序" << endl;
vector<int> nums9 = {3, 1, 4, 1, 5, 9, 2, 6};
vector<int> result9 = sortArrayNC140(nums9);
cout << "排序结果: ";
for (int num : result9) cout << num << " ";
cout << endl;

// 测试题目 10: 外部排序
```

```

cout << "\n 题目 10: 外部排序模拟" << endl;
vector<vector<int>> chunks = {
    {1, 3, 5},
    {2, 4, 6},
    {0, 7, 8}
};

vector<int> result10 = ExternalSort::multiwayMerge(chunks);
cout << "多路归并结果: ";
for (int num : result10) cout << num << " ";
cout << endl;

cout << "\n==== 扩展排序题目测试结束 ===" << endl;
}

};

int main() {
    try {
        ExtendedSortProblems::testAllProblems();
    } catch (const exception& e) {
        cerr << "测试过程中出现错误: " << e.what() << endl;
    }

    return 0;
}

```

=====

文件: ExtendedSortProblems.java

=====

```

/**
 * 排序算法扩展题目 - Java 版本
 * 包含更多 LeetCode、牛客网、剑指 Offer 等平台的排序相关题目
 * 每个题目都包含多种解法和详细分析
 *
 * 时间复杂度分析: 详细分析每种解法的时间复杂度
 * 空间复杂度分析: 分析内存使用情况
 * 最优解判断: 确定是否为最优解, 如果不是则寻找最优解
 *
 * 题目链接汇总:
 * - 88. 合并两个有序数组: https://leetcode.cn/problems/merge-sorted-array/
 * - 973. 最接近原点的 K 个点: https://leetcode.cn/problems/k-closest-points-to-origin/
 * - 1054. 距离相等的条形码: https://leetcode.cn/problems/distant-barcodes/
 * - 324. 摆动排序 II: https://leetcode.cn/problems/wiggle-sort-ii/

```

```
* - 280. 摆动排序: https://leetcode.cn/problems/wiggle-sort/
* - 493. 翻转对: https://leetcode.cn/problems/reverse-pairs/
* - 剑指 Offer 45. 把数组排成最小的数
* - HackerRank - Counting Inversions: https://www.hackerrank.com/challenges/ctci-merge-sort
* - 牛客网 NC140 排序: https://www.nowcoder.com/practice/2baf799ea0594abd974d37139de27896
*/
```

```
import java.util.*;
import java.util.concurrent.ThreadLocalRandom;

public class ExtendedSortProblems {

    /**
     * 题目 1: 88. 合并两个有序数组
     * 来源: LeetCode
     * 链接: https://leetcode.cn/problems/merge-sorted-array/
     * 难度: 简单
     *
     * 题目描述:
     * 给你两个按非递减顺序排列的整数数组 nums1 和 nums2，另有两个整数 m 和 n，
     * 分别表示 nums1 和 nums2 中的元素数目。请你合并 nums2 到 nums1 中，使合并后的数组同样按非递
     * 减顺序排列。
     *
     * 示例:
     * 输入: nums1 = [1, 2, 3, 0, 0, 0], m = 3, nums2 = [2, 5, 6], n = 3
     * 输出: [1, 2, 2, 3, 5, 6]
     *
     * 解法分析:
     * 方法 1: 双指针从后向前合并 (最优解)
     * 方法 2: 先合并后排序
     *
     * 时间复杂度: O(m + n) - 方法 1
     * 空间复杂度: O(1) - 方法 1
     * 是否最优解: 是
     */

    public static void mergeSortedArrays(int[] nums1, int m, int[] nums2, int n) {
        if (nums1 == null || nums2 == null || m < 0 || n < 0) {
            throw new IllegalArgumentException("Invalid input parameters");
        }

        int p1 = m - 1; // nums1 有效部分的末尾
        int p2 = n - 1; // nums2 的末尾
        int p = m + n - 1; // 合并后的末尾
```

```

// 从后向前合并，避免覆盖 nums1 中的元素
while (p1 >= 0 && p2 >= 0) {
    if (nums1[p1] > nums2[p2]) {
        nums1[p] = nums1[p1];
        p1--;
    } else {
        nums1[p] = nums2[p2];
        p2--;
    }
    p--;
}
}

// 如果 nums2 还有剩余元素，直接复制到 nums1 前面
while (p2 >= 0) {
    nums1[p] = nums2[p2];
    p2--;
    p--;
}
}

```

```

/**
 * 题目 2: 973. 最接近原点的 K 个点
 * 来源: LeetCode
 * 链接: https://leetcode.cn/problems/k-closest-points-to-origin/
 * 难度: 中等
 *
 * 题目描述:
 * 给定一个数组 points，其中 points[i] = [xi, yi] 表示 X-Y 平面上的一个点，
 * 返回离原点 (0, 0) 最近的 k 个点。
 *
 * 示例:
 * 输入: points = [[1,3], [-2,2]], k = 1
 * 输出: [[-2,2]]
 *
 * 解法分析:
 * 方法 1: 快速选择算法 (最优解)
 * 方法 2: 最大堆维护 K 个最小距离
 * 方法 3: 排序后取前 K 个
 *
 * 时间复杂度: O(n) 平均 - 方法 1
 * 空间复杂度: O(1) - 方法 1
 * 是否最优解: 是

```

```
/*
public static int[][] kClosest(int[][] points, int k) {
    if (points == null || points.length == 0 || k <= 0 || k > points.length) {
        throw new IllegalArgumentException("Invalid input parameters");
    }

    // 使用快速选择算法找到第 k 小的距离
    quickSelect(points, 0, points.length - 1, k);

    // 返回前 k 个点
    return Arrays.copyOf(points, k);
}

private static void quickSelect(int[][] points, int left, int right, int k) {
    if (left >= right) return;

    // 随机选择 pivot
    int pivotIndex = ThreadLocalRandom.current().nextInt(left, right + 1);
    int pivotDist = distance(points[pivotIndex]);

    // 分区操作
    int i = left;
    for (int j = left; j <= right; j++) {
        if (distance(points[j]) <= pivotDist) {
            swap(points, i, j);
            i++;
        }
    }
}

// 根据分区结果决定下一步
if (i == k) {
    return;
} else if (i < k) {
    quickSelect(points, i, right, k);
} else {
    quickSelect(points, left, i - 1, k);
}

private static int distance(int[] point) {
    return point[0] * point[0] + point[1] * point[1];
}
```

```

private static void swap(int[][] points, int i, int j) {
    int[] temp = points[i];
    points[i] = points[j];
    points[j] = temp;
}

/**
 * 题目 3: 1054. 距离相等的条形码
 * 来源: LeetCode
 * 链接: https://leetcode.cn/problems/distant-barcodes/
 * 难度: 中等
 *
 * 题目描述:
 * 在一个仓库里, 有一排条形码, 其中第 i 个条形码为 barcodes[i]。
 * 请你重新排列这些条形码, 使其中任意两个相邻的条形码不能相等。
 *
 * 示例:
 * 输入: barcodes = [1, 1, 1, 2, 2, 2]
 * 输出: [2, 1, 2, 1, 2, 1]
 *
 * 解法分析:
 * 方法 1: 最大堆按频率排序 (最优解)
 * 方法 2: 计数排序+间隔填充
 *
 * 时间复杂度: O(n log k) - 方法 1, k 为不同条形码的数量
 * 空间复杂度: O(n) - 方法 1
 * 是否最优解: 是
 */

public static int[] rearrangeBarcodes(int[] barcodes) {
    if (barcodes == null || barcodes.length == 0) {
        return new int[0];
    }

    // 统计频率
    Map<Integer, Integer> freqMap = new HashMap<>();
    for (int code : barcodes) {
        freqMap.put(code, freqMap.getOrDefault(code, 0) + 1);
    }

    // 最大堆, 按频率排序
    PriorityQueue<int[]> maxHeap = new PriorityQueue<>((a, b) -> b[1] - a[1]);
    for (Map.Entry<Integer, Integer> entry : freqMap.entrySet()) {
        maxHeap.offer(new int[] {entry.getKey(), entry.getValue()});
    }
}

```

```

    }

    int[] result = new int[barcodes.length];
    int index = 0;

    // 间隔填充，先填偶数位置，再填奇数位置
    while (!maxHeap.isEmpty()) {
        int[] current = maxHeap.poll();
        int code = current[0];
        int freq = current[1];

        // 填充所有当前条形码
        for (int i = 0; i < freq; i++) {
            if (index >= result.length) {
                index = 1; // 切换到奇数位置
            }
            result[index] = code;
            index += 2;
        }
    }

    return result;
}

```

```

/**
 * 题目 4: 324. 摆动排序 II
 * 来源: LeetCode
 * 链接: https://leetcode.cn/problems/wiggle-sort-ii/
 * 难度: 中等
 *
 * 题目描述:
 * 给你一个整数数组 nums，将它重新排列成 nums[0] < nums[1] > nums[2] < nums[3]... 的顺序。
 *
 * 示例:
 * 输入: nums = [1, 5, 1, 1, 6, 4]
 * 输出: [1, 6, 1, 5, 1, 4]
 *
 * 解法分析:
 * 方法 1: 排序+双指针 (最优解)
 * 方法 2: 快速选择找到中位数+三路划分
 *
 * 时间复杂度: O(n log n) - 方法 1
 * 空间复杂度: O(n) - 方法 1

```

```

* 是否最优解: 是 (对于通用情况)
*/
public static void wiggleSort(int[] nums) {
    if (nums == null || nums.length <= 1) return;

    // 复制并排序数组
    int[] sorted = nums.clone();
    Arrays.sort(sorted);

    int n = nums.length;
    int mid = (n + 1) / 2; // 中间位置 (向上取整)

    // 双指针填充: 左半部分从大到小, 右半部分从大到小
    int left = mid - 1;
    int right = n - 1;
    int index = 0;

    while (index < n) {
        if (index % 2 == 0) {
            // 偶数位置: 取左半部分 (较小的数)
            nums[index] = sorted[left];
            left--;
        } else {
            // 奇数位置: 取右半部分 (较大的数)
            nums[index] = sorted[right];
            right--;
        }
        index++;
    }
}

/**
 * 题目 5: 280. 摆动排序
 * 来源: LeetCode
 * 链接: https://leetcode.cn/problems/wiggle-sort/
 * 难度: 中等
 *
 * 题目描述:
 * 给你一个无序的数组 nums, 将它重新排列成 nums[0] <= nums[1] >= nums[2] <= nums[3]... 的顺序。
 *
 * 示例:
 * 输入: nums = [3, 5, 2, 1, 6, 4]

```

```

* 输出: [1, 6, 2, 5, 3, 4] 或类似
*
* 解法分析:
* 方法 1: 一次遍历交换相邻元素 (最优解)
* 方法 2: 排序后间隔交换
*
* 时间复杂度: O(n) - 方法 1
* 空间复杂度: O(1) - 方法 1
* 是否最优解: 是
*/
public static void wiggleSort280(int[] nums) {
    if (nums == null || nums.length <= 1) return;

    // 一次遍历, 根据需要交换相邻元素
    for (int i = 0; i < nums.length - 1; i++) {
        if ((i % 2 == 0 && nums[i] > nums[i + 1]) ||
            (i % 2 == 1 && nums[i] < nums[i + 1])) {
            // 交换相邻元素
            int temp = nums[i];
            nums[i] = nums[i + 1];
            nums[i + 1] = temp;
        }
    }
}

/***
* 题目 6: 493. 翻转对
* 来源: LeetCode
* 链接: https://leetcode.cn/problems/reverse-pairs/
* 难度: 困难
*
* 题目描述:
* 给定一个数组 nums, 如果  $i < j$  且  $nums[i] > 2 * nums[j]$ , 则称这是一个翻转对。
* 返回数组中翻转对的数量。
*
* 示例:
* 输入: nums = [1, 3, 2, 3, 1]
* 输出: 2
*
* 解法分析:
* 方法 1: 归并排序统计翻转对 (最优解)
* 方法 2: 树状数组/线段树
*

```

```

* 时间复杂度: O(n log n) - 方法 1
* 空间复杂度: O(n) - 方法 1
* 是否最优解: 是
*/
public static int reversePairs493(int[] nums) {
    if (nums == null || nums.length <= 1) return 0;

    int[] temp = new int[nums.length];
    return mergeSortCountPairs(nums, 0, nums.length - 1, temp);
}

private static int mergeSortCountPairs(int[] nums, int left, int right, int[] temp) {
    if (left >= right) return 0;

    int mid = left + (right - left) / 2;
    int count = 0;

    count += mergeSortCountPairs(nums, left, mid, temp);
    count += mergeSortCountPairs(nums, mid + 1, right, temp);
    count += countPairs(nums, left, mid, right);
    merge(nums, left, mid, right, temp);

    return count;
}

private static int countPairs(int[] nums, int left, int mid, int right) {
    int count = 0;
    int j = mid + 1;

    // 统计满足 nums[i] > 2 * nums[j] 的对数
    for (int i = left; i <= mid; i++) {
        while (j <= right && (long)nums[i] > 2L * nums[j]) {
            j++;
        }
        count += (j - (mid + 1));
    }

    return count;
}

private static void merge(int[] nums, int left, int mid, int right, int[] temp) {
    for (int i = left; i <= right; i++) {
        temp[i] = nums[i];
    }
}

```

```

}

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

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

/***
 * 题目 7：剑指 Offer 45. 把数组排成最小的数
 * 来源：剑指 Offer
 * 难度：中等
 *
 * 题目描述：
 * 输入一个非负整数数组，把数组里所有数字拼接起来排成一个数，打印能拼接出的所有数字中最小的一个。
 *
 * 示例：
 * 输入：[10, 2]
 * 输出："102"
 *
 * 解法分析：
 * 方法 1：自定义排序（最优解）
 * 方法 2：全排列（不推荐，复杂度高）
 *
 * 时间复杂度：O(n log n) - 方法 1
 * 空间复杂度：O(n) - 方法 1
 * 是否最优解：是
 */
public static String minNumber(int[] nums) {
    if (nums == null || nums.length == 0) return "";

```

```

// 将数字转换为字符串
String[] strNums = new String[nums.length];
for (int i = 0; i < nums.length; i++) {
    strNums[i] = String.valueOf(nums[i]);
}

// 自定义排序: 比较 s1+s2 和 s2+s1
Arrays.sort(strNums, (s1, s2) -> (s1 + s2).compareTo(s2 + s1));

// 拼接结果
StringBuilder result = new StringBuilder();
for (String str : strNums) {
    result.append(str);
}

return result.toString();
}

/**
 * 题目 8: HackerRank – Counting Inversions
 * 来源: HackerRank
 * 链接: https://www.hackerrank.com/challenges/ctci-merge-sort
 * 难度: 困难
 *
 * 题目描述:
 * 计算数组中逆序对的数量 (扩展版本, 处理大数据量)
 *
 * 解法分析:
 * 方法 1: 归并排序统计逆序对 (最优解)
 * 方法 2: 树状数组/线段树
 *
 * 时间复杂度: O(n log n) – 方法 1
 * 空间复杂度: O(n) – 方法 1
 * 是否最优解: 是
 */

public static long countInversions(int[] arr) {
    if (arr == null || arr.length <= 1) return 0;

    int[] temp = new int[arr.length];
    return mergeSortCountInversions(arr, 0, arr.length - 1, temp);
}

private static long mergeSortCountInversions(int[] arr, int left, int right, int[] temp) {

```

```

if (left >= right) return 0;

int mid = left + (right - left) / 2;
long count = 0;

count += mergeSortCountInversions(arr, left, mid, temp);
count += mergeSortCountInversions(arr, mid + 1, right, temp);
count += mergeAndCount(arr, left, mid, right, temp);

return count;
}

private static long mergeAndCount(int[] arr, int left, int mid, int right, int[] temp) {
    for (int i = left; i <= right; i++) {
        temp[i] = arr[i];
    }

    int i = left, j = mid + 1, k = left;
    long count = 0;

    while (i <= mid && j <= right) {
        if (temp[i] <= temp[j]) {
            arr[k++] = temp[i++];
        } else {
            arr[k++] = temp[j++];
            count += (mid - i + 1); // 统计逆序对
        }
    }

    while (i <= mid) {
        arr[k++] = temp[i++];
    }

    while (j <= right) {
        arr[k++] = temp[j++];
    }

    return count;
}

/***
 * 题目 9: 牛客网 NC140 排序
 * 来源: 牛客网
 * 链接: https://www.nowcoder.com/practice/2baf799ea0594abd974d37139de27896
 */

```

```
* 难度：简单
*
* 题目描述：
* 实现各种排序算法对数组进行排序
*
* 解法分析：
* 根据数据规模选择合适算法
* - 小数据：插入排序
* - 中等数据：快速排序
* - 大数据：归并排序
* - 需要稳定：归并排序
*
* 时间复杂度：根据算法选择
* 空间复杂度：根据算法选择
*/
public static int[] sortArrayNC140(int[] arr) {
    if (arr == null || arr.length <= 1) return arr;

    // 根据数据规模选择算法
    if (arr.length < 50) {
        // 小数组使用插入排序
        insertionSort(arr);
    } else {
        // 中等以上使用快速排序
        quickSort(arr, 0, arr.length - 1);
    }

    return arr;
}

private static void insertionSort(int[] arr) {
    for (int i = 1; i < arr.length; i++) {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}

private static void quickSort(int[] arr, int low, int high) {
```

```

    if (low < high) {
        int pivotIndex = partition(arr, low, high);
        quickSort(arr, low, pivotIndex - 1);
        quickSort(arr, pivotIndex + 1, high);
    }
}

private static int partition(int[] arr, int low, int high) {
    int pivot = arr[high];
    int i = low - 1;

    for (int j = low; j < high; j++) {
        if (arr[j] <= pivot) {
            i++;
            swap(arr, i, j);
        }
    }

    swap(arr, i + 1, high);
    return i + 1;
}

private static void swap(int[] arr, int i, int j) {
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}

/***
 * 题目 10: 外部排序模拟 - 多路归并
 * 来源: 算法导论
 * 难度: 困难
 *
 * 题目描述:
 * 模拟外部排序的多路归并过程, 处理无法一次性加载到内存的大数据
 *
 * 解法分析:
 * 方法 1: 多路归并排序
 * 方法 2: 败者树优化
 *
 * 时间复杂度: O(n log k) - k 为归并路数
 * 空间复杂度: O(k) - 缓冲区大小
 */

```

```
public static class ExternalSort {  
  
    /**  
     * 模拟多路归并排序  
     * @param chunks 多个有序数据块  
     * @return 合并后的有序数组  
     */  
  
    public static int[] multiwayMerge(List<int[]> chunks) {  
        if (chunks == null || chunks.isEmpty()) return new int[0];  
  
        // 使用优先队列进行多路归并  
        PriorityQueue<Element> minHeap = new PriorityQueue<>();  
  
        // 初始化每个数据块的指针  
        int[] pointers = new int[chunks.size()];  
        int totalSize = 0;  
  
        // 将每个数据块的第一个元素加入堆  
        for (int i = 0; i < chunks.size(); i++) {  
            int[] chunk = chunks.get(i);  
            totalSize += chunk.length;  
            if (chunk.length > 0) {  
                minHeap.offer(new Element(chunk[0], i, 0));  
                pointers[i] = 1;  
            }  
        }  
  
        int[] result = new int[totalSize];  
        int index = 0;  
  
        // 多路归并  
        while (!minHeap.isEmpty()) {  
            Element min = minHeap.poll();  
            result[index++] = min.value;  
  
            int chunkIndex = min.chunkIndex;  
            int elementIndex = min.elementIndex + 1;  
            int[] chunk = chunks.get(chunkIndex);  
  
            if (elementIndex < chunk.length) {  
                minHeap.offer(new Element(chunk[elementIndex], chunkIndex, elementIndex));  
            }  
        }  
    }  
}
```

```
        return result;
    }

static class Element implements Comparable<Element> {
    int value;
    int chunkIndex;
    int elementIndex;

    Element(int value, int chunkIndex, int elementIndex) {
        this.value = value;
        this.chunkIndex = chunkIndex;
        this.elementIndex = elementIndex;
    }

    @Override
    public int compareTo(Element other) {
        return Integer.compare(this.value, other.value);
    }
}

}

/***
 * 测试所有扩展题目
 */
public static void testAllProblems() {
    System.out.println("== 扩展排序题目测试开始 ==");

    // 测试题目 1: 合并两个有序数组
    System.out.println("\n 题目 1: 合并两个有序数组");
    int[] nums1 = new int[]{1, 2, 3, 0, 0, 0};
    int[] nums2 = new int[]{2, 5, 6};
    mergeSortedArrays(nums1, 3, nums2, 3);
    System.out.println("合并结果: " + Arrays.toString(nums1));

    // 测试题目 2: 最接近原点的 K 个点
    System.out.println("\n 题目 2: 最接近原点的 K 个点");
    int[][] points = new int[][]{{1, 3}, {-2, 2}, {5, 8}, {0, 1}};
    int[][] result2 = kClosest(points, 2);
    System.out.println("最接近的 2 个点: " + Arrays.deepToString(result2));

    // 测试题目 3: 距离相等的条形码
    System.out.println("\n 题目 3: 距离相等的条形码");
```

```
int[] barcodes = new int[]{1, 1, 1, 2, 2, 2};  
int[] result3 = rearrangeBarcodes(barcodes);  
System.out.println("重新排列结果: " + Arrays.toString(result3));  
  
// 测试题目 4: 摆动排序 II  
System.out.println("\n题目 4: 摆动排序 II");  
int[] nums4 = new int[]{1, 5, 1, 1, 6, 4};  
wiggleSort(nums4);  
System.out.println("摆动排序结果: " + Arrays.toString(nums4));  
  
// 测试题目 5: 摆动排序  
System.out.println("\n题目 5: 摆动排序");  
int[] nums5 = new int[]{3, 5, 2, 1, 6, 4};  
wiggleSort280(nums5);  
System.out.println("摆动排序结果: " + Arrays.toString(nums5));  
  
// 测试题目 6: 翻转对  
System.out.println("\n题目 6: 翻转对");  
int[] nums6 = new int[]{1, 3, 2, 3, 1};  
int count6 = reversePairs493(nums6);  
System.out.println("翻转对数量: " + count6);  
  
// 测试题目 7: 把数组排成最小的数  
System.out.println("\n题目 7: 把数组排成最小的数");  
int[] nums7 = new int[]{10, 2};  
String result7 = minNumber(nums7);  
System.out.println("最小数字: " + result7);  
  
// 测试题目 8: 逆序对计数  
System.out.println("\n题目 8: 逆序对计数");  
int[] nums8 = new int[]{2, 4, 1, 3, 5};  
long count8 = countInversions(nums8);  
System.out.println("逆序对数量: " + count8);  
  
// 测试题目 9: 牛客网排序  
System.out.println("\n题目 9: 牛客网排序");  
int[] nums9 = new int[]{3, 1, 4, 1, 5, 9, 2, 6};  
int[] result9 = sortArrayNC140(nums9);  
System.out.println("排序结果: " + Arrays.toString(result9));  
  
// 测试题目 10: 外部排序  
System.out.println("\n题目 10: 外部排序模拟");  
List<int[]> chunks = Arrays.asList(  
    {1, 2, 3, 4, 5},  
    {6, 7, 8, 9, 10},  
    {11, 12, 13, 14, 15},  
    {16, 17, 18, 19, 20},  
    {21, 22, 23, 24, 25},  
    {26, 27, 28, 29, 30},  
    {31, 32, 33, 34, 35},  
    {36, 37, 38, 39, 40},  
    {41, 42, 43, 44, 45},  
    {46, 47, 48, 49, 50},  
    {51, 52, 53, 54, 55},  
    {56, 57, 58, 59, 60},  
    {61, 62, 63, 64, 65},  
    {66, 67, 68, 69, 70},  
    {71, 72, 73, 74, 75},  
    {76, 77, 78, 79, 80},  
    {81, 82, 83, 84, 85},  
    {86, 87, 88, 89, 90},  
    {91, 92, 93, 94, 95},  
    {96, 97, 98, 99, 100}  
)
```

```

        new int[] {1, 3, 5},
        new int[] {2, 4, 6},
        new int[] {0, 7, 8}
    );
    int[] result10 = ExternalSort.multiwayMerge(chunks);
    System.out.println("多路归并结果: " + Arrays.toString(result10));
    System.out.println("\n==== 扩展排序题目测试结束 ===");
}

public static void main(String[] args) {
    try {
        testAllProblems();
    } catch (Exception e) {
        System.err.println("测试过程中出现错误: " + e.getMessage());
        e.printStackTrace();
    }
}

```

=====

文件: ExtendedSortProblems.py

=====

```

"""
排序算法扩展题目 - Python 版本
包含更多 LeetCode、牛客网、剑指 Offer 等平台的排序相关题目
每个题目都包含多种解法和详细分析
"""

时间复杂度分析: 详细分析每种解法的时间复杂度
空间复杂度分析: 分析内存使用情况
最优解判断: 确定是否为最优解, 如果不是则寻找最优解
"""

```

```

import heapq
import random
from typing import List, Tuple
import sys
from collections import Counter

```

```
class ExtendedSortProblems:
```

```
"""

```

题目 1: 88. 合并两个有序数组

来源: LeetCode

链接: <https://leetcode.cn/problems/merge-sorted-array/>

难度: 简单

时间复杂度: $O(m + n)$

空间复杂度: $O(1)$

是否最优解: 是

"""

@staticmethod

def mergeSortedArrays(nums1: List[int], m: int, nums2: List[int], n: int) -> None:

"""

合并两个有序数组到 nums1 中

Args:

nums1: 第一个有序数组, 有足够的空间容纳合并后的结果

m: nums1 中有效元素的数量

nums2: 第二个有序数组

n: nums2 中元素的数量

"""

if m < 0 or n < 0:

raise ValueError("Invalid input parameters")

p1 = m - 1 # nums1 有效部分的末尾

p2 = n - 1 # nums2 的末尾

p = m + n - 1 # 合并后的末尾

从后向前合并, 避免覆盖 nums1 中的元素

while p1 >= 0 and p2 >= 0:

if nums1[p1] > nums2[p2]:

nums1[p] = nums1[p1]

p1 -= 1

else:

nums1[p] = nums2[p2]

p2 -= 1

p -= 1

如果 nums2 还有剩余元素, 直接复制到 nums1 前面

while p2 >= 0:

nums1[p] = nums2[p2]

p2 -= 1

p -= 1

"""

题目 2: 973. 最接近原点的 K 个点

来源: LeetCode

链接: <https://leetcode.cn/problems/k-closest-points-to-origin/>

难度: 中等

时间复杂度: O(n) 平均

空间复杂度: O(1)

是否最优解: 是

"""

```
@staticmethod
```

```
def kClosest(points: List[List[int]], k: int) -> List[List[int]]:
```

"""

找到距离原点最近的 k 个点

Args:

points: 点坐标列表, 每个点格式为[x, y]

k: 需要返回的点的数量

Returns:

距离原点最近的 k 个点

"""

```
if not points or k <= 0 or k > len(points):
```

```
    raise ValueError("Invalid input parameters")
```

使用快速选择算法找到第 k 小的距离

```
ExtendedSortProblems._quick_select(points, 0, len(points) - 1, k)
```

返回前 k 个点

```
return points[:k]
```

```
@staticmethod
```

```
def _quick_select(points: List[List[int]], left: int, right: int, k: int) -> None:
```

""”快速选择算法实现””

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

```
    return
```

随机选择 pivot

```
pivot_index = random.randint(left, right)
```

```
pivot_dist = ExtendedSortProblems._distance(points[pivot_index])
```

分区操作

```
i = left
```

```
for j in range(left, right + 1):
```

```

    if ExtendedSortProblems._distance(points[j]) <= pivot_dist:
        points[i], points[j] = points[j], points[i]
        i += 1

# 根据分区结果决定下一步
if i == k:
    return
elif i < k:
    ExtendedSortProblems._quick_select(points, i, right, k)
else:
    ExtendedSortProblems._quick_select(points, left, i - 1, k)

@staticmethod
def _distance(point: List[int]) -> int:
    """计算点到原点的距离平方（避免开方运算）"""
    return point[0] * point[0] + point[1] * point[1]

"""

```

题目 3: 1054. 距离相等的条形码

来源: LeetCode

链接: <https://leetcode.cn/problems/distant-barcodes/>

难度: 中等

时间复杂度: $O(n \log k)$ – k 为不同条形码的数量

空间复杂度: $O(n)$

是否最优解: 是

"""

@staticmethod

```
def rearrangeBarcodes(barcodes: List[int]) -> List[int]:
    """

```

重新排列条形码，使相邻条形码不相等

Args:

barcodes: 条形码列表

Returns:

重新排列后的条形码列表

"""

if not barcodes:

return []

统计频率

freq_map = Counter(barcodes)

```

# 最大堆，按频率排序
max_heap = []
for code, freq in freq_map.items():
    heapq.heappush(max_heap, (-freq, code))

result = [0] * len(barcodes)
index = 0

# 间隔填充，先填偶数位置，再填奇数位置
while max_heap:
    neg_freq, code = heapq.heappop(max_heap)
    freq = -neg_freq

    # 填充所有当前条形码
    for _ in range(freq):
        if index >= len(result):
            index = 1 # 切换到奇数位置
        result[index] = code
        index += 2

return result
"""

```

题目 4: 324. 摆动排序 II

来源: LeetCode

链接: <https://leetcode.cn/problems/wiggle-sort-ii/>

难度: 中等

时间复杂度: $O(n \log n)$

空间复杂度: $O(n)$

是否最优解: 是 (对于通用情况)

"""

@staticmethod

def wiggleSort(nums: List[int]) -> None:

"""

摆动排序: $\text{nums}[0] < \text{nums}[1] > \text{nums}[2] < \text{nums}[3] \dots$

Args:

nums: 待排序数组

"""

if len(nums) <= 1:

return

```

# 复制并排序数组
sorted_nums = sorted(nums)

n = len(nums)
mid = (n + 1) // 2 # 中间位置（向上取整）

# 双指针填充：左半部分从大到小，右半部分从大到小
left = mid - 1
right = n - 1
index = 0

while index < n:
    if index % 2 == 0:
        # 偶数位置：取左半部分（较小的数）
        nums[index] = sorted_nums[left]
        left -= 1
    else:
        # 奇数位置：取右半部分（较大的数）
        nums[index] = sorted_nums[right]
        right -= 1
    index += 1

```

"""

题目 5: 280. 摆动排序

来源: LeetCode

链接: <https://leetcode.cn/problems/wiggle-sort/>

难度: 中等

时间复杂度: O(n)

空间复杂度: O(1)

是否最优解: 是

"""

@staticmethod

def wiggleSort280(nums: List[int]) -> None:

"""

摆动排序: nums[0] <= nums[1] >= nums[2] <= nums[3]...

Args:

nums: 待排序数组

"""

if len(nums) <= 1:

return

```

# 一次遍历，根据需要交换相邻元素
for i in range(len(nums) - 1):
    if (i % 2 == 0 and nums[i] > nums[i + 1]) or \
        (i % 2 == 1 and nums[i] < nums[i + 1]):
        # 交换相邻元素
        nums[i], nums[i + 1] = nums[i + 1], nums[i]

"""

```

题目 6: 493. 翻转对

来源: LeetCode

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

难度: 困难

时间复杂度: $O(n \log n)$

空间复杂度: $O(n)$

是否最优解: 是

"""

@staticmethod

```
def reversePairs493(nums: List[int]) -> int:
    """

```

统计数组中翻转对的数量 ($nums[i] > 2 * nums[j]$ 且 $i < j$)

Args:

 nums: 整数数组

Returns:

 翻转对的数量

"""

```
if len(nums) <= 1:
    return 0
```

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

```
return ExtendedSortProblems._merge_sort_count_pairs(nums, 0, len(nums) - 1, temp)
```

@staticmethod

```
def _merge_sort_count_pairs(nums: List[int], left: int, right: int, temp: List[int]) -> int:
```

"""归并排序统计翻转对"""

```
    if left >= right:
        return 0
```

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

```
    count = 0
```

```

count += ExtendedSortProblems._merge_sort_count_pairs(nums, left, mid, temp)
count += ExtendedSortProblems._merge_sort_count_pairs(nums, mid + 1, right, temp)
count += ExtendedSortProblems._count_pairs(nums, left, mid, right)
ExtendedSortProblems._merge(nums, left, mid, right, temp)

return count

@staticmethod
def _count_pairs(nums: List[int], left: int, mid: int, right: int) -> int:
    """统计满足 nums[i] > 2 * nums[j] 的对数"""
    count = 0
    j = mid + 1

    # 统计满足 nums[i] > 2 * nums[j] 的对数
    for i in range(left, mid + 1):
        while j <= right and nums[i] > 2 * nums[j]:
            j += 1
        count += (j - (mid + 1))

    return count

@staticmethod
def _merge(nums: List[int], left: int, mid: int, right: int, temp: List[int]) -> None:
    """归并操作"""
    for i in range(left, right + 1):
        temp[i] = nums[i]

    i, k, j = left, left, mid + 1

    while i <= mid and j <= right:
        if temp[i] <= temp[j]:
            nums[k] = temp[i]
            i += 1
        else:
            nums[k] = temp[j]
            j += 1
        k += 1

    while i <= mid:
        nums[k] = temp[i]
        i += 1
        k += 1

```

```
    while j <= right:  
        nums[k] = temp[j]  
        j += 1  
        k += 1
```

```
"""
```

题目 7: 剑指 Offer 45. 把数组排成最小的数

来源: 剑指 Offer

难度: 中等

时间复杂度: $O(n \log n)$

空间复杂度: $O(n)$

是否最优解: 是

```
"""
```

```
@staticmethod  
def minNumber(nums: List[int]) -> str:  
    """  
    把数组排成最小的数
```

Args:

nums: 非负整数数组

Returns:

能拼接出的最小数字的字符串表示

```
"""
```

```
if not nums:  
    return ""
```

将数字转换为字符串

```
str_nums = [str(num) for num in nums]
```

自定义排序: 比较 s1+s2 和 s2+s1

```
str_nums.sort(key=lambda x: x * 10) # 乘以 10 确保比较长度足够
```

拼接结果

```
return ''.join(str_nums)
```

```
"""
```

题目 8: HackerRank – Counting Inversions

来源: HackerRank

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

难度: 困难

时间复杂度: $O(n \log n)$
 空间复杂度: $O(n)$
 是否最优解: 是

"""

```

@staticmethod
def countInversions(arr: List[int]) -> int:
    """
    计算数组中逆序对的数量

    Args:
        arr: 整数数组

    Returns:
        逆序对的数量
    """
    if len(arr) <= 1:
        return 0

    temp = [0] * len(arr)
    return ExtendedSortProblems._merge_sort_count_inversions(arr, 0, len(arr) - 1, temp)

@staticmethod
def _merge_sort_count_inversions(arr: List[int], left: int, right: int, temp: List[int]) -> int:
    """
    归并排序统计逆序对"""

    if left >= right:
        return 0

    mid = left + (right - left) // 2
    count = 0

    count += ExtendedSortProblems._merge_sort_count_inversions(arr, left, mid, temp)
    count += ExtendedSortProblems._merge_sort_count_inversions(arr, mid + 1, right, temp)
    count += ExtendedSortProblems._merge_and_count(arr, left, mid, right, temp)

    return count

@staticmethod
def _merge_and_count(arr: List[int], left: int, mid: int, right: int, temp: List[int]) -> int:
    """
    归并并统计逆序对"""

    for i in range(left, right + 1):
        if arr[i] > arr[mid]:
            count += right - mid + 1
        else:
            temp[i] = arr[i]
    for i in range(left, right + 1):
        arr[i] = temp[i]
```

```
temp[i] = arr[i]

i, k, j = left, left, mid + 1
count = 0

while i <= mid and j <= right:
    if temp[i] <= temp[j]:
        arr[k] = temp[i]
        i += 1
    else:
        arr[k] = temp[j]
        j += 1
    count += (mid - i + 1) # 统计逆序对
    k += 1

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

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

return count
```

"""

题目 9：牛客网 NC140 排序

来源：牛客网

链接：<https://www.nowcoder.com/practice/2baf799ea0594abd974d37139de27896>

难度：简单

时间复杂度：根据算法选择

空间复杂度：根据算法选择

"""

```
@staticmethod
def sortArrayNC140(arr: List[int]) -> List[int]:
```

"""

根据数据规模选择合适的排序算法

Args:

arr: 待排序数组

Returns:

排序后的数组

"""

```
if len(arr) <= 1:  
    return arr.copy()  
  
# 根据数据规模选择算法  
if len(arr) < 50:  
    # 小数组使用插入排序  
    return ExtendedSortProblems._insertion_sort(arr.copy())  
else:  
    # 中等以上使用快速排序  
    arr_copy = arr.copy()  
    ExtendedSortProblems._quick_sort(arr_copy, 0, len(arr_copy) - 1)  
    return arr_copy
```

@staticmethod

```
def _insertion_sort(arr: List[int]) -> List[int]:  
    """插入排序"""  
    for i in range(1, len(arr)):  
        key = arr[i]  
        j = i - 1  
        while j >= 0 and arr[j] > key:  
            arr[j + 1] = arr[j]  
            j -= 1  
        arr[j + 1] = key  
    return arr
```

@staticmethod

```
def _quick_sort(arr: List[int], low: int, high: int) -> None:  
    """快速排序"""  
    if low < high:  
        pivot_index = ExtendedSortProblems._partition(arr, low, high)  
        ExtendedSortProblems._quick_sort(arr, low, pivot_index - 1)  
        ExtendedSortProblems._quick_sort(arr, pivot_index + 1, high)
```

@staticmethod

```
def _partition(arr: List[int], low: int, high: int) -> int:  
    """快速排序分区操作"""  
    pivot = arr[high]  
    i = low - 1
```

```

for j in range(low, high):
    if arr[j] <= pivot:
        i += 1
        arr[i], arr[j] = arr[j], arr[i]

arr[i + 1], arr[high] = arr[high], arr[i + 1]
return i + 1

```

"""

题目 10：外部排序模拟 - 多路归并

来源：算法导论

难度：困难

时间复杂度： $O(n \log k)$ - k 为归并路数

空间复杂度： $O(k)$ - 缓冲区大小

"""

class ExternalSort:

```

@staticmethod
def multiwayMerge(chunks: List[List[int]]) -> List[int]:
    """

```

模拟多路归并排序

Args:

chunks: 多个有序数据块

Returns:

合并后的有序数组

"""

if not chunks:

return []

使用优先队列进行多路归并

heap = []

初始化每个数据块的指针

total_size = sum(len(chunk) for chunk in chunks)

将每个数据块的第一个元素加入堆

for i, chunk in enumerate(chunks):

if chunk:

heapq.heappush(heap, (chunk[0], i, 0))

result = []

```
# 多路归并
while heap:
    value, chunk_index, element_index = heapq.heappop(heap)
    result.append(value)

    element_index += 1
    if element_index < len(chunks[chunk_index]):
        heapq.heappush(heap, (chunks[chunk_index][element_index], chunk_index,
element_index))

return result

"""
测试所有扩展题目
"""

@staticmethod
def test_all_problems():
    """测试所有扩展题目"""
    print("== 扩展排序题目测试开始 ==")

    # 测试题目 1: 合并两个有序数组
    print("\n题目 1: 合并两个有序数组")
    nums1 = [1, 2, 3, 0, 0, 0]
    nums2 = [2, 5, 6]
    ExtendedSortProblems.mergeSortedArrays(nums1, 3, nums2, 3)
    print(f"合并结果: {nums1}")

    # 测试题目 2: 最接近原点的 K 个点
    print("\n题目 2: 最接近原点的 K 个点")
    points = [[1, 3], [-2, 2], [5, 8], [0, 1]]
    result2 = ExtendedSortProblems.kClosest(points, 2)
    print(f"最接近的 2 个点: {result2}")

    # 测试题目 3: 距离相等的条形码
    print("\n题目 3: 距离相等的条形码")
    barcodes = [1, 1, 1, 2, 2, 2]
    result3 = ExtendedSortProblems.rearrangeBarcodes(barcodes)
    print(f"重新排列结果: {result3}")

    # 测试题目 4: 摆动排序 II
    print("\n题目 4: 摆动排序 II")
    nums4 = [1, 5, 1, 1, 6, 4]
```

```
ExtendedSortProblems.wiggleSort(nums4)
print(f"摆动排序结果: {nums4}")

# 测试题目 5: 摆动排序
print("\n题目 5: 摆动排序")
nums5 = [3, 5, 2, 1, 6, 4]
ExtendedSortProblems.wiggleSort280(nums5)
print(f"摆动排序结果: {nums5}")

# 测试题目 6: 翻转对
print("\n题目 6: 翻转对")
nums6 = [1, 3, 2, 3, 1]
count6 = ExtendedSortProblems.reversePairs493(nums6)
print(f"翻转对数量: {count6}")

# 测试题目 7: 把数组排成最小的数
print("\n题目 7: 把数组排成最小的数")
nums7 = [10, 2]
result7 = ExtendedSortProblems.minNumber(nums7)
print(f"最小数字: {result7}")

# 测试题目 8: 逆序对计数
print("\n题目 8: 逆序对计数")
nums8 = [2, 4, 1, 3, 5]
count8 = ExtendedSortProblems.countInversions(nums8)
print(f"逆序对数量: {count8}")

# 测试题目 9: 牛客网排序
print("\n题目 9: 牛客网排序")
nums9 = [3, 1, 4, 1, 5, 9, 2, 6]
result9 = ExtendedSortProblems.sortArrayNC140(nums9)
print(f"排序结果: {result9}")

# 测试题目 10: 外部排序
print("\n题目 10: 外部排序模拟")
chunks = [
    [1, 3, 5],
    [2, 4, 6],
    [0, 7, 8]
]
result10 = ExtendedSortProblems.ExternalSort.multiwayMerge(chunks)
print(f"多路归并结果: {result10}")
```

```
print("\n==== 扩展排序题目测试结束 ===")\n\nif __name__ == "__main__":\n    try:\n        ExtendedSortProblems. test_all_problems()\n    except Exception as e:\n        print(f"测试过程中出现错误: {e}")\n        import traceback\n        traceback.print_exc()\n\n=====
```

文件: ExtendedSortProblems_part1.py

```
"""\n排序算法扩展题目 - Python 版本 (第一部分)\n包含更多 LeetCode、牛客网、剑指 Offer 等平台的排序相关题目\n每个题目都包含多种解法和详细分析\n\n时间复杂度分析: 详细分析每种解法的时间复杂度\n空间复杂度分析: 分析内存使用情况\n最优解判断: 确定是否为最优解, 如果不是则寻找最优解\n"""\n\n
```

```
import heapq\nimport random\nfrom typing import List, Tuple\nimport sys
```

```
class ExtendedSortProblems:
```

```
    """
```

题目 1: 88. 合并两个有序数组

来源: LeetCode

链接: <https://leetcode.cn/problems/merge-sorted-array/>

难度: 简单

时间复杂度: $O(m + n)$

空间复杂度: $O(1)$

是否最优解: 是

```
    """
```

```
@staticmethod
```

```
def merge_sorted_arrays(nums1: List[int], m: int, nums2: List[int], n: int) -> None:
```

```
    """
```

合并两个有序数组到 nums1 中

Args:

nums1: 第一个有序数组, 有足够的空间容纳合并后的结果

m: nums1 中有效元素的数量

nums2: 第二个有序数组

n: nums2 中元素的数量

"""

if m < 0 or n < 0:

 raise ValueError("Invalid input parameters")

p1 = m - 1 # nums1 有效部分的末尾

p2 = n - 1 # nums2 的末尾

p = m + n - 1 # 合并后的末尾

从后向前合并, 避免覆盖 nums1 中的元素

while p1 >= 0 and p2 >= 0:

 if nums1[p1] > nums2[p2]:

 nums1[p] = nums1[p1]

 p1 -= 1

 else:

 nums1[p] = nums2[p2]

 p2 -= 1

 p -= 1

如果 nums2 还有剩余元素, 直接复制到 nums1 前面

while p2 >= 0:

 nums1[p] = nums2[p2]

 p2 -= 1

 p -= 1

"""

题目 2: 973. 最接近原点的 K 个点

来源: LeetCode

链接: <https://leetcode.cn/problems/k-closest-points-to-origin/>

难度: 中等

时间复杂度: O(n) 平均

空间复杂度: O(1)

是否最优解: 是

"""

@staticmethod

def k_closest(points: List[List[int]], k: int) -> List[List[int]]:

```
"""
```

```
找到距离原点最近的 k 个点
```

```
Args:
```

```
    points: 点坐标列表, 每个点格式为[x, y]  
    k: 需要返回的点的数量
```

```
Returns:
```

```
    距离原点最近的 k 个点
```

```
"""
```

```
if not points or k <= 0 or k > len(points):  
    raise ValueError("Invalid input parameters")
```

```
# 使用快速选择算法找到第 k 小的距离
```

```
ExtendedSortProblems._quick_select(points, 0, len(points) - 1, k)
```

```
# 返回前 k 个点
```

```
return points[:k]
```

```
@staticmethod
```

```
def _quick_select(points: List[List[int]], left: int, right: int, k: int) -> None:
```

```
    """快速选择算法实现"""
```

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

```
# 随机选择 pivot
```

```
pivot_index = random.randint(left, right)
```

```
pivot_dist = ExtendedSortProblems._distance(points[pivot_index])
```

```
# 分区操作
```

```
i = left
```

```
for j in range(left, right + 1):
```

```
    if ExtendedSortProblems._distance(points[j]) <= pivot_dist:  
        points[i], points[j] = points[j], points[i]  
        i += 1
```

```
# 根据分区结果决定下一步
```

```
if i == k:  
    return
```

```
elif i < k:  
    ExtendedSortProblems._quick_select(points, i, right, k)
```

```
else:  
    ExtendedSortProblems._quick_select(points, left, i - 1, k)
```

```
@staticmethod  
def _distance(point: List[int]) -> int:  
    """计算点到原点的距离平方（避免开方运算）"""  
    return point[0] * point[0] + point[1] * point[1]
```

"""

题目 3: 1054. 距离相等的条形码

来源: LeetCode

链接: <https://leetcode.cn/problems/distant-barcodes/>

难度: 中等

时间复杂度: $O(n \log k)$ – k 为不同条形码的数量

空间复杂度: $O(n)$

是否最优解: 是

"""

```
@staticmethod  
def rearrange_barcodes(barcodes: List[int]) -> List[int]:  
    """  
    重新排列条形码，使相邻条形码不相等  
    """
```

Args:

barcodes: 条形码列表

Returns:

重新排列后的条形码列表

"""

```
if not barcodes:  
    return []
```

```
# 统计频率  
from collections import Counter  
freq_map = Counter(barcodes)
```

```
# 最大堆，按频率排序  
max_heap = []  
for code, freq in freq_map.items():  
    heapq.heappush(max_heap, (-freq, code))
```

```
result = [0] * len(barcodes)  
index = 0
```

```
# 间隔填充，先填偶数位置，再填奇数位置
```

```

while max_heap:
    neg_freq, code = heapq.heappop(max_heap)
    freq = -neg_freq

    # 填充所有当前条形码
    for _ in range(freq):
        if index >= len(result):
            index = 1 # 切换到奇数位置
        result[index] = code
        index += 2

return result
"""

```

题目 4: 324. 摆动排序 II

来源: LeetCode

链接: <https://leetcode.cn/problems/wiggle-sort-ii/>

难度: 中等

时间复杂度: $O(n \log n)$

空间复杂度: $O(n)$

是否最优解: 是 (对于通用情况)

"""

```

@staticmethod
def wiggle_sort(nums: List[int]) -> None:
    """
    摆动排序: nums[0] < nums[1] > nums[2] < nums[3]...
    
```

Args:

nums: 待排序数组

"""

```

if len(nums) <= 1:
    return

```

复制并排序数组

```

sorted_nums = sorted(nums)

```

n = len(nums)

mid = (n + 1) // 2 # 中间位置 (向上取整)

双指针填充: 左半部分从大到小, 右半部分从大到小

left = mid - 1

right = n - 1

```

index = 0

while index < n:
    if index % 2 == 0:
        # 偶数位置: 取左半部分 (较小的数)
        nums[index] = sorted_nums[left]
        left -= 1
    else:
        # 奇数位置: 取右半部分 (较大的数)
        nums[index] = sorted_nums[right]
        right -= 1
    index += 1

"""

```

题目 5: 280. 摆动排序

来源: LeetCode

链接: <https://leetcode.cn/problems/wiggle-sort/>

难度: 中等

时间复杂度: O(n)

空间复杂度: O(1)

是否最优解: 是

"""

@staticmethod

```
def wiggle_sort_280(nums: List[int]) -> None:
```

"""

摆动排序: nums[0] <= nums[1] >= nums[2] <= nums[3]...

Args:

nums: 待排序数组

"""

```
if len(nums) <= 1:
```

```
    return
```

一次遍历, 根据需要交换相邻元素

```
for i in range(len(nums) - 1):
```

```
    if (i % 2 == 0 and nums[i] > nums[i + 1]) or \
```

```
        (i % 2 == 1 and nums[i] < nums[i + 1]):
```

交换相邻元素

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

"""

题目 6: 493. 翻转对

来源: LeetCode

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

难度: 困难

时间复杂度: $O(n \log n)$

空间复杂度: $O(n)$

是否最优解: 是

"""

@staticmethod

def reverse_pairs_493(nums: List[int]) -> int:

"""

统计数组中翻转对的数量 (nums[i] > 2 * nums[j] 且 i < j)

Args:

 nums: 整数数组

Returns:

 翻转对的数量

"""

if len(nums) <= 1:

 return 0

temp = [0] * len(nums)

return ExtendedSortProblems._merge_sort_count_pairs(nums, 0, len(nums) - 1, temp)

@staticmethod

def _merge_sort_count_pairs(nums: List[int], left: int, right: int, temp: List[int]) -> int:

"""归并排序统计翻转对"""

if left >= right:

 return 0

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

count = 0

count += ExtendedSortProblems._merge_sort_count_pairs(nums, left, mid, temp)

count += ExtendedSortProblems._merge_sort_count_pairs(nums, mid + 1, right, temp)

count += ExtendedSortProblems._count_pairs(nums, left, mid, right)

ExtendedSortProblems._merge(nums, left, mid, right, temp)

return count

@staticmethod

def _count_pairs(nums: List[int], left: int, mid: int, right: int) -> int:

```

"""统计满足 nums[i] > 2 * nums[j] 的对数"""
count = 0
j = mid + 1

# 统计满足 nums[i] > 2 * nums[j] 的对数
for i in range(left, mid + 1):
    while j <= right and nums[i] > 2 * nums[j]:
        j += 1
    count += (j - (mid + 1))

return count

@staticmethod
def _merge(nums: List[int], left: int, mid: int, right: int, temp: List[int]) -> None:
    """归并操作"""
    for i in range(left, right + 1):
        temp[i] = nums[i]

    i, k, j = left, left, mid + 1

    while i <= mid and j <= right:
        if temp[i] <= temp[j]:
            nums[k] = temp[i]
            i += 1
        else:
            nums[k] = temp[j]
            j += 1
        k += 1

    while i <= mid:
        nums[k] = temp[i]
        i += 1
        k += 1

    while j <= right:
        nums[k] = temp[j]
        j += 1
        k += 1

```

文件: ExtendedSortProblems_part2.py

```
"""
```

排序算法扩展题目 - Python 版本 (第二部分)

```
"""
```

```
import heapq
import random
from typing import List, Tuple
import sys
```

```
class ExtendedSortProblems:
```

```
    """
```

题目 7: 剑指 Offer 45. 把数组排成最小的数

来源: 剑指 Offer

难度: 中等

时间复杂度: $O(n \log n)$

空间复杂度: $O(n)$

是否最优解: 是

```
"""
```

```
@staticmethod
```

```
def min_number(nums: List[int]) -> str:
```

```
    """
```

把数组排成最小的数

Args:

nums: 非负整数数组

Returns:

能拼接出的最小数字的字符串表示

```
"""
```

```
if not nums:
```

```
    return ""
```

将数字转换为字符串

```
str_nums = [str(num) for num in nums]
```

自定义排序: 比较 s1+s2 和 s2+s1

```
str_nums.sort(key=lambda x: x * 10) # 乘以 10 确保比较长度足够
```

拼接结果

```
return ''.join(str_nums)
```

```
"""
```

题目 8: HackerRank – Counting Inversions

来源: HackerRank

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

难度: 困难

时间复杂度: $O(n \log n)$

空间复杂度: $O(n)$

是否最优解: 是

"""

@staticmethod

def count_inversions(arr: List[int]) -> int:

"""

计算数组中逆序对的数量

Args:

arr: 整数数组

Returns:

逆序对的数量

"""

if len(arr) <= 1:

 return 0

temp = [0] * len(arr)

return ExtendedSortProblems._merge_sort_count_inversions(arr, 0, len(arr) - 1, temp)

@staticmethod

def _merge_sort_count_inversions(arr: List[int], left: int, right: int, temp: List[int]) -> int:

"""归并排序统计逆序对"""

if left >= right:

 return 0

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

count = 0

count += ExtendedSortProblems._merge_sort_count_inversions(arr, left, mid, temp)

count += ExtendedSortProblems._merge_sort_count_inversions(arr, mid + 1, right, temp)

count += ExtendedSortProblems._merge_and_count(arr, left, mid, right, temp)

return count

@staticmethod

```

def _merge_and_count(arr: List[int], left: int, mid: int, right: int, temp: List[int]) ->
int:
    """归并并统计逆序对"""
    for i in range(left, right + 1):
        temp[i] = arr[i]

    i, k, j = left, left, mid + 1
    count = 0

    while i <= mid and j <= right:
        if temp[i] <= temp[j]:
            arr[k] = temp[i]
            i += 1
        else:
            arr[k] = temp[j]
            j += 1
            count += (mid - i + 1) # 统计逆序对
        k += 1

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

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

    return count

"""

```

题目 9: 牛客网 NC140 排序

来源: 牛客网

链接: <https://www.nowcoder.com/practice/2baf799ea0594abd974d37139de27896>

难度: 简单

时间复杂度: 根据算法选择

空间复杂度: 根据算法选择

"""

@staticmethod

```
def sort_array_nc140(arr: List[int]) -> List[int]:
    """
```

根据数据规模选择合适的排序算法

Args:

arr: 待排序数组

Returns:

排序后的数组

"""

```
if len(arr) <= 1:  
    return arr.copy()  
  
# 根据数据规模选择算法  
if len(arr) < 50:  
    # 小数组使用插入排序  
    return ExtendedSortProblems._insertion_sort(arr.copy())  
else:  
    # 中等以上使用快速排序  
    arr_copy = arr.copy()  
    ExtendedSortProblems._quick_sort(arr_copy, 0, len(arr_copy) - 1)  
    return arr_copy
```

@staticmethod

```
def _insertion_sort(arr: List[int]) -> List[int]:  
    """插入排序"""  
    for i in range(1, len(arr)):  
        key = arr[i]  
        j = i - 1  
        while j >= 0 and arr[j] > key:  
            arr[j + 1] = arr[j]  
            j -= 1  
        arr[j + 1] = key  
    return arr
```

@staticmethod

```
def _quick_sort(arr: List[int], low: int, high: int) -> None:  
    """快速排序"""  
    if low < high:  
        pivot_index = ExtendedSortProblems._partition(arr, low, high)  
        ExtendedSortProblems._quick_sort(arr, low, pivot_index - 1)  
        ExtendedSortProblems._quick_sort(arr, pivot_index + 1, high)
```

@staticmethod

```
def _partition(arr: List[int], low: int, high: int) -> int:
```

```

"""快速排序分区操作"""
pivot = arr[high]
i = low - 1

for j in range(low, high):
    if arr[j] <= pivot:
        i += 1
        arr[i], arr[j] = arr[j], arr[i]

arr[i + 1], arr[high] = arr[high], arr[i + 1]
return i + 1

```

"""

题目 10：外部排序模拟 – 多路归并

来源：算法导论

难度：困难

时间复杂度： $O(n \log k)$ – k 为归并路数

空间复杂度： $O(k)$ – 缓冲区大小

"""

```

class ExternalSort:
    @staticmethod
    def multiway_merge(chunks: List[List[int]]) -> List[int]:
        """
        模拟多路归并排序
        
```

Args:

chunks: 多个有序数据块

Returns:

合并后的有序数组

"""

```

if not chunks:
    return []

```

使用优先队列进行多路归并

heap = []

初始化每个数据块的指针

pointers = [0] * len(chunks)

total_size = sum(len(chunk) for chunk in chunks)

将每个数据块的第一个元素加入堆

```

for i, chunk in enumerate(chunks):
    if chunk:
        heapq.heappush(heap, (chunk[0], i, 0))
        pointers[i] = 1

    result = []

# 多路归并
while heap:
    value, chunk_index, element_index = heapq.heappop(heap)
    result.append(value)

    element_index += 1
    if element_index < len(chunks[chunk_index]):
        heapq.heappush(heap, (chunks[chunk_index][element_index], chunk_index,
element_index))

return result

```

"""

测试所有扩展题目

"""

@staticmethod

def test_all_problems():
 """测试所有扩展题目"""
 print("== 扩展排序题目测试开始 ==")

测试题目 1: 合并两个有序数组

print("\n 题目 1: 合并两个有序数组")

nums1 = [1, 2, 3, 0, 0, 0]

nums2 = [2, 5, 6]

ExtendedSortProblems.merge_sorted_arrays(nums1, 3, nums2, 3)

print(f"合并结果: {nums1}")

测试题目 2: 最接近原点的 K 个点

print("\n 题目 2: 最接近原点的 K 个点")

points = [[1, 3], [-2, 2], [5, 8], [0, 1]]

result2 = ExtendedSortProblems.k_closest(points, 2)

print(f"最接近的 2 个点: {result2}")

测试题目 3: 距离相等的条形码

print("\n 题目 3: 距离相等的条形码")

barcodes = [1, 1, 1, 2, 2, 2]

```
result3 = ExtendedSortProblems.rearrange_barcodes(barcodes)
print(f"重新排列结果: {result3}")
```

```
# 测试题目 4: 摆动排序 II
print("\n题目 4: 摆动排序 II")
nums4 = [1, 5, 1, 1, 6, 4]
ExtendedSortProblems.wiggle_sort(nums4)
print(f"摆动排序结果: {nums4}")
```

```
# 测试题目 5: 摆动排序
print("\n题目 5: 摆动排序")
nums5 = [3, 5, 2, 1, 6, 4]
ExtendedSortProblems.wiggle_sort_280(nums5)
print(f"摆动排序结果: {nums5}")
```

```
# 测试题目 6: 翻转对
print("\n题目 6: 翻转对")
nums6 = [1, 3, 2, 3, 1]
count6 = ExtendedSortProblems.reverse_pairs_493(nums6)
print(f"翻转对数量: {count6}")
```

```
# 测试题目 7: 把数组排成最小的数
print("\n题目 7: 把数组排成最小的数")
nums7 = [10, 2]
result7 = ExtendedSortProblems.min_number(nums7)
print(f"最小数字: {result7}")
```

```
# 测试题目 8: 逆序对计数
print("\n题目 8: 逆序对计数")
nums8 = [2, 4, 1, 3, 5]
count8 = ExtendedSortProblems.count_inversions(nums8)
print(f"逆序对数量: {count8}")
```

```
# 测试题目 9: 牛客网排序
print("\n题目 9: 牛客网排序")
nums9 = [3, 1, 4, 1, 5, 9, 2, 6]
result9 = ExtendedSortProblems.sort_array_nc140(nums9)
print(f"排序结果: {result9}")
```

```
# 测试题目 10: 外部排序
print("\n题目 10: 外部排序模拟")
chunks = [
    [1, 3, 5],
```

```
[2, 4, 6],  
[0, 7, 8]  
]  
result10 = ExtendedSortProblems.ExternalSort.multiway_merge(chunks)  
print(f"多路归并结果: {result10}")  
  
print("\n==== 扩展排序题目测试结束 ===")  
  
if __name__ == "__main__":  
    try:  
        ExtendedSortProblems.test_all_problems()  
    except Exception as e:  
        print(f"测试过程中出现错误: {e}")  
        import traceback  
        traceback.print_exc()  
  
=====
```

文件: LanguageConversion.java

```
package class001;  
  
// 测试链接 : https://leetcode.cn/problems/sort-an-array/  
// 此时不要求掌握, 因为这些排序后续的课都会讲到的  
// 这里只是想说明代码语言的转换并不困难  
// 整个系列虽然都是 java 讲的, 但使用不同语言的同学听懂思路之后, 想理解代码真的不是问题  
// 语言问题并不是学习算法的障碍, 有了人工智能工具之后, 就更不是障碍了  
public class LanguageConversion {  
  
    class Solution {  
  
        public static int[] sortArray(int[] nums) {  
            if (nums.length > 1) {  
                mergeSort(nums);  
            }  
            return nums;  
        }  
  
        public static int MAXN = 50001;  
  
        // 以下是归并排序  
        public static int[] help = new int[MAXN];  
    }  
}
```

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

```
public static void merge(int[] nums, int l, int m, int r) {  
    int p1 = l;  
    int p2 = m + 1;  
    int i = l;  
    while (p1 <= m && p2 <= r) {  
        help[i++] = nums[p1] <= nums[p2] ? nums[p1++] : nums[p2++];  
    }  
    while (p1 <= m) {  
        help[i++] = nums[p1++];  
    }  
    while (p2 <= r) {  
        help[i++] = nums[p2++];  
    }  
    for (i = l; i <= r; i++) {  
        nums[i] = help[i];  
    }  
}
```

```
// 以下是随机快速排序  
public static void quickSort(int[] arr) {  
    sort(arr, 0, arr.length - 1);  
}
```

```
public static void sort(int[] arr, int l, int r) {  
    if (l >= r) {  
        return;  
    }
```

```

// 随机这一下，常数时间比较大
// 但只有这一下随机，才能在概率上把快速排序的时间复杂度收敛到 O(n * log n)
int x = arr[1 + (int) (Math.random() * (r - 1 + 1))];
partition(arr, 1, r, x);
int left = first;
int right = last;
sort(arr, 1, left - 1);
sort(arr, right + 1, r);
}

public static int first, last;

public static void partition(int[] nums, int l, int r, int x) {
    first = l;
    last = r;
    int i = l;
    while (i <= last) {
        if (nums[i] == x) {
            i++;
        } else if (nums[i] < x) {
            swap(nums, first++, i++);
        } else {
            swap(nums, i, last--);
        }
    }
}

public static void swap(int[] arr, int i, int j) {
    int tmp = arr[i];
    arr[i] = arr[j];
    arr[j] = tmp;
}

// 以下是堆排序
public static void heapSort(int[] nums) {
    int n = nums.length;
    for (int i = n - 1; i >= 0; i--) {
        heapify(nums, i, n);
    }
    while (n > 1) {
        swap(nums, 0, --n);
        heapify(nums, 0, n);
    }
}

```

```

}

// 这个方法虽然堆排序用不上，但是堆结构里是重要方法，所以这里保留
// 后面的课会讲
public static void heapInsert(int[] nums, int i) {
    while (nums[i] > nums[(i - 1) / 2]) {
        swap(nums, i, (i - 1) / 2);
        i = (i - 1) / 2;
    }
}

public static void heapify(int[] nums, int i, int s) {
    int l = i * 2 + 1;
    while (l < s) {
        int best = l + 1 < s && nums[l + 1] > nums[l] ? l + 1 : l;
        best = nums[best] > nums[i] ? best : i;
        if (best == i) {
            break;
        }
        swap(nums, best, i);
        i = best;
        l = i * 2 + 1;
    }
}

}

}

```

}

=====

文件: SortAlgorithms.cpp

=====

```

/***
 * 排序算法实现 - C++版本
 * 包含归并排序、快速排序、堆排序的完整实现
 * 时间复杂度、空间复杂度详细分析
 * 工程化考量：异常处理、边界条件、性能优化
 *
 * 时间复杂度分析：
 * - 归并排序：O(n log n) 平均和最坏情况
 * - 快速排序：O(n log n) 平均, O(n2) 最坏
 * - 堆排序：O(n log n) 平均和最坏情况

```

*

- * 空间复杂度分析:

- * - 归并排序: $O(n)$ 需要辅助数组
- * - 快速排序: $O(\log n)$ 递归栈空间
- * - 堆排序: $O(1)$ 原地排序

*

- * 稳定性分析:

- * - 归并排序: 稳定
- * - 快速排序: 不稳定
- * - 堆排序: 不稳定

*

- * 题目相关:

- * - 912. 排序数组: <https://leetcode.cn/problems/sort-an-array/>
- * - 215. 数组中的第 K 个最大元素: <https://leetcode.cn/problems/kth-largest-element-in-an-array/>
- * - 75. 颜色分类: <https://leetcode.cn/problems/sort-colors/>
- * - 56. 合并区间: <https://leetcode.cn/problems/merge-intervals/>
- * - ALDS1_2_A: Bubble Sort: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_2_A
- * - ALDS1_2_B: Selection Sort: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_2_B
- * - ALDS1_2_C: Stable Sort: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_2_C
- * - ALDS1_2_D: Shell Sort: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_2_D
- * - ALDS1_5_B: Merge Sort: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_5_B
- * - ALDS1_5_D: The Number of Inversions: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_5_D
- * - ALDS1_6_B: Partition: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_6_B
- * - ALDS1_6_C: Quick Sort: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_6_C
- * - ALDS1_6_D: Minimum Cost Sort: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_6_D
- * - ALDS1_9_A: Complete Binary Tree: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_9_A
- * - ALDS1_9_B: Maximum Heap: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_9_B
- * - ALDS1_9_C: Priority Queue: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_9_C

*

- * 工程化考量:

- * - 异常处理: 对空数组、单元素数组进行特殊处理
- * - 边界条件: 处理各种边界情况, 如已排序、逆序、全相同数组

- * - 性能优化:

- * - 快速排序的小数组优化 (长度小于 16 时使用插入排序)
- * - 三数取中法选择基准值避免最坏情况
- * - 随机化避免最坏情况
- * - 归并排序提供迭代版本避免栈溢出
- * - 内存管理: 合理使用临时空间, 避免不必要的拷贝

* - 可读性：清晰的注释和命名规范

*

* 算法选择建议：

* - 小数据量：插入排序

* - 一般情况：快速排序（带优化）

* - 需要稳定排序：归并排序

* - 内存受限：堆排序

* - 最坏情况要求：堆排序

*/

```
#include <iostream>
```

```
#include <vector>
```

```
#include <algorithm>
```

```
#include <random>
```

```
#include <chrono>
```

```
#include <cassert>
```

```
#include <queue>
```

```
#include <functional>
```

```
#include <memory>
```

```
using namespace std;
```

```
class SortAlgorithms {
```

```
public:
```

```
/**
```

* 归并排序 - 递归版本

* 时间复杂度: $O(n \log n)$ - 在所有情况下都是这个复杂度，包括最好、平均和最坏情况

* 空间复杂度: $O(n)$ - 需要一个与原数组相同大小的辅助数组

* 稳定性: 稳定 - 相等元素的相对位置在排序后不会改变

*

* 算法原理:

* 1. 分治法: 将数组不断二分直到只有一个元素

* 2. 合并: 将两个有序数组合并成一个有序数组

* 3. 递归处理: 自底向上构建有序数组

*

* 适用场景:

* - 需要稳定排序

* - 链表排序

* - 外部排序（数据量大无法全部加载到内存）

*

* 相关题目:

* - ALDS1_5_B: Merge Sort: [https://judge.u-](https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_5_B)

aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_5_B

```

* - ALDS1_5_D: The Number of Inversions: https://judge.u-
aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_5_D

*
* 工程化考量:
* - 递归深度: 可能导致栈溢出, 可使用迭代版本
* - 内存使用: 需要额外 O(n) 空间
* - 缓存友好性: 相对较好
* - 可以复用辅助数组避免频繁创建销毁
*
* @param nums 待排序数组
*/
static void mergeSort(vector<int>& nums) {
    // 边界条件检查
    if (nums.size() <= 1) return;

    vector<int> temp(nums.size());
    mergeSortHelper(nums, temp, 0, nums.size() - 1);
}

private:
    static void mergeSortHelper(vector<int>& nums, vector<int>& temp, int left, int right) {
        if (left >= right) return;

        int mid = left + (right - left) / 2;

        // 分治递归
        mergeSortHelper(nums, temp, left, mid);
        mergeSortHelper(nums, temp, mid + 1, right);

        // 合并有序数组
        merge(nums, temp, left, mid, right);
    }

    static void merge(vector<int>& nums, vector<int>& temp, int left, int mid, int right) {
        int i = left, j = mid + 1, k = left;

        // 复制到临时数组
        for (int idx = left; idx <= right; idx++) {
            temp[idx] = nums[idx];
        }

        // 合并两个有序数组
        while (i <= mid && j <= right) {

```

```

    // 注意: 使用<=保证稳定性
    if (temp[i] <= temp[j]) {
        nums[k++] = temp[i++];
    } else {
        nums[k++] = temp[j++];
    }
}

// 处理剩余元素
while (i <= mid) {
    nums[k++] = temp[i++];
}
while (j <= right) {
    nums[k++] = temp[j++];
}
}

public:
/***
 * 归并排序 - 迭代版本 (自底向上)
 * 避免递归调用, 节省栈空间
 * 时间复杂度: O(n log n)
 * 空间复杂度: O(n)
 * 稳定性: 稳定
 *
 * 相关题目:
 * - ALDS1_5_B: Merge Sort: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\_5\_B
 *
 * 工程化考量:
 * - 避免递归深度过大导致的栈溢出
 * - 适合处理大数据集
 * - 缓存局部性可能不如递归版本
 */
static void mergeSortIterative(vector<int>& nums) {
    int n = nums.size();
    if (n <= 1) return;

    vector<int> temp(n);

    // 从 1 开始, 每次倍增
    for (int size = 1; size < n; size *= 2) {
        for (int left = 0; left < n - size; left += 2 * size) {

```

```

        int mid = left + size - 1;
        int right = min(left + 2 * size - 1, n - 1);
        merge(nums, temp, left, mid, right);
    }
}

/**
 * 快速排序 - 基础版本
 * 时间复杂度: O(n log n) 平均, O(n2) 最坏
 * 空间复杂度: O(log n) 平均, O(n) 最坏 (递归栈)
 * 稳定性: 不稳定
 *
 * 优化策略: 三数取中、随机化、小数组优化
 *
 * 相关题目:
 * - ALDS1_6_B: Partition: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\_6\_B
 * - ALDS1_6_C: Quick Sort: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\_6\_C
 *
 * 工程化考量:
 * - 平均性能优秀, 实际应用广泛
 * - 最坏情况需要避免, 可通过随机化或三数取中
 * - 小数组可使用插入排序优化
 */
static void quickSort(vector<int>& nums) {
    if (nums.size() <= 1) return;

    // 随机化避免最坏情况
    random_device rd;
    mt19937 gen(rd());
    shuffle(nums.begin(), nums.end(), gen);

    quickSortHelper(nums, 0, nums.size() - 1);
}

private:
    static void quickSortHelper(vector<int>& nums, int low, int high) {
        // 小数组优化: 使用插入排序
        if (high - low + 1 <= 16) {
            insertionSort(nums, low, high);
            return;
        }
    }
}

```

```

    }

    if (low < high) {
        int pivotIndex = partition(nums, low, high);
        quickSortHelper(nums, low, pivotIndex - 1);
        quickSortHelper(nums, pivotIndex + 1, high);
    }
}

static int partition(vector<int>& nums, int low, int high) {
    // 三数取中法选择基准
    int mid = low + (high - low) / 2;
    if (nums[mid] < nums[low]) swap(nums[low], nums[mid]);
    if (nums[high] < nums[low]) swap(nums[low], nums[high]);
    if (nums[high] < nums[mid]) swap(nums[mid], nums[high]);

    int pivot = nums[mid];
    swap(nums[mid], nums[high]); // 将基准放到最后

    int i = low;
    for (int j = low; j < high; j++) {
        if (nums[j] <= pivot) {
            swap(nums[i], nums[j]);
            i++;
        }
    }
    swap(nums[i], nums[high]);
    return i;
}

/**
 * 快速排序 - 三路划分版本
 * 针对大量重复元素的优化
 * 时间复杂度: O(n log n) 平均
 * 空间复杂度: O(log n) 平均
 * 稳定性: 不稳定
 *
 * 工程化考量:
 * - 对大量重复元素的数据集性能优秀
 * - 相比普通快排，减少了不必要的比较和交换
 */
static void quickSort3Way(vector<int>& nums, int low, int high) {
    if (low >= high) return;
}

```

```

// 随机选择基准
random_device rd;
mt19937 gen(rd());
uniform_int_distribution<> dis(low, high);
int pivotIndex = dis(gen);
int pivot = nums[pivotIndex];

int lt = low;      // 小于区域的右边界
int gt = high;    // 大于区域的左边界
int i = low;       // 当前指针

while (i <= gt) {
    if (nums[i] < pivot) {
        swap(nums[lt++], nums[i++]);
    } else if (nums[i] > pivot) {
        swap(nums[i], nums[gt--]);
    } else {
        i++;
    }
}

quickSort3Way(nums, low, lt - 1);
quickSort3Way(nums, gt + 1, high);
}

/***
 * 插入排序 - 用于小数组优化
 * 时间复杂度: O(n2) 最坏, O(n) 最好 (已排序)
 * 空间复杂度: O(1)
 * 稳定性: 稳定
 *
 * 工程化考量:
 * - 对小数组效率高, 常数因子小
 * - 稳定排序算法
 * - 适合部分有序的数据
 */
static void insertionSort(vector<int>& nums, int low, int high) {
    for (int i = low + 1; i <= high; i++) {
        int key = nums[i];
        int j = i - 1;

        // 将大于 key 的元素向右移动

```

```

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

public:
    /**
     * 堆排序
     * 时间复杂度: O(n log n) - 在所有情况下都是这个复杂度
     * 空间复杂度: O(1) - 原地排序算法
     * 稳定性: 不稳定 - 元素的相对位置可能改变
     *
     * 优势: 原地排序、最坏情况 O(n log n)
     * 劣势: 缓存不友好、常数项较大
     *
     * 相关题目:
     * - ALDS1_9_A: Complete Binary Tree: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\_9\_A
     * - ALDS1_9_B: Maximum Heap: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\_9\_B
     * - ALDS1_9_C: Priority Queue: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\_9\_C
     *
     * 工程化考量:
     * - 原地排序, 内存使用效率高
     * - 最坏情况时间复杂度有保证
     * - 常数因子相对较大, 实际性能可能不如快速排序
     * - 缓存局部性较差
    */
static void heapSort(vector<int>& nums) {
    int n = nums.size();
    if (n <= 1) return;

    // 构建最大堆
    for (int i = n / 2 - 1; i >= 0; i--) {
        heapify(nums, n, i);
    }

    // 逐个提取最大元素
    for (int i = n - 1; i > 0; i--) {

```

```

        swap(nums[0], nums[i]); // 将当前最大元素移到末尾
        heapify(nums, i, 0); // 调整剩余堆
    }
}

private:
    static void heapify(vector<int>& nums, int n, int i) {
        int largest = i; // 初始化最大元素为根
        int left = 2 * i + 1; // 左子节点
        int right = 2 * i + 2; // 右子节点

        // 如果左子节点大于根
        if (left < n && nums[left] > nums[largest]) {
            largest = left;
        }

        // 如果右子节点大于当前最大
        if (right < n && nums[right] > nums[largest]) {
            largest = right;
        }

        // 如果最大元素不是根
        if (largest != i) {
            swap(nums[i], nums[largest]);
            // 递归调整受影响的子树
            heapify(nums, n, largest);
        }
    }
}

public:
    /**
     * 冒泡排序 - 基础版本 (用于对比)
     * 时间复杂度: O(n2)
     * 空间复杂度: O(1)
     * 稳定性: 稳定
     *
     * 工程化考量:
     * - 简单易懂, 适合教学
     * - 性能较差, 不适用于实际项目
     * - 稳定排序算法
     */
    static void bubbleSort(vector<int>& nums) {
        int n = nums.size();

```

```

    for (int i = 0; i < n - 1; i++) {
        bool swapped = false;
        for (int j = 0; j < n - i - 1; j++) {
            if (nums[j] > nums[j + 1]) {
                swap(nums[j], nums[j + 1]);
                swapped = true;
            }
        }
        // 如果没有交换，说明已经有序
        if (!swapped) break;
    }
}

```

```

/**
 * 选择排序 - 基础版本（用于对比）
 * 时间复杂度: O(n2)
 * 空间复杂度: O(1)
 * 稳定性: 不稳定
 *
 * 工程化考量:
 * - 简单易懂，适合教学
 * - 性能较差，不适用于实际项目
 * - 不稳定排序算法
 * - 交换次数少，适合交换代价高的场景
 */

```

```

static void selectionSort(vector<int>& nums) {
    int n = nums.size();
    for (int i = 0; i < n - 1; i++) {
        int minIndex = i;
        for (int j = i + 1; j < n; j++) {
            if (nums[j] < nums[minIndex]) {
                minIndex = j;
            }
        }
        swap(nums[i], nums[minIndex]);
    }
}

```

```

/**
 * 测试函数 - 验证排序算法的正确性
 */
static void testAllAlgorithms() {
    cout << "==== 排序算法测试 ===" << endl;
}

```

```

// 测试用例
vector<vector<int>> testCases = {
    {},                                // 空数组
    {1},                                // 单元素
    {3, 1, 2},                           // 小数组
    {5, 2, 8, 1, 9, 3},                 // 中等数组
    {1, 2, 3, 4, 5},                   // 已排序
    {5, 4, 3, 2, 1},                   // 逆序
    {4, 2, 2, 8, 3, 3, 1},              // 重复元素
    {1, 1, 1, 1, 1}                    // 全相同
};

vector<string> algorithmNames = {
    "归并排序", "快速排序", "堆排序", "冒泡排序", "选择排序"
};

vector<function<void(vector<int>&)>> algorithms = {
    mergeSort, quickSort, heapSort, bubbleSort, selectionSort
};

for (int i = 0; i < testCases.size(); i++) {
    cout << "\n测试用例 " << (i + 1) << ": ";
    printVector(testCases[i]);

    for (int j = 0; j < algorithms.size(); j++) {
        vector<int> arr = testCases[i];
        vector<int> expected = testCases[i];
        sort(expected.begin(), expected.end()); // 标准库排序作为基准

        try {
            algorithms[j](arr);
            bool correct = isSorted(arr) && arr == expected;
            cout << algorithmNames[j] << ": "
                << (correct ? "✓" : "✗") << " ";
        } catch (const exception& e) {
            cout << algorithmNames[j] << ": 异常(" << e.what() << ")";
        }
    }
    cout << endl;
}
}

```

```
/**  
 * 性能测试 - 比较不同算法的执行时间  
 */  
  
static void performanceTest() {  
    cout << "\n==== 性能测试 ===" << endl;  
  
    vector<int> sizes = {100, 1000, 10000, 50000};  
    vector<string> algorithmNames = {  
        "归并排序", "快速排序", "堆排序", "std::sort"  
    };  
  
    vector<function<void(vector<int>&)>> algorithms = {  
        mergeSort, quickSort, heapSort,  
        [](vector<int>& arr) { sort(arr.begin(), arr.end()); }  
    };  
  
    random_device rd;  
    mt19937 gen(rd());  
  
    for (int size : sizes) {  
        cout << "\n数据规模: " << size << endl;  
  
        // 生成随机测试数据  
        vector<int> testData(size);  
        uniform_int_distribution<int> dis(0, size * 10);  
        for (int i = 0; i < size; i++) {  
            testData[i] = dis(gen);  
        }  
  
        for (int i = 0; i < algorithms.size(); i++) {  
            vector<int> arr = testData;  
            auto start = chrono::high_resolution_clock::now();  
  
            algorithms[i](arr);  
  
            auto end = chrono::high_resolution_clock::now();  
            auto duration = chrono::duration_cast<chrono::microseconds>(end - start);  
  
            cout << algorithmNames[i] << ":" << duration.count() << " μs" << endl;  
        }  
    }  
}
```

```

private:
    /**
     * 辅助函数: 检查数组是否已排序
     */
    static bool isSorted(const vector<int>& nums) {
        for (int i = 1; i < nums.size(); i++) {
            if (nums[i] < nums[i - 1]) {
                return false;
            }
        }
        return true;
    }

    /**
     * 辅助函数: 打印数组
     */
    static void printVector(const vector<int>& nums) {
        cout << "[";
        for (int i = 0; i < nums.size(); i++) {
            cout << nums[i];
            if (i < nums.size() - 1) cout << ", ";
        }
        cout << "]";
    }
};

/**
 * 扩展问题: 第 K 大元素 (快速选择算法)
 */
class KthLargest {
public:
    /**
     * 快速选择算法 - 寻找第 K 大的元素
     * 时间复杂度: O(n) 平均, O(n2) 最坏
     * 空间复杂度: O(1)
     *
     * 工程化考量:
     * - 平均性能优秀, 是 Top-K 问题的最优解法
     * - 可通过随机化避免最坏情况
     * - 原地操作, 空间效率高
     */
    static int findKthLargest(vector<int>& nums, int k) {
        if (nums.empty() || k < 1 || k > nums.size()) {

```

```

        throw invalid_argument("Invalid input");
    }

    int left = 0, right = nums.size() - 1;
    k = nums.size() - k; // 转换为第 k 小的索引

    random_device rd;
    mt19937 gen(rd());
}

while (left <= right) {
    uniform_int_distribution<int> dis(left, right);
    int pivotIndex = dis(gen);
    int pivotPos = partition(nums, left, right, pivotIndex);

    if (pivotPos == k) {
        return nums[pivotPos];
    } else if (pivotPos < k) {
        left = pivotPos + 1;
    } else {
        right = pivotPos - 1;
    }
}

return -1; // 不会执行到这里
}

private:
static int partition(vector<int>& nums, int left, int right, int pivotIndex) {
    int pivotValue = nums[pivotIndex];
    swap(nums[pivotIndex], nums[right]);

    int storeIndex = left;
    for (int i = left; i < right; i++) {
        if (nums[i] < pivotValue) {
            swap(nums[storeIndex], nums[i]);
            storeIndex++;
        }
    }
    swap(nums[storeIndex], nums[right]);
    return storeIndex;
}
};

```

```
/**  
 * 扩展问题：颜色分类（荷兰国旗问题）  
 */  
  
class SortColors {  
public:  
    /**  
     * 三指针法解决荷兰国旗问题  
     * 时间复杂度：O(n)  
     * 空间复杂度：O(1)  
     *  
     * 工程化考量：  
     * - 一次遍历完成排序，效率高  
     * - 原地操作，空间效率高  
     * - 适合处理有限类别数据的排序  
     */  
  
    static void sortColors(vector<int>& nums) {  
        if (nums.size() <= 1) return;  
  
        int left = 0;           // 0 的右边界  
        int right = nums.size() - 1; // 2 的左边界  
        int current = 0;         // 当前指针  
  
        while (current <= right) {  
            if (nums[current] == 0) {  
                swap(nums[left], nums[current]);  
                left++;  
                current++;  
            } else if (nums[current] == 2) {  
                swap(nums[current], nums[right]);  
                right--;  
                // current 不增加，需要检查交换过来的元素  
            } else {  
                current++;  
            }  
        }  
    }  
};  
  
// 主函数 - 测试所有功能  
int main() {  
    try {  
        // 测试基本排序算法  
        SortAlgorithms::testAllAlgorithms();  
    }
```

```
// 性能测试
SortAlgorithms::performanceTest();

// 测试扩展问题
cout << "\n==== 扩展问题测试 ===" << endl;

// 测试第 K 大元素
vector<int> nums = {3, 2, 1, 5, 6, 4};
int k = 2;
int result = KthLargest::findKthLargest(nums, k);
cout << "第" << k << "大元素: " << result << endl;

// 测试颜色分类
vector<int> colors = {2, 0, 2, 1, 1, 0};
cout << "颜色分类前: ";
// SortAlgorithms::printVector(colors);
cout << endl;

SortColors::sortColors(colors);
cout << "颜色分类后: ";
// SortAlgorithms::printVector(colors);
cout << endl;

} catch (const exception& e) {
    cerr << "错误: " << e.what() << endl;
    return 1;
}

return 0;
}

/***
 * 工程化考量总结:
 *
 * 1. 异常处理: 对空数组、非法输入进行验证
 * 2. 边界条件: 处理单元素、已排序、逆序等特殊情况
 * 3. 性能优化:
 *     - 小数组使用插入排序
 *     - 快速排序使用随机化和三数取中
 *     - 归并排序提供迭代版本避免栈溢出
 * 4. 内存管理: 合理使用临时空间, 避免不必要的拷贝
 * 5. 代码可读性: 清晰的注释和命名规范
*/
```

- * 6. 测试覆盖：包含各种边界情况和性能测试
- * 7. 算法选择：根据数据特征选择最优算法
- *
- * 时间复杂度分析：
 - * - 归并排序：稳定 $O(n \log n)$ ，适合需要稳定性的场景
 - * - 快速排序：平均 $O(n \log n)$ ，原地排序，常数项较小
 - * - 堆排序：最坏 $O(n \log n)$ ，适合对最坏情况有要求的场景
- *
- * 实际应用建议：
 - * - 小数据：插入排序或选择排序
 - * - 一般数据：快速排序（随机化版本）
 - * - 大数据且需要稳定：归并排序
 - * - 内存敏感：堆排序或快速排序
 - * - 大量重复：三路快速排序
- */

=====

文件：SortAlgorithms.java

=====

```
import java.util.Arrays;
import java.util.Random;

/**
 * 排序算法完整实现 - Java 版本
 * 包含归并排序、快速排序、堆排序的完整实现和详细注释
 *
 * 时间复杂度分析：
 * - 归并排序： $O(n \log n)$  平均和最坏情况
 * - 快速排序： $O(n \log n)$  平均， $O(n^2)$  最坏
 * - 堆排序： $O(n \log n)$  平均和最坏情况
 *
 * 空间复杂度分析：
 * - 归并排序： $O(n)$  需要辅助数组
 * - 快速排序： $O(\log n)$  递归栈空间
 * - 堆排序： $O(1)$  原地排序
 *
 * 稳定性分析：
 * - 归并排序：稳定
 * - 快速排序：不稳定
 * - 堆排序：不稳定
 *
 * 题目相关：
```

```
* - 912. 排序数组: https://leetcode.cn/problems/sort-an-array/
* - 215. 数组中的第 K 个最大元素: https://leetcode.cn/problems/kth-largest-element-in-an-array/
* - 75. 颜色分类: https://leetcode.cn/problems/sort-colors/
* - 56. 合并区间: https://leetcode.cn/problems/merge-intervals/
* - 148. 排序链表: https://leetcode.cn/problems/sort-list/
* - 剑指 Offer 51. 数组中的逆序对: https://leetcode.cn/problems/shu-zu-zhong-de-ni-xu-dui-lcof/
* - ALDS1_2_A: Bubble Sort: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\_2\_A
* - ALDS1_2_B: Selection Sort: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\_2\_B
* - ALDS1_2_C: Stable Sort: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\_2\_C
* - ALDS1_2_D: Shell Sort: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\_2\_D
* - ALDS1_5_B: Merge Sort: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\_5\_B
* - ALDS1_5_D: The Number of Inversions: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\_5\_D
* - ALDS1_6_B: Partition: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\_6\_B
* - ALDS1_6_C: Quick Sort: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\_6\_C
* - ALDS1_6_D: Minimum Cost Sort: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\_6\_D
* - ALDS1_9_A: Complete Binary Tree: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\_9\_A
* - ALDS1_9_B: Maximum Heap: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\_9\_B
* - ALDS1_9_C: Priority Queue: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\_9\_C
*
* 工程化考量:
* - 异常处理: 对空数组、单元素数组进行特殊处理
* - 边界条件: 处理各种边界情况, 如已排序、逆序、全相同数组
* - 性能优化:
*   - 快速排序的小数组优化 (长度小于 16 时使用插入排序)
*   - 三数取中法选择基准值避免最坏情况
*   - 归并排序使用全局辅助数组避免频繁创建销毁
*   - 稳定性: 归并排序保证稳定性, 快速排序和堆排序不稳定
*   - 可读性: 清晰的函数命名和详细注释
*
* 算法选择建议:
* - 数据量小 ( $n < 50$ ): 插入排序
* - 需要稳定排序: 归并排序
* - 一般情况: 快速排序 (带优化)
* - 内存受限: 堆排序
* - 最坏情况要求: 堆排序
*/
public class SortAlgorithms {
```

```

// 归并排序辅助数组，避免频繁创建销毁
private static int[] mergeHelper;

/**
 * 归并排序主函数
 * 时间复杂度: O(n log n) - 在所有情况下都是这个复杂度，包括最好、平均和最坏情况
 * 空间复杂度: O(n) - 需要一个与原数组相同大小的辅助数组
 * 稳定性: 稳定 - 相等元素的相对位置在排序后不会改变
 *
 * 算法原理:
 * 1. 分治法: 将数组不断二分直到只有一个元素
 * 2. 合并: 将两个有序数组合并成一个有序数组
 * 3. 递归处理: 自底向上构建有序数组
 *
 * 适用场景:
 * - 需要稳定排序
 * - 链表排序
 * - 外部排序 (数据量大无法全部加载到内存)
 *
 * 相关题目:
 * - ALDS1_5_B: Merge Sort: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\_5\_B
 * - ALDS1_5_D: The Number of Inversions: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\_5\_D
 *
 * 工程化考量:
 * - 对于小数组 (长度小于 16)，可考虑使用插入排序优化
 * - 可以使用迭代版本避免递归栈溢出
 * - 可以复用辅助数组避免频繁创建销毁
 *
 * @param arr 待排序数组
 */
public static void mergeSort(int[] arr) {
    // 边界条件检查: 空数组或单元素数组无需排序
    if (arr == null || arr.length <= 1) {
        return;
    }

    // 初始化辅助数组，避免在递归中频繁创建销毁
    mergeHelper = new int[arr.length];
    // 调用递归实现
    mergeSort(arr, 0, arr.length - 1);
}

```

```
/**  
 * 归并排序递归实现  
 * 核心思想：分治法，将数组分成两半分别排序，然后合并  
 *  
 * @param arr 待排序数组  
 * @param left 左边界  
 * @param right 右边界  
 */  
  
private static void mergeSort(int[] arr, int left, int right) {  
    if (left >= right) {  
        return;  
    }  
  
    int mid = left + (right - left) / 2;  
  
    // 递归排序左半部分  
    mergeSort(arr, left, mid);  
    // 递归排序右半部分  
    mergeSort(arr, mid + 1, right);  
    // 合并两个有序数组  
    merge(arr, left, mid, right);  
}  
  
/**  
 * 合并两个有序数组  
 * 关键步骤：双指针合并，保证稳定性  
 *  
 * @param arr 原数组  
 * @param left 左边界  
 * @param mid 中间位置  
 * @param right 右边界  
 */  
  
private static void merge(int[] arr, int left, int mid, int right) {  
    // 复制数据到辅助数组  
    for (int i = left; i <= right; i++) {  
        mergeHelper[i] = arr[i];  
    }  
  
    int i = left;      // 左半部分指针  
    int j = mid + 1;   // 右半部分指针  
    int k = left;      // 原数组指针  
  
    // 合并两个有序数组
```

```

        while (i <= mid && j <= right) {
            // 相等时取左边的元素，保证稳定性
            if (mergeHelper[i] <= mergeHelper[j]) {
                arr[k++] = mergeHelper[i++];
            } else {
                arr[k++] = mergeHelper[j++];
            }
        }

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

/***
 * 快速排序主函数
 * 时间复杂度：平均  $O(n \log n)$ ，最坏  $O(n^2)$ 
 * 空间复杂度： $O(\log n)$  平均情况下的递归栈空间，最坏  $O(n)$ 
 * 稳定性：不稳定 - 元素的相对位置可能改变
 *
 * 算法原理：
 * 1. 分治法：选取基准值，将数组分成两部分
 * 2. 递归处理：对左右两部分分别进行快速排序
 * 3. 合并：由于是原地排序，不需要合并操作
 *
 * 适用场景：
 * - 一般情况下的排序
 * - 内存受限时（不需要稳定排序）
 * - 数据量较大时（平均性能较好）
 *
 * 相关题目：
 * - ALDS1_6_B: Partition: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\_6\_B
 * - ALDS1_6_C: Quick Sort: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\_6\_C
 *
 * 工程化考量：
 * - 小数组优化：当数组长度较小时使用插入排序
 * - 三数取中法选择基准值，避免最坏情况
 */

```

```
* - 随机化基准选择也可避免最坏情况
* - 可以使用迭代版本避免递归栈溢出
*
* @param arr 待排序数组
*/
public static void quickSort(int[] arr) {
    // 边界条件检查：空数组或单元素数组无需排序
    if (arr == null || arr.length <= 1) {
        return;
    }
    // 调用递归实现
    quickSort(arr, 0, arr.length - 1);
}

/**
 * 快速排序递归实现
 * 核心思想：分治法，选取基准值，将数组分成两部分
 *
 * @param arr 待排序数组
 * @param left 左边界
 * @param right 右边界
*/
private static void quickSort(int[] arr, int left, int right) {
    // 递归终止条件
    if (left >= right) {
        return;
    }

    // 小数组优化：当数组长度较小时使用插入排序
    if (right - left + 1 < 16) {
        insertionSort(arr, left, right);
        return;
    }

    // 三数取中法选择基准值，避免最坏情况
    int pivot = medianOfThree(arr, left, right);

    // 分区操作
    int[] partition = partition(arr, left, right, pivot);
    int equalLeft = partition[0];
    int equalRight = partition[1];

    // 递归排序左右部分
}
```

```

        quickSort(arr, left, equalLeft - 1);
        quickSort(arr, equalRight + 1, right);
    }

/***
 * 三数取中法选择基准值
 * 优化策略：避免快速排序的最坏情况
 *
 * @param arr 数组
 * @param left 左边界
 * @param right 右边界
 * @return 基准值
 */
private static int medianOfThree(int[] arr, int left, int right) {
    int mid = left + (right - left) / 2;

    // 对左中右三个数排序
    if (arr[left] > arr[mid]) {
        swap(arr, left, mid);
    }
    if (arr[left] > arr[right]) {
        swap(arr, left, right);
    }
    if (arr[mid] > arr[right]) {
        swap(arr, mid, right);
    }

    // 将中间值放到 right-1 位置，作为基准值
    swap(arr, mid, right - 1);
    return arr[right - 1];
}

/***
 * 快速排序分区操作
 * 荷兰国旗问题变种：将数组分成小于、等于、大于基准值的三部分
 *
 * @param arr 数组
 * @param left 左边界
 * @param right 右边界
 * @param pivot 基准值
 * @return 等于区域的左右边界
 */
private static int[] partition(int[] arr, int left, int right, int pivot) {

```

```

int less = left - 1;           // 小于区域右边界
int more = right;             // 大于区域左边界
int i = left;                 // 当前指针

while (i < more) {
    if (arr[i] < pivot) {
        swap(arr, ++less, i++);
    } else if (arr[i] > pivot) {
        swap(arr, --more, i);
    } else {
        i++;
    }
}

// 将基准值放回等于区域
swap(arr, more, right - 1);

return new int[] {less + 1, more};
}

/***
 * 插入排序（用于小数组优化）
 * 时间复杂度: O(n2) 但常数项很小
 *
 * @param arr 数组
 * @param left 左边界
 * @param right 右边界
 */
private static void insertionSort(int[] arr, int left, int right) {
    for (int i = left + 1; i <= right; i++) {
        int key = arr[i];
        int j = i - 1;

        // 将大于 key 的元素向后移动
        while (j >= left && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}

/***

```

- * 堆排序主函数
- * 时间复杂度: $O(n \log n)$ – 在所有情况下都是这个复杂度
- * 空间复杂度: $O(1)$ – 原地排序算法
- * 稳定性: 不稳定 – 元素的相对位置可能改变
- *
- * 算法原理:
- * 1. 构建最大堆: 从最后一个非叶子节点开始
- * 2. 逐个提取堆顶元素: 将堆顶元素 (最大值) 与当前末尾元素交换
- * 3. 重新堆化: 对剩余元素重新堆化
- * 4. 重复步骤 2 和 3 直到堆为空
- *
- * 适用场景:
- * - 内存受限时 (不需要稳定排序)
- * - 最坏情况要求
- * - 数据量较大时 (平均性能较好)
- *
- * 相关题目:
- * - ALDS1_9_A: Complete Binary Tree: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_9_A
- * - ALDS1_9_B: Maximum Heap: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_9_B
- * - ALDS1_9_C: Priority Queue: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_9_C
- *
- * 工程化考量:
- * - 原地排序, 内存使用效率高
- * - 最坏情况时间复杂度有保证
- * - 常数因子相对较大, 实际性能可能不如快速排序
- *
- * @param arr 待排序数组
- */

```

public static void heapSort(int[] arr) {
    // 边界条件检查: 空数组或单元素数组无需排序
    if (arr == null || arr.length <= 1) {
        return;
    }

    int n = arr.length;

    // 构建最大堆: 从最后一个非叶子节点开始
    // 最后一个非叶子节点的索引是  $n/2 - 1$ 
    for (int i = n / 2 - 1; i >= 0; i--) {
        heapify(arr, n, i);
    }
}

```

```
}

// 逐个提取堆顶元素
for (int i = n - 1; i > 0; i--) {
    // 将堆顶元素（最大值）与当前末尾元素交换
    swap(arr, 0, i);
    // 对剩余元素重新堆化，堆大小减 1
    heapify(arr, i, 0);
}

/***
 * 堆化操作：维护最大堆性质
 * 核心思想：下沉操作，确保父节点大于等于子节点
 *
 * @param arr 数组
 * @param n 堆大小
 * @param i 当前节点索引
 */
private static void heapify(int[] arr, int n, int i) {
    int largest = i;           // 假设当前节点最大
    int left = 2 * i + 1;      // 左子节点
    int right = 2 * i + 2;     // 右子节点

    // 比较左子节点
    if (left < n && arr[left] > arr[largest]) {
        largest = left;
    }

    // 比较右子节点
    if (right < n && arr[right] > arr[largest]) {
        largest = right;
    }

    // 如果最大值不是当前节点，需要交换并继续堆化
    if (largest != i) {
        swap(arr, i, largest);
        heapify(arr, n, largest);
    }
}

/***
 * 交换数组元素
*/
```

```

* 基础操作，但要注意边界检查
*
* @param arr 数组
* @param i 索引 1
* @param j 索引 2
*/
private static void swap(int[] arr, int i, int j) {
    if (i == j) return; // 相同索引不需要交换

    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}

/**
* 测试函数：验证排序算法的正确性
* 包含边界测试、性能测试、稳定性测试
*/
public static void testSortAlgorithms() {
    System.out.println("==> 排序算法测试开始 ==>");

    // 测试用例设计
    int[][] testCases = {
        {},                                // 空数组
        {1},                                // 单元素
        {1, 2, 3},                           // 已排序
        {3, 2, 1},                           // 逆序
        {1, 1, 1},                           // 全相同
        {5, 2, 8, 1, 9},                    // 普通情况
        {3, 1, 4, 1, 5, 9, 2, 6}           // 重复元素
    };

    String[] algorithms = {"归并排序", "快速排序", "堆排序"};

    for (int i = 0; i < testCases.length; i++) {
        System.out.println("\n测试用例 " + (i + 1) + ":" + Arrays.toString(testCases[i]));

        for (int j = 0; j < algorithms.length; j++) {
            int[] arr = testCases[i].clone();
            int[] expected = testCases[i].clone();
            Arrays.sort(expected); // 使用系统排序作为基准

            switch (j) {

```

```

        case 0:
            mergeSort(arr);
            break;
        case 1:
            quickSort(arr);
            break;
        case 2:
            heapSort(arr);
            break;
    }

    boolean correct = Arrays.equals(arr, expected);
    System.out.printf("%s: %s - %s%n",
        algorithms[j],
        Arrays.toString(arr),
        correct ? "✓" : "✗"
    );
}

if (!correct) {
    System.out.println("预期: " + Arrays.toString(expected));
}
}

// 性能测试
performanceTest();

System.out.println("==> 排序算法测试结束 ==>");
}

/**
 * 性能测试: 比较不同排序算法在大数据量下的表现
 */
private static void performanceTest() {
    System.out.println("\n==> 性能测试 ==>");

    int size = 10000;
    int[] data = generateRandomArray(size);

    String[] algorithms = {"归并排序", "快速排序", "堆排序"};
    Runnable[] sorts = {
        () -> mergeSort(data.clone()),
        () -> quickSort(data.clone()),
        () -> heapSort(data.clone())
    };
}

```

```

() -> heapSort(data.clone())
};

for (int i = 0; i < algorithms.length; i++) {
    int[] testData = data.clone();
    long startTime = System.nanoTime();

    switch (i) {
        case 0: mergeSort(testData); break;
        case 1: quickSort(testData); break;
        case 2: heapSort(testData); break;
    }

    long endTime = System.nanoTime();
    double duration = (endTime - startTime) / 1e6; // 转换为毫秒

    System.out.printf("%s: %.2f ms%n", algorithms[i], duration);

    // 验证排序正确性
    boolean correct = isSorted(testData);
    System.out.printf(" 排序正确性: %s%n", correct ? "√" : "✗");
}
}

/**
 * 生成随机测试数组
 *
 * @param size 数组大小
 * @return 随机数组
 */
private static int[] generateRandomArray(int size) {
    Random random = new Random();
    int[] arr = new int[size];
    for (int i = 0; i < size; i++) {
        arr[i] = random.nextInt(size * 10);
    }
    return arr;
}

/**
 * 检查数组是否已排序
 *
 * @param arr 数组

```

```
* @return 是否已排序
*/
private static boolean isSorted(int[] arr) {
    for (int i = 1; i < arr.length; i++) {
        if (arr[i] < arr[i - 1]) {
            return false;
        }
    }
    return true;
}

/**
 * 主函数：演示排序算法的使用
 */
public static void main(String[] args) {
    // 基础功能演示
    int[] arr = {64, 34, 25, 12, 22, 11, 90};
    System.out.println("原始数组：" + Arrays.toString(arr));

    // 测试不同排序算法
    int[] arr1 = arr.clone();
    mergeSort(arr1);
    System.out.println("归并排序：" + Arrays.toString(arr1));

    int[] arr2 = arr.clone();
    quickSort(arr2);
    System.out.println("快速排序：" + Arrays.toString(arr2));

    int[] arr3 = arr.clone();
    heapSort(arr3);
    System.out.println("堆排序：" + Arrays.toString(arr3));

    // 运行完整测试套件
    testSortAlgorithms();

    // 运行扩展题目测试
    System.out.println("\n==== 扩展题目测试 ====");
    testExtendedProblems();
}

// =====
// 扩展题目实现部分
// =====
```

```
/**  
 * 题目 1: 912. 排序数组  
 * 来源: LeetCode  
 * 链接: https://leetcode.cn/problems/sort-an-array/  
 *  
 * 题目描述:  
 * 给定一个整数数组 nums，将该数组升序排列。  
 *  
 * 示例:  
 * 输入: nums = [5, 2, 3, 1]  
 * 输出: [1, 2, 3, 5]  
 *  
 * 约束条件:  
 *  $1 \leq \text{nums.length} \leq 5 * 10^4$   
 *  $-5 * 10^4 \leq \text{nums}[i] \leq 5 * 10^4$   
 *  
 * 思路:  
 * 可以使用快速排序、归并排序或堆排序  
 * 本题要求时间复杂度  $O(n \log n)$ ，三种算法都满足  
 *  
 * 时间复杂度:  $O(n \log n)$  - 所有情况  
 * 空间复杂度:  $O(n)$  - 归并排序需要辅助数组;  $O(\log n)$  - 快速排序递归栈;  $O(1)$  - 堆排序  
 *  
 * 是否最优解: 是。基于比较的排序算法下界是  $O(n \log n)$   
 *  
 * 相关题目:  
 * - ALDS1_5_B: Merge Sort: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\_5\_B  
 *  
 * 工程化考量:  
 * - 根据数据规模选择合适的排序算法  
 * - 可以结合多种算法的优点进行优化  
 */  
  
public static int[] sortArray(int[] nums) {  
    // 边界条件检查  
    if (nums == null || nums.length <= 1) {  
        return nums;  
    }  
  
    // 根据数据规模选择算法  
    if (nums.length < 50) {  
        // 小数组使用插入排序  
    }  
}
```

```

        insertionSort(nums, 0, nums.length - 1);
    } else {
        // 大数组使用快速排序(带优化)
        quickSort(nums);
    }

    return nums;
}

/***
 * 题目 2: 215. 数组中的第 K 个最大元素
 * 来源: LeetCode
 * 链接: https://leetcode.cn/problems/kth-largest-element-in-an-array/
 *
 * 题目描述:
 * 给定整数数组 nums 和整数 k, 请返回数组中第 k 个最大的元素。
 * 请注意, 你需要找的是数组排序后的第 k 个最大的元素, 而不是第 k 个不同的元素。
 *
 * 示例:
 * 输入: nums = [3,2,1,5,6,4], k = 2
 * 输出: 5
 *
 * 思路对比:
 * 方法 1: 完全排序后返回第 k 个 - O(n log n)
 * 方法 2: 使用最小堆维护 k 个最大元素 - O(n log k)
 * 方法 3: 快速选择算法 - O(n) 平均, O(n2) 最坏
 *
 * 时间复杂度: O(n) - 平均情况(快速选择)
 * 空间复杂度: O(1) - 原地操作
 *
 * 是否最优解: 是。快速选择是找第 K 大元素的最优算法(平均 O(n))
 *
 * 相关题目:
 * - ALDS1_6_B: Partition: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\_6\_B
 */

public static int findKthLargest(int[] nums, int k) {
    if (nums == null || nums.length == 0 || k < 1 || k > nums.length) {
        throw new IllegalArgumentException("Invalid input");
    }

    // 使用快速选择算法
    // 第 k 大 = 第 (n-k) 小 (0-based)
}

```

```
        return quickSelect(nums, 0, nums.length - 1, nums.length - k);
    }

/***
 * 快速选择算法
 * 核心思想：类似快速排序的分区操作，但只递归处理包含目标的一侧
 *
 * @param nums 数组
 * @param left 左边界
 * @param right 右边界
 * @param k 要找的第 k 小元素的索引 (0-based)
 * @return 第 k 小的元素值
 */
private static int quickSelect(int[] nums, int left, int right, int k) {
    if (left == right) {
        return nums[left];
    }

    // 随机选择 pivot，避免最坏情况
    Random rand = new Random();
    int pivotIndex = left + rand.nextInt(right - left + 1);
    swap(nums, pivotIndex, right);

    // 分区操作
    int pivot = nums[right];
    int i = left;
    for (int j = left; j < right; j++) {
        if (nums[j] < pivot) {
            swap(nums, i, j);
            i++;
        }
    }
    swap(nums, i, right);

    // 根据 pivot 位置决定下一步
    if (i == k) {
        return nums[i];
    } else if (i < k) {
        return quickSelect(nums, i + 1, right, k);
    } else {
        return quickSelect(nums, left, i - 1, k);
    }
}
```

```
/**  
 * 题目 3: 75. 颜色分类 (荷兰国旗问题)  
 * 来源: LeetCode  
 * 链接: https://leetcode.cn/problems/sort-colors/  
 *  
 * 题目描述:  
 * 给定一个包含红色、白色和蓝色，一共 n 个元素的数组，  
 * 原地对它们进行排序，使得相同颜色的元素相邻，并按照红色、白色、蓝色顺序排列。  
 * 此题中，我们使用整数 0、1 和 2 分别表示红色、白色和蓝色。  
 *  
 * 示例:  
 * 输入: nums = [2, 0, 2, 1, 1, 0]  
 * 输出: [0, 0, 1, 1, 2, 2]  
 *  
 * 进阶:  
 * - 你可以不使用代码库中的排序函数来解决这道题吗?  
 * - 你能想出一个仅使用常数空间的一趟扫描算法吗?  
 *  
 * 思路:  
 * 使用三指针法(荷兰国旗问题的经典解法):  
 * - p0: 指向下一个 0 应该放置的位置  
 * - curr: 当前遍历的位置  
 * - p2: 指向下一个 2 应该放置的位置  
 *  
 * 时间复杂度: O(n) - 一趟扫描  
 * 空间复杂度: O(1) - 原地操作  
 *  
 * 是否最优解: 是。一趟扫描，原地排序是该问题的最优解法  
 *  
 * 相关题目:  
 * - ALDS1_2_C: Stable Sort: https://judge.u-  
 aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_2_C  
 */  
  
public static void sortColors(int[] nums) {  
    if (nums == null || nums.length <= 1) {  
        return;  
    }  
  
    int p0 = 0;           // 下一个 0 的位置  
    int curr = 0;          // 当前指针  
    int p2 = nums.length - 1; // 下一个 2 的位置
```

```

while (curr <= p2) {
    if (nums[curr] == 0) {
        // 遇到 0, 与 p0 位置交换, p0 和 curr 都前进
        swap(nums, curr, p0);
        p0++;
        curr++;
    } else if (nums[curr] == 2) {
        // 遇到 2, 与 p2 位置交换, p2 后退, curr 不动(因为交换来的元素还需要判断)
        swap(nums, curr, p2);
        p2--;
    } else {
        // 遇到 1, curr 前进
        curr++;
    }
}
}

```

/**

* 题目 4: 56. 合并区间

* 来源: LeetCode

* 链接: <https://leetcode.cn/problems/merge-intervals/>

*

* 题目描述:

* 以数组 intervals 表示若干个区间的集合, 其中单个区间为 intervals[i] = [starti, endi]。

* 请你合并所有重叠的区间, 并返回一个不重叠的区间数组, 该数组需恰好覆盖输入中的所有区间。

*

* 示例:

* 输入: intervals = [[1, 3], [2, 6], [8, 10], [15, 18]]

* 输出: [[1, 6], [8, 10], [15, 18]]

* 解释: 区间 [1, 3] 和 [2, 6] 重叠, 将它们合并为 [1, 6].

*

* 思路:

* 1. 按区间起始位置排序

* 2. 遍历排序后的区间, 判断当前区间与上一个合并区间是否重叠

* 3. 重叠则合并, 不重叠则加入结果

*

* 时间复杂度: $O(n \log n)$ - 排序时间

* 空间复杂度: $O(\log n)$ - 排序的递归栈空间

*

* 是否最优解: 是。排序是必需的, 整体复杂度已达下界

*

* 相关题目:

* - ALDS1_6_D: Minimum Cost Sort: <a href="https://judge.u-

aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_6_D

```
/*
public static int[][] merge(int[][] intervals) {
    if (intervals == null || intervals.length <= 1) {
        return intervals;
    }

    // 按起始位置排序
    Arrays.sort(intervals, (a, b) -> Integer.compare(a[0], b[0]));

    java.util.List<int[]> result = new java.util.ArrayList<>();
    int[] currentInterval = intervals[0];
    result.add(currentInterval);

    for (int i = 1; i < intervals.length; i++) {
        int[] interval = intervals[i];

        // 判断是否重叠：当前区间的起始位置 <= 上一个合并区间的结束位置
        if (interval[0] <= currentInterval[1]) {
            // 重叠，合并：更新结束位置为两者最大值
            currentInterval[1] = Math.max(currentInterval[1], interval[1]);
        } else {
            // 不重叠，添加新区间
            currentInterval = interval;
            result.add(currentInterval);
        }
    }

    return result.toArray(new int[result.size()][]);
}

/***
 * 题目 5: 148. 排序链表
 * 来源: LeetCode
 * 链接: https://leetcode.cn/problems/sort-list/
 *
 * 题目描述:
 * 给你链表的头结点 head，请将其按 升序 排列并返回 排序后的链表。
 *
 * 进阶: 你可以在 O(n log n) 时间复杂度和常数级空间复杂度下，对链表进行排序吗？
 *
 * 示例:
 * 输入: head = [4,2,1,3]
```

```

* 输出: [1, 2, 3, 4]
*
* 思路:
* 归并排序特别适合链表:
* 1. 自顶向下: 递归找中点, 分割链表, 归并
* 2. 自底向上: 迭代合并, 空间 O(1)
*
* 时间复杂度: O(n log n)
* 空间复杂度: O(1) - 自底向上; O(log n) - 自顶向下的递归栈
*
* 是否最优解: 是。满足题目时间和空间要求
*
* 相关题目:
* - ALDS1_5_B: Merge Sort: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1\_5\_B
*/
public static class ListNode {
    int val;
    ListNode next;
    ListNode() {}
    ListNode(int val) { this.val = val; }
    ListNode(int val, ListNode next) { this.val = val; this.next = next; }
}

public static ListNode sortList(ListNode head) {
    if (head == null || head.next == null) {
        return head;
    }

    // 使用快慢指针找到中点
    ListNode slow = head, fast = head, prev = null;
    while (fast != null && fast.next != null) {
        prev = slow;
        slow = slow.next;
        fast = fast.next.next;
    }

    // 断开链表
    prev.next = null;

    // 递归排序两半
    ListNode left = sortList(head);
    ListNode right = sortList(slow);
}

```

```

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

/**
 * 合并两个有序链表
 */
private static ListNode mergeTwoLists(ListNode l1, ListNode l2) {
    ListNode dummy = new ListNode(0);
    ListNode curr = dummy;

    while (l1 != null && l2 != null) {
        if (l1.val <= l2.val) {
            curr.next = l1;
            l1 = l1.next;
        } else {
            curr.next = l2;
            l2 = l2.next;
        }
        curr = curr.next;
    }

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

/**
 * 题目 6：剑指 Offer 51. 数组中的逆序对
 * 来源：LeetCode
 * 链接：https://leetcode.cn/problems/shu-zu-zhong-de-ni-xu-dui-lcof/
 *
 * 题目描述：
 * 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：(7, 5), (7, 6), (7, 4), (5, 4), (6, 4)
 *
 * 思路：
 * 利用归并排序的过程统计逆序对：

```

- * 在合并两个有序数组时，如果左边元素大于右边元素，
- * 则左边该元素及之后的所有元素都与右边元素构成逆序对
- *
- * 时间复杂度: $O(n \log n)$ – 归并排序时间
- * 空间复杂度: $O(n)$ – 辅助数组
- *
- * 是否最优解: 是。暴力 $O(n^2)$ ，归并排序优化到 $O(n \log n)$ 是最优
- *
- * 相关题目:
- * - ALDS1_5_D: The Number of Inversions: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_5_D
- */

```

public static int reversePairs(int[] nums) {
    if (nums == null || nums.length <= 1) {
        return 0;
    }

    int[] temp = new int[nums.length];
    return mergeCountPairs(nums, 0, nums.length - 1, temp);
}
}

private static int mergeCountPairs(int[] nums, int left, int right, int[] temp) {
    if (left >= right) {
        return 0;
    }

    int mid = left + (right - left) / 2;
    int count = 0;

    // 递归统计左半部分和右半部分的逆序对
    count += mergeCountPairs(nums, left, mid, temp);
    count += mergeCountPairs(nums, mid + 1, right, temp);

    // 合并时统计跨越左右的逆序对
    count += mergeAndCount(nums, left, mid, right, temp);

    return count;
}

private static int mergeAndCount(int[] nums, int left, int mid, int right, int[] temp) {
    // 复制到临时数组
    for (int i = left; i <= right; i++) {
        temp[i] = nums[i];
    }
}

```

```

}

int i = left, j = mid + 1, k = left;
int count = 0;

while (i <= mid && j <= right) {
    if (temp[i] <= temp[j]) {
        nums[k++] = temp[i++];
    } else {
        // temp[i] > temp[j], 形成逆序对
        // 从 i 到 mid 的所有元素都大于 temp[j]
        count += (mid - i + 1);
        nums[k++] = temp[j++];
    }
}

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

return count;
}

/***
 * 测试扩展题目
 */
private static void testExtendedProblems() {
    // 测试题目 1: 排序数组
    System.out.println("\n题目 1: 排序数组");
    int[] arr1 = {5, 2, 3, 1};
    System.out.println("输入: " + Arrays.toString(arr1));
    sortArray(arr1);
    System.out.println("输出: " + Arrays.toString(arr1));

    // 测试题目 2: 第 K 个最大元素
    System.out.println("\n题目 2: 第 K 个最大元素");
    int[] arr2 = {3, 2, 1, 5, 6, 4};
    int k = 2;
    System.out.println("输入: " + Arrays.toString(arr2) + ", k=" + k);
    System.out.println("输出: " + findKthLargest(arr2, k));
}

```

```

// 测试题目 3: 颜色分类
System.out.println("\n 题目 3: 颜色分类");
int[] arr3 = {2, 0, 2, 1, 1, 0};
System.out.println("输入: " + Arrays.toString(arr3));
sortColors(arr3);
System.out.println("输出: " + Arrays.toString(arr3));

// 测试题目 4: 合并区间
System.out.println("\n 题目 4: 合并区间");
int[][] intervals = {{1, 3}, {2, 6}, {8, 10}, {15, 18}};
System.out.println("输入: " + Arrays.deepToString(intervals));
int[][] merged = merge(intervals);
System.out.println("输出: " + Arrays.deepToString(merged));

// 测试题目 6: 逆序对
System.out.println("\n 题目 6: 数组中的逆序对");
int[] arr6 = {7, 5, 6, 4};
System.out.println("输入: " + Arrays.toString(arr6));
System.out.println("逆序对数量: " + reversePairs(arr6.clone()));
}

}
=====

文件: SortAlgorithms.py
=====

#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""

排序算法完整实现 - Python 版本
包含归并排序、快速排序、堆排序的完整实现和详细注释

```

时间复杂度分析:

- 归并排序: $O(n \log n)$ 平均和最坏情况
- 快速排序: $O(n \log n)$ 平均, $O(n^2)$ 最坏
- 堆排序: $O(n \log n)$ 平均和最坏情况

空间复杂度分析:

- 归并排序: $O(n)$ 需要辅助数组
- 快速排序: $O(\log n)$ 递归栈空间
- 堆排序: $O(1)$ 原地排序

稳定性分析:

- 归并排序: 稳定
- 快速排序: 不稳定
- 堆排序: 不稳定

题目相关:

- 912. 排序数组: <https://leetcode.cn/problems/sort-an-array/>
- 215. 数组中的第 K 个最大元素: <https://leetcode.cn/problems/kth-largest-element-in-an-array/>
- 75. 颜色分类: <https://leetcode.cn/problems/sort-colors/>
- 56. 合并区间: <https://leetcode.cn/problems/merge-intervals/>
- 剑指 Offer 51. 数组中的逆序对: <https://leetcode.cn/problems/shu-zu-zhong-de-ni-xu-dui-lcof/>
- ALDS1_2_A: Bubble Sort: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_2_A
- ALDS1_2_B: Selection Sort: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_2_B
- ALDS1_2_C: Stable Sort: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_2_C
- ALDS1_2_D: Shell Sort: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_2_D
- ALDS1_5_B: Merge Sort: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_5_B
- ALDS1_5_D: The Number of Inversions: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_5_D
- ALDS1_6_B: Partition: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_6_B
- ALDS1_6_C: Quick Sort: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_6_C
- ALDS1_6_D: Minimum Cost Sort: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_6_D
- ALDS1_9_A: Complete Binary Tree: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_9_A
- ALDS1_9_B: Maximum Heap: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_9_B
- ALDS1_9_C: Priority Queue: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_9_C

工程化考量:

- 异常处理: 对空数组、单元素数组进行特殊处理
- 边界条件: 处理各种边界情况, 如已排序、逆序、全相同数组
- 性能优化:
 - 快速排序的小数组优化 (长度小于 16 时使用插入排序)
 - 三数取中法选择基准值避免最坏情况
 - 归并排序使用辅助数组避免频繁创建销毁
- 稳定性: 归并排序保证稳定性, 快速排序和堆排序不稳定
- 可读性: 清晰的函数命名和详细注释

算法选择建议:

- 数据量小 ($n < 50$): 插入排序
- 需要稳定排序: 归并排序
- 一般情况: 快速排序 (带优化)
- 内存受限: 堆排序

- 最坏情况要求：堆排序

"""

```
import random
import time
import sys
from typing import List, Tuple

class SortAlgorithms:
    """
    排序算法类，包含多种经典排序算法的实现
    """
```

工程化特性：

1. 所有方法都是静态方法，便于直接调用
2. 包含详细的输入验证和异常处理
3. 提供性能测试和正确性验证
4. 支持稳定性分析和演示

"""

```
@staticmethod
```

```
def merge_sort(arr: List[int]) -> None:
```

"""

归并排序主函数

时间复杂度： $O(n \log n)$ – 在所有情况下都是这个复杂度，包括最好、平均和最坏情况

空间复杂度： $O(n)$ – 需要一个与原数组相同大小的辅助数组

稳定性：稳定 – 相等元素的相对位置在排序后不会改变

算法原理：

1. 分治法：将数组不断二分直到只有一个元素
2. 合并：将两个有序数组合并成一个有序数组
3. 递归处理：自底向上构建有序数组

适用场景：

- 需要稳定排序
- 链表排序
- 外部排序（数据量大无法全部加载到内存）

相关题目：

- ALDS1_5_B: Merge Sort: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_5_B
- ALDS1_5_D: The Number of Inversions: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_5_D

工程化考量:

- 对于小数组 (长度小于 16), 可考虑使用插入排序优化
- 可以使用迭代版本避免递归栈溢出
- 可以复用辅助数组避免频繁创建销毁

Args:

arr: 待排序数组

"""

边界条件检查: 空数组或单元素数组无需排序

if len(arr) <= 1:

 return

创建辅助数组用于归并操作, 避免在递归中频繁创建销毁

helper = [0] * len(arr)

SortAlgorithms._merge_sort(arr, 0, len(arr) - 1, helper)

@staticmethod

def _merge_sort(arr: List[int], left: int, right: int, helper: List[int]) -> None:

"""

归并排序递归实现

核心思想: 分治法, 将数组分成两半分别排序, 然后合并

Args:

arr: 待排序数组

left: 左边界

right: 右边界

helper: 辅助数组

"""

递归终止条件: 子数组只有一个元素或为空

if left >= right:

 return

计算中点, 避免整数溢出

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

递归排序左半部分

SortAlgorithms._merge_sort(arr, left, mid, helper)

递归排序右半部分

SortAlgorithms._merge_sort(arr, mid + 1, right, helper)

合并两个有序数组

SortAlgorithms._merge(arr, left, mid, right, helper)

@staticmethod

```
def _merge(arr: List[int], left: int, mid: int, right: int, helper: List[int]) -> None:
    """
    合并两个有序数组
    关键步骤：双指针合并，保证稳定性
    """

    Args:
        arr: 原数组
        left: 左边界
        mid: 中间位置
        right: 右边界
        helper: 辅助数组
    """

    # 复制数据到辅助数组，避免在合并过程中覆盖未处理的数据
    for i in range(left, right + 1):
        helper[i] = arr[i]

    # 初始化三个指针：
    # i: 左半部分指针
    # j: 右半部分指针
    # k: 原数组指针
    i, j, k = left, mid + 1, left

    # 合并两个有序数组
    while i <= mid and j <= right:
        # 相等时取左边的元素，保证稳定性
        if helper[i] <= helper[j]:
            arr[k] = helper[i]
            i += 1
        else:
            arr[k] = helper[j]
            j += 1
        k += 1

    # 处理剩余元素
    # 左半部分剩余元素
    while i <= mid:
        arr[k] = helper[i]
        i += 1
        k += 1

    # 右半部分剩余元素
    while j <= right:
        arr[k] = helper[j]
```

```

j += 1
k += 1

@staticmethod
def quick_sort(arr: List[int]) -> None:
    """
    快速排序主函数
    时间复杂度: 平均  $O(n \log n)$ , 最坏  $O(n^2)$ 
    空间复杂度:  $O(\log n)$  平均情况下的递归栈空间, 最坏  $O(n)$ 
    稳定性: 不稳定 - 元素的相对位置可能改变

```

算法原理:

1. 分治法: 选取基准值, 将数组分成两部分
2. 递归处理: 对左右两部分分别进行快速排序
3. 合并: 不需要额外合并操作

适用场景:

- 一般情况下的排序
- 数据量较大

相关题目:

- ALDS1_6_B: Partition: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_6_B
- ALDS1_6_C: Quick Sort: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_6_C

工程化考量:

- 小数组优化: 当数组长度较小时使用插入排序
- 三数取中法选择基准值, 避免最坏情况
- 随机化基准选择也可避免最坏情况
- 可以使用迭代版本避免递归栈溢出

Args:

arr: 待排序数组

```

"""
# 边界条件检查: 空数组或单元素数组无需排序
if len(arr) <= 1:
    return

```

SortAlgorithms._quick_sort(arr, 0, len(arr) - 1)

@staticmethod

```
def _quick_sort(arr: List[int], left: int, right: int) -> None:
```

```

"""
快速排序递归实现
核心思想：分治法，选取基准值，将数组分成两部分

Args:
    arr: 待排序数组
    left: 左边界
    right: 右边界
"""

# 递归终止条件
if left >= right:
    return

# 小数组优化：当数组长度较小时使用插入排序
if right - left + 1 < 16:
    SortAlgorithms._insertion_sort(arr, left, right)
    return

# 三数取中法选择基准值，避免最坏情况
pivot = SortAlgorithms._median_of_three(arr, left, right)

# 分区操作
equal_left, equal_right = SortAlgorithms._partition(arr, left, right, pivot)

# 递归排序左右部分
SortAlgorithms._quick_sort(arr, left, equal_left - 1)
SortAlgorithms._quick_sort(arr, equal_right + 1, right)

@staticmethod
def _median_of_three(arr: List[int], left: int, right: int) -> int:
    """
    三数取中法选择基准值
    优化策略：避免快速排序的最坏情况

    Args:
        arr: 数组
        left: 左边界
        right: 右边界

    Returns:
        基准值
    """

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

```

```
# 对左中右三个数排序
if arr[left] > arr[mid]:
    SortAlgorithms._swap(arr, left, mid)
if arr[left] > arr[right]:
    SortAlgorithms._swap(arr, left, right)
if arr[mid] > arr[right]:
    SortAlgorithms._swap(arr, mid, right)

# 将中间值放到 right-1 位置，作为基准值
SortAlgorithms._swap(arr, mid, right - 1)
return arr[right - 1]
```

@staticmethod

```
def _partition(arr: List[int], left: int, right: int, pivot: int) -> Tuple[int, int]:
    """
```

快速排序分区操作

荷兰国旗问题变种：将数组分成小于、等于、大于基准值的三部分

Args:

- arr: 数组
- left: 左边界
- right: 右边界
- pivot: 基准值

Returns:

等于区域的左右边界

"""

```
less = left - 1          # 小于区域右边界
more = right             # 大于区域左边界
i = left                 # 当前指针
```

```
while i < more:
```

```
    if arr[i] < pivot:
        less += 1
        SortAlgorithms._swap(arr, less, i)
        i += 1
    elif arr[i] > pivot:
        more -= 1
        SortAlgorithms._swap(arr, more, i)
    else:
        i += 1
```

```
# 将基准值放回等于区域
SortAlgorithms._swap(arr, more, right - 1)

return less + 1, more

@staticmethod
def _insertion_sort(arr: List[int], left: int, right: int) -> None:
    """
    插入排序（用于小数组优化）
    时间复杂度: O(n2) 但常数项很小
    """
```

Args:

```
    arr: 数组
    left: 左边界
    right: 右边界
    """
for i in range(left + 1, right + 1):
    key = arr[i]
    j = i - 1
```

```
# 将大于 key 的元素向后移动
while j >= left and arr[j] > key:
    arr[j + 1] = arr[j]
    j -= 1
arr[j + 1] = key
```

```
@staticmethod
def heap_sort(arr: List[int]) -> None:
    """
    堆排序主函数
    时间复杂度: O(n log n) - 在所有情况下都是这个复杂度
    空间复杂度: O(1) - 原地排序算法
    稳定性: 不稳定 - 元素的相对位置可能改变
    """
```

算法原理:

1. 构建最大堆: 从最后一个非叶子节点开始
2. 逐个提取堆顶元素
3. 重新堆化: 将剩余元素重新构建成最大堆

适用场景:

- 内存受限
- 最坏情况要求

相关题目：

- ALDS1_9_A: Complete Binary Tree: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_9_A
- ALDS1_9_B: Maximum Heap: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_9_B
- ALDS1_9_C: Priority Queue: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_9_C

工程化考量：

- 原地排序，内存使用效率高
- 最坏情况时间复杂度有保证
- 常数因子相对较大，实际性能可能不如快速排序

Args:

```
arr: 待排序数组
"""
# 边界条件检查：空数组或单元素数组无需排序
if len(arr) <= 1:
    return

n = len(arr)

# 构建最大堆：从最后一个非叶子节点开始
# 最后一个非叶子节点的索引是 n//2 - 1
for i in range(n // 2 - 1, -1, -1):
    SortAlgorithms._heapify(arr, n, i)

# 逐个提取堆顶元素
for i in range(n - 1, 0, -1):
    # 将堆顶元素（最大值）与当前末尾元素交换
    SortAlgorithms._swap(arr, 0, i)
    # 对剩余元素重新堆化，堆大小减 1
    SortAlgorithms._heapify(arr, i, 0)

@staticmethod
def _heapify(arr: List[int], n: int, i: int) -> None:
"""
堆化操作：维护最大堆性质
核心思想：下沉操作，确保父节点大于等于子节点

```

Args:

```
arr: 数组
n: 堆大小
```

```
i: 当前节点索引
"""
largest = i      # 假设当前节点最大
left = 2 * i + 1 # 左子节点
right = 2 * i + 2 # 右子节点

# 比较左子节点
if left < n and arr[left] > arr[largest]:
    largest = left

# 比较右子节点
if right < n and arr[right] > arr[largest]:
    largest = right

# 如果最大值不是当前节点，需要交换并继续堆化
if largest != i:
    SortAlgorithms._swap(arr, i, largest)
    SortAlgorithms._heapify(arr, n, largest)
```

```
@staticmethod
def _swap(arr: List[int], i: int, j: int) -> None:
    """
    交换数组元素
    基础操作，但要注意边界检查
```

```
Args:
    arr: 数组
    i: 索引 1
    j: 索引 2
"""
if i == j: # 相同索引不需要交换
    return

arr[i], arr[j] = arr[j], arr[i]
```

```
@staticmethod
def generate_random_array(size: int) -> List[int]:
    """
    生成随机测试数组
```

```
Args:
    size: 数组大小
```

Returns:

随机数组

"""

```
return [random.randint(0, size * 10) for _ in range(size)]
```

@staticmethod

```
def is_sorted(arr: List[int]) -> bool:
```

"""

检查数组是否已排序

Args:

arr: 数组

Returns:

是否已排序

"""

```
for i in range(1, len(arr)):
    if arr[i] < arr[i - 1]:
        return False
return True
```

@staticmethod

```
def print_array(arr: List[int], name: str = "") -> None:
```

"""

打印数组

Args:

arr: 数组

name: 数组名称

"""

if name:

```
    print(f"{name}: ", end="")
```

```
print(arr)
```

@staticmethod

```
def test_sort_algorithms() -> None:
```

"""

测试函数：验证排序算法的正确性

包含边界测试、性能测试、稳定性测试

"""

```
print("== 排序算法测试开始 ==")
```

测试用例设计

```
test_cases = [
    [],                      # 空数组
    [1],                     # 单元素
    [1, 2, 3],               # 已排序
    [3, 2, 1],               # 逆序
    [1, 1, 1],               # 全相同
    [5, 2, 8, 1, 9],         # 普通情况
    [3, 1, 4, 1, 5, 9, 2, 6] # 重复元素
]

algorithms = ["归并排序", "快速排序", "堆排序"]
algorithm_funcs = [
    SortAlgorithms.merge_sort,
    SortAlgorithms.quick_sort,
    SortAlgorithms.heap_sort
]

for i, test_case in enumerate(test_cases):
    print(f"\n测试用例 {i + 1}: {test_case}")

    for j, (algo_name, algo_func) in enumerate(zip(algorithms, algorithm_funcs)):
        arr = test_case.copy()
        expected = sorted(test_case) # 使用内置排序作为基准

        algo_func(arr)

        correct = arr == expected
        print(f"{algo_name}: {arr} - {'✓' if correct else '✗'}")

    if not correct:
        print(f"预期: {expected}")

# 性能测试
SortAlgorithms.performance_test()

print("== 排序算法测试结束 ==")

@staticmethod
def performance_test() -> None:
    """
    性能测试: 比较不同排序算法在大数据量下的表现
    """
    print("\n== 性能测试 ==")
```

```
size = 10000
data = SortAlgorithms.generate_random_array(size)

algorithms = ["归并排序", "快速排序", "堆排序"]
algorithm_funcs = [
    SortAlgorithms.merge_sort,
    SortAlgorithms.quick_sort,
    SortAlgorithms.heap_sort
]

for algo_name, algo_func in zip(algorithms, algorithm_funcs):
    test_data = data.copy()
    start_time = time.time()

    algo_func(test_data)

    end_time = time.time()
    duration = (end_time - start_time) * 1000 # 转换为毫秒

    print(f'{algo_name}: {duration:.2f} ms')

    # 验证排序正确性
    correct = SortAlgorithms.is_sorted(test_data)
    print(f' 排序正确性: {'✓' if correct else '✗'}')

@staticmethod
def demonstrate_stability() -> None:
    """
    演示排序算法的稳定性
    稳定性: 相等元素的相对顺序在排序后保持不变
    """
    print("\n==== 稳定性测试 ====")

    # 创建包含重复元素的测试数据, 每个元素包含值和原始索引
    test_data = [
        (3, 'a'), (1, 'b'), (2, 'c'), (1, 'd'), (3, 'e'), (2, 'f')
    ]

    print("原始数据:", test_data)

    # 测试归并排序的稳定性
    merge_data = test_data.copy()
```

```
# 使用归并排序，只比较第一个元素（数值）
merge_data.sort(key=lambda x: x[0]) # Python 内置的归并排序是稳定的
print("稳定排序结果:", merge_data)

# 演示不稳定排序可能的结果
print("注意：快速排序和堆排序是不稳定的")
print("不稳定排序可能的结果示例:")
unstable_example = [
    (1, 'b'), (1, 'd'), (2, 'c'), (2, 'f'), (3, 'a'), (3, 'e')
]
print("稳定结果:", unstable_example)

# 可能的不稳定结果（相对顺序改变）
unstable_possible = [
    (1, 'd'), (1, 'b'), (2, 'f'), (2, 'c'), (3, 'e'), (3, 'a')
]
print("不稳定结果:", unstable_possible)

def main():
    """
    主函数：演示排序算法的使用
    """
    # 基础功能演示
    arr = [64, 34, 25, 12, 22, 11, 90]
    print("原始数组:", arr)

    # 测试不同排序算法
    arr1 = arr.copy()
    SortAlgorithms.merge_sort(arr1)
    print("归并排序:", arr1)

    arr2 = arr.copy()
    SortAlgorithms.quick_sort(arr2)
    print("快速排序:", arr2)

    arr3 = arr.copy()
    SortAlgorithms.heap_sort(arr3)
    print("堆排序:", arr3)

    # 运行完整测试套件，包括基础算法和扩展题目
    run_all_tests_with_extended()
```

```

# =====
# 扩展题目实现 - Python 版本
# =====

class ExtendedProblems:

    """
    排序相关的扩展题目实现
    """

    @staticmethod
    def sort_array(nums):
        """
        题目 1: 912. 排序数组
        来源: LeetCode
        链接: https://leetcode.cn/problems/sort-an-array/

        时间复杂度: O(n log n)
        空间复杂度: O(n)
        是否最优解: 是
    """

    def quick_select(nums, left, right, k):
        """
        快速选择算法
        用于找第 K 个元素
        """

        if left == right:
            return nums[left]

        # 随机选择 pivot
        import random

```

```

pivot_index = random.randint(left, right)
nums[pivot_index], nums[right] = nums[right], nums[pivot_index]

pivot = nums[right]
i = left
for j in range(left, right):
    if nums[j] < pivot:
        nums[i], nums[j] = nums[j], nums[i]
        i += 1
nums[i], nums[right] = nums[right], nums[i]

if i == k:
    return nums[i]
elif i < k:
    return ExtendedProblems.quick_select(nums, i + 1, right, k)
else:
    return ExtendedProblems.quick_select(nums, left, i - 1, k)

```

@staticmethod

def find_kth_largest(nums, k):

"""

题目 2: 215. 数组中的第 K 个最大元素

来源: LeetCode

链接: <https://leetcode.cn/problems/kth-largest-element-in-an-array/>

使用快速选择算法

时间复杂度: O(n) 平均情况

空间复杂度: O(1)

是否最优解: 是

"""

```

if not nums or k < 1 or k > len(nums):
    raise ValueError("Invalid input")

```

```

return ExtendedProblems.quick_select(nums, 0, len(nums) - 1, len(nums) - k)

```

@staticmethod

def sort_colors(nums):

"""

题目 3: 75. 颜色分类 (荷兰国旗问题)

来源: LeetCode

链接: <https://leetcode.cn/problems/sort-colors/>

三指针法

时间复杂度: $O(n)$

空间复杂度: $O(1)$

是否最优解: 是

"""

```
if not nums or len(nums) <= 1:  
    return
```

$p0 = 0$ # 下一个 0 的位置

$curr = 0$ # 当前指针

$p2 = \text{len}(nums) - 1$ # 下一个 2 的位置

while curr <= p2:

if $\text{nums}[curr] == 0$:

```
    nums[curr], nums[p0] = nums[p0], nums[curr]
```

$p0 += 1$

$curr += 1$

elif $\text{nums}[curr] == 2$:

```
    nums[curr], nums[p2] = nums[p2], nums[curr]
```

$p2 -= 1$

else:

$curr += 1$

@staticmethod

```
def merge_intervals(intervals):
```

"""

题目 4: 56. 合并区间

来源: LeetCode

链接: <https://leetcode.cn/problems/merge-intervals/>

相关题目:

- ALDS1_6_D: Minimum Cost Sort: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_6_D

时间复杂度: $O(n \log n)$

空间复杂度: $O(\log n)$

是否最优解: 是

"""

```
if not intervals or len(intervals) <= 1:  
    return intervals
```

按起始位置排序

```
intervals.sort(key=lambda x: x[0])
```

```

result = [intervals[0]]

for i in range(1, len(intervals)):
    interval = intervals[i]
    last = result[-1]

    if interval[0] <= last[1]:
        # 重叠, 合并
        last[1] = max(last[1], interval[1])
    else:
        # 不重叠, 添加新区间
        result.append(interval)

return result

@staticmethod
def merge_and_count(nums, left, mid, right, temp):
    """
    合并并统计逆序对
    """
    for i in range(left, right + 1):
        temp[i] = nums[i]

    i, j, k = left, mid + 1, left
    count = 0

    while i <= mid and j <= right:
        if temp[i] <= temp[j]:
            nums[k] = temp[i]
            i += 1
        else:
            count += (mid - i + 1)
            nums[k] = temp[j]
            j += 1
        k += 1

    while i <= mid:
        nums[k] = temp[i]
        i += 1
        k += 1

    while j <= right:
        nums[k] = temp[j]
        j += 1
        k += 1

```

```

        j += 1
        k += 1

    return count

@staticmethod
def merge_count_pairs(nums, left, right, temp):
    """
    递归统计逆序对
    """
    if left >= right:
        return 0

    mid = left + (right - left) // 2
    count = 0

    count += ExtendedProblems.merge_count_pairs(nums, left, mid, temp)
    count += ExtendedProblems.merge_count_pairs(nums, mid + 1, right, temp)
    count += ExtendedProblems.merge_and_count(nums, left, mid, right, temp)

    return count

```

```

@staticmethod
def reverse_pairs(nums):
    """
    题目 6: 剑指 Offer 51. 数组中的逆序对
    来源: LeetCode
    链接: https://leetcode.cn/problems/shu-zu-zhong-de-ni-xu-dui-lcof/

```

相关题目:

- ALDS1_5_D: The Number of Inversions: https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_5_D

使用归并排序统计逆序对

时间复杂度: $O(n \log n)$

空间复杂度: $O(n)$

是否最优解: 是

"""

```
if not nums or len(nums) <= 1:
```

```
    return 0
```

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

```
return ExtendedProblems.merge_count_pairs(nums, 0, len(nums) - 1, temp)
```

```
@staticmethod
def test():
    """
    测试所有扩展题目
    """
    print("\n==== 扩展题目测试 ===")

    # 测试题目 1: 排序数组
    print("\n题目 1: 排序数组")
    arr1 = [5, 2, 3, 1]
    print(f"输入: {arr1}")
    ExtendedProblems.sort_array(arr1)
    print(f"输出: {arr1}")

    # 测试题目 2: 第 K 个最大元素
    print("\n题目 2: 第 K 个最大元素")
    arr2 = [3, 2, 1, 5, 6, 4]
    k = 2
    print(f"输入: {arr2}, k={k}")
    result2 = ExtendedProblems.find_kth_largest(arr2.copy(), k)
    print(f"输出: {result2}")

    # 测试题目 3: 颜色分类
    print("\n题目 3: 颜色分类")
    arr3 = [2, 0, 2, 1, 1, 0]
    print(f"输入: {arr3}")
    ExtendedProblems.sort_colors(arr3)
    print(f"输出: {arr3}")

    # 测试题目 4: 合并区间
    print("\n题目 4: 合并区间")
    intervals = [[1, 3], [2, 6], [8, 10], [15, 18]]
    print(f"输入: {intervals}")
    merged = ExtendedProblems.merge_intervals(intervals)
    print(f"输出: {merged}")

    # 测试题目 6: 逆序对
    print("\n题目 6: 数组中的逆序对")
    arr6 = [7, 5, 6, 4]
    print(f"输入: {arr6}")
    result6 = ExtendedProblems.reverse_pairs(arr6.copy())
    print(f"逆序对数量: {result6}")
```

```

# =====
# 扩展题目实现 - Python 版本
# =====

class MergeKSortedLists:
    """
    题目 6: 23. 合并 K 个排序链表
    来源: LeetCode
    链接: https://leetcode.cn/problems/merge-k-sorted-lists/
    """

    解法分析:
    1. 优先队列法 (最优解) - 时间复杂度: O(n log k), 空间复杂度: O(k)
    2. 分治法 - 时间复杂度: O(n log k), 空间复杂度: O(log k)
    """

```

```

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

    @staticmethod
    def merge_k_lists_priority_queue(lists):
        """
        解法 1: 优先队列法
        时间复杂度: O(n log k) - n 是所有节点总数, k 是链表数量
        空间复杂度: O(k) - 优先队列大小
        """

        if not lists:
            return None

        import heapq

        # 优先队列
        min_heap = []

        # 将所有链表头节点加入优先队列
        for i, head in enumerate(lists):
            if head:
                # 使用元组(值, 索引, 节点)以处理相同值的情况
                heapq.heappush(min_heap, (head.val, i, head))

        return min_heap[0][2]

```

```

# 虚拟头节点
dummy = MergeKSortedLists.ListNode(0)
current = dummy

# 依次取出最小节点
while min_heap:
    val, i, node = heapq.heappop(min_heap)
    current.next = node
    current = current.next

    # 如果有后续节点，加入优先队列
    if node.next:
        heapq.heappush(min_heap, (node.next.val, i, node.next))

return dummy.next

@staticmethod
def merge_two_lists(l1, l2):
    """
    合并两个有序链表
    """

    dummy = MergeKSortedLists.ListNode(0)
    current = dummy

    while l1 and l2:
        if l1.val <= l2.val:
            current.next = l1
            l1 = l1.next
        else:
            current.next = l2
            l2 = l2.next
        current = current.next

    current.next = l1 if l1 else l2
    return dummy.next

@staticmethod
def merge_k_lists_helper(lists, left, right):
    """
    分治辅助函数
    """

    if left == right:
        return lists[left]

```

```

if left + 1 == right:
    return MergeKSortedLists.merge_two_lists(lists[left], lists[right])

mid = left + (right - left) // 2
l1 = MergeKSortedLists.merge_k_lists_helper(lists, left, mid)
l2 = MergeKSortedLists.merge_k_lists_helper(lists, mid + 1, right)

return MergeKSortedLists.merge_two_lists(l1, l2)

@staticmethod
def merge_k_lists_divide_and_conquer(lists):
    """
    解法 2: 分治法
    时间复杂度: O(n log k)
    空间复杂度: O(log k) - 递归栈空间
    """
    if not lists:
        return None
    return MergeKSortedLists.merge_k_lists_helper(lists, 0, len(lists) - 1)

@staticmethod
def print_list(head):
    """
    打印链表
    """
    result = []
    current = head
    while current:
        result.append(str(current.val))
        current = current.next
    print(' '.join(result))

@staticmethod
def test():
    """
    测试函数
    """
    print("\n==== 合并 K 个排序链表测试 ====")

    # 创建测试数据
    l1 = MergeKSortedLists.ListNode(1)
    l1.next = MergeKSortedLists.ListNode(4)
    l1.next.next = MergeKSortedLists.ListNode(5)

```

```

12 = MergeKSortedLists.ListNode(1)
12.next = MergeKSortedLists.ListNode(3)
12.next.next = MergeKSortedLists.ListNode(4)

13 = MergeKSortedLists.ListNode(2)
13.next = MergeKSortedLists.ListNode(6)

lists = [11, 12, 13]

# 测试优先队列法
result1 = MergeKSortedLists.merge_k_lists_priority_queue(lists)
print("优先队列法结果:", end=" ")
MergeKSortedLists.print_list(result1)

```

```
class MaximumGap:
```

```
"""
```

题目 7: 164. 最大间距

来源: LeetCode

链接: <https://leetcode.cn/problems/maximum-gap/>

解法分析:

1. 基数排序 (最优解) - 时间复杂度: $O(n)$, 空间复杂度: $O(n)$
2. 排序后遍历 - 时间复杂度: $O(n \log n)$

```
"""
```

```
@staticmethod
```

```
def counting_sort_by_digit(nums, exp):
```

```
"""
```

按位计数排序

```
"""
```

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

```
output = [0] * n
```

```
count = [0] * 10
```

统计每个数字出现的次数

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

```
    index = (nums[i] // exp) % 10
```

```
    count[index] += 1
```

计算累积计数

```
for i in range(1, 10):
```

```

        count[i] += count[i - 1]

# 从后向前遍历，保证稳定性
for i in range(n - 1, -1, -1):
    index = (nums[i] // exp) % 10
    output[count[index] - 1] = nums[i]
    count[index] -= 1

# 复制到原数组
for i in range(n):
    nums[i] = output[i]

@staticmethod
def radix_sort(nums):
    """
    基数排序
    """

    if not nums:
        return

    # 找出最大值
    max_val = max(nums)

    # 按位排序
    exp = 1
    while max_val // exp > 0:
        MaximumGap.counting_sort_by_digit(nums, exp)
        exp *= 10

@staticmethod
def maximum_gap_radix_sort(nums):
    """
    解法 1：基数排序
    时间复杂度：O(n)
    空间复杂度：O(n)
    """

    if len(nums) < 2:
        return 0

    # 基数排序
    sorted_nums = nums.copy()
    MaximumGap.radix_sort(sorted_nums)

```

```
# 计算最大间距
max_gap = 0
for i in range(1, len(sorted_nums)):
    max_gap = max(max_gap, sorted_nums[i] - sorted_nums[i - 1])

return max_gap

@staticmethod
def maximum_gap_sort(nums):
    """
    解法 2：排序后遍历
    时间复杂度：O(n log n)
    """
    if len(nums) < 2:
        return 0

    sorted_nums = sorted(nums)

    max_gap = 0
    for i in range(1, len(sorted_nums)):
        max_gap = max(max_gap, sorted_nums[i] - sorted_nums[i - 1])

    return max_gap

@staticmethod
def test():
    """
    测试函数
    """
    print("\n==== 最大间距测试 ====")

    nums = [3, 6, 9, 1]
    print(f"数组: {nums}")

    result1 = MaximumGap.maximum_gap_radix_sort(nums)
    result2 = MaximumGap.maximum_gap_sort(nums)

    print(f"基数排序结果: {result1}")
    print(f"普通排序结果: {result2}")

class TopKFrequentElements:
    """
```

题目 8: 347. 前 K 个高频元素

来源: LeetCode

链接: <https://leetcode.cn/problems/top-k-frequent-elements/>

解法分析:

1. 最小堆法 (最优解) - 时间复杂度: $O(n \log k)$, 空间复杂度: $O(n)$
2. 桶排序 - 时间复杂度: $O(n)$, 空间复杂度: $O(n)$

"""

```
@staticmethod
def top_k_frequent_min_heap(nums, k):
    """
    解法 1: 最小堆
    时间复杂度: O(n log k)
    空间复杂度: O(n)
    """

    if not nums or k <= 0 or k > len(nums):
        return []

    # 统计频率
    frequency_map = {}
    for num in nums:
        frequency_map[num] = frequency_map.get(num, 0) + 1

    # 最小堆
    import heapq
    min_heap = []

    # 保持堆的大小为 k
    for num, freq in frequency_map.items():
        heapq.heappush(min_heap, (freq, num))
        if len(min_heap) > k:
            heapq.heappop(min_heap)

    # 提取结果
    result = [0] * k
    for i in range(k - 1, -1, -1):
        result[i] = heapq.heappop(min_heap)[1]

    return result
```

```
@staticmethod
def top_k_frequent_bucket_sort(nums, k):
```

```

"""
解法 2: 桶排序
时间复杂度: O(n)
空间复杂度: O(n)
"""

if not nums or k <= 0 or k > len(nums):
    return []

# 统计频率
frequency_map = {}
for num in nums:
    frequency_map[num] = frequency_map.get(num, 0) + 1

# 创建桶
n = len(nums)
buckets = [[] for _ in range(n + 1)]

for num, freq in frequency_map.items():
    buckets[freq].append(num)

# 从后向前收集 k 个元素
result = []
for i in range(n, -1, -1):
    if buckets[i]:
        for num in buckets[i]:
            result.append(num)
            if len(result) == k:
                return result

return result

@staticmethod
def test():
    """
    测试函数
    """
    print("\n==== 前 K 个高频元素测试 ====")

    nums = [1, 1, 1, 2, 2, 3]
    k = 2

    print(f"数组: {nums}")
    print(f"k = {k}")

```

```
result1 = TopKFrequentElements.top_k_frequent_min_heap(nums, k)
result2 = TopKFrequentElements.top_k_frequent_bucket_sort(nums, k)

print(f"最小堆结果: {result1}")
print(f"桶排序结果: {result2}")

# 添加新的测试函数

def run_extended_tests():
    """
    运行所有扩展题目的测试
    """
    MergeKSortedLists.test()
    MaximumGap.test()
    TopKFrequentElements.test()
    ExtendedProblems.test()

def run_all_tests_with_extended():
    """
    运行所有测试，包括基础算法和扩展题目
    """
    print("===== 基础排序算法测试 =====")
    # 运行原有的测试
    SortAlgorithms.test_sort_algorithms()
    SortAlgorithms.demonstrate_stability()

    print("\n===== 扩展题目测试 =====")
    run_extended_tests()

if __name__ == "__main__":
    main()
```

文件: SortAlgorithms_part1.py

=====
排序算法实现 - Python 版本 (第一部分)
包含归并排序、快速排序的基础实现

```
"""
```

```
import random
import time
import sys
from typing import List, Callable
from heapq import heappush, heappop
```

```
class SortAlgorithms:
```

```
    """
    排序算法类 - 包含多种排序算法的 Python 实现
    每种算法都包含详细的时间复杂度分析和工程化考量
    """
```

```
@staticmethod
```

```
def merge_sort(nums: List[int]) -> List[int]:
```

```
    """
    归并排序 - 递归版本
    时间复杂度: O(n log n)
    空间复杂度: O(n)
    稳定性: 稳定
    """
```

```
    适用场景: 需要稳定排序、链表排序、外部排序
    工程考量: 递归深度、内存使用、缓存友好性
    """
```

```
# 输入验证
```

```
    if not isinstance(nums, list):
        raise TypeError("输入必须是列表")
    if not all(isinstance(x, int) for x in nums):
        raise TypeError("列表必须只包含整数")
```

```
# 边界条件处理
```

```
    if len(nums) <= 1:
        return nums.copy()
```

```
# 分治递归
```

```
    mid = len(nums) // 2
    left = SortAlgorithms.merge_sort(nums[:mid])
    right = SortAlgorithms.merge_sort(nums[mid:])
```

```
# 合并有序数组
```

```
    return SortAlgorithms._merge(left, right)
```

```

@staticmethod
def _merge(left: List[int], right: List[int]) -> List[int]:
    """合并两个有序数组"""
    result = []
    i = j = 0

    # 比较并合并
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1

    # 添加剩余元素
    result.extend(left[i:])
    result.extend(right[j:])

    return result

@staticmethod
def merge_sort_iterative(nums: List[int]) -> List[int]:
    """
    归并排序 - 迭代版本（自底向上）
    避免递归调用，节省栈空间
    时间复杂度: O(n log n)
    空间复杂度: O(n)
    """
    if len(nums) <= 1:
        return nums.copy()

    n = len(nums)
    result = nums.copy()
    temp = [0] * n

    size = 1
    while size < n:
        for left in range(0, n - size, 2 * size):
            mid = left + size - 1
            right = min(left + 2 * size - 1, n - 1)
            SortAlgorithms._merge_iterative(result, temp, left, mid, right)
        size *= 2

```

```

    return result

@staticmethod
def _merge_iterative(nums: List[int], temp: List[int], left: int, mid: int, right: int):
    """迭代版本的合并操作"""
    i, j, k = left, mid + 1, left

    # 复制到临时数组
    for idx in range(left, right + 1):
        temp[idx] = nums[idx]

    # 合并
    while i <= mid and j <= right:
        if temp[i] <= temp[j]:
            nums[k] = temp[i]
            i += 1
        else:
            nums[k] = temp[j]
            j += 1
        k += 1

    # 处理剩余元素
    while i <= mid:
        nums[k] = temp[i]
        i += 1
        k += 1

    while j <= right:
        nums[k] = temp[j]
        j += 1
        k += 1

@staticmethod
def quick_sort(nums: List[int]) -> List[int]:
    """
    快速排序 - 基础版本
    时间复杂度: O(n log n) 平均, O(n2) 最坏
    空间复杂度: O(log n) 平均, O(n) 最坏 (递归栈)
    稳定性: 不稳定
    """

    if len(nums) <= 1:
        return nums.copy()

```

```

# 小数组优化：使用插入排序
if len(nums) <= 16:
    return SortAlgorithms.insertion_sort(nums)

# 随机化避免最坏情况
nums_copy = nums.copy()
random.shuffle(nums_copy)

pivot = nums_copy[0]
left = [x for x in nums_copy[1:] if x <= pivot]
right = [x for x in nums_copy[1:] if x > pivot]

return SortAlgorithms.quick_sort(left) + [pivot] + SortAlgorithms.quick_sort(right)

@staticmethod
def quick_sort_inplace(nums: List[int], low: int = 0, high: int = None) -> None:
    """
    快速排序 - 原地排序版本
    节省空间，直接修改原数组
    """
    if high is None:
        high = len(nums) - 1

    if low >= high:
        return

    # 小数组优化
    if high - low + 1 <= 16:
        SortAlgorithms._insertion_sort_range(nums, low, high)
        return

    pivot_index = SortAlgorithms._partition(nums, low, high)
    SortAlgorithms.quick_sort_inplace(nums, low, pivot_index - 1)
    SortAlgorithms.quick_sort_inplace(nums, pivot_index + 1, high)

@staticmethod
def _partition(nums: List[int], low: int, high: int) -> int:
    """
    快速排序的分区操作"""
    # 三数取中法选择基准
    mid = (low + high) // 2
    if nums[mid] < nums[low]:
        nums[low], nums[mid] = nums[mid], nums[low]
    if nums[high] < nums[low]:

```

```

        nums[low], nums[high] = nums[high], nums[low]
        if nums[high] < nums[mid]:
            nums[mid], nums[high] = nums[high], nums[mid]

    pivot = nums[mid]
    nums[mid], nums[high] = nums[high], nums[mid]

    i = low
    for j in range(low, high):
        if nums[j] <= pivot:
            nums[i], nums[j] = nums[j], nums[i]
            i += 1

    nums[i], nums[high] = nums[high], nums[i]
    return i

@staticmethod
def quick_sort_3way(nums: List[int]) -> List[int]:
    """
    快速排序 - 三路划分版本
    针对大量重复元素的优化
    时间复杂度: O(n log n) 平均
    """
    if len(nums) <= 1:
        return nums.copy()

    pivot = random.choice(nums)

    left = [x for x in nums if x < pivot]
    middle = [x for x in nums if x == pivot]
    right = [x for x in nums if x > pivot]

    return SortAlgorithms.quick_sort_3way(left) + middle +
SortAlgorithms.quick_sort_3way(right)

```

文件: SortAlgorithms_part2.py

```

=====
"""
排序算法实现 - Python 版本 (第二部分)
包含堆排序、插入排序、冒泡排序、选择排序等
"""

```

```
import random
import time
import sys
import os
from typing import List

# 添加第一部分导入路径
sys.path.append(os.path.dirname(__file__))
from SortAlgorithms_part1 import SortAlgorithms

class SortAlgorithmsPart2:
    """排序算法第二部分 - 堆排序和其他基础排序"""

    @staticmethod
    def heap_sort(nums: List[int]) -> List[int]:
        """
        堆排序
        时间复杂度: O(n log n)
        空间复杂度: O(1) 原地排序
        稳定性: 不稳定
        """

        if len(nums) <= 1:
            return nums.copy()

        n = len(nums)
        result = nums.copy()

        # 构建最大堆
        for i in range(n // 2 - 1, -1, -1):
            SortAlgorithmsPart2._heapify(result, n, i)

        # 逐个提取最大元素
        for i in range(n - 1, 0, -1):
            result[0], result[i] = result[i], result[0]
            SortAlgorithmsPart2._heapify(result, i, 0)

        return result

    @staticmethod
    def _heapify(nums: List[int], n: int, i: int) -> None:
        """
        堆调整函数
        """
        largest = i
```

```

left = 2 * i + 1
right = 2 * i + 2

if left < n and nums[left] > nums[largest]:
    largest = left

if right < n and nums[right] > nums[largest]:
    largest = right

if largest != i:
    nums[i], nums[largest] = nums[largest], nums[i]
    SortAlgorithmsPart2._heapify(nums, n, largest)

```

```

@staticmethod
def insertion_sort(nums: List[int]) -> List[int]:
    """

```

插入排序

时间复杂度: $O(n^2)$ 最坏, $O(n)$ 最好 (已排序)

空间复杂度: $O(1)$

稳定性: 稳定

"""

```

if len(nums) <= 1:
    return nums.copy()

```

```

result = nums.copy()

```

```

for i in range(1, len(result)):

```

```

    key = result[i]

```

```

    j = i - 1

```

```

    while j >= 0 and result[j] > key:

```

```

        result[j + 1] = result[j]

```

```

        j -= 1

```

```

    result[j + 1] = key

```

```

return result

```

```

@staticmethod

```

```

def _insertion_sort_range(nums: List[int], low: int, high: int) -> None:

```

"""指定范围的插入排序"""

```

for i in range(low + 1, high + 1):

```

```

    key = nums[i]

```

```

    j = i - 1

```

```

    while j >= low and nums[j] > key:

```

```

        nums[j + 1] = nums[j]
        j -= 1
        nums[j + 1] = key

@staticmethod
def bubble_sort(nums: List[int]) -> List[int]:
    """
    冒泡排序
    时间复杂度: O(n2)
    空间复杂度: O(1)
    稳定性: 稳定
    """

    if len(nums) <= 1:
        return nums.copy()

    result = nums.copy()
    n = len(result)

    for i in range(n - 1):
        swapped = False
        for j in range(n - i - 1):
            if result[j] > result[j + 1]:
                result[j], result[j + 1] = result[j + 1], result[j]
                swapped = True

        if not swapped:
            break

    return result

@staticmethod
def selection_sort(nums: List[int]) -> List[int]:
    """
    选择排序
    时间复杂度: O(n2)
    空间复杂度: O(1)
    稳定性: 不稳定
    """

    if len(nums) <= 1:
        return nums.copy()

    result = nums.copy()
    n = len(result)

```

```

for i in range(n - 1):
    min_index = i
    for j in range(i + 1, n):
        if result[j] < result[min_index]:
            min_index = j

    result[i], result[min_index] = result[min_index], result[i]

return result
}

class SortAlgorithmsComplete(SortAlgorithmsPart2):
    """完整的排序算法类 - 继承第一部分和第二部分"""

    @staticmethod
    def test_all_algorithms() -> None:
        """测试所有排序算法的正确性"""
        print("== 排序算法测试 ==")

        test_cases = [
            [],                      # 空数组
            [1],                     # 单元素
            [3, 1, 2],               # 小数组
            [5, 2, 8, 1, 9, 3],     # 中等数组
            [1, 2, 3, 4, 5],        # 已排序
            [5, 4, 3, 2, 1],        # 逆序
            [4, 2, 2, 8, 3, 3, 1],  # 重复元素
            [1, 1, 1, 1, 1]         # 全相同
        ]

        algorithms = {
            "归并排序": SortAlgorithms.merge_sort,
            "快速排序": SortAlgorithms.quick_sort,
            "堆排序": SortAlgorithmsComplete.heap_sort,
            "插入排序": SortAlgorithmsComplete.insertion_sort,
            "冒泡排序": SortAlgorithmsComplete.bubble_sort,
            "选择排序": SortAlgorithmsComplete.selection_sort
        }

        for i, test_case in enumerate(test_cases, 1):
            print(f"\n测试用例 {i}: {test_case}")

```

```

for name, algorithm in algorithms.items():
    try:
        result = algorithm(test_case)
        expected = sorted(test_case)
        correct = result == expected
        status = "✓" if correct else "✗"
        print(f"{name}: {status}", end=" ")
    except Exception as e:
        print(f"{name}: 异常({e})", end=" ")
print()

@staticmethod
def performance_test() -> None:
    """性能测试 - 比较不同算法的执行时间"""
    print("\n==== 性能测试 ====")

    sizes = [100, 1000, 10000]
    algorithms = {
        "归并排序": SortAlgorithms.merge_sort,
        "快速排序": SortAlgorithms.quick_sort,
        "堆排序": SortAlgorithmsComplete.heap_sort,
        "内置排序": sorted
    }

    for size in sizes:
        print(f"\n数据规模: {size}")

        test_data = [random.randint(0, size * 10) for _ in range(size)]

        for name, algorithm in algorithms.items():
            start_time = time.time()
            result = algorithm(test_data)
            end_time = time.time()

            duration = (end_time - start_time) * 1000
            print(f"{name}: {duration:.2f} ms")

    @staticmethod
    def _is_sorted(nums: List[int]) -> bool:
        """检查数组是否已排序"""
        return all(nums[i] <= nums[i + 1] for i in range(len(nums) - 1))

```

```
class KthLargest:
    """第 K 大元素相关算法"""

    @staticmethod
    def find_kth_largest(nums: List[int], k: int) -> int:
        """
        快速选择算法 - 寻找第 K 大的元素
        时间复杂度: O(n) 平均, O(n2) 最坏
        空间复杂度: O(1)
        """

        if not nums or k < 1 or k > len(nums):
            raise ValueError("Invalid input parameters")

        return KthLargest._quick_select(nums, 0, len(nums) - 1, len(nums) - k)

    @staticmethod
    def _quick_select(nums: List[int], left: int, right: int, k: int) -> int:
        """
        快速选择算法的核心实现"""
        if left == right:
            return nums[left]

        pivot_index = random.randint(left, right)
        pivot_index = KthLargest._partition(nums, left, right, pivot_index)

        if k == pivot_index:
            return nums[k]
        elif k < pivot_index:
            return KthLargest._quick_select(nums, left, pivot_index - 1, k)
        else:
            return KthLargest._quick_select(nums, pivot_index + 1, right, k)

    @staticmethod
    def _partition(nums: List[int], left: int, right: int, pivot_index: int) -> int:
        """
        分区操作"""
        pivot_value = nums[pivot_index]
        nums[pivot_index], nums[right] = nums[right], nums[pivot_index]

        store_index = left
        for i in range(left, right):
            if nums[i] < pivot_value:
                nums[store_index], nums[i] = nums[i], nums[store_index]
                store_index += 1
```

```
    nums[store_index], nums[right] = nums[right], nums[store_index]
    return store_index
```

```
if __name__ == "__main__":
    # 测试所有算法
    SortAlgorithmsComplete.test_all_algorithms()
```

```
# 性能测试
SortAlgorithmsComplete.performance_test()
```

```
# 测试第 K 大元素
nums = [3, 2, 1, 5, 6, 4]
k = 2
result = KthLargest.find_kth_largest(nums, k)
print(f"\n第{k}大元素: {result}")
```

=====