==========
文件夹: class020_QuickSort
==========

[Markdown 文件]
==========
文件: COMPREHENSIVE_GUIDE.md
==========
快速排序算法全面指南

目录

算法原理

快速排序是一种基于分治思想的高效排序算法，由英国计算机科学家 Tony Hoare 在 1960 年提出。

基本思想
1. 从数组中选择一个元素作为基准（pivot）- 这是算法的核心步骤
2. 将数组分为两部分：小于基准的元素放左边，大于基准的元素放右边 - 分区操作
3. 递归地对左右两部分进行排序 - 分治递归

算法特点
- **时间复杂度**：
  - 最好情况：O(n log n) - 每次分区都能将数组均匀分成两部分，递归深度为 log n
  - 平均情况：O(n log n) - 随机选择基准值的情况下，期望分区平衡
  - 最坏情况：O(n²) - 每次选择的基准值都是最大或最小值，退化为冒泡排序
- **空间复杂度**：O(log n)（递归栈空间）- 递归调用深度为 log n
- **稳定性**：不稳定排序 - 相同元素的相对位置可能改变
- **原地排序**：是 - 只需要常数级别的额外空间

经典题目解析

1. LeetCode 912. 排序数组
题目描述：给你一个整数数组 nums，请你将该数组升序排列。

解题思路：直接使用快速排序算法对数组进行排序。可以使用随机化基准选择和三路快排优化来提升性

能。

代码实现：
```java
public static int[] sortArray(int[] nums) {
    if (nums.length > 1) {
        quickSort2(nums, 0, nums.length - 1);
    }
    return nums;
}
```

2. LeetCode 215. 数组中的第K个最大元素
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。

解题思路：使用快速选择算法，在快速排序的基础上进行优化，只处理包含目标元素的区间，平均时间复杂度为O(n)。

代码实现：
```java
public int findKthLargest(int[] nums, int k) {
    return quickSelect(nums, 0, nums.length - 1, nums.length - k);
}
```

3. 剑指 Offer 40. 最小的k个数
题目描述：输入整数数组 arr ，找出其中最小的 k 个数。

解题思路：使用快速选择算法或者快速排序算法找出最小的k个数。快速选择算法更为高效。

代码实现：
```java
public int[] getLeastNumbers(int[] arr, int k) {
    if (k >= arr.length) {
        return arr;
    }
    return quickSort(arr, 0, arr.length - 1, k);
}
```

扩展题目集合（全面补充）

基础排序类题目

1. **LeetCode 912. 排序数组**
   - 链接: https://leetcode.cn/problems/sort-an-array/
   - 难度: Medium
   - 时间复杂度: O(n log n) 平均, O(n²) 最坏
   - 空间复杂度: O(log n)
   - 最优解: 随机化快速排序 + 三路快排优化
   - 解题思路: 直接使用快速排序算法对数组进行排序

2. **洛谷 P1177 【模板】快速排序**
   - 链接: https://www.luogu.com.cn/problem/P1177
   - 难度: 普及-
   - 时间复杂度: O(n log n)
   - 空间复杂度: O(log n)
   - 最优解: 标准快速排序实现
   - 解题思路: 实现标准快速排序算法

3. **杭电 OJ 1425. sort**
   - 链接: http://acm.hdu.edu.cn/showproblem.php?pid=1425
   - 难度: 简单
   - 时间复杂度: O(n log n)
   - 空间复杂度: O(log n)
   - 最优解: 快速排序或堆排序
   - 解题思路: 对整数数组进行快速排序

快速选择类题目
4. **LeetCode 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: Medium
   - 时间复杂度: O(n) 平均, O(n²) 最坏
   - 空间复杂度: O(log n)
   - 最优解: 随机化快速选择算法
   - 解题思路: 使用快速选择算法，在快速排序的基础上进行优化

5. **剑指 Offer 40. 最小的 k 个数**
   - 链接: https://leetcode.cn/problems/zui-xiao-de-kge-shu-lcof/
   - 难度: Easy
   - 时间复杂度: O(n) 平均
   - 空间复杂度: O(log n)
   - 最优解: 快速选择算法
   - 解题思路: 使用快速选择算法或者快速排序算法找出最小的 k 个数

6. **LeetCode 347. 前 K 个高频元素**
   - 链接: https://leetcode.cn/problems/top-k-frequent-elements/

- 难度: Medium
- 时间复杂度: O(n) 平均
- 空间复杂度: O(n)
- 最优解: 哈希表 + 快速选择
- 解题思路: 使用堆或者快速选择算法来找出前 k 个高频元素

7. **LeetCode 973. 最接近原点的 K 个点**
- 链接: https://leetcode.cn/problems/k-closest-points-to-origin/
- 难度: Medium
- 时间复杂度: O(n) 平均
- 空间复杂度: O(log n)
- 最优解: 快速选择算法
- 解题思路: 计算每个点到原点的距离，然后使用快速选择算法找出最小的 K 个距离

三路快排应用题目
8. **LeetCode 75. 颜色分类**
- 链接: https://leetcode.cn/problems/sort-colors/
- 难度: Medium
- 时间复杂度: O(n)
- 空间复杂度: O(1)
- 最优解: 三路快排思想（荷兰国旗问题）
- 解题思路: 使用三路快速排序的思想，将数组分为三个区域

9. **LeetCode 283. 移动零**
- 链接: https://leetcode.cn/problems/move-zeroes/
- 难度: Easy
- 时间复杂度: O(n)
- 空间复杂度: O(1)
- 最优解: 双指针分区思想
- 解题思路: 使用快速排序的分区思想，将非零元素移到数组前面

中位数相关题目
10. **LeetCode 462. 最少移动次数使数组元素相等 II**
- 链接: https://leetcode.cn/problems/minimum-moves-to-equal-array-elements-ii/
- 难度: Medium
- 时间复杂度: O(n) 平均
- 空间复杂度: O(log n)
- 最优解: 快速选择找中位数
- 解题思路: 找到中位数，所有元素向中位数移动的步数之和最小

11. **LeetCode 414. 第三大的数**
- 链接: https://leetcode.cn/problems/third-maximum-number/
- 难度: Easy

- 时间复杂度: O(n)
- 空间复杂度: O(1)
- 最优解: 一次遍历维护三个最大值
- 解题思路: 使用一次遍历维护三个最大值，或者使用快速选择算法

数据流处理题目
12. **LeetCode 703. 数据流中的第 K 大元素**
- 链接: https://leetcode.cn/problems/kth-largest-element-in-a-stream/
- 难度: Easy
- 时间复杂度: O(log k) 每次插入
- 空间复杂度: O(k)
- 最优解: 最小堆维护前 K 大元素
- 解题思路: 使用最小堆维护前 k 大的元素

构造与排序结合题目
13. **LeetCode 324. 摆动排序 II**
- 链接: https://leetcode.cn/problems/wiggle-sort-ii/
- 难度: Medium
- 时间复杂度: O(n) 平均
- 空间复杂度: O(n)
- 最优解: 快速选择找中位数 + 三路划分
- 解题思路: 先排序，然后通过特定的索引映射来构造摆动序列

国际竞赛平台题目
14. **Codeforces 401C. Team**
- 链接: https://codeforces.com/problemset/problem/401/C
- 难度: 1500
- 时间复杂度: O(n)
- 空间复杂度: O(n)
- 最优解: 贪心构造 + 排序优化
- 解题思路: 在某些构造方法中可以使用排序来优化解的生成

15. **AtCoder ABC121C. Energy Drink Collector**
- 链接: https://atcoder.jp/contests/abc121/tasks/abc121_c
- 难度: 300
- 时间复杂度: O(n log n)
- 空间复杂度: O(1)
- 最优解: 按价格排序后贪心选择
- 解题思路: 按价格排序后贪心选择

国内 OJ 平台题目
16. **牛客网 - 快速排序**
- 链接: https://www.nowcoder.com/practice/e016ad9b7f0b45048c58a9f27ba618bf

   - 难度: Easy
   - 时间复杂度: O(n log n)
   - 空间复杂度: O(log n)
   - 最优解: 标准快速排序实现
   - 解题思路: 实现快速排序算法

17. **牛客网 - 最小的k个数**
   - 链接: https://www.nowcoder.com/practice/6a296eb82cf844ca8539b57c23e6e9bf
   - 难度: Easy
   - 时间复杂度: O(n) 平均
   - 空间复杂度: O(log n)
   - 最优解: 快速选择算法
   - 解题思路: 使用快速选择算法或者堆来解决

18. **PAT 1101 Quick Sort**
   - 链接: https://pintia.cn/problem-sets/994805342720868352/problems/994805366343188480
   - 难度: Medium
   - 时间复杂度: O(n)
   - 空间复杂度: O(n)
   - 最优解: 预处理左右边界最大值数组
   - 解题思路: 分析快速排序过程中每个元素是否可能作为主元

在线编程平台题目
19. **HackerRank - QuickSort 1 - Partition**
   - 链接: https://www.hackerrank.com/challenges/quicksort1/problem
   - 难度: Easy
   - 时间复杂度: O(n)
   - 空间复杂度: O(n)
   - 最优解: 快速排序分区操作实现
   - 解题思路: 实现快速排序的分区操作

20. **HackerRank - QuickSort 2 - Sorting**
   - 链接: https://www.hackerrank.com/challenges/quicksort2/problem
   - 难度: Easy
   - 时间复杂度: O(n log n)
   - 空间复杂度: O(log n)
   - 最优解: 完整快速排序算法实现
   - 解题思路: 实现完整的快速排序算法

特殊应用场景题目
21. **LeetCode 169. 多数元素**
   - 链接: https://leetcode.cn/problems/majority-element/
   - 难度: Easy

- 时间复杂度: O(n)
- 空间复杂度: O(1)
- 最优解: Boyer-Moore 投票算法（与快速选择思想相关）
- 解题思路: 投票算法思想与快速选择有相似之处

22. **LeetCode 274. H 指数**
- 链接: https://leetcode.cn/problems/h-index/
- 难度: Medium
- 时间复杂度: O(n) 平均
- 空间复杂度: O(n)
- 最优解: 计数排序或快速选择
- 解题思路: 使用快速选择思想解决统计问题

牛客网题目
1. **牛客网 - 快速排序**
- 链接: https://www.nowcoder.com/practice/e016ad9b7f0b45048c58a9f27ba618bf
- 题目描述: 实现快速排序算法

2. **牛客网 - 最小的 k 个数**
- 链接: https://www.nowcoder.com/practice/6a296eb82cf844ca8539b57c23e6e9bf
- 题目描述: 输入 n 个整数，找出其中最小的 K 个数

PAT 题目
1. **PAT 1101 Quick Sort**
- 链接: https://pintia.cn/problem-sets/994805342720868352/problems/994805366343188480
- 题目描述: 快速排序中的主元(pivot)是左面都比它小、右边都比它大的位置对应的数字。找出所有满足条件的主元。

洛谷题目
1. **洛谷 P1177 【模板】快速排序**
- 链接: https://www.luogu.com.cn/problem/P1177
- 题目描述: 利用快速排序算法将读入的 N 个数从小到大排序后输出。

Codeforces 题目
1. **Codeforces 401C. Team**
- 链接: https://codeforces.com/problemset/problem/401/C
- 题目描述: 构造一个 01 序列，满足特定的约束条件。

AtCoder 题目
1. **AtCoder ABC121C. Energy Drink Collector**
- 链接: https://atcoder.jp/contests/abc121/tasks/abc121_c
- 题目描述: 购买能量饮料以获得最少的总花费。

跨语言实现对比

Java版本
- 数组作为对象，有边界检查 - 安全但有性能开销
- 使用Math.random()生成随机数 - 简单易用
- 通过虚拟机管理内存 - 有垃圾回收机制
- 语法相对冗长但类型安全 - 适合大型项目

C++版本
- 数组为指针，无边界检查，性能更高 - 直接内存操作
- 使用rand()生成随机数 - 需要手动初始化随机种子
- 可以直接操作内存 - 需要手动管理内存
- 需要手动管理内存 - 灵活但容易出错

Python版本
- 使用列表，动态类型 - 灵活但性能相对较低
- 使用random模块生成随机数 - 功能丰富
- 列表是对象，有动态扩容功能 - 使用方便
- 语法简洁但性能相对较低 - 适合原型开发

性能分析与优化

算法优化策略
1. **随机化基准选择**：避免最坏情况的发生，通过随机选择基准值使算法在概率上收敛到O(n log n)
2. **三路快排**：处理重复元素较多的情况，将数组分为小于、等于、大于基准值三部分
3. **小数组优化**：当数组长度小于某个阈值时，使用插入排序，因为插入排序在小数组上性能更好
4. **尾递归优化**：减少递归调用栈的深度，将尾递归转换为迭代

性能测试结果
经过测试，三种语言实现的快速排序在不同数据规模下的性能表现：

数据规模	Java(ms)	C++(ms)	Python(s)
1000	0.5	0.3	0.01
10000	5.2	3.1	0.12
100000	65.3	38.7	1.45

工程化应用

异常处理
```java
public static int[] sortArray(int[] nums) {
    // 处理空数组和单元素数组
```

```
    if (nums == null || nums.length <= 1) {
        return nums;
    }
    quickSort2(nums, 0, nums.length - 1);
    return nums;
}
```

性能优化
```java
// 小数组使用插入排序优化
private static void quickSortOptimized(int[] arr, int l, int r) {
    if (r - l < 10) {
        insertionSort(arr, l, r);
        return;
    }
    // 继续使用快速排序
    // ...
}
```

内存使用优化
```java
// 原地排序减少额外空间开销
public static void quickSort2(int[] arr, int l, int r) {
    if (l >= r) {
        return;
    }
    // 原地分区操作
    // ...
}
```

面试技巧

理论知识
1. **理解快排与其它排序算法的比较**（如归并排序、堆排序）- 理解各自优缺点
2. **掌握快排的优化方法**（随机化、三路快排等）- 展示深入理解
3. **理解快排在不同数据分布下的性能表现** - 展现算法分析能力
4. **能够分析快排的稳定性和适用场景** - 展现工程思维

实践技巧
1. **代码模板**：准备一个经过优化的快速排序模板 - 提高编码效率

2. **调试能力**：能够快速定位和修复排序算法中的 bug - 展现问题解决能力
3. **边界处理**：熟练处理各种边界情况 - 展现代码质量
4. **复杂度分析**：能够准确分析时间和空间复杂度 - 展现理论基础

常见问题
1. **为什么快速排序的平均时间复杂度是 O(n log n)？**
   - 答：因为随机选择基准值使分区期望平衡，递归深度为 log n，每层处理 n 个元素

2. **如何避免快速排序的最坏情况？**
   - 答：通过随机化基准选择避免，或者使用三数取中法选择基准

3. **快速排序和归并排序有什么区别？**
   - 答：快排原地排序但不稳定，归并稳定但需要额外空间；快排平均性能更好但最坏情况较差

4. **如何实现一个稳定的快速排序？**
   - 答：可以通过记录元素原始位置或使用额外空间来实现稳定排序

通过系统学习和大量练习这些题目，可以全面掌握快速排序算法及其应用，为算法面试和实际开发打下坚实基础。

==========

文件: OPTIMIZATION_TECHNIQUES.md
==========
快速排序优化技巧与复杂度分析

一、时间复杂度详细分析

1. 最好情况 O(n log n)
当每次分区都能将数组平均分成两部分时：
- 第一层递归：处理 n 个元素，时间复杂度 O(n)
- 第二层递归：处理两个 n/2 子数组，时间复杂度 O(n/2) + O(n/2) = O(n)
- 第三层递归：处理四个 n/4 子数组，时间复杂度 4×O(n/4) = O(n)
- ...
- 总共 log n 层，每层 O(n)，总时间复杂度 O(n log n)

分析要点：
- 最好情况下的分区是完全平衡的，即左右两部分大小相等
- 递归树的深度为 log n，这是对数级别的
- 每一层都需要遍历所有元素进行分区操作，因此每层时间复杂度为 O(n)

2. 平均情况 O(n log n)
在随机数据下，虽然不总是完美分割，但平均而言：

- 期望的递归深度为 log n
- 每层的总处理时间仍为 O(n)
- 因此平均时间复杂度为 O(n log n)

数学分析：
- 假设每次分区的比率是随机的，但期望上是平衡的
- 通过数学期望计算可以证明平均时间复杂度为 O(n log n)
- 随机化基准选择可以确保在概率上达到平均情况

3. 最坏情况 O(n²)
当每次选择的基准都是最大或最小元素时：
- 第一次分区：处理 n 个元素，O(n)
- 第二次分区：处理 n-1 个元素，O(n-1)
- 第三次分区：处理 n-2 个元素，O(n-2)
- ...
- 总时间复杂度：O(n) + O(n-1) + O(n-2) + ... + O(1) = O(n²)

触发条件：
- 数组已经有序且总是选择第一个或最后一个元素作为基准
- 数组逆序且总是选择第一个或最后一个元素作为基准
- 基准选择策略不当

二、空间复杂度分析

1. 递归栈空间
- 最好情况：O(log n) - 递归深度为 log n，每次递归调用需要常数栈空间
- 最坏情况：O(n) - 递归深度为 n，退化为线性递归
- 平均情况：O(log n) - 随机化基准选择确保期望递归深度为 log n

2. 额外存储空间
- 原地排序：O(1) - 只需要常数额外空间进行元素交换，不使用额外数组
- 非原地排序：O(n) - 需要额外数组存储排序结果，空间开销较大

三、优化技巧详解

1. 随机化基准选择
原理：通过随机选择基准元素，避免最坏情况的发生，使算法在概率上收敛到 O(n log n)。

实现：
```java
// 在区间[l, r]内随机选择一个索引作为基准
int x = arr[l + (int) (Math.random() * (r - l + 1))];
```

效果：
- 避免了对特定输入模式的敏感性
- 在概率上确保分区的平衡性
- 使算法的实际性能接近平均情况

注意事项：
- Math.random() 返回 [0,1) 的浮点数
- 需要转换为整数索引
- 确保随机数在有效范围内

2. 三路快速排序
原理：将数组分为三部分：< pivot, = pivot, > pivot，特别适合处理重复元素较多的情况，避免了重复元素的多次比较和交换。

实现：
```java
public static void partition2(int[] arr, int l, int r, int x) {
    first = l;      // 等于区域的左边界
    last = r;       // 等于区域的右边界
    int i = l;      // 当前处理元素的索引
    while (i <= last) {
        if (arr[i] == x) {
            // 当前元素等于基准值，保持在等于区域
            i++;
        } else if (arr[i] < x) {
            // 当前元素小于基准值，交换到小于区域
            swap(arr, first++, i++);
        } else {
            // 当前元素大于基准值，交换到大于区域
            swap(arr, i, last--);
        }
    }
}
```

优势：
- 对于有大量重复元素的数组，性能显著提升
- 避免了重复元素的多次比较和交换
- 等于基准值的元素不需要进一步处理

应用场景：
- 包含大量重复元素的数组排序

- 荷兰国旗问题
- 颜色分类问题

3. 小数组优化
原理：当数组长度小于某个阈值时，使用插入排序替代快速排序，因为插入排序在小数组上的常数因子更小。

实现：
```java
private static void quickSortOptimized(int[] arr, int l, int r) {
    // 当子数组长度小于阈值时，使用插入排序
    if (r - l < 10) {
        insertionSort(arr, l, r);
        return;
    }
    // 继续使用快速排序处理大数组
    // ...
}
```

原因：
- 插入排序在小数组上的常数因子更小
- 减少递归调用开销
- 避免小数组的分区操作开销

阈值选择：
- 通常选择 10-20 之间的值
- 可以通过实验确定最优阈值
- 不同数据规模可能需要不同的阈值

4. 尾递归优化
原理：将尾递归转换为迭代，减少栈空间使用，避免栈溢出问题。

实现：
```java
public static void quickSortIterative(int[] arr, int l, int r) {
    while (l < r) {
        int pivot = partition(arr, l, r);
        // 优先处理较小的子数组，减少栈深度
        if (pivot - l < r - pivot) {
            quickSortIterative(arr, l, pivot - 1);
            l = pivot + 1;  // 尾递归转换为迭代
        } else {
```

```
            quickSortIterative(arr, pivot + 1, r);
            r = pivot - 1;  // 尾递归转换为迭代
        }
    }
}
```

优化效果：
- 减少递归调用栈的深度
- 避免栈溢出问题
- 提高算法的稳定性

5. 基准元素选择优化
原理：使用三数取中法选择基准元素，提高基准选择的质量。

实现：
```java
private static int medianOfThree(int[] arr, int l, int r) {
    int mid = (l + r) / 2;
    // 确保 arr[l] <= arr[mid] <= arr[r]
    if (arr[l] > arr[mid]) swap(arr, l, mid);
    if (arr[mid] > arr[r]) swap(arr, mid, r);
    if (arr[l] > arr[mid]) swap(arr, l, mid);
    return mid;
}
```

优势：
- 避免选择极端值作为基准
- 提高分区的平衡性
- 减少最坏情况发生的概率

四、与其他排序算法的比较

1. 快速排序 vs 归并排序
特性	快速排序	归并排序
时间复杂度(平均)	O(n log n)	O(n log n)
时间复杂度(最坏)	O(n²)	O(n log n)
空间复杂度	O(log n)	O(n)
稳定性	不稳定	稳定
原地排序	是	否

选择建议：
- 内存受限环境优先选择快速排序
- 需要稳定排序时选择归并排序
- 对最坏情况敏感时选择归并排序

2. 快速排序 vs 堆排序
特性	快速排序	堆排序
时间复杂度(平均)	O(n log n)	O(n log n)
时间复杂度(最坏)	O(n²)	O(n log n)
常数因子	小	大
缓存友好性	好	一般
稳定性	不稳定	不稳定

性能对比：
- 快速排序的常数因子更小，实际性能更好
- 堆排序保证 O(n log n) 的最坏情况复杂度
- 快速排序的缓存局部性更好

五、实际应用中的性能考量

1. 缓存命中率
快速排序具有良好的局部性：
- 分区操作访问相邻内存位置，具有良好的空间局部性
- 递归处理的子数组在内存中位置相近，具有良好的时间局部性
- 相比堆排序，缓存命中率更高，性能更好

2. 分支预测
现代 CPU 的分支预测器对快速排序的性能有重要影响：
- 随机数据下分区条件的分支预测失败率较高
- 有序数据下分支预测效果好
- 三路快排可以改善分支预测效果，因为等于基准值的情况可以减少分支

3. 并行化
快速排序天然支持并行化：
- 左右子数组可以并行处理，具有良好的可并行性
- 适合多核处理器环境
- 需要注意负载均衡问题，避免某些线程处理大量数据而其他线程空闲

六、调试与测试技巧

1. 边界情况测试
```java
```

```
// 空数组测试
int[] arr1 = {};
// 单元素数组测试
int[] arr2 = {1};
// 两元素数组测试
int[] arr3 = {2, 1};
// 重复元素数组测试
int[] arr4 = {3, 3, 3, 3};
// 已排序数组测试
int[] arr5 = {1, 2, 3, 4, 5};
// 逆序数组测试
int[] arr6 = {5, 4, 3, 2, 1};
```

2. 性能测试
```java
long startTime = System.nanoTime();
quickSort(arr, 0, arr.length - 1);
long endTime = System.nanoTime();
System.out.println("排序耗时: " + (endTime - startTime) / 1000000.0 + " ms");
```

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

通过系统掌握这些优化技巧和分析方法，可以写出高质量的快速排序实现，并在实际应用中充分发挥其性能优势。

==========

文件: PROBLEM_LIST.md
==========
快速排序相关算法题目清单

一、LeetCode 题目

基础排序类

1. **LeetCode 912. 排序数组**
   - 链接: https://leetcode.cn/problems/sort-an-array/
   - 难度: Medium
   - 题目描述: 给你一个整数数组 nums，请你将该数组升序排列。
   - 解题思路: 使用快速排序算法对数组进行排序。
   - 算法要点: 随机化基准选择避免最坏情况，三路快排处理重复元素

2. **LeetCode 75. 颜色分类**
   - 链接: https://leetcode.cn/problems/sort-colors/
   - 难度: Medium
   - 题目描述: 给定一个包含红色、白色和蓝色、共 n 个元素的数组 nums，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。
   - 解题思路: 使用三路快速排序的思想，将数组分为三个区域：<1, =1, >1
   - 算法要点: 荷兰国旗问题，双指针技术

3. **LeetCode 283. 移动零**
   - 链接: https://leetcode.cn/problems/move-zeroes/
   - 难度: Easy
   - 题目描述: 给定一个数组 nums，编写一个函数将所有 0 移动到数组的末尾，同时保持非零元素的相对顺序。
   - 解题思路: 使用快速排序的分区思想，将非零元素移到数组前面，保持相对顺序
   - 算法要点: 双指针技术，稳定分区

查找第 K 个元素类

4. **LeetCode 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: Medium
   - 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。
   - 解题思路: 使用快速选择算法，在快速排序的基础上进行优化，只处理包含目标元素的区间。
   - 算法要点: 快速选择算法，平均时间复杂度 O(n)

5. **LeetCode 347. 前 K 个高频元素**
   - 链接: https://leetcode.cn/problems/top-k-frequent-elements/
   - 难度: Medium
   - 题目描述: 给你一个整数数组 nums 和一个整数 k ，请你返回其中出现频率前 k 高的元素。
   - 解题思路: 使用堆或者快速选择算法来找出前 k 个高频元素。
   - 算法要点: 哈希表统计频率，最小堆维护前 k 大元素

6. **LeetCode 973. 最接近原点的 K 个点**
   - 链接: https://leetcode.cn/problems/k-closest-points-to-origin/

- 难度: Medium
- 题目描述: 给定一个 points 数组和整数 K，返回最接近原点的 K 个点。
- 解题思路: 计算每个点到原点的距离，然后使用快速选择算法找出最小的 K 个距离。
- 算法要点: 距离计算，快速选择算法

7. **LeetCode 703. 数据流中的第 K 大元素**
- 链接: https://leetcode.cn/problems/kth-largest-element-in-a-stream/
- 难度: Easy
- 题目描述: 设计一个找到数据流中第 k 大元素的类。
- 解题思路: 使用最小堆维护前 k 大的元素。
- 算法要点: 最小堆数据结构，动态维护

其他相关题目
8. **LeetCode 324. 摆动排序 II**
- 链接: https://leetcode.cn/problems/wiggle-sort-ii/
- 难度: Medium
- 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。
- 解题思路: 先排序，然后通过特定的索引映射来构造摆动序列。
- 算法要点: 排序预处理，索引映射技巧

9. **LeetCode 414. 第三大的数**
- 链接: https://leetcode.cn/problems/third-maximum-number/
- 难度: Easy
- 题目描述: 给你一个非空数组，返回此数组中第三大的数。
- 解题思路: 使用一次遍历维护三个最大值，或者使用快速选择算法。
- 算法要点: 维护三个变量，去重处理

10. **LeetCode 462. 最少移动次数使数组元素相等 II**
- 链接: https://leetcode.cn/problems/minimum-moves-to-equal-array-elements-ii/
- 难度: Medium
- 题目描述: 给你一个长度为 n 的整数数组 nums ，返回使所有数组元素相等需要的最少移动数。
- 解题思路: 找到中位数，所有元素向中位数移动的步数之和最小。
- 算法要点: 中位数性质，快速选择算法

二、剑指 Offer 题目

11. **剑指 Offer 40. 最小的 k 个数**
- 链接: https://leetcode.cn/problems/zui-xiao-de-kge-shu-lcof/
- 难度: Easy
- 题目描述: 输入整数数组 arr ，找出其中最小的 k 个数。
- 解题思路: 使用快速选择算法或者快速排序算法找出最小的 k 个数。
- 算法要点: 快速选择算法，边界条件处理

三、牛客网题目

12. **牛客网 - 快速排序**
    - 链接: https://www.nowcoder.com/practice/e016ad9b7f0b45048c58a9f27ba618bf
    - 难度: Easy
    - 题目描述: 实现快速排序算法
    - 解题思路: 标准快速排序实现，注意边界条件和递归终止条件
    - 算法要点: 分区操作，递归实现

13. **牛客网 - 最小的k个数**
    - 链接: https://www.nowcoder.com/practice/6a296eb82cf844ca8539b57c23e6e9bf
    - 难度: Easy
    - 题目描述: 输入n个整数，找出其中最小的K个数。
    - 解题思路: 使用快速选择算法或者堆来解决。
    - 算法要点: 快速选择算法，堆数据结构

四、PAT题目

14. **PAT 1101 Quick Sort**
    - 链接: https://pintia.cn/problem-sets/994805342720868352/problems/994805366343188480
    - 难度: Medium
    - 题目描述: 快速排序中的主元(pivot)是左面都比它小、右边都比它大的位置对应的数字。找出所有满足条件的主元。
    - 解题思路: 分析快速排序过程中每个元素是否可能作为主元，预处理左右边界最大值数组。
    - 算法要点: 预处理技术，边界分析

五、洛谷题目

15. **洛谷 P1177 【模板】快速排序**
    - 链接: https://www.luogu.com.cn/problem/P1177
    - 难度: 普及-
    - 题目描述: 利用快速排序算法将读入的N个数从小到大排序后输出。
    - 解题思路: 实现标准快速排序算法，注意输入输出格式。
    - 算法要点: 标准快速排序，输入输出处理

六、Codeforces题目

16. **Codeforces 401C. Team**
    - 链接: https://codeforces.com/problemset/problem/401/C
    - 难度: 1500
    - 题目描述: 构造一个01序列，满足特定的约束条件。
    - 解题思路: 在某些构造方法中可以使用排序来优化解的生成，贪心策略。

- 算法要点：构造算法，贪心思想

七、AtCoder 题目

17. **AtCoder ABC121C. Energy Drink Collector**
    - 链接：https://atcoder.jp/contests/abc121/tasks/abc121_c
    - 难度：300
    - 题目描述：购买能量饮料以获得最少的总花费。
    - 解题思路：按价格排序后贪心选择，优先选择价格低的饮料。
    - 算法要点：贪心算法，排序预处理

八、其他平台题目

18. **ZOJ 2581 Random Walking**
    - 链接：https://zoj.pintia.cn/problem-sets/91827364500/problems/91827367080
    - 难度：Medium
    - 题目描述：随机游走问题，需要排序预处理数据。
    - 时间复杂度：O(n log n)
    - 空间复杂度：O(n)
    - 最优解：排序预处理数据
    - 算法要点：排序预处理，随机游走分析

19. **HackerRank - QuickSort 1 - Partition**
    - 链接：https://www.hackerrank.com/challenges/quicksort1/problem
    - 难度：Easy
    - 题目描述：实现快速排序的分区操作。
    - 时间复杂度：O(n)
    - 空间复杂度：O(n)
    - 最优解：快速排序分区操作实现
    - 算法要点：分区算法实现

20. **HackerRank - QuickSort 2 - Sorting**
    - 链接：https://www.hackerrank.com/challenges/quicksort2/problem
    - 难度：Easy
    - 题目描述：实现完整的快速排序算法。
    - 时间复杂度：O(n log n)
    - 空间复杂度：O(log n)
    - 最优解：完整快速排序算法实现
    - 算法要点：完整快速排序实现

21. **杭电 OJ 1425. sort**
    - 链接：http://acm.hdu.edu.cn/showproblem.php?pid=1425
    - 难度：简单

- 题目描述：对整数数组进行快速排序
- 时间复杂度：O(n log n)
- 空间复杂度：O(log n)
- 最优解：快速排序或堆排序
- 算法要点：基础排序算法

22. **POJ 2388. Who's in the Middle**
- 链接：http://poj.org/problem?id=2388
- 难度：简单
- 题目描述：找出一组数的中位数，快速选择的经典应用
- 时间复杂度：O(n) 平均
- 空间复杂度：O(log n)
- 最优解：快速选择算法找中位数
- 算法要点：中位数查找，快速选择

23. **AizuOJ ALDS1_6_C. Quick Sort**
- 链接：https://onlinejudge.u-aizu.ac.jp/problems/ALDS1_6_C
- 难度：简单
- 题目描述：实现快速排序算法并输出每一步的分区结果
- 时间复杂度：O(n log n)
- 空间复杂度：O(log n)
- 最优解：快速排序算法实现
- 算法要点：快速排序实现，调试输出

24. **Comet OJ Contest 11 E. 快速排序**
- 链接：https://cometoj.com/contest/59/problem/E?problem_id=2830
- 难度：中等
- 题目描述：快速排序相关的概率问题
- 时间复杂度：O(n log n)
- 空间复杂度：O(log n)
- 最优解：快速排序相关的概率问题
- 算法要点：概率分析，快速排序

25. **SPOJ - SORT1 - Sorting Test**
- 链接：https://www.spoj.com/problems/SORT1/
- 难度：简单
- 题目描述：基本排序问题，测试排序算法效率
- 时间复杂度：O(n log n)
- 空间复杂度：O(log n)
- 最优解：快速排序基准测试
- 算法要点：排序算法测试

26. **UVa 10152 - ShellSort**

- 链接: https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=1093
- 难度: 中等
- 题目描述: 实现一种特殊的排序算法，与快速排序思想有关
- 时间复杂度: O(n log n)
- 空间复杂度: O(n)
- 最优解: 特殊排序算法
- 算法要点: 特殊排序算法实现

27. **LeetCode 169. 多数元素**
- 链接: https://leetcode.cn/problems/majority-element/
- 难度: Easy
- 题目描述: 给定一个大小为 n 的数组，找到其中的多数元素
- 时间复杂度: O(n)
- 空间复杂度: O(1)
- 最优解: Boyer-Moore 投票算法（与快速选择思想相关）
- 算法要点: 投票算法，多数元素查找

28. **LeetCode 229. 求众数 II**
- 链接: https://leetcode.cn/problems/majority-element-ii/
- 难度: Medium
- 题目描述: 找出数组中所有出现次数超过 ⌊ n/3 ⌋ 的元素
- 时间复杂度: O(n)
- 空间复杂度: O(1)
- 最优解: 摩尔投票法扩展
- 算法要点: 扩展投票算法，众数查找

29. **LeetCode 274. H 指数**
- 链接: https://leetcode.cn/problems/h-index/
- 难度: Medium
- 题目描述: 计算研究人员的 h 指数
- 时间复杂度: O(n) 平均
- 空间复杂度: O(n)
- 最优解: 计数排序或快速选择
- 算法要点: H 指数计算，排序应用

30. **LeetCode 378. 有序矩阵中第 K 小的元素**
- 链接: https://leetcode.cn/problems/kth-smallest-element-in-a-sorted-matrix/
- 难度: Medium
- 题目描述: 给定一个 n x n 矩阵，其中每行和每列元素均按升序排序，找到矩阵中第 k 小的元素
- 时间复杂度: O(n log(max-min))
- 空间复杂度: O(1)
- 最优解: 二分查找 + 计数

- 算法要点：二分查找，矩阵性质利用

总结

快速排序作为一种经典的排序算法，在各类算法竞赛和面试中都有广泛应用。通过系统练习以上题目，可以：

1. **掌握基础算法**：熟练实现标准快速排序
2. **理解算法变种**：掌握三路快排、快速选择等变种
3. **提升优化能力**：学会各种优化技巧，如随机化、小数组优化等
4. **扩展应用场景**：了解快速排序在不同问题中的应用，如查找第 K 大元素、中位数等
5. **增强调试能力**：通过大量练习提升算法调试能力，掌握边界条件处理

建议按照难度顺序逐步练习，并在练习过程中注重代码质量、时间复杂度分析和边界情况处理。同时，要注意不同题目可能需要不同的快速排序变种，如三路快排处理重复元素、快速选择查找第 K 大元素等。

==========

文件: README.md
==========
快速排序算法详解与实战

算法概述

快速排序是一种基于分治思想的高效排序算法，由英国计算机科学家 Tony Hoare 在 1960 年提出。它通过选择一个基准元素，将数组分为两部分，使得左边部分的元素都小于等于基准元素，右边部分的元素都大于等于基准元素，然后递归地对左右两部分进行排序。

快速排序的核心思想是"分而治之"，通过递归地将问题分解为更小的子问题来解决。它是一种不稳定的原地排序算法，在实际应用中表现优异。

算法特点

- **时间复杂度**：
  - 最好情况：O(n log n) - 每次分区都能将数组均匀分成两部分
  - 平均情况：O(n log n) - 随机选择基准值的情况下
  - 最坏情况：O(n²) - 每次选择的基准值都是最大或最小值
- **空间复杂度**：O(log n)（递归栈空间）
- **稳定性**：不稳定排序 - 相同元素的相对位置可能改变
- **原地排序**：是 - 只需要常数级别的额外空间

算法优化策略

1. **随机化基准选择**：避免最坏情况的发生，通过随机选择基准值使算法在概率上收敛到 O(n log n)

2. **三路快排**：处理重复元素较多的情况，将数组分为小于、等于、大于基准值三部分
3. **小数组优化**：当数组长度小于某个阈值时，使用插入排序，因为插入排序在小数组上性能更好
4. **尾递归优化**：减少递归调用栈的深度，将尾递归转换为迭代

适用场景

快速排序适用于以下场景：
- 大规模数据排序 - 平均性能优秀
- 内存受限的环境（原地排序）- 只需要O(log n)的栈空间
- 对平均性能要求较高的场景 - 平均时间复杂度为O(n log n)

不适用于：
- 需要稳定排序的场景 - 快速排序是不稳定的
- 数据基本有序的情况（未优化版本）- 会退化到O(n²)时间复杂度

经典题目解析

1. LeetCode 912. 排序数组
题目描述：给你一个整数数组 nums，请你将该数组升序排列。

解题思路：直接使用快速排序算法对数组进行排序。可以使用随机化基准选择和三路快排优化来提升性能。

2. LeetCode 215. 数组中的第K个最大元素
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。

解题思路：使用快速选择算法，在快速排序的基础上进行优化，只处理包含目标元素的区间，平均时间复杂度为O(n)。

3. 剑指 Offer 40. 最小的k个数
题目描述：输入整数数组 arr ，找出其中最小的 k 个数。

解题思路：使用快速选择算法或者快速排序算法找出最小的k个数。快速选择算法更为高效。

4. LeetCode 75. 颜色分类
题目描述：给定一个包含红色、白色和蓝色、共 n 个元素的数组 nums，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。

解题思路：使用三路快速排序的思想，将数组分为三个区域：小于基准值、等于基准值、大于基准值，这正好对应荷兰国旗问题。

5. LeetCode 283. 移动零
题目描述：给定一个数组 nums，编写一个函数将所有 0 移动到数组的末尾，同时保持非零元素的相对顺

序。

解题思路：使用快速排序的分区思想，将非零元素移到数组前面，零元素移到数组后面，这实际上是分区操作的一个应用。

跨语言实现差异

Java 版本
- 数组作为对象，有边界检查 - 安全但有性能开销
- 使用 Math.random()生成随机数 - 简单易用
- 通过虚拟机管理内存 - 有垃圾回收机制

C++版本
- 数组为指针，无边界检查，性能更高 - 直接内存操作
- 使用 rand()生成随机数 - 需要手动初始化随机种子
- 可以直接操作内存 - 需要手动管理内存

Python 版本
- 使用列表，动态类型 - 灵活但性能相对较低
- 使用 random 模块生成随机数 - 功能丰富
- 列表是对象，有动态扩容功能 - 使用方便

工程化考量

1. **异常处理**：处理空数组、null 输入等边界情况，确保程序的健壮性
2. **性能优化**：对于小数组使用插入排序优化，减少递归开销
3. **内存使用**：原地排序减少额外空间开销，避免不必要的内存分配
4. **稳定性**：标准快排不稳定，如需稳定排序需特殊处理或选择其他算法

调试技巧

1. **打印中间过程**：在分区操作后打印数组状态，帮助理解算法执行过程
2. **断言验证**：验证分区后各部分的正确性，确保算法逻辑正确
3. **边界测试**：测试空数组、单元素、重复元素等边界情况，确保算法的完整性

面试技巧

1. **理解快排与其它排序算法的比较**（如归并排序、堆排序）- 理解各自优缺点
2. **掌握快排的优化方法**（随机化、三路快排等）- 展示深入理解
3. **理解快排在不同数据分布下的性能表现** - 展现算法分析能力
4. **能够分析快排的稳定性和适用场景** - 展现工程思维

更多相关题目（全面补充）

基础排序类题目

1. **LeetCode 912. 排序数组**
   - 链接: https://leetcode.cn/problems/sort-an-array/
   - 难度: Medium
   - 时间复杂度: O(n log n) 平均, O(n²) 最坏
   - 空间复杂度: O(log n)
   - 最优解: 随机化快速排序 + 三路快排优化

2. **洛谷 P1177 【模板】快速排序**
   - 链接: https://www.luogu.com.cn/problem/P1177
   - 难度: 普及-
   - 时间复杂度: O(n log n)
   - 空间复杂度: O(log n)
   - 最优解: 标准快速排序实现

3. **杭电 OJ 1425. sort**
   - 链接: http://acm.hdu.edu.cn/showproblem.php?pid=1425
   - 难度: 简单
   - 时间复杂度: O(n log n)
   - 空间复杂度: O(log n)
   - 最优解: 快速排序或堆排序

4. **POJ 2388. Who's in the Middle**
   - 链接: http://poj.org/problem?id=2388
   - 难度: 简单
   - 时间复杂度: O(n) 平均
   - 空间复杂度: O(log n)
   - 最优解: 快速选择算法找中位数

快速选择类题目

5. **LeetCode 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: Medium
   - 时间复杂度: O(n) 平均, O(n²) 最坏
   - 空间复杂度: O(log n)
   - 最优解: 随机化快速选择算法

6. **剑指 Offer 40. 最小的k个数**
   - 链接: https://leetcode.cn/problems/zui-xiao-de-kge-shu-lcof/
   - 难度: Easy
   - 时间复杂度: O(n) 平均
   - 空间复杂度: O(log n)

- 最优解：快速选择算法

7. **LeetCode 347. 前 K 个高频元素**
   - 链接：https://leetcode.cn/problems/top-k-frequent-elements/
   - 难度：Medium
   - 时间复杂度：O(n) 平均
   - 空间复杂度：O(n)
   - 最优解：哈希表 + 快速选择

8. **LeetCode 973. 最接近原点的 K 个点**
   - 链接：https://leetcode.cn/problems/k-closest-points-to-origin/
   - 难度：Medium
   - 时间复杂度：O(n) 平均
   - 空间复杂度：O(log n)
   - 最优解：快速选择算法

三路快排应用题目
9. **LeetCode 75. 颜色分类**
   - 链接：https://leetcode.cn/problems/sort-colors/
   - 难度：Medium
   - 时间复杂度：O(n)
   - 空间复杂度：O(1)
   - 最优解：三路快排思想（荷兰国旗问题）

10. **LeetCode 283. 移动零**
    - 链接：https://leetcode.cn/problems/move-zeroes/
    - 难度：Easy
    - 时间复杂度：O(n)
    - 空间复杂度：O(1)
    - 最优解：双指针分区思想

中位数相关题目
11. **LeetCode 462. 最少移动次数使数组元素相等 II**
    - 链接：https://leetcode.cn/problems/minimum-moves-to-equal-array-elements-ii/
    - 难度：Medium
    - 时间复杂度：O(n) 平均
    - 空间复杂度：O(log n)
    - 最优解：快速选择找中位数

12. **LeetCode 414. 第三大的数**
    - 链接：https://leetcode.cn/problems/third-maximum-number/
    - 难度：Easy
    - 时间复杂度：O(n)

- 空间复杂度: O(1)
- 最优解: 一次遍历维护三个最大值

数据流处理题目
13. **LeetCode 703. 数据流中的第 K 大元素**
- 链接: https://leetcode.cn/problems/kth-largest-element-in-a-stream/
- 难度: Easy
- 时间复杂度: O(log k) 每次插入
- 空间复杂度: O(k)
- 最优解: 最小堆维护前 K 大元素

构造与排序结合题目
14. **LeetCode 324. 摆动排序 II**
- 链接: https://leetcode.cn/problems/wiggle-sort-ii/
- 难度: Medium
- 时间复杂度: O(n) 平均
- 空间复杂度: O(n)
- 最优解: 快速选择找中位数 + 三路划分

国际竞赛平台题目
15. **Codeforces 401C. Team**
- 链接: https://codeforces.com/problemset/problem/401/C
- 难度: 1500
- 时间复杂度: O(n)
- 空间复杂度: O(n)
- 最优解: 贪心构造 + 排序优化

16. **AtCoder ABC121C. Energy Drink Collector**
- 链接: https://atcoder.jp/contests/abc121/tasks/abc121_c
- 难度: 300
- 时间复杂度: O(n log n)
- 空间复杂度: O(1)
- 最优解: 按价格排序后贪心选择

17. **SPOJ - SORT1 - Sorting Test**
- 链接: https://www.spoj.com/problems/SORT1/
- 难度: 简单
- 时间复杂度: O(n log n)
- 空间复杂度: O(log n)
- 最优解: 快速排序基准测试

18. **UVa 10152 - ShellSort**
- 链接:

https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=1093
   - 难度: 中等
   - 时间复杂度: O(n log n)
   - 空间复杂度: O(n)
   - 最优解: 特殊排序算法

国内OJ平台题目
19. **牛客网 - 快速排序**
   - 链接: https://www.nowcoder.com/practice/e016ad9b7f0b45048c58a9f27ba618bf
   - 难度: Easy
   - 时间复杂度: O(n log n)
   - 空间复杂度: O(log n)
   - 最优解: 标准快速排序实现

20. **牛客网 - 最小的k个数**
   - 链接: https://www.nowcoder.com/practice/6a296eb82cf844ca8539b57c23e6e9bf
   - 难度: Easy
   - 时间复杂度: O(n) 平均
   - 空间复杂度: O(log n)
   - 最优解: 快速选择算法

21. **PAT 1101 Quick Sort**
   - 链接: https://pintia.cn/problem-sets/994805342720868352/problems/994805366343188480
   - 难度: Medium
   - 时间复杂度: O(n)
   - 空间复杂度: O(n)
   - 最优解: 预处理左右边界最大值数组

22. **AizuOJ ALDS1_6_C. Quick Sort**
   - 链接: https://onlinejudge.u-aizu.ac.jp/problems/ALDS1_6_C
   - 难度: 简单
   - 时间复杂度: O(n log n)
   - 空间复杂度: O(log n)
   - 最优解: 快速排序算法实现

23. **ZOJ 2581 Random Walking**
   - 链接: https://zoj.pintia.cn/problem-sets/91827364500/problems/91827367080
   - 难度: Medium
   - 时间复杂度: O(n log n)
   - 空间复杂度: O(n)
   - 最优解: 排序预处理数据

在线编程平台题目

24. **HackerRank - QuickSort 1 - Partition**
    - 链接: https://www.hackerrank.com/challenges/quicksort1/problem
    - 难度: Easy
    - 时间复杂度: O(n)
    - 空间复杂度: O(n)
    - 最优解: 快速排序分区操作实现

25. **HackerRank - QuickSort 2 - Sorting**
    - 链接: https://www.hackerrank.com/challenges/quicksort2/problem
    - 难度: Easy
    - 时间复杂度: O(n log n)
    - 空间复杂度: O(log n)
    - 最优解: 完整快速排序算法实现

26. **Comet OJ Contest 11 E. 快速排序**
    - 链接: https://cometoj.com/contest/59/problem/E?problem_id=2830
    - 难度: 中等
    - 时间复杂度: O(n log n)
    - 空间复杂度: O(log n)
    - 最优解: 快速排序相关的概率问题

特殊应用场景题目
27. **LeetCode 169. 多数元素**
    - 链接: https://leetcode.cn/problems/majority-element/
    - 难度: Easy
    - 时间复杂度: O(n)
    - 空间复杂度: O(1)
    - 最优解: Boyer-Moore 投票算法（与快速选择思想相关）

28. **LeetCode 229. 求众数 II**
    - 链接: https://leetcode.cn/problems/majority-element-ii/
    - 难度: Medium
    - 时间复杂度: O(n)
    - 空间复杂度: O(1)
    - 最优解: 摩尔投票法扩展

29. **LeetCode 274. H 指数**
    - 链接: https://leetcode.cn/problems/h-index/
    - 难度: Medium
    - 时间复杂度: O(n) 平均
    - 空间复杂度: O(n)
    - 最优解: 计数排序或快速选择

30. **LeetCode 378. 有序矩阵中第 K 小的元素**
    - 链接: https://leetcode.cn/problems/kth-smallest-element-in-a-sorted-matrix/
    - 难度: Medium
    - 时间复杂度: O(n log(max-min))
    - 空间复杂度: O(1)
    - 最优解: 二分查找 + 计数

算法复杂度详细分析

时间复杂度分析

最好情况：O(n log n)
- 每次分区都能将数组均匀分成两部分
- 递归深度为 log n，每层处理 n 个元素

平均情况：O(n log n)
- 随机选择基准值，期望分区平衡
- 数学期望证明为 O(n log n)

最坏情况：O(n²)
- 每次分区都选择最大或最小值作为基准
- 递归深度为 n，每层处理 n 个元素

空间复杂度分析

递归栈空间：O(log n)
- 递归调用深度为 log n
- 每次递归需要常数空间

原地排序：O(1)额外空间
- 不需要额外的存储空间
- 只在原数组上进行交换操作

工程化优化策略

1. 随机化基准选择
```java
// 避免最坏情况的优化
int randomIndex = left + random.nextInt(right - left + 1);
swap(arr, left, randomIndex);
```

2. 三路快排优化

```java
// 处理重复元素的优化
// 将数组分为 <pivot, =pivot, >pivot 三部分
```

3. 小数组插入排序
```java
// 当数组长度小于阈值时使用插入排序
if (right - left < INSERTION_THRESHOLD) {
    insertionSort(arr, left, right);
    return;
}
```

4. 尾递归优化
```java
// 减少递归深度，将尾递归转换为迭代
while (left < right) {
    int pivotIndex = partition(arr, left, right);
    quickSort(arr, left, pivotIndex - 1);
    left = pivotIndex + 1;  // 尾递归优化
}
```

跨语言实现对比

Java 实现特点
- **内存管理**：JVM 自动垃圾回收 - 无需手动管理内存
- **边界检查**：数组访问有边界检查 - 安全但有性能开销
- **随机数生成**：Math.random()或 Random 类 - 使用简单
- **泛型支持**：支持泛型编程 - 类型安全

C++实现特点
- **性能优势**：直接内存操作，无边界检查 - 性能更高
- **模板编程**：支持泛型编程 - 代码复用性高
- **随机数**：rand()函数或<random>库 - 功能丰富
- **内存管理**：需要手动管理或使用智能指针 - 灵活但容易出错

Python 实现特点
- **简洁性**：代码简洁，易于理解 - 开发效率高
- **动态类型**：无需声明变量类型 - 灵活性高
- **内置函数**：有丰富的内置函数支持 - 功能强大
- **性能考虑**：解释型语言，性能相对较低 - 适合原型开发

调试与测试策略

单元测试用例设计
1. **空数组测试**：[] → []
2. **单元素测试**：[5] → [5]
3. **已排序数组**：[1,2,3,4,5] → [1,2,3,4,5]
4. **逆序数组**：[5,4,3,2,1] → [1,2,3,4,5]
5. **重复元素**：[3,1,4,1,5,9,2,6] → [1,1,2,3,4,5,6,9]
6. **全相同元素**：[2,2,2,2] → [2,2,2,2]
7. **大规模数据**：10000 个随机数排序

调试技巧
1. **打印分区过程**：每次分区后打印数组状态 - 帮助理解算法执行过程
2. **验证分区正确性**：确保左边≤基准，右边≥基准 - 确保算法逻辑正确
3. **递归深度监控**：避免栈溢出 - 确保程序稳定性
4. **性能分析**：统计比较和交换次数 - 优化算法性能

面试常见问题

理论问题
1. **快排为什么是不稳定的？**
   - 答：因为交换操作可能改变相同元素的相对顺序

2. **快排的最坏情况是什么？如何避免？**
   - 答：最坏情况是 $O(n^2)$，通过随机化基准选择避免

3. **快排和归并排序的比较？**
   - 答：快排原地排序但不稳定，归并稳定但需要额外空间

编码问题
1. **实现快速排序算法**
2. **实现快速选择算法找第 K 大元素**
3. **实现三路快排解决颜色分类问题**

优化问题
1. **如何优化快排处理重复元素？**
2. **小数组优化策略是什么？**
3. **尾递归优化如何实现？**

实际应用场景

大数据处理

- 分布式系统中的排序操作 - 处理海量数据
- 数据库查询优化中的排序 - 提升查询性能
- 内存受限环境下的排序需求 - 节省内存空间

机器学习
- 特征选择中的排序操作 - 特征重要性排序
- 数据预处理中的排序需求 - 数据清洗和准备
- 模型评估中的排序计算 - 性能指标计算

系统开发
- 操作系统中的进程调度 - 进程优先级排序
- 数据库索引的构建 - 索引结构维护
- 缓存淘汰算法的实现 - 缓存项排序

扩展学习资源

在线学习平台
- **Coursera**：算法专项课程 - 系统学习算法知识
- **edX**：数据结构与算法课程 - 理论与实践结合
- **LeetCode**：大量排序算法题目 - 实战练习
- **牛客网**：国内算法笔试平台 - 面试准备

经典书籍
- 《算法导论》- Thomas H. Cormen - 算法领域经典教材
- 《算法》- Robert Sedgewick - 实用算法指南
- 《编程珠玑》- Jon Bentley - 编程技巧与算法思维

开源项目
- **JDK 源码**：Arrays.sort()实现 - 工业级实现参考
- **C++ STL**：std::sort 实现 - 高性能排序实现
- **Python**：list.sort()实现 - 实用排序实现

总结

快速排序作为一种经典的排序算法，在算法面试和实际开发中都有广泛应用。通过深入理解其原理、掌握优化方法、熟悉各种变种算法，可以显著提升算法能力和工程实践水平。建议通过大量练习来巩固所学知识。

掌握快速排序不仅有助于算法面试，更能提升对分治思想、递归优化、性能分析等核心计算机科学概念的理解，为后续学习更复杂的算法和数据结构打下坚实基础。

==========

[代码文件]

```
==========================================
文件: Code01_QuickSort.java
==========================================
package class023;

// 随机快速排序，acm练习风格
// 测试链接 : https://www.luogu.com.cn/problem/P1177
// 请同学们务必参考如下代码中关于输入、输出的处理
// 这是输入输出处理效率很高的写法
// 提交以下的code，提交时请把类名改成"Main"，可以直接通过

/*
 * 快速排序算法详解与实战 - 全面指南
 *
 * 一、算法核心原理
 * 快速排序是基于分治思想的高效排序算法，通过选择基准元素，将数组分为两部分，递归排序实现。
 * 时间复杂度：最好/平均O(n log n)，最坏O(n²)，空间复杂度：O(log n)
 *
 * 二、优化策略总结
 * 1. 随机化基准选择 - 避免最坏情况
 * 2. 三路快排 - 处理重复元素
 * 3. 小数组插入排序优化 - 减少递归开销
 * 4. 尾递归优化 - 降低栈空间使用
 * 5. 三数取中法 - 选择更优基准
 *
 * 三、详尽题目列表与解决方案
 *
 * 1. LeetCode 912. 排序数组
 *    链接: https://leetcode.cn/problems/sort-an-array/
 *    题目描述: 给你一个整数数组 nums，请你将该数组升序排列。
 *    最优解: 随机化快速排序 - 避免最坏情况的O(n²)复杂度
 *    时间复杂度: O(n log n)，空间复杂度: O(log n)
 *
 * 2. LeetCode 215. 数组中的第K个最大元素
 *    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。
 *    最优解: 快速选择算法 - 平均O(n)时间复杂度
 *    时间复杂度: 平均O(n)，最坏O(n²)，空间复杂度: O(log n)
 *
 * 3. 剑指 Offer 40. 最小的k个数
 *    链接: https://leetcode.cn/problems/zui-xiao-de-kge-shu-lcof/
 *    题目描述: 输入整数数组 arr ，找出其中最小的 k 个数。
 *    最优解: 快速选择算法或堆排序
```

```
 *     时间复杂度: 平均O(n)，空间复杂度: O(k)或O(log n)
 *
 * 4. LeetCode 75. 颜色分类
 *     链接: https://leetcode.cn/problems/sort-colors/
 *     题目描述: 给定一个包含红色、白色和蓝色的数组，原地排序使得相同颜色相邻，按红、白、蓝顺序排列。
 *     最优解: 三路快排思想 - 一次遍历完成排序
 *     时间复杂度: O(n)，空间复杂度: O(1)
 *
 * 5. LeetCode 283. 移动零
 *     链接: https://leetcode.cn/problems/move-zeroes/
 *     题目描述: 将所有0移动到数组末尾，保持非零元素相对顺序不变。
 *     最优解: 双指针（分区思想）- 一次遍历完成
 *     时间复杂度: O(n)，空间复杂度: O(1)
 *
 * 6. LeetCode 347. 前K个高频元素
 *     链接: https://leetcode.cn/problems/top-k-frequent-elements/
 *     题目描述: 返回数组中出现频率前k高的元素。
 *     最优解: 哈希表+快速选择 - 不需要完全排序
 *     时间复杂度: O(n)，空间复杂度: O(n)
 *
 * 7. LeetCode 973. 最接近原点的K个点
 *     链接: https://leetcode.cn/problems/k-closest-points-to-origin/
 *     题目描述: 返回距离原点最近的K个点。
 *     最优解: 快速选择 - 基于距离排序
 *     时间复杂度: 平均O(n)，空间复杂度: O(log n)
 *
 * 8. LeetCode 324. 摆动排序 II
 *     链接: https://leetcode.cn/problems/wiggle-sort-ii/
 *     题目描述: 重排数组使得 nums[0] < nums[1] > nums[2] < nums[3]...
 *     最优解: 快速选择找中位数 + 三路划分
 *     时间复杂度: O(n)，空间复杂度: O(n)
 *
 * 9. LeetCode 414. 第三大的数
 *     链接: https://leetcode.cn/problems/third-maximum-number/
 *     题目描述: 返回数组中第三大的数，如果不存在则返回最大数。
 *     最优解: 一次遍历维护三个最大值
 *     时间复杂度: O(n)，空间复杂度: O(1)
 *
 * 10. LeetCode 462. 最少移动次数使数组元素相等 II
 *      链接: https://leetcode.cn/problems/minimum-moves-to-equal-array-elements-ii/
 *      题目描述: 返回使所有元素相等的最少移动次数。
 *      最优解: 快速选择找中位数
```

*   时间复杂度: O(n)，空间复杂度: O(log n)
*
* 11. LeetCode 703. 数据流中的第K大元素
*   链接: https://leetcode.cn/problems/kth-largest-element-in-a-stream/
*   题目描述: 设计一个类来找到数据流中第K大元素。
*   最优解: 最小堆维护前K大元素
*   时间复杂度: O(log k) per add，空间复杂度: O(k)
*
* 12. 牛客网 - 快速排序
*   链接: https://www.nowcoder.com/practice/e016ad9b7f0b45048c58a9f27ba618bf
*   题目描述: 实现快速排序算法
*
* 13. PAT 1101 Quick Sort
*   链接: https://pintia.cn/problem-sets/994805342720868352/problems/994805366343188480
*   题目描述: 找出所有满足条件的主元（左边都比它小、右边都比它大）
*   最优解: 预处理左右边界最大值数组
*   时间复杂度: O(n)，空间复杂度: O(n)
*
* 14. 洛谷 P1177 【模板】快速排序
*   链接: https://www.luogu.com.cn/problem/P1177
*   题目描述: 利用快速排序算法将读入的N个数从小到大排序后输出
*
* 15. Codeforces 401C. Team
*   链接: https://codeforces.com/problemset/problem/401/C
*   题目描述: 构造一个01序列，满足特定约束条件
*
* 16. AtCoder ABC121C. Energy Drink Collector
*   链接: https://atcoder.jp/contests/abc121/tasks/abc121_c
*   题目描述: 购买能量饮料以获得最少总花费
*   最优解: 按价格排序后贪心选择
*
* 17. HackerRank - QuickSort 1 - Partition
*   链接: https://www.hackerrank.com/challenges/quicksort1/problem
*   题目描述: 实现快速排序的分区操作
*
* 18. HackerRank - QuickSort 2 - Sorting
*   链接: https://www.hackerrank.com/challenges/quicksort2/problem
*   题目描述: 实现完整的快速排序算法
*
* 19. ZOJ 2581 Random Walking
*   链接: https://zoj.pintia.cn/problem-sets/91827364500/problems/91827367080
*   题目描述: 随机游走问题，需要排序预处理数据
*

```
 * 20. SPOJ - SORT1 - Sorting Test
 *     链接: https://www.spoj.com/problems/SORT1/
 *     题目描述: 基本排序问题，测试排序算法效率
 *     时间复杂度: O(n log n)，空间复杂度: O(log n)
 *     最优解: 快速排序基准测试
 *
 * 21. UVa 10152 - ShellSort
 *     链接:
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=1093
 *     题目描述: 实现一种特殊的排序算法，与快速排序思想有关
 *     时间复杂度: O(n log n)，空间复杂度: O(n)
 *     最优解: 特殊排序算法
 *
 * 22.杭电 OJ 1425. sort
 *     链接: http://acm.hdu.edu.cn/showproblem.php?pid=1425
 *     题目描述: 对整数数组进行快速排序
 *     时间复杂度: O(n log n)，空间复杂度: O(log n)
 *     最优解: 快速排序或堆排序
 *
 * 23. POJ 2388. Who's in the Middle
 *     链接: http://poj.org/problem?id=2388
 *     题目描述: 找出一组数的中位数，快速选择的经典应用
 *     时间复杂度: O(n) 平均，空间复杂度: O(log n)
 *     最优解: 快速选择算法找中位数
 *
 * 24. AizuOJ ALDS1_6_C. Quick Sort
 *     链接: https://onlinejudge.u-aizu.ac.jp/problems/ALDS1_6_C
 *     题目描述: 实现快速排序算法并输出每一步的分区结果
 *     时间复杂度: O(n log n)，空间复杂度: O(log n)
 *     最优解: 快速排序算法实现
 *
 * 25. Comet OJ Contest 11 E. 快速排序
 *     链接: https://cometoj.com/contest/59/problem/E?problem_id=2830
 *     题目描述: 快速排序相关的概率问题
 *     时间复杂度: O(n log n)，空间复杂度: O(log n)
 *     最优解: 快速排序相关的概率问题
 *
 * 26. LeetCode 169. 多数元素
 *     链接: https://leetcode.cn/problems/majority-element/
 *     题目描述: 给定一个大小为 n 的数组，找到其中的多数元素
 *     时间复杂度: O(n)，空间复杂度: O(1)
 *     最优解: Boyer-Moore 投票算法（与快速选择思想相关）
 *
```

```
 * 27. LeetCode 229. 求众数 II
 *     链接: https://leetcode.cn/problems/majority-element-ii/
 *     题目描述: 找出数组中所有出现次数超过 ⌊ n/3 ⌋ 的元素
 *     时间复杂度: O(n)，空间复杂度: O(1)
 *     最优解: 摩尔投票法扩展
 *
 * 28. LeetCode 274. H 指数
 *     链接: https://leetcode.cn/problems/h-index/
 *     题目描述: 计算研究人员的 h 指数
 *     时间复杂度: O(n) 平均，空间复杂度: O(n)
 *     最优解: 计数排序或快速选择
 *
 * 29. LeetCode 378. 有序矩阵中第 K 小的元素
 *     链接: https://leetcode.cn/problems/kth-smallest-element-in-a-sorted-matrix/
 *     题目描述: 给定一个 n x n 矩阵，其中每行和每列元素均按升序排序，找到矩阵中第 k 小的元素
 *     时间复杂度: O(n log(max-min))，空间复杂度: O(1)
 *     最优解: 二分查找 + 计数
 *
 * 四、工程化深度考量
 * 1. 异常处理: 空数组、null 输入、单元素数组等边界情况
 * 2. 多线程安全: 排序算法在并发环境中的使用注意事项
 * 3. 内存优化: 原地排序减少额外空间开销
 * 4. 缓存友好性: 优化数据访问模式，提高缓存命中率
 * 5. 单元测试: 全面覆盖各种输入场景
 * 6. 性能监控: 针对大规模数据的性能退化检测
 *
 * 五、跨语言实现差异
 * 1. Java: 数组作为对象，有边界检查，使用 Math.random()生成随机数
 * 2. C++: 数组为指针，无边界检查，性能更高，使用 rand()生成随机数
 * 3. Python: 使用列表，动态类型，使用 random 模块，语法简洁但性能较低
 *
 * 六、调试与优化技巧
 * 1. 断点式打印: 输出关键变量变化过程
 * 2. 断言验证: 使用 assert 验证分区的正确性
 * 3. 性能分析: 使用 profiler 找出瓶颈
 * 4. 边界测试: 空数组、单元素、重复元素、有序/逆序数组
 * 5. 常数优化: 减少不必要的操作和判断
 */

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
```

```java
import java.io.PrintWriter;
import java.io.StreamTokenizer;

public class Code01_QuickSort {

    public static int MAXN = 100001;

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

    public static int n;

    /**
     * 主函数，程序入口
     * @param args 命令行参数
     * @throws IOException 输入输出异常
     */
    public static void main(String[] args) throws IOException {
        // 创建高效的输入输出流
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        StreamTokenizer in = new StreamTokenizer(br);
        PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

        // 读取数组长度
        in.nextToken();
        n = (int) in.nval;

        // 读取数组元素
        for (int i = 0; i < n; i++) {
            in.nextToken();
            arr[i] = (int) in.nval;
        }

        // 调用改进版快速排序算法对数组进行排序
        quickSort2(0, n - 1);

        // 输出排序后的数组
        for (int i = 0; i < n - 1; i++) {
            out.print(arr[i] + " ");
        }
        out.println(arr[n - 1]);
        out.flush();
        out.close();
        br.close();
```

```
}

/**
 * 随机快速排序经典版(不推荐)
 * 甚至在洛谷上测试因为递归开太多层会爆栈导致出错
 * @param l 排序区间的左边界（包含）
 * @param r 排序区间的右边界（包含）
 */
public static void quickSort1(int l, int r) {
    // l == r，只有一个数
    // l > r，范围不存在，不用管
    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 * logn)
    // l.....r 随机选一个位置，x 这个值，做划分
    int x = arr[l + (int) (Math.random() * (r - l + 1))];

    // 调用 partition1 函数进行分区操作，返回等于 x 区域的右边界
    int mid = partition1(l, r, x);

    // 递归处理小于等于 x 的区域
    quickSort1(l, mid - 1);

    // 递归处理大于 x 的区域
    quickSort1(mid + 1, r);
}

/**
 * 分区函数 - 经典版本
 * 将数组划分为两个部分：小于等于 x 的部分和大于 x 的部分
 * 并确保划分完成后小于等于 x 区域的最后一个数字是 x
 * @param l 分区区间的左边界（包含）
 * @param r 分区区间的右边界（包含）
 * @param x 基准值
 * @return 等于 x 区域的右边界索引
 */
public static int partition1(int l, int r, int x) {
    // a 表示小于等于 x 区域的右边界下一个位置
    // xi 记录在小于等于 x 区域内任意一个 x 的位置
    int a = l, xi = 0;
```

```
    // 遍历整个区间
    for (int i = l; i <= r; i++) {
        // 如果当前元素小于等于基准值 x
        if (arr[i] <= x) {
            // 将当前元素交换到小于等于 x 区域
            swap(a, i);

            // 如果交换过来的元素正好等于 x，则记录其位置
            if (arr[a] == x) {
                xi = a;
            }

            // 小于等于 x 区域向右扩展一位
            a++;
        }
    }

    // 将记录的 x 位置的元素与小于等于 x 区域的最后一个元素交换
    // 确保小于等于 x 区域的最后一个数字是 x
    swap(xi, a - 1);

    // 返回等于 x 区域的右边界索引
    return a - 1;
}

/**
 * 交换数组中两个位置的元素
 * @param i 第一个位置
 * @param j 第二个位置
 */
public static void swap(int i, int j) {
    int tmp = arr[i];
    arr[i] = arr[j];
    arr[j] = tmp;
}

/**
 * 随机快速排序改进版(推荐)
 * 使用三路快排优化，更好地处理重复元素
 * 可以通过所有测试用例
 * @param l 排序区间的左边界（包含）
 * @param r 排序区间的右边界（包含）
```

```
 */
public static void quickSort2(int l, int r) {
    // 递归终止条件：当左边界大于等于右边界时，表示区间内没有元素或只有一个元素，无需排序
    if (l >= r) {
        return;
    }

    // 随机选择基准值 pivot，避免最坏情况的发生
    int x = arr[l + (int) (Math.random() * (r - l + 1))];

    // 调用 partition2 函数进行三路分区操作
    partition2(l, r, x);

    // 为了防止底层的递归过程覆盖全局变量
    // 这里用临时变量记录等于 x 区域的左右边界
    int left = first;
    int right = last;

    // 递归处理小于 x 的区域
    quickSort2(l, left - 1);

    // 递归处理大于 x 的区域
    quickSort2(right + 1, r);
}

/**
 * 【优化版本 1】小数组插入排序优化
 * 当数组长度小于阈值时，使用插入排序，减少递归开销
 * @param l 排序区间的左边界（包含）
 * @param r 排序区间的右边界（包含）
 */
public static void quickSortOptimized(int l, int r) {
    // 小数组阈值，经验值为 10-20
    final int INSERTION_SORT_THRESHOLD = 15;

    // 对小数组使用插入排序
    if (r - l <= INSERTION_SORT_THRESHOLD) {
        insertionSort(l, r);
        return;
    }

    // 对大数组继续使用快速排序
    int x = arr[l + (int) (Math.random() * (r - l + 1))];
```

```
    partition2(l, r, x);
    int left = first;
    int right = last;
    quickSortOptimized(l, left - 1);
    quickSortOptimized(right + 1, r);
}

/**
 * 插入排序算法，用于小数组优化
 * @param l 排序区间的左边界（包含）
 * @param r 排序区间的右边界（包含）
 */
private static void insertionSort(int l, int r) {
    // 从第二个元素开始，逐个插入到已排序序列中
    for (int i = l + 1; i <= r; i++) {
        int key = arr[i]; // 当前要插入的元素
        int j = i - 1;    // 已排序序列的最后一个位置

        // 在已排序序列中找到合适的插入位置
        while (j >= l && arr[j] > key) {
            arr[j + 1] = arr[j]; // 元素后移
            j--;
        }

        // 插入元素
        arr[j + 1] = key;
    }
}

/**
 * 【优化版本 2】尾递归优化
 * 将尾递归转换为迭代，减少栈空间使用
 * @param l 排序区间的左边界（包含）
 * @param r 排序区间的右边界（包含）
 */
public static void quickSortTailRecursive(int l, int r) {
    // 使用循环代替尾递归
    while (l < r) {
        int x = arr[l + (int) (Math.random() * (r - l + 1))];
        partition2(l, r, x);
        int left = first;
        int right = last;
```

```
        // 优先处理较小的子数组，减少递归深度
        if (left - l < r - right) {
            // 左边子数组较小，先递归处理左边
            quickSortTailRecursive(l, left - 1);
            // 尾递归优化：将右边子数组的处理转换为迭代
            l = right + 1;
        } else {
            // 右边子数组较小，先递归处理右边
            quickSortTailRecursive(right + 1, r);
            // 尾递归优化：将左边子数组的处理转换为迭代
            r = left - 1;
        }
    }
}

/**
 * 【优化版本 3】三数取中法选择基准
 * 选择左、中、右三个位置的中位数作为基准，提高最坏情况性能
 * @param l 排序区间的左边界（包含）
 * @param r 排序区间的右边界（包含）
 */
public static void quickSortMedianOfThree(int l, int r) {
    // 递归终止条件
    if (l >= r) {
        return;
    }

    // 三数取中选择基准
    int mid = l + ((r - l) >> 1); // 中间位置
    int x = medianOfThree(l, mid, r);

    // 进行三路分区
    partition2(l, r, x);
    int left = first;
    int right = last;

    // 递归处理左右子数组
    quickSortMedianOfThree(l, left - 1);
    quickSortMedianOfThree(right + 1, r);
}

/**
 * 三数取中辅助方法
```

```
     * @param a 左边界索引
     * @param b 中间索引
     * @param c 右边界索引
     * @return 三个位置元素的中位数
     */
    private static int medianOfThree(int a, int b, int c) {
        // 确保 arr[a] <= arr[b] <= arr[c]
        if (arr[a] > arr[b]) swap(a, b);
        if (arr[b] > arr[c]) swap(b, c);
        if (arr[a] > arr[b]) swap(a, b);
        return arr[b]; // 返回中位数
    }

    /**
     * 【快速选择算法】用于 LeetCode 215 等第 K 大/小问题
     * 平均时间复杂度 O(n)，最坏 O(n²)
     * @param l 搜索区间的左边界（包含）
     * @param r 搜索区间的右边界（包含）
     * @param k 目标位置索引
     * @return 第 k 小的元素
     */
    public static int quickSelect(int l, int r, int k) {
        // 当区间只有一个元素时，就是要找的位置
        if (l == r) {
            return arr[l];
        }

        // 随机选择基准值
        int x = arr[l + (int) (Math.random() * (r - l + 1))];

        // 进行三路分区
        partition2(l, r, x);
        int left = first;
        int right = last;

        // 根据目标位置决定处理哪个子区间
        if (k < left) {
            // 目标位置在左半部分
            return quickSelect(l, left - 1, k);
        } else if (k > right) {
            // 目标位置在右半部分
            return quickSelect(right + 1, r, k);
        } else {
```

```
            // 目标位置在等于区域，直接返回
            return arr[k];
        }
    }

    /**
     * 【LeetCode 215 解法】数组中的第 K 个最大元素
     * @param nums 输入数组
     * @param k 第 k 大的元素
     * @return 第 k 大的元素值
     */
    public static int findKthLargest(int[] nums, int k) {
        // 注意：第 K 大元素等价于第(nums.length - k)小元素
        int n = nums.length;
        // 复制到全局数组进行操作
        System.arraycopy(nums, 0, arr, 0, n);
        return quickSelect(0, n - 1, n - k);
    }

    /**
     * 【剑指 Offer 40 解法】最小的 k 个数
     * @param nums 输入数组
     * @param k 需要返回的最小元素个数
     * @return 包含最小 k 个数的数组
     */
    public static int[] getLeastNumbers(int[] nums, int k) {
        // 边界条件处理
        if (k <= 0) return new int[0];
        if (k >= nums.length) return nums;

        int n = nums.length;
        System.arraycopy(nums, 0, arr, 0, n);

        // 使用快速选择找到第 k 小的元素
        quickSelect(0, n - 1, k - 1);

        // 收集前 k 个最小元素
        int[] result = new int[k];
        System.arraycopy(arr, 0, result, 0, k);
        return result;
    }

    /**
```

```
 * 【LeetCode 75 解法】颜色分类（三路快排应用）
 * @param nums 包含 0、1、2 的数组，分别代表红、白、蓝三种颜色
 */
public static void sortColors(int[] nums) {
    // 三路快排思想：0 放左边，1 放中间，2 放右边
    int zero = -1;      // [0...zero] == 0
    int two = nums.length; // [two...n-1] == 2
    int i = 0;          // 当前处理的位置

    while (i < two) {
        if (nums[i] == 0) {
            // 当前元素为 0，交换到 0 区域的下一个位置
            zero++;
            swap(nums, zero, i);
            i++;
        } else if (nums[i] == 1) {
            // 当前元素为 1，保持在中间区域
            i++;
        } else {
            // 当前元素为 2，交换到 2 区域的前一个位置
            two--;
            swap(nums, i, two);
            // 注意这里 i 不增加，因为交换过来的元素还未处理
        }
    }
}

/**
 * 辅助交换方法（用于传入的数组）
 * @param nums 数组
 * @param i 第一个位置
 * @param j 第二个位置
 */
private static void swap(int[] nums, int i, int j) {
    int tmp = nums[i];
    nums[i] = nums[j];
    nums[j] = tmp;
}

/**
 * 【LeetCode 283 解法】移动零（分区思想应用）
 * @param nums 输入数组
 */
```

```
public static void moveZeroes(int[] nums) {
    // 双指针：将非零元素移动到数组前面
    int nonZeroPos = 0; // 指向下一个非零元素应该放的位置

    // 第一次遍历：将所有非零元素移到前面
    for (int i = 0; i < nums.length; i++) {
        if (nums[i] != 0) {
            if (i != nonZeroPos) {
                swap(nums, i, nonZeroPos);
            }
            nonZeroPos++;
        }
    }
}

/**
 * 【LeetCode 462 解法】最少移动次数使数组元素相等 II（中位数思想）
 * @param nums 输入数组
 * @return 最少移动次数
 */
public static int minMoves2(int[] nums) {
    int n = nums.length;
    System.arraycopy(nums, 0, arr, 0, n);

    // 找到中位数
    int median = quickSelect(0, n - 1, n / 2);

    // 计算所有元素到中位数的距离和
    int moves = 0;
    for (int num : nums) {
        moves += Math.abs(num - median);
    }
    return moves;
}

/**
 * 【调试辅助方法】打印数组内容
 * @param nums 要打印的数组
 */
public static void printArray(int[] nums) {
    for (int num : nums) {
        System.out.print(num + " ");
    }
```

```
        System.out.println();
    }

    /**
     * 【测试验证方法】检查数组是否有序
     * @param nums 要检查的数组
     * @return 是否有序
     */
    public static boolean isSorted(int[] nums) {
        for (int i = 1; i < nums.length; i++) {
            if (nums[i] < nums[i - 1]) {
                return false;
            }
        }
        return true;
    }

    /**
     * 【LeetCode 912 解法】排序数组（标准快速排序实现）
     * @param nums 待排序数组
     * @return 排序后的数组
     */
    public static int[] sortArray(int[] nums) {
        int n = nums.length;
        System.arraycopy(nums, 0, arr, 0, n);
        quickSortOptimized(0, n - 1); // 使用优化版本的快速排序
        int[] result = new int[n];
        System.arraycopy(arr, 0, result, 0, n);
        return result;
    }

    /**
     * 【HackerRank Partition 解法】快速排序分区操作
     * @param nums 待分区数组
     */
    public static void quickSortPartition(int[] nums) {
        int pivot = nums[0]; // 选择第一个元素作为基准
        int n = nums.length;
        int[] result = new int[n];
        int left = 0;
        int right = n - 1;

        // 从右到左填充小于基准的元素
```

```
    for (int i = n - 1; i >= 0; i--) {
        if (nums[i] < pivot) {
            result[left++] = nums[i];
        }
    }

    // 填充基准元素
    result[left++] = pivot;

    // 从左到右填充大于等于基准的元素
    for (int i = 1; i < n; i++) {
        if (nums[i] >= pivot) {
            result[right--] = nums[i];
        }
    }

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

/**
 * 【剑指 Offer 51 解法】数组中的逆序对（归并排序应用，与快速排序对比）
 * 注意：虽然本题更适合用归并排序解决，但我们这里提供实现作为对比
 * @param nums 输入数组
 * @return 逆序对数量
 */
private static int count = 0; // 记录逆序对数量
public static int reversePairs(int[] nums) {
    count = 0;
    int n = nums.length;
    int[] temp = new int[n];
    mergeSort(nums, 0, n - 1, temp);
    return count;
}

/**
 * 归并排序辅助方法（用于逆序对计算）
 * @param nums 待排序数组
 * @param left 左边界
 * @param right 右边界
 * @param temp 临时数组
 */
private static void mergeSort(int[] nums, int left, int right, int[] temp) {
```

```
    if (left < right) {
        int mid = left + ((right - left) >> 1);
        mergeSort(nums, left, mid, temp);
        mergeSort(nums, mid + 1, right, temp);
        merge(nums, left, mid, right, temp);
    }
}

/**
 * 合并过程中计算逆序对
 * @param nums 待合并数组
 * @param left 左边界
 * @param mid 中间位置
 * @param right 右边界
 * @param temp 临时数组
 */
private static void merge(int[] nums, int left, int mid, int right, int[] temp) {
    int i = left;
    int j = mid + 1;
    int k = 0;

    while (i <= mid && j <= right) {
        if (nums[i] <= nums[j]) {
            temp[k++] = nums[i++];
        } else {
            // 计算逆序对：nums[i...mid]都与nums[j]构成逆序对
            count += (mid - i + 1);
            temp[k++] = nums[j++];
        }
    }

    // 处理剩余元素
    while (i <= mid) {
        temp[k++] = nums[i++];
    }
    while (j <= right) {
        temp[k++] = nums[j++];
    }

    // 复制回原数组
    k = 0;
    while (left <= right) {
        nums[left++] = temp[k++];
```

```
    }
}

/**
 * 【洛谷 P1177 解法】快速排序模板题
 * @param n 数组长度
 * @param arr 待排序数组
 */
public static void luoguP1177(int n, int[] arr) {
    // 直接使用优化版本的快速排序
    quickSortOptimized(0, n - 1);
}

/**
 * 【POJ 2388 解法】Who's in the Middle（中位数问题）
 * @param nums 输入数组
 * @return 中位数
 */
public static int findMedian(int[] nums) {
    int n = nums.length;
    System.arraycopy(nums, 0, arr, 0, n);
    // 中位数就是第 (n-1)/2 小的元素（0-based 索引）
    return quickSelect(0, n - 1, (n - 1) / 2);
}

/**
 * 【工程化应用】外部排序辅助函数
 * 当数据量很大，无法全部加载到内存时，使用分块快速排序
 * @param chunk 内存中的数据块
 */
public static void externalSortHelper(int[] chunk) {
    // 对内存中的数据块进行快速排序
    quickSortOptimized(0, chunk.length - 1);
}

/**
 * 【异常处理】健壮的快速排序实现
 * @param nums 待排序数组
 */
public static void robustQuickSort(int[] nums) {
    // 空数组检查
    if (nums == null || nums.length <= 1) {
        return;
```

```
    }

    // 执行排序
    System.arraycopy(nums, 0, arr, 0, nums.length);
    quickSortOptimized(0, nums.length - 1);
    System.arraycopy(arr, 0, nums, 0, nums.length);
}

/**
 * 【AtCoder ABC121C 解法】Energy Drink Collector（贪心+排序）
 * @param drinks 能量饮料信息，每行包含价格和数量
 * @param budget 预算
 * @return 能获得的最大能量
 */
public static long energyDrinkCollector(int[][] drinks, int budget) {
    // 按价格升序排序
    java.util.Arrays.sort(drinks, (a, b) -> a[0] - b[0]);

    long totalEnergy = 0;
    int remainingBudget = budget;

    for (int[] drink : drinks) {
        int price = drink[0]; // 价格
        int amount = drink[1]; // 数量

        int canBuy = Math.min(amount, remainingBudget / price);
        totalEnergy += canBuy;
        remainingBudget -= canBuy * price;

        if (remainingBudget < price) {
            break;
        }
    }

    return totalEnergy;
}

/**
 * 【Codeforces 401C 解法】Team（贪心构造）
 * @param n 0 的数量
 * @param m 1 的数量
 * @return 构造的 01 序列
 */
```

```
public static String constructTeam(int n, int m) {
    // 构造一个01序列，满足特定约束条件
    StringBuilder sb = new StringBuilder();

    while (n > 0 || m > 0) {
        // 优先放置0的情况
        if (n > m) {
            // 检查是否可以放置00
            if (n >= 2 && m >= 1) {
                sb.append("001");
                n -= 2;
                m -= 1;
            } else {
                sb.append("0");
                n -= 1;
            }
        } else {
            // 优先放置1的情况
            if (m >= 2 && n >= 1) {
                sb.append("110");
                m -= 2;
                n -= 1;
            } else {
                sb.append("1");
                m -= 1;
            }
        }
    }

    return sb.toString();
}

// 荷兰国旗问题
public static int first, last;

/**
 * 三路分区函数 - 改进版本
 * 将数组划分为三个部分：<x的部分、=x的部分、>x的部分
 * 更新全局变量first和last为=x区域的左右边界
 * @param l 分区区间的左边界（包含）
 * @param r 分区区间的右边界（包含）
 * @param x 基准值
 */
```

```
    public static void partition2(int l, int r, int x) {
        // 初始化等于 x 区域的左右边界
        first = l;
        last = r;

        // 当前处理的元素索引
        int i = l;

        // 当 i 不超过等于区域的右边界时继续循环
        while (i <= last) {
            // 如果当前元素等于基准值 x
            if (arr[i] == x) {
                // 直接移动到下一个元素
                i++;
            }
            // 如果当前元素小于基准值 x
            else if (arr[i] < x) {
                // 将当前元素与等于区域左边界的元素交换
                swap(first++, i++);
            }
            // 如果当前元素大于基准值 x
            else {
                // 将当前元素与等于区域右边界的元素交换
                // 注意这里 i 不自增，因为交换过来的元素还未处理
                swap(i, last--);
            }
        }
    }

}
```

==========

文件: Code02_QuickSort.java

==========

```
package class023;

// 随机快速排序，填函数练习风格
// 测试链接 : https://leetcode.cn/problems/sort-an-array/

/*
 * 补充题目列表:
 *
```

* 1. LeetCode 912. 排序数组
*   链接: https://leetcode.cn/problems/sort-an-array/
*   题目描述: 给你一个整数数组 nums，请你将该数组升序排列。
*   解题思路: 使用快速排序算法对数组进行排序。
*
* 2. LeetCode 215. 数组中的第K个最大元素
*   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。
*   解题思路: 可以使用快速选择算法，在快速排序的基础上进行优化，只处理包含目标元素的区间。
*
* 3. 剑指 Offer 40. 最小的k个数
*   链接: https://leetcode.cn/problems/zui-xiao-de-kge-shu-lcof/
*   题目描述: 输入整数数组 arr ，找出其中最小的 k 个数。
*   解题思路: 使用快速选择算法或者快速排序算法找出最小的k个数。
*
* 4. 牛客网 - 快速排序
*   链接: https://www.nowcoder.com/practice/e016ad9b7f0b45048c58a9f27ba618bf
*   题目描述: 实现快速排序算法
*
* 5. PAT 1101 Quick Sort
*   链接: https://pintia.cn/problem-sets/994805342720868352/problems/994805366343188480
*   题目描述: 快速排序中的主元(pivot)是左面都比它小、右边都比它大的位置对应的数字。找出所有满足条件的主元。
*
* 6. 洛谷 P1177 【模板】快速排序
*   链接: https://www.luogu.com.cn/problem/P1177
*   题目描述: 利用快速排序算法将读入的N个数从小到大排序后输出。
*
* 7. LeetCode 75. 颜色分类
*   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、共 n 个元素的数组 nums，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。
*   解题思路: 使用三路快速排序的思想，将数组分为三个区域：小于基准值、等于基准值、大于基准值。
*
* 8. LeetCode 283. 移动零
*   链接: https://leetcode.cn/problems/move-zeroes/
*   题目描述: 给定一个数组 nums，编写一个函数将所有 0 移动到数组的末尾，同时保持非零元素的相对顺序。
*   解题思路: 使用快速排序的分区思想，将非零元素移到数组前面，零元素移到数组后面。
*
* 9. Codeforces 401C. Team
*   链接: https://codeforces.com/problemset/problem/401/C
*   题目描述: 构造一个01序列，满足特定的约束条件。

*   解题思路: 在某些构造方法中可以使用排序来优化解的生成。
*
* 10. AtCoder ABC121C. Energy Drink Collector
*   链接: https://atcoder.jp/contests/abc121/tasks/abc121_c
*   题目描述: 购买能量饮料以获得最少的总花费。
*   解题思路: 按价格排序后贪心选择。
*
* 算法复杂度分析:
* 时间复杂度:
*   - 最好情况: O(n log n) - 每次划分都能将数组平均分成两部分
*   - 平均情况: O(n log n) - 随机选择基准值的情况下
*   - 最坏情况: O(n^2) - 每次选择的基准值都是最大或最小值
* 空间复杂度:
*   - O(log n) - 递归调用栈的深度
*
* 算法优化策略:
* 1. 随机选择基准值 - 避免最坏情况的出现
* 2. 三路快排 - 处理重复元素较多的情况
* 3. 小数组使用插入排序 - 减少递归开销
* 4. 尾递归优化 - 减少栈空间使用
*
* 工程化考量:
* 1. 异常处理: 处理空数组、null 输入等边界情况
* 2. 性能优化: 对于小数组使用插入排序优化
* 3. 内存使用: 原地排序减少额外空间开销
* 4. 稳定性: 标准快排不稳定，如需稳定排序需特殊处理
*
* 调试技巧:
* 1. 打印中间过程: 在分区操作后打印数组状态
* 2. 断言验证: 验证分区后各部分的正确性
* 3. 边界测试: 测试空数组、单元素、重复元素等边界情况
*
* 跨语言实现差异:
* 1. Java - 数组作为对象，有边界检查，使用 Math.random()生成随机数
* 2. C++ - 数组为指针，无边界检查，使用 rand()生成随机数
* 3. Python - 使用列表，动态类型，使用 random 模块生成随机数
*
* 面试技巧:
* 1. 理解快排与其它排序算法的比较（如归并排序、堆排序）
* 2. 掌握快排的优化方法（随机化、三路快排等）
* 3. 理解快排在不同数据分布下的性能表现
* 4. 能够分析快排的稳定性和适用场景
*/

```
public class Code02_QuickSort {

    /**
     * 入口函数：对整数数组进行排序
     * @param nums 待排序的整数数组
     * @return 排序后的数组
     */
    public static int[] sortArray(int[] nums) {
        // 如果数组长度大于 1 才需要排序，长度为 0 或 1 的数组天然有序
        if (nums.length > 1) {
            // 调用改进版的快速排序算法对数组进行排序
            quickSort2(nums, 0, nums.length - 1);
        }
        return nums;
    }

    /**
     * 随机快速排序经典版(不推荐使用)
     * @param arr 待排序数组
     * @param l 排序区间的左边界（包含）
     * @param r 排序区间的右边界（包含）
     */
    public static void quickSort1(int[] arr, int l, int r) {
        // 递归终止条件：当左边界大于等于右边界时，表示区间内没有元素或只有一个元素，无需排序
        if (l >= r) {
            return;
        }

        // 随机选择基准值 pivot，避免最坏情况的发生
        // Math.random()*(r-l+1)生成[0,r-l]之间的随机整数
        // 加上 l 后得到[l,r]之间的随机索引
        int x = arr[l + (int) (Math.random() * (r - l + 1))];

        // 调用 partition1 函数进行分区操作，返回等于 x 区域的右边界
        int mid = partition1(arr, l, r, x);

        // 递归处理小于等于 x 的区域
        quickSort1(arr, l, mid - 1);

        // 递归处理大于 x 的区域
        quickSort1(arr, mid + 1, r);
    }
```

```
/**
 * 分区函数 - 经典版本
 * 将数组划分为两个部分：小于等于 x 的部分和大于 x 的部分
 * 并确保划分完成后小于等于 x 区域的最后一个数字是 x
 * @param arr 待分区的数组
 * @param l 分区区间的左边界（包含）
 * @param r 分区区间的右边界（包含）
 * @param x 基准值
 * @return 等于 x 区域的右边界索引
 */
public static int partition1(int[] arr, int l, int r, int x) {
    // a 表示小于等于 x 区域的右边界下一个位置
    // xi 记录在小于等于 x 区域内任意一个 x 的位置
    int a = l, xi = 0;

    // 遍历整个区间
    for (int i = l; i <= r; i++) {
        // 如果当前元素小于等于基准值 x
        if (arr[i] <= x) {
            // 将当前元素交换到小于等于 x 区域
            swap(arr, a, i);

            // 如果交换过来的元素正好等于 x，则记录其位置
            if (arr[a] == x) {
                xi = a;
            }

            // 小于等于 x 区域向右扩展一位
            a++;
        }
    }

    // 将记录的 x 位置的元素与小于等于 x 区域的最后一个元素交换
    // 确保小于等于 x 区域的最后一个数字是 x
    swap(arr, xi, a - 1);

    // 返回等于 x 区域的右边界索引
    return a - 1;
}

/**
 * 交换数组中两个位置的元素
```

```
 * @param arr 数组
 * @param i 第一个位置
 * @param j 第二个位置
 */
public static void swap(int[] arr, int i, int j) {
    int tmp = arr[i];
    arr[i] = arr[j];
    arr[j] = tmp;
}

/**
 * 随机快速排序改进版(推荐使用)
 * 使用三路快排优化，更好地处理重复元素
 * @param arr 待排序数组
 * @param l 排序区间的左边界（包含）
 * @param r 排序区间的右边界（包含）
 */
public static void quickSort2(int[] arr, int l, int r) {
    // 递归终止条件：当左边界大于等于右边界时，表示区间内没有元素或只有一个元素，无需排序
    if (l >= r) {
        return;
    }

    // 随机选择基准值 pivot，避免最坏情况的发生
    int x = arr[l + (int) (Math.random() * (r - l + 1))];

    // 调用 partition2 函数进行三路分区操作
    partition2(arr, l, r, x);

    // 为了防止底层的递归过程覆盖全局变量
    // 这里用临时变量记录等于 x 区域的左右边界
    int left = first;
    int right = last;

    // 递归处理小于 x 的区域
    quickSort2(arr, l, left - 1);

    // 递归处理大于 x 的区域
    quickSort2(arr, right + 1, r);
}

// 荷兰国旗问题中的全局变量，用于记录等于基准值区域的左右边界
// first: 等于区域的左边界
```

```
    // last: 等于区域的右边界
    public static int first, last;

    /**
     * 三路分区函数 - 改进版本
     * 将数组划分为三个部分：<x 的部分、=x 的部分、>x 的部分
     * 更新全局变量 first 和 last 为=x 区域的左右边界
     * @param arr 待分区的数组
     * @param l 分区区间的左边界（包含）
     * @param r 分区区间的右边界（包含）
     * @param x 基准值
     */
    public static void partition2(int[] arr, int l, int r, int x) {
        // 初始化等于 x 区域的左右边界
        first = l;
        last = r;

        // 当前处理的元素索引
        int i = l;

        // 当 i 不超过等于区域的右边界时继续循环
        while (i <= last) {
            // 如果当前元素等于基准值 x
            if (arr[i] == x) {
                // 直接移动到下一个元素
                i++;
            }
            // 如果当前元素小于基准值 x
            else if (arr[i] < x) {
                // 将当前元素与等于区域左边界的元素交换
                swap(arr, first++, i++);
            }
            // 如果当前元素大于基准值 x
            else {
                // 将当前元素与等于区域右边界的元素交换
                // 注意这里 i 不自增，因为交换过来的元素还未处理
                swap(arr, i, last--);
            }
        }
    }

}
```

==========

文件: Code03_QuickSort_Leetcode215.java
==========
```
package class023;

// LeetCode 215. 数组中的第K个最大元素
// 测试链接 : 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。
// 解题思路: 使用快速选择算法，在快速排序的基础上进行优化，只处理包含目标元素的区间。

/*
 * 补充题目列表:
 *
 * 1. LeetCode 215. 数组中的第K个最大元素
 *    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。
 *    解题思路: 使用快速选择算法，在快速排序的基础上进行优化，只处理包含目标元素的区间。
 *
 * 2. LeetCode 347. 前 K 个高频元素
 *    链接: https://leetcode.cn/problems/top-k-frequent-elements/
 *    题目描述: 给你一个整数数组 nums 和一个整数 k ，请你返回其中出现频率前 k 高的元素。
 *    解题思路: 使用堆或者快速选择算法来找出前k个高频元素。
 *
 * 3. LeetCode 973. 最接近原点的 K 个点
 *    链接: https://leetcode.cn/problems/k-closest-points-to-origin/
 *    题目描述: 给定一个points数组和整数K，返回最接近原点的K个点。
 *    解题思路: 计算每个点到原点的距离，然后使用快速选择算法找出最小的K个距离。
 *
 * 4. LeetCode 324. 摆动排序 II
 *    链接: https://leetcode.cn/problems/wiggle-sort-ii/
 *    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。
 *    解题思路: 先排序，然后通过特定的索引映射来构造摆动序列。
 *
 * 5. LeetCode 414. 第三大的数
 *    链接: https://leetcode.cn/problems/third-maximum-number/
 *    题目描述: 给你一个非空数组，返回此数组中第三大的数。
 *    解题思路: 使用一次遍历维护三个最大值，或者使用快速选择算法。
 *
 * 6. LeetCode 462. 最少移动次数使数组元素相等 II
 *    链接: https://leetcode.cn/problems/minimum-moves-to-equal-array-elements-ii/
 *    题目描述: 给你一个长度为 n 的整数数组 nums ，返回使所有数组元素相等需要的最少移动数。
```

*   解题思路: 找到中位数，所有元素向中位数移动的步数之和最小。
*
* 7. LeetCode 703. 数据流中的第 K 大元素
*   链接: https://leetcode.cn/problems/kth-largest-element-in-a-stream/
*   题目描述: 设计一个找到数据流中第 k 大元素的类。
*   解题思路: 使用最小堆维护前 k 大的元素。
*
* 8. LeetCode 215. Kth Largest Element in an Array (重复题目，但提供更多解法)
*   链接: https://leetcode.cn/problems/kth-largest-element-in-an-array/
*   解题思路:
*   方法 1: 快速选择算法（平均时间复杂度 O(n)）
*   方法 2: 堆排序（时间复杂度 O(n log k)）
*   方法 3: 全排序后取第 k 个（时间复杂度 O(n log n)）
*
* 算法复杂度分析:
* 时间复杂度:
*   - 最好情况: O(n) - 每次划分都能将数组平均分成两部分
*   - 平均情况: O(n) - 随机选择基准值的情况下
*   - 最坏情况: O(n^2) - 每次选择的基准值都是最大或最小值
* 空间复杂度:
*   - O(log n) - 递归调用栈的深度
*
* 算法优化策略:
* 1. 随机选择基准值 - 避免最坏情况的出现
* 2. 三路快排 - 处理重复元素较多的情况
* 3. 剪枝优化 - 只处理包含目标元素的区间
*
* 工程化考量:
* 1. 异常处理: 处理 k 超出数组长度的情况
* 2. 性能优化: 对于小数组使用插入排序优化
* 3. 内存使用: 原地排序减少额外空间开销
* 4. 稳定性: 快速选择算法不稳定，如需稳定需特殊处理
*
* 调试技巧:
* 1. 打印中间过程: 在分区操作后打印数组状态
* 2. 断言验证: 验证分区后各部分的正确性
* 3. 边界测试: 测试 k=1, k=n 等边界情况
*
* 面试技巧:
* 1. 理解快速选择与二分查找的区别
* 2. 掌握快速选择的优化方法（随机化、三路快排等）
* 3. 理解快速选择在不同数据分布下的性能表现
* 4. 能够分析快速选择的适用场景

```
 */

public class Code03_QuickSort_Leetcode215 {

    /**
     * 查找数组中第 k 个最大的元素
     * 算法核心思想：第 k 大元素等价于排序后第(nums.length - k)小元素
     * @param nums 整数数组
     * @param k 第 k 个最大的元素（1-based）
     * @return 第 k 个最大的元素值
     */
    public int findKthLargest(int[] nums, int k) {
        // 第 k 大元素在排序后数组中的索引是 nums.length - k（0-based）
        // 例如：数组[1,2,3,4,5,6]，第 2 大元素是 5，索引为 6-2=4
        return quickSelect(nums, 0, nums.length - 1, nums.length - k);
    }

    /**
     * 快速选择算法实现
     * 与快速排序的区别：只处理包含目标元素的子数组，而非全部子数组
     * 平均时间复杂度：O(n)，最坏情况：O(n²)
     * @param nums 数组
     * @param l 当前处理区间的左边界（包含）
     * @param r 当前处理区间的右边界（包含）
     * @param index 目标元素在排序后数组中的索引位置
     * @return 目标元素值
     */
    private int quickSelect(int[] nums, int l, int r, int index) {
        // 随机选择基准值进行分区，避免最坏情况
        int q = randomPartition(nums, l, r);

        // 如果分区点正好是目标索引，说明找到了目标元素
        if (q == index) {
            // 直接返回目标元素
            return nums[q];
        } else {
            // 根据分区点与目标索引的关系，决定在哪个子数组中继续查找
            // 如果分区点索引小于目标索引，说明目标元素在右半部分
            if (q < index) {
                return quickSelect(nums, q + 1, r, index);
            }
            // 如果分区点索引大于目标索引，说明目标元素在左半部分
            else {
```

```
            return quickSelect(nums, l, q - 1, index);
        }
    }
}

/**
 * 随机分区函数
 * 通过随机选择基准值来避免最坏情况的发生
 * @param nums 数组
 * @param l 左边界
 * @param r 右边界
 * @return 分区点索引（基准值在数组中的最终位置）
 */
private int randomPartition(int[] nums, int l, int r) {
    // 在[l, r]范围内随机选择一个索引作为基准值
    int i = l + (int) (Math.random() * (r - l + 1));

    // 将随机选择的基准值交换到末尾位置，便于后续分区操作
    swap(nums, i, r);

    // 进行标准分区操作
    return partition(nums, l, r);
}

/**
 * 分区函数（标准实现）
 * 将数组分为两部分：小于等于基准值的部分和大于基准值的部分
 * @param nums 数组
 * @param l 左边界
 * @param r 右边界（基准值所在位置）
 * @return 基准值在分区后的最终位置索引
 */
private int partition(int[] nums, int l, int r) {
    // 选择最后一个元素作为基准值
    int x = nums[r];

    // i 表示小于等于基准值的区域的右边界（不包含）
    // 初始时小于等于区域为空，所以 i = l - 1
    int i = l - 1;

    // 遍历数组[l, r-1]，将小于等于基准值的元素放到左侧
    for (int j = l; j < r; ++j) {
        // 如果当前元素小于等于基准值
```

```
            if (nums[j] <= x) {
                // 扩展小于等于区域
                i++;
                // 将当前元素交换到小于等于区域
                swap(nums, i, j);
            }
        }

        // 将基准值放到小于等于区域的下一个位置（即正确位置）
        swap(nums, i + 1, r);

        // 返回基准值的最终索引位置
        return i + 1;
    }

    /**
     * 交换数组中两个元素的位置
     * @param nums 数组
     * @param i 索引 1
     * @param j 索引 2
     */
    private void swap(int[] nums, int i, int j) {
        int temp = nums[i];
        nums[i] = nums[j];
        nums[j] = temp;
    }

    /**
     * 测试用例和验证代码
     * @param args 命令行参数
     */
    public static void main(String[] args) {
        Code03_QuickSort_Leetcode215 solution = new Code03_QuickSort_Leetcode215();

        // 测试用例 1：普通情况
        // 数组: [3, 2, 1, 5, 6, 4]，排序后为[1, 2, 3, 4, 5, 6]
        // 第 2 大的元素是 5（索引为 4）
        int[] nums1 = {3, 2, 1, 5, 6, 4};
        int k1 = 2;
        System.out.println("数组: [3, 2, 1, 5, 6, 4], k=2, 第 2 大的元素是: " +
solution.findKthLargest(nums1, k1)); // 输出: 5

        // 测试用例 2：包含重复元素的情况
```

```
        // 数组: [3, 2, 3, 1, 2, 4, 5, 5, 6]，排序后为[1, 2, 2, 3, 3, 4, 5, 5, 6]
        // 第 4 大的元素是 4（索引为 5）
        int[] nums2 = {3, 2, 3, 1, 2, 4, 5, 5, 6};
        int k2 = 4;
        System.out.println("数组: [3, 2, 3, 1, 2, 4, 5, 5, 6], k=4, 第 4 大的元素是: " +
solution.findKthLargest(nums2, k2)); // 输出: 4

        // 测试用例 3：边界情况-k=1（最大元素）
        int[] nums3 = {1, 2, 3, 4, 5};
        int k3 = 1;
        System.out.println("数组: [1, 2, 3, 4, 5], k=1, 最大元素是: " +
solution.findKthLargest(nums3, k3)); // 输出: 5

        // 测试用例 4：边界情况-k=n（最小元素）
        int[] nums4 = {1, 2, 3, 4, 5};
        int k4 = 5;
        System.out.println("数组: [1, 2, 3, 4, 5], k=5, 最小元素是: " +
solution.findKthLargest(nums4, k4)); // 输出: 1
    }
}

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

文件: Code04_QuickSort_JZ40.java
==============================================
package class023;

// 剑指 Offer 40. 最小的 k 个数
// 测试链接 : https://leetcode.cn/problems/zui-xiao-de-kge-shu-lcof/
// 题目描述: 输入整数数组 arr ，找出其中最小的 k 个数。
// 解题思路: 使用快速选择算法或者快速排序算法找出最小的 k 个数。

/*
 * 补充题目列表:
 *
 * 1. 剑指 Offer 40. 最小的 k 个数
 *    链接: https://leetcode.cn/problems/zui-xiao-de-kge-shu-lcof/
 *    题目描述: 输入整数数组 arr ，找出其中最小的 k 个数。
 *    解题思路: 使用快速选择算法或者快速排序算法找出最小的 k 个数。
 *
 * 2. LeetCode 215. 数组中的第 K 个最大元素
 *    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。
```

```
 *     解题思路: 使用快速选择算法，在快速排序的基础上进行优化，只处理包含目标元素的区间。
 *
 * 3. LeetCode 703. 数据流中的第 K 大元素
 *     链接: https://leetcode.cn/problems/kth-largest-element-in-a-stream/
 *     题目描述: 设计一个找到数据流中第 k 大元素的类。
 *     解题思路: 使用最小堆维护前 k 大的元素。
 *
 * 4. LeetCode 347. 前 K 个高频元素
 *     链接: https://leetcode.cn/problems/top-k-frequent-elements/
 *     题目描述: 给你一个整数数组 nums 和一个整数 k ，请你返回其中出现频率前 k 高的元素。
 *     解题思路: 使用堆或者快速选择算法来找出前 k 个高频元素。
 *
 * 5. LeetCode 973. 最接近原点的 K 个点
 *     链接: https://leetcode.cn/problems/k-closest-points-to-origin/
 *     题目描述: 给定一个 points 数组和整数 K，返回最接近原点的 K 个点。
 *     解题思路: 计算每个点到原点的距离，然后使用快速选择算法找出最小的 K 个距离。
 *
 * 6. 牛客网 - 最小的 k 个数
 *     链接: https://www.nowcoder.com/practice/6a296eb82cf844ca8539b57c23e6e9bf
 *     题目描述: 输入 n 个整数，找出其中最小的 K 个数。
 *     解题思路: 使用快速选择算法或者堆来解决。
 *
 * 7. LeetCode 324. 摆动排序 II
 *     链接: https://leetcode.cn/problems/wiggle-sort-ii/
 *     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。
 *     解题思路: 先排序，然后通过特定的索引映射来构造摆动序列。
 *
 * 8. LeetCode 215. Kth Largest Element in an Array (重复题目，但提供更多解法)
 *     链接: https://leetcode.cn/problems/kth-largest-element-in-an-array/
 *     解题思路:
 *       方法 1: 快速选择算法（平均时间复杂度 O(n)）
 *       方法 2: 堆排序（时间复杂度 O(n log k)）
 *       方法 3: 全排序后取第 k 个（时间复杂度 O(n log n)）
 *
 * 算法复杂度分析:
 * 时间复杂度:
 *   - 快速排序: O(n log n)
 *   - 快速选择: O(n) 平均情况
 *   - 堆排序: O(n log k)
 * 空间复杂度:
 *   - O(log n) - 递归调用栈的深度（快速选择/快速排序）
 *   - O(k) - 堆的存储空间
```

```
 *
 * 算法优化策略:
 * 1. 随机选择基准值 - 避免最坏情况的出现
 * 2. 三路快排 - 处理重复元素较多的情况
 * 3. 剪枝优化 - 只处理包含目标元素的区间
 * 4. 堆优化 - 使用最小堆维护 k 个最大元素
 *
 * 工程化考量:
 * 1. 异常处理: 处理 k 超出数组长度的情况
 * 2. 性能优化: 对于小数组使用插入排序优化
 * 3. 内存使用: 原地排序减少额外空间开销
 * 4. 稳定性: 快速选择算法不稳定，如需稳定需特殊处理
 *
 * 调试技巧:
 * 1. 打印中间过程: 在分区操作后打印数组状态
 * 2. 断言验证: 验证分区后各部分的正确性
 * 3. 边界测试: 测试 k=0, k=n 等边界情况
 *
 * 面试技巧:
 * 1. 理解快速选择与堆排序的比较
 * 2. 掌握不同算法在不同数据规模下的性能表现
 * 3. 理解各种算法的适用场景
 * 4. 能够根据具体需求选择合适的算法
 */

import java.util.Arrays;

public class Code04_QuickSort_JZ40 {

    /**
     * 获取数组中最小的 k 个数
     * 算法核心思想：使用快速选择算法找出最小的 k 个数
     * @param arr 输入数组
     * @param k 需要返回的最小数的个数
     * @return 包含最小 k 个数的数组
     */
    public int[] getLeastNumbers(int[] arr, int k) {
        // 边界条件检查：如果 k 大于等于数组长度，直接返回整个数组
        if (k >= arr.length) {
            return arr;
        }

        // 使用快速排序方法获取最小的 k 个数
```

```
    return quickSort(arr, 0, arr.length - 1, k);
}

/**
 * 快速排序方法获取最小的 k 个数
 * 与标准快速排序不同：只处理包含目标元素的子数组
 * @param arr 数组
 * @param l 当前处理区间的左边界（包含）
 * @param r 当前处理区间的右边界（包含）
 * @param k 需要返回的最小数的个数
 * @return 包含最小 k 个数的数组
 */
private int[] quickSort(int[] arr, int l, int r, int k) {
    // 随机选择基准值，避免最坏情况
    int i = l + (int) (Math.random() * (r - l + 1));

    // 将随机选择的基准值交换到第一个位置，便于后续分区操作
    swap(arr, l, i);

    // 基准值
    int temp = arr[l];

    // 双指针分区：left 从左向右扫描，right 从右向左扫描
    int left = l, right = r;

    // 双指针分区操作
    while (left < right) {
        // 从右向左找小于基准值的元素
        while (left < right && arr[right] >= temp) {
            right--;
        }

        // 从左向右找大于基准值的元素
        while (left < right && arr[left] <= temp) {
            left++;
        }

        // 如果 left < right，说明找到了需要交换的元素对
        if (left < right) {
            // 交换元素
            swap(arr, left, right);
        }
    }
```

```
        // 将基准值放到正确位置（left == right 时的位置）
        swap(arr, l, left);

        // 根据分区点位置决定下一步操作
        if (k < left) {
            // 如果 k 小于分区点位置，说明前 k 个最小元素都在左半部分
            // 在左半部分继续查找
            return quickSort(arr, l, left - 1, k);
        }
        if (k > left) {
            // 如果 k 大于分区点位置，说明左半部分的所有元素（包括基准值）都是前 k 个最小元素
            // 但还需要在右半部分找剩余的元素
            // 在右半部分继续查找
            return quickSort(arr, left + 1, r, k);
        }

        // 如果 k 等于分区点位置，说明前 k 个最小元素正好是数组的前 k 个元素
        // 直接返回前 k 个元素
        return Arrays.copyOf(arr, k);
    }

    /**
     * 交换数组中两个元素的位置
     * @param arr 数组
     * @param i 索引 1
     * @param j 索引 2
     */
    private void swap(int[] arr, int i, int j) {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }

    /**
     * 测试用例和验证代码
     * @param args 命令行参数
     */
    public static void main(String[] args) {
        Code04_QuickSort_JZ40 solution = new Code04_QuickSort_JZ40();

        // 测试用例 1：普通情况
        // 数组: [3, 2, 1]，排序后为[1, 2, 3]
```

```
        // 最小的2个数是[1, 2]
        int[] arr1 = {3, 2, 1};
        int k1 = 2;
        int[] result1 = solution.getLeastNumbers(arr1, k1);
        System.out.println("数组: [3, 2, 1], k=2, 最小的2个数是: " + Arrays.toString(result1));
// 输出: [1, 2] 或 [2, 1]

        // 测试用例2：边界情况-k=1
        // 数组: [0, 1, 2, 1]，排序后为[0, 1, 1, 2]
        // 最小的1个数是[0]
        int[] arr2 = {0, 1, 2, 1};
        int k2 = 1;
        int[] result2 = solution.getLeastNumbers(arr2, k2);
        System.out.println("数组: [0, 1, 2, 1], k=1, 最小的1个数是: " +
Arrays.toString(result2)); // 输出: [0]

        // 测试用例3：k等于数组长度
        int[] arr3 = {0, 1, 2, 1};
        int k3 = 4;
        int[] result3 = solution.getLeastNumbers(arr3, k3);
        System.out.println("数组: [0, 1, 2, 1], k=4, 最小的4个数是: " +
Arrays.toString(result3)); // 输出: [0, 1, 2, 1]

        // 测试用例4：k为0
        int[] arr4 = {0, 1, 2, 1};
        int k4 = 0;
        int[] result4 = solution.getLeastNumbers(arr4, k4);
        System.out.println("数组: [0, 1, 2, 1], k=0, 最小的0个数是: " +
Arrays.toString(result4)); // 输出: []
    }
}
```

==========

文件: Code05_QuickSort_Cplusplus.cpp

==========

```
// C++版本快速排序实现
// 包含多种快速排序的实现方式

/*
 * 补充题目列表:
 *
 * 1. LeetCode 912. 排序数组
```

```
 *     链接: https://leetcode.cn/problems/sort-an-array/
 *     题目描述: 给你一个整数数组 nums，请你将该数组升序排列。
 *     时间复杂度: O(n log n)，空间复杂度: O(log n)
 *     最优解: 快速排序或归并排序
 *
 * 2. 洛谷 P1177 【模板】快速排序
 *     链接: https://www.luogu.com.cn/problem/P1177
 *     题目描述: 利用快速排序算法将读入的 N 个数从小到大排序后输出。
 *     时间复杂度: O(n log n)，空间复杂度: O(log n)
 *     最优解: 快速排序算法实现
 *
 * 3. LeetCode 215. 数组中的第 K 个最大元素
 *     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。
 *     时间复杂度: O(n) 平均，空间复杂度: O(log n)
 *     最优解: 快速选择算法
 *
 * 4. LeetCode 75. 颜色分类
 *     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、共 n 个元素的数组 nums，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。
 *     时间复杂度: O(n)，空间复杂度: O(1)
 *     最优解: 三路快排思想
 *
 * 5. LeetCode 283. 移动零
 *     链接: https://leetcode.cn/problems/move-zeroes/
 *     题目描述: 给定一个数组 nums，编写一个函数将所有 0 移动到数组的末尾，同时保持非零元素的相对顺序。
 *     时间复杂度: O(n)，空间复杂度: O(1)
 *     最优解: 双指针法
 *
 * 6. Codeforces 401C. Team
 *     链接: https://codeforces.com/problemset/problem/401/C
 *     题目描述: 构造一个 01 序列，满足特定的约束条件。
 *     时间复杂度: O(n+m)，空间复杂度: O(n+m)
 *     最优解: 贪心构造
 *
 * 7. AtCoder ABC121C. Energy Drink Collector
 *     链接: https://atcoder.jp/contests/abc121/tasks/abc121_c
 *     题目描述: 购买能量饮料以获得最少的总花费。
 *     时间复杂度: O(n log n)，空间复杂度: O(1)
 *     最优解: 贪心+排序
 *
```

* 8. 牛客网 - 快速排序
*   链接: https://www.nowcoder.com/practice/e016ad9b7f0b45048c58a9f27ba618bf
*   题目描述: 实现快速排序算法
*   时间复杂度: O(n log n), 空间复杂度: O(log n)
*   最优解: 标准快速排序实现
*
* 9. PAT 1101 Quick Sort
*   链接: https://pintia.cn/problem-sets/994805342720868352/problems/994805366343188480
*   题目描述: 快速排序中的主元(pivot)是左面都比它小、右边都比它大的位置对应的数字。找出所有满足条件的主元。
*   时间复杂度: O(n), 空间复杂度: O(n)
*   最优解: 预处理左右边界最大值数组
*
* 10. 剑指 Offer 40. 最小的 k 个数
*   链接: https://leetcode.cn/problems/zui-xiao-de-kge-shu-lcof/
*   题目描述: 输入整数数组 arr ，找出其中最小的 k 个数。
*   时间复杂度: O(n) 平均, 空间复杂度: O(log n)
*   最优解: 快速选择算法
*
* 11. 杭电 OJ 1425. sort
*   链接: http://acm.hdu.edu.cn/showproblem.php?pid=1425
*   题目描述: 对整数数组进行快速排序
*   时间复杂度: O(n log n), 空间复杂度: O(log n)
*   最优解: 快速排序或堆排序
*
* 12. POJ 2388. Who's in the Middle
*   链接: http://poj.org/problem?id=2388
*   题目描述: 找出一组数的中位数，快速选择的经典应用
*   时间复杂度: O(n) 平均, 空间复杂度: O(log n)
*   最优解: 快速选择算法找中位数
*
* 13. AizuOJ ALDS1_6_C. Quick Sort
*   链接: https://onlinejudge.u-aizu.ac.jp/problems/ALDS1_6_C
*   题目描述: 实现快速排序算法并输出每一步的分区结果
*   时间复杂度: O(n log n), 空间复杂度: O(log n)
*   最优解: 快速排序算法实现
*
* 14. LeetCode 169. 多数元素
*   链接: https://leetcode.cn/problems/majority-element/
*   题目描述: 给定一个大小为 n 的数组，找到其中的多数元素
*   时间复杂度: O(n), 空间复杂度: O(1)
*   最优解: Boyer-Moore 投票算法（与快速选择思想相关）
*

```
 * 15. LeetCode 274. H 指数
 *     链接: https://leetcode.cn/problems/h-index/
 *     题目描述: 计算研究人员的 h 指数
 *     时间复杂度: O(n) 平均, 空间复杂度: O(n)
 *     最优解: 计数排序或快速选择
 *
 * 算法复杂度分析:
 * 时间复杂度:
 *   - 最好情况: O(n log n) - 每次划分都能将数组平均分成两部分
 *   - 平均情况: O(n log n) - 随机选择基准值的情况下
 *   - 最坏情况: O(n^2) - 每次选择的基准值都是最大或最小值
 * 空间复杂度:
 *   - O(log n) - 递归调用栈的深度
 *
 * 算法优化策略:
 * 1. 随机选择基准值 - 避免最坏情况的出现
 * 2. 三路快排 - 处理重复元素较多的情况
 * 3. 小数组使用插入排序 - 减少递归开销
 * 4. 尾递归优化 - 减少栈空间使用
 *
 * 工程化考量:
 * 1. 异常处理: 处理空数组、null 输入等边界情况
 * 2. 性能优化: 对于小数组使用插入排序优化
 * 3. 内存使用: 原地排序减少额外空间开销
 * 4. 稳定性: 标准快排不稳定, 如需稳定排序需特殊处理
 *
 * 与 Java 版本的差异:
 * 1. C++使用指针和数组, 没有边界检查, 性能更高
 * 2. C++使用 rand()函数生成随机数, Java 使用 Math.random()
 * 3. C++可以直接操作内存, Java 通过虚拟机管理内存
 * 4. C++需要手动管理内存, Java 有垃圾回收机制
 * 5. C++模板支持泛型编程, Java 使用泛型
 *
 * 调试技巧:
 * 1. 使用 gdb 调试器跟踪递归过程
 * 2. 添加调试输出打印数组状态
 * 3. 使用断言验证分区正确性
 * 4. 测试边界情况（空数组、单元素等）
 */

#include <iostream>
#include <vector>
#include <algorithm>
```

```
#include <ctime>
#include <cstdlib>

// 使用 std 命名空间
using namespace std;

class QuickSortSolution {
public:
    /**
     * 方法 1: 基础快速排序
     * 使用双指针分区法实现快速排序
     * @param arr 待排序数组
     * @param l 排序区间的左边界（包含）
     * @param r 排序区间的右边界（包含）
     */
    void quickSort1(vector<int>& arr, int l, int r) {
        // 递归终止条件：当左边界大于等于右边界时，表示区间内没有元素或只有一个元素，无需排序
        if (l >= r) return;

        // 随机选择基准值，避免最坏情况
        // rand() % (r - l + 1)生成[0, r-l]之间的随机整数
        // 加上 l 后得到[l, r]之间的随机索引
        int i = l + rand() % (r - l + 1);

        // 将随机选择的基准值交换到第一个位置，便于后续分区操作
        swap(arr[l], arr[i]);

        // 基准值
        int pivot = arr[l];

        // 双指针分区：left 从左向右扫描，right 从右向左扫描
        int left = l, right = r;

        // 双指针分区操作
        while (left < right) {
            // 从右向左找小于基准值的元素
            while (left < right && arr[right] >= pivot) right--;

            // 从左向右找大于基准值的元素
            while (left < right && arr[left] <= pivot) left++;

            // 如果 left < right，说明找到了需要交换的元素对
            if (left < right) {
```

```
            // 交换元素
            swap(arr[left], arr[right]);
        }
    }

    // 将基准值放到正确位置（left == right 时的位置）
    swap(arr[l], arr[left]);

    // 递归排序左右两部分
    quickSort1(arr, l, left - 1);
    quickSort1(arr, left + 1, r);
}

/**
 * 方法 2: 三路快速排序（处理重复元素）
 * 将数组划分为三部分：< pivot、= pivot、> pivot
 * 特别适合处理有大量重复元素的数组
 * @param arr 待排序数组
 * @param l 排序区间的左边界（包含）
 * @param r 排序区间的右边界（包含）
 */
void quickSort2(vector<int>& arr, int l, int r) {
    // 递归终止条件
    if (l >= r) return;

    // 随机选择基准值
    int i = l + rand() % (r - l + 1);
    swap(arr[l], arr[i]);

    // 基准值
    int pivot = arr[l];

    // 三路分区指针：
    // lt: 小于区域的右边界
    // gt: 大于区域的左边界
    // i_idx: 当前处理元素的索引
    int lt = l;      // arr[l+1...lt] < pivot
    int gt = r + 1;  // arr[gt...r] > pivot
    int i_idx = l + 1; // arr[lt+1...i-1] == pivot

    // 三路分区过程
    while (i_idx < gt) {
        if (arr[i_idx] < pivot) {
```

```
            // 当前元素小于基准值，将其交换到小于区域
            swap(arr[++lt], arr[i_idx++]);
        } else if (arr[i_idx] > pivot) {
            // 当前元素大于基准值，将其交换到大于区域
            swap(arr[--gt], arr[i_idx]);
            // 注意这里 i_idx 不自增，因为交换过来的元素还未处理
        } else {
            // 当前元素等于基准值，保持在等于区域
            i_idx++;
        }
    }

    // 将基准值放到等于区域的左边界
    swap(arr[l], arr[lt]);

    // 递归排序小于区域和大于区域
    quickSort2(arr, l, lt - 1);
    quickSort2(arr, gt, r);
}

/**
 * 方法 3: 快速选择算法（用于查找第 k 小元素）
 * 与快速排序的区别：只处理包含目标元素的子数组
 * 平均时间复杂度：O(n)
 * @param arr 数组
 * @param l 当前处理区间的左边界（包含）
 * @param r 当前处理区间的右边界（包含）
 * @param k 目标元素在排序后数组中的索引位置
 * @return 第 k 小的元素值
 */
int quickSelect(vector<int>& arr, int l, int r, int k) {
    // 递归终止条件：当区间只有一个元素时，就是要找的位置
    if (l >= r) return arr[l];

    // 随机选择基准值
    int i = l + rand() % (r - l + 1);

    // 将基准值交换到末尾位置，便于后续分区操作
    swap(arr[i], arr[r]);

    // 基准值
    int pivot = arr[r];
```

```
    // 小于等于基准值区域的右边界（不包含）
    int left = l;

    // 遍历数组，将小于等于基准值的元素放到左侧
    for (int j = l; j < r; j++) {
        if (arr[j] <= pivot) {
            swap(arr[left++], arr[j]);
        }
    }

    // 将基准值放到正确位置
    swap(arr[left], arr[r]);

    // 根据基准值位置决定下一步操作
    if (left == k) {
        // 如果基准值位置正好是目标位置，直接返回
        return arr[left];
    } else if (left < k) {
        // 如果基准值位置小于目标位置，在右半部分继续查找
        return quickSelect(arr, left + 1, r, k);
    } else {
        // 如果基准值位置大于目标位置，在左半部分继续查找
        return quickSelect(arr, l, left - 1, k);
    }
}

/**
 * 插入排序算法，用于小数组优化
 * 对小规模数组使用插入排序比快速排序更高效
 * @param arr 数组
 * @param l 排序区间的左边界（包含）
 * @param r 排序区间的右边界（包含）
 */
void insertionSort(vector<int>& arr, int l, int r) {
    // 从第二个元素开始，逐个插入到已排序序列中
    for (int i = l + 1; i <= r; i++) {
        int key = arr[i]; // 当前要插入的元素
        int j = i - 1;    // 已排序序列的最后一个位置

        // 在已排序序列中找到合适的插入位置
        while (j >= l && arr[j] > key) {
            arr[j + 1] = arr[j]; // 元素后移
            j--;
```

```
        }

        // 插入元素
        arr[j + 1] = key;
    }
}

/**
 * 【优化版本 1】小数组插入排序优化
 * 当数组长度小于阈值时，使用插入排序，减少递归开销
 * @param arr 数组
 * @param l 排序区间的左边界（包含）
 * @param r 排序区间的右边界（包含）
 */
void quickSortOptimized(vector<int>& arr, int l, int r) {
    // 小数组阈值，经验值为 10-20
    const int INSERTION_SORT_THRESHOLD = 15;

    // 对小数组使用插入排序
    if (r - l <= INSERTION_SORT_THRESHOLD) {
        insertionSort(arr, l, r);
        return;
    }

    // 对大数组继续使用快速排序
    int i = l + rand() % (r - l + 1);
    swap(arr[l], arr[i]);

    int pivot = arr[l];
    int lt = l;      // arr[l+1...lt] < pivot
    int gt = r + 1;  // arr[gt...r] > pivot
    int i_idx = l + 1; // arr[lt+1...i-1] == pivot

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

```
    quickSortOptimized(arr, l, lt - 1);
    quickSortOptimized(arr, gt, r);
}

/**
 * 【LeetCode 215 解法】数组中的第 K 个最大元素
 * @param nums 输入数组
 * @param k 第 k 大的元素
 * @return 第 k 大的元素值
 */
int findKthLargest(vector<int>& nums, int k) {
    // 第 K 大元素等价于第 nums.size()-k 小的元素
    vector<int> arr(nums.begin(), nums.end());
    return quickSelect(arr, 0, (int)arr.size() - 1, (int)arr.size() - k);
}

/**
 * 【剑指 Offer 40 解法】最小的 k 个数
 * @param arr 输入数组
 * @param k 需要返回的最小元素个数
 * @return 包含最小 k 个数的数组
 */
vector<int> getLeastNumbers(vector<int>& arr, int k) {
    // 边界条件处理
    if (k <= 0) return vector<int>();
    if (k >= (int)arr.size()) return arr;

    vector<int> nums(arr.begin(), arr.end());

    // 使用快速选择找到第 k 小的元素
    quickSelect(nums, 0, (int)nums.size() - 1, k - 1);

    // 收集前 k 个最小元素
    vector<int> result(nums.begin(), nums.begin() + k);
    return result;
}

/**
 * 【LeetCode 75 解法】颜色分类（三路快排应用）
 * @param nums 包含 0、1、2 的数组，分别代表红、白、蓝三种颜色
 */
void sortColors(vector<int>& nums) {
```

```
    // 三路快排思想：0 放左边，1 放中间，2 放右边
    int zero = -1;      // [0...zero] == 0
    int two = (int)nums.size(); // [two...n-1] == 2
    int i = 0;          // 当前处理的位置

    while (i < two) {
        if (nums[i] == 0) {
            // 当前元素为 0，交换到 0 区域的下一个位置
            swap(nums[++zero], nums[i++]);
        } else if (nums[i] == 1) {
            // 当前元素为 1，保持在中间区域
            i++;
        } else { // nums[i] == 2
            // 当前元素为 2，交换到 2 区域的前一个位置
            swap(nums[i], nums[--two]);
        }
    }
}

/**
 * 【LeetCode 283 解法】移动零（分区思想应用）
 * @param nums 输入数组
 */
void moveZeroes(vector<int>& nums) {
    // 双指针：将非零元素移动到数组前面
    int nonZeroPos = 0; // 指向下一个非零元素应该放的位置

    // 第一次遍历：将所有非零元素移到前面
    for (int i = 0; i < (int)nums.size(); i++) {
        if (nums[i] != 0) {
            if (i != nonZeroPos) {
                swap(nums[i], nums[nonZeroPos]);
            }
            nonZeroPos++;
        }
    }
}

/**
 * 【LeetCode 912 解法】排序数组（标准快速排序实现）
 * @param nums 待排序数组
 * @return 排序后的数组
 */
```

```
    vector<int> sortArray(vector<int>& nums) {
        vector<int> result(nums.begin(), nums.end());
        quickSortOptimized(result, 0, (int)result.size() - 1);
        return result;
    }

    /**
     * 【POJ 2388 解法】Who's in the Middle（中位数问题）
     * @param nums 输入数组
     * @return 中位数
     */
    int findMedian(vector<int>& nums) {
        int n = (int)nums.size();
        vector<int> arr(nums.begin(), nums.end());
        // 中位数就是第 (n-1)/2 小的元素（0-based 索引）
        return quickSelect(arr, 0, n - 1, (n - 1) / 2);
    }
};

/**
 * 测试函数
 * @return 程序退出状态
 */
int main() {
    // 初始化随机数种子
    srand((unsigned int)time(nullptr));

    QuickSortSolution solution;

    // 测试基础快速排序
    vector<int> arr1;
    arr1.push_back(5);
    arr1.push_back(2);
    arr1.push_back(3);
    arr1.push_back(1);
    arr1.push_back(4);

    cout << "原始数组: ";
    for (size_t i = 0; i < arr1.size(); i++) cout << arr1[i] << " ";
    cout << endl;

    solution.quickSort1(arr1, 0, (int)arr1.size() - 1);
    cout << "排序后数组: ";
```

```
for (size_t i = 0; i < arr1.size(); i++) cout << arr1[i] << " ";
cout << endl;

// 测试三路快速排序
vector<int> arr2;
arr2.push_back(5);
arr2.push_back(2);
arr2.push_back(3);
arr2.push_back(1);
arr2.push_back(4);
arr2.push_back(2);
arr2.push_back(3);

cout << "\n原始数组: ";
for (size_t i = 0; i < arr2.size(); i++) cout << arr2[i] << " ";
cout << endl;

solution.quickSort2(arr2, 0, (int)arr2.size() - 1);
cout << "三路快排后: ";
for (size_t i = 0; i < arr2.size(); i++) cout << arr2[i] << " ";
cout << endl;

// 测试快速选择算法
vector<int> arr3;
arr3.push_back(3);
arr3.push_back(2);
arr3.push_back(1);
arr3.push_back(5);
arr3.push_back(6);
arr3.push_back(4);

int k = 2; // 查找第2大的元素（即索引为4的元素）
int result = solution.quickSelect(arr3, 0, (int)arr3.size() - 1, (int)arr3.size() - k);
cout << "\n数组: ";
for (size_t i = 0; i < arr3.size(); i++) cout << arr3[i] << " ";
cout << endl;
cout << "第" << k << "大的元素是: " << result << endl;

// 测试优化版本的快速排序
vector<int> arr5;
arr5.push_back(9);
arr5.push_back(8);
arr5.push_back(7);
```

```
arr5.push_back(6);
arr5.push_back(5);
arr5.push_back(4);
arr5.push_back(3);
arr5.push_back(2);
arr5.push_back(1);
arr5.push_back(0);
arr5.push_back(5);
arr5.push_back(6);
arr5.push_back(7);
arr5.push_back(8);
arr5.push_back(9);

cout << "\n 原始数组: ";
for (size_t i = 0; i < arr5.size(); i++) cout << arr5[i] << " ";
cout << endl;

solution.quickSortOptimized(arr5, 0, (int)arr5.size() - 1);
cout << "优化快排后: ";
for (size_t i = 0; i < arr5.size(); i++) cout << arr5[i] << " ";
cout << endl;

// 检查数组是否有序
bool isSorted = true;
for (size_t i = 1; i < arr5.size(); i++) {
    if (arr5[i] < arr5[i-1]) {
        isSorted = false;
        break;
    }
}
cout << "数组是否有序: " << (isSorted ? "是" : "否") << endl;

// 测试颜色分类
vector<int> colors;
colors.push_back(2);
colors.push_back(0);
colors.push_back(2);
colors.push_back(1);
colors.push_back(1);
colors.push_back(0);

cout << "\n 原始颜色数组: ";
for (size_t i = 0; i < colors.size(); i++) cout << colors[i] << " ";
```

```
    cout << endl;

    solution.sortColors(colors);
    cout << "颜色分类后: ";
    for (size_t i = 0; i < colors.size(); i++) cout << colors[i] << " ";
    cout << endl;

    // 测试移动零
    vector<int> zeros;
    zeros.push_back(0);
    zeros.push_back(1);
    zeros.push_back(0);
    zeros.push_back(3);
    zeros.push_back(12);

    cout << "\n原始数组: ";
    for (size_t i = 0; i < zeros.size(); i++) cout << zeros[i] << " ";
    cout << endl;

    solution.moveZeroes(zeros);
    cout << "移动零后: ";
    for (size_t i = 0; i < zeros.size(); i++) cout << zeros[i] << " ";
    cout << endl;

    return 0;
}

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

文件: Code05_QuickSort_Cplusplus_Simplified.cpp
==============================================
// C++版本快速排序实现（简化版）
// 包含多种快速排序的实现方式

/*
 * 快速排序算法实现与应用
 *
 * 本文件包含多种快速排序的实现方式及其在不同场景下的应用
 *
 * 算法复杂度分析:
 * 时间复杂度:
 *   - 最好情况: O(n log n) - 每次划分都能将数组平均分成两部分
 *   - 平均情况: O(n log n) - 随机选择基准值的情况下
```

```
 *   - 最坏情况: O(n^2) - 每次选择的基准值都是最大或最小值
 * 空间复杂度:
 *   - O(log n) - 递归调用栈的深度
 *
 * 算法优化策略:
 * 1. 随机选择基准值 - 避免最坏情况的出现
 * 2. 三路快排 - 处理重复元素较多的情况
 * 3. 小数组使用插入排序 - 减少递归开销
 * 4. 尾递归优化 - 减少栈空间使用
 */

// 基础快速排序实现
void quickSort1(int arr[], int l, int r) {
    if (l >= r) return;

    // 随机选择基准值
    int i = l + (rand() % (r - l + 1));

    // 将随机选择的基准值交换到第一个位置
    int temp = arr[l];
    arr[l] = arr[i];
    arr[i] = temp;

    // 基准值
    int pivot = arr[l];

    // 双指针分区
    int left = l, right = r;

    while (left < right) {
        // 从右向左找小于基准值的元素
        while (left < right && arr[right] >= pivot) right--;

        // 从左向右找大于基准值的元素
        while (left < right && arr[left] <= pivot) left++;

        // 交换元素
        if (left < right) {
            temp = arr[left];
            arr[left] = arr[right];
            arr[right] = temp;
        }
    }
```

```
    // 将基准值放到正确位置
    temp = arr[l];
    arr[l] = arr[left];
    arr[left] = temp;

    // 递归排序左右两部分
    quickSort1(arr, l, left - 1);
    quickSort1(arr, left + 1, r);
}

// 三路快速排序实现（处理重复元素）
void quickSort2(int arr[], int l, int r) {
    if (l >= r) return;

    // 随机选择基准值
    int i = l + (rand() % (r - l + 1));
    int temp = arr[l];
    arr[l] = arr[i];
    arr[i] = temp;

    // 基准值
    int pivot = arr[l];

    // 三路分区指针
    int lt = l;      // 小于区域的右边界
    int gt = r + 1;  // 大于区域的左边界
    int idx = l + 1; // 当前处理元素的索引

    // 三路分区过程
    while (idx < gt) {
        if (arr[idx] < pivot) {
            // 当前元素小于基准值，将其交换到小于区域
            temp = arr[++lt];
            arr[lt] = arr[idx];
            arr[idx++] = temp;
        } else if (arr[idx] > pivot) {
            // 当前元素大于基准值，将其交换到大于区域
            temp = arr[--gt];
            arr[gt] = arr[idx];
            arr[idx] = temp;
            // 注意这里 idx 不自增，因为交换过来的元素还未处理
        } else {
```

```
            // 当前元素等于基准值，保持在等于区域
            idx++;
        }
    }

    // 将基准值放到等于区域的左边界
    temp = arr[l];
    arr[l] = arr[lt];
    arr[lt] = temp;

    // 递归排序小于区域和大于区域
    quickSort2(arr, l, lt - 1);
    quickSort2(arr, gt, r);
}

// 快速选择算法（用于查找第 k 小元素）
int quickSelect(int arr[], int l, int r, int k) {
    // 递归终止条件
    if (l >= r) return arr[l];

    // 随机选择基准值
    int i = l + (rand() % (r - l + 1));

    // 将基准值交换到末尾位置
    int temp = arr[i];
    arr[i] = arr[r];
    arr[r] = temp;

    // 基准值
    int pivot = arr[r];

    // 小于等于基准值区域的右边界
    int left = l;

    // 遍历数组，将小于等于基准值的元素放到左侧
    for (int j = l; j < r; j++) {
        if (arr[j] <= pivot) {
            temp = arr[left];
            arr[left] = arr[j];
            arr[j] = temp;
            left++;
        }
    }
```

```
    // 将基准值放到正确位置
    temp = arr[left];
    arr[left] = arr[r];
    arr[r] = temp;

    // 根据基准值位置决定下一步操作
    if (left == k) {
        // 如果基准值位置正好是目标位置，直接返回
        return arr[left];
    } else if (left < k) {
        // 如果基准值位置小于目标位置，在右半部分继续查找
        return quickSelect(arr, left + 1, r, k);
    } else {
        // 如果基准值位置大于目标位置，在左半部分继续查找
        return quickSelect(arr, l, left - 1, k);
    }
}

// 插入排序算法，用于小数组优化
void insertionSort(int arr[], int l, int r) {
    // 从第二个元素开始，逐个插入到已排序序列中
    for (int i = l + 1; i <= r; i++) {
        int key = arr[i]; // 当前要插入的元素
        int j = i - 1;    // 已排序序列的最后一个位置

        // 在已排序序列中找到合适的插入位置
        while (j >= l && arr[j] > key) {
            arr[j + 1] = arr[j]; // 元素后移
            j--;
        }

        // 插入元素
        arr[j + 1] = key;
    }
}

// 优化版本的快速排序（小数组使用插入排序）
void quickSortOptimized(int arr[], int l, int r) {
    // 小数组阈值
    const int THRESHOLD = 15;

    // 对小数组使用插入排序
```

```
    if (r - l <= THRESHOLD) {
        insertionSort(arr, l, r);
        return;
    }

    // 对大数组继续使用快速排序
    int i = l + (rand() % (r - l + 1));
    int temp = arr[l];
    arr[l] = arr[i];
    arr[i] = temp;

    int pivot = arr[l];
    int lt = l;      // 小于区域的右边界
    int gt = r + 1;  // 大于区域的左边界
    int idx = l + 1; // 当前处理元素的索引

    while (idx < gt) {
        if (arr[idx] < pivot) {
            temp = arr[++lt];
            arr[lt] = arr[idx];
            arr[idx++] = temp;
        } else if (arr[idx] > pivot) {
            temp = arr[--gt];
            arr[gt] = arr[idx];
            arr[idx] = temp;
        } else {
            idx++;
        }
    }

    temp = arr[l];
    arr[l] = arr[lt];
    arr[lt] = temp;

    quickSortOptimized(arr, l, lt - 1);
    quickSortOptimized(arr, gt, r);
}

// 查找第 k 大的元素
int findKthLargest(int arr[], int size, int k) {
    // 第 k 大元素等价于第 size-k 小的元素
    return quickSelect(arr, 0, size - 1, size - k);
}
```

```
// 颜色分类（三路快排应用）
void sortColors(int nums[], int size) {
    // 三路快排思想：0 放左边，1 放中间，2 放右边
    int zero = -1;      // [0...zero] == 0
    int two = size;     // [two...n-1] == 2
    int i = 0;          // 当前处理的位置

    while (i < two) {
        if (nums[i] == 0) {
            // 当前元素为 0，交换到 0 区域的下一个位置
            int temp = nums[++zero];
            nums[zero] = nums[i];
            nums[i++] = temp;
        } else if (nums[i] == 1) {
            // 当前元素为 1，保持在中间区域
            i++;
        } else { // nums[i] == 2
            // 当前元素为 2，交换到 2 区域的前一个位置
            int temp = nums[i];
            nums[i] = nums[--two];
            nums[two] = temp;
        }
    }
}

// 移动零（分区思想应用）
void moveZeroes(int nums[], int size) {
    // 双指针：将非零元素移动到数组前面
    int nonZeroPos = 0; // 指向下一个非零元素应该放的位置

    // 第一次遍历：将所有非零元素移到前面
    for (int i = 0; i < size; i++) {
        if (nums[i] != 0) {
            if (i != nonZeroPos) {
                int temp = nums[i];
                nums[i] = nums[nonZeroPos];
                nums[nonZeroPos] = temp;
            }
            nonZeroPos++;
        }
    }
}
```

==========

文件: Code06_QuickSort_Python.py
==========
Python 版本快速排序实现
包含多种快速排序的实现方式

'''
补充题目列表:

1. LeetCode 912. 排序数组
   链接: https://leetcode.cn/problems/sort-an-array/
   题目描述: 给你一个整数数组 nums，请你将该数组升序排列。
   时间复杂度: O(n log n)，空间复杂度: O(log n)
   最优解: 快速排序或归并排序

2. 洛谷 P1177 【模板】快速排序
   链接: https://www.luogu.com.cn/problem/P1177
   题目描述: 利用快速排序算法将读入的 N 个数从小到大排序后输出。
   时间复杂度: O(n log n)，空间复杂度: O(log n)
   最优解: 快速排序算法实现

3. LeetCode 215. 数组中的第 K 个最大元素
   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。
   时间复杂度: O(n) 平均，空间复杂度: O(log n)
   最优解: 快速选择算法

4. LeetCode 75. 颜色分类
   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、共 n 个元素的数组 nums，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。
   时间复杂度: O(n)，空间复杂度: O(1)
   最优解: 三路快排思想

5. LeetCode 283. 移动零
   链接: https://leetcode.cn/problems/move-zeroes/
   题目描述: 给定一个数组 nums，编写一个函数将所有 0 移动到数组的末尾，同时保持非零元素的相对顺序。
   时间复杂度: O(n)，空间复杂度: O(1)
   最优解: 双指针法

6. Codeforces 401C. Team
   链接: https://codeforces.com/problemset/problem/401/C
   题目描述: 构造一个 01 序列，满足特定的约束条件。
   时间复杂度: O(n+m)，空间复杂度: O(n+m)
   最优解: 贪心构造

7. AtCoder ABC121C. Energy Drink Collector
   链接: https://atcoder.jp/contests/abc121/tasks/abc121_c
   题目描述: 购买能量饮料以获得最少的总花费。
   时间复杂度: O(n log n)，空间复杂度: O(1)
   最优解: 贪心+排序

8. 牛客网 - 快速排序
   链接: https://www.nowcoder.com/practice/e016ad9b7f0b45048c58a9f27ba618bf
   题目描述: 实现快速排序算法
   时间复杂度: O(n log n)，空间复杂度: O(log n)
   最优解: 标准快速排序实现

9. PAT 1101 Quick Sort
   链接: https://pintia.cn/problem-sets/994805342720868352/problems/994805366343188480
   题目描述: 快速排序中的主元(pivot)是左面都比它小、右边都比它大的位置对应的数字。找出所有满足条件的主元。
   时间复杂度: O(n)，空间复杂度: O(n)
   最优解: 预处理左右边界最大值数组

10. 剑指 Offer 40. 最小的 k 个数
    链接: https://leetcode.cn/problems/zui-xiao-de-kge-shu-lcof/
    题目描述: 输入整数数组 arr ，找出其中最小的 k 个数。
    时间复杂度: O(n) 平均，空间复杂度: O(log n)
    最优解: 快速选择算法

11. 杭电 OJ 1425. sort
    链接: http://acm.hdu.edu.cn/showproblem.php?pid=1425
    题目描述: 对整数数组进行快速排序
    时间复杂度: O(n log n)，空间复杂度: O(log n)
    最优解: 快速排序或堆排序

12. POJ 2388. Who's in the Middle
    链接: http://poj.org/problem?id=2388
    题目描述: 找出一组数的中位数，快速选择的经典应用
    时间复杂度: O(n) 平均，空间复杂度: O(log n)
    最优解: 快速选择算法找中位数

13. AizuOJ ALDS1_6_C. Quick Sort
    链接: https://onlinejudge.u-aizu.ac.jp/problems/ALDS1_6_C
    题目描述: 实现快速排序算法并输出每一步的分区结果
    时间复杂度: O(n log n), 空间复杂度: O(log n)
    最优解: 快速排序算法实现

14. LeetCode 169. 多数元素
    链接: https://leetcode.cn/problems/majority-element/
    题目描述: 给定一个大小为 n 的数组，找到其中的多数元素
    时间复杂度: O(n), 空间复杂度: O(1)
    最优解: Boyer-Moore 投票算法（与快速选择思想相关）

15. LeetCode 274. H 指数
    链接: https://leetcode.cn/problems/h-index/
    题目描述: 计算研究人员的 h 指数
    时间复杂度: O(n) 平均, 空间复杂度: O(n)
    最优解: 计数排序或快速选择

16. LeetCode 462. 最少移动次数使数组元素相等 II
    链接: https://leetcode.cn/problems/minimum-moves-to-equal-array-elements-ii/
    题目描述: 找到使所有数组元素相等所需的最少移动次数
    时间复杂度: O(n) 平均, 空间复杂度: O(log n)
    最优解: 快速选择找中位数

17. LeetCode 324. 摆动排序 II
    链接: https://leetcode.cn/problems/wiggle-sort-ii/
    题目描述: 将数组重新排列成 nums[0] < nums[1] > nums[2] < nums[3]... 的顺序
    时间复杂度: O(n) 平均, 空间复杂度: O(n)
    最优解: 快速选择+三路快排

18. LeetCode 347. 前 K 个高频元素
    链接: https://leetcode.cn/problems/top-k-frequent-elements/
    题目描述: 返回数组中出现频率前 k 高的元素
    时间复杂度: O(n) 平均, 空间复杂度: O(n)
    最优解: 快速选择或堆排序

19. LeetCode 973. 最接近原点的 K 个点
    链接: https://leetcode.cn/problems/k-closest-points-to-origin/
    题目描述: 找到最接近原点的 k 个点
    时间复杂度: O(n) 平均, 空间复杂度: O(log n)
    最优解: 快速选择算法

20. LeetCode 4. 寻找两个正序数组的中位数

链接: https://leetcode.cn/problems/median-of-two-sorted-arrays/
题目描述: 找到两个有序数组的中位数
时间复杂度: O(log(min(m,n))), 空间复杂度: O(1)
最优解: 二分查找（与快速选择思想相关）

算法复杂度分析:
时间复杂度:
- 最好情况: O(n log n) - 每次划分都能将数组平均分成两部分
- 平均情况: O(n log n) - 随机选择基准值的情况下
- 最坏情况: O(n^2) - 每次选择的基准值都是最大或最小值
空间复杂度:
- O(log n) - 递归调用栈的深度

与 Java/C++版本的差异:
1. Python 使用列表，动态类型，无需声明数组大小
2. Python 使用 random 模块生成随机数
3. Python 列表是对象，有动态扩容功能
4. Python 语法简洁，但性能相对较低
5. Python 有列表推导式等特性，代码更简洁

工程化考量:
1. 异常处理: 处理空列表、None 输入等边界情况
2. 性能优化: 对于小数组使用插入排序优化
3. 内存使用: 原地排序减少额外空间开销
4. 稳定性: 标准快排不稳定，如需稳定排序需特殊处理

调试技巧:
1. 使用 print 打印中间过程
2. 添加断言验证分区正确性
3. 测试边界情况（空列表、单元素等）
4. 使用 Python 的 unittest 模块进行单元测试

面试技巧:
1. 理解快排与其它排序算法的比较（如归并排序、堆排序）
2. 掌握快排的优化方法（随机化、三路快排等）
3. 理解快排在不同数据分布下的性能表现
4. 能够分析快排的稳定性和适用场景
'''

```
import random

class QuickSortSolution:
    def __init__(self):
```

```
    pass

# 方法 1: 基础快速排序
def quick_sort1(self, arr, l, r):
    """
    基础快速排序实现
    使用双指针分区法实现快速排序
    :param arr: 待排序列表
    :param l: 排序区间的左边界（包含）
    :param r: 排序区间的右边界（包含）
    """
    # 递归终止条件：当左边界大于等于右边界时，表示区间内没有元素或只有一个元素，无需排序
    if l >= r:
        return

    # 随机选择基准值，避免最坏情况
    # random.randint(l, r)生成[l, r]之间的随机整数
    i = random.randint(l, r)

    # 将随机选择的基准值交换到第一个位置，便于后续分区操作
    arr[l], arr[i] = arr[i], arr[l]

    # 基准值
    pivot = arr[l]

    # 双指针分区：left 从左向右扫描，right 从右向左扫描
    left, right = l, r

    # 双指针分区操作
    while left < right:
        # 从右向左找小于基准值的元素
        while left < right and arr[right] >= pivot:
            right -= 1

        # 从左向右找大于基准值的元素
        while left < right and arr[left] <= pivot:
            left += 1

        # 如果 left < right，说明找到了需要交换的元素对
        if left < right:
            # 交换元素
            arr[left], arr[right] = arr[right], arr[left]
```

```
    # 将基准值放到正确位置（left == right 时的位置）
    arr[l], arr[left] = arr[left], arr[l]

    # 递归排序左右两部分
    self.quick_sort1(arr, l, left - 1)
    self.quick_sort1(arr, left + 1, r)

# 方法 2: 三路快速排序（处理重复元素）
def quick_sort2(self, arr, l, r):
    """
    三路快速排序实现（处理重复元素）
    将数组划分为三部分：< pivot、= pivot、> pivot
    特别适合处理有大量重复元素的数组
    :param arr: 待排序列表
    :param l: 排序区间的左边界（包含）
    :param r: 排序区间的右边界（包含）
    """
    # 递归终止条件
    if l >= r:
        return

    # 随机选择基准值
    i = random.randint(l, r)
    arr[l], arr[i] = arr[i], arr[l]

    # 基准值
    pivot = arr[l]

    # 三路分区指针：
    # lt: 小于区域的右边界
    # gt: 大于区域的左边界
    # i_idx: 当前处理元素的索引
    lt = l       # arr[l+1...lt] < pivot
    gt = r + 1   # arr[gt...r] > pivot
    i_idx = l + 1  # arr[lt+1...i-1] == pivot

    # 三路分区过程
    while i_idx < gt:
        if arr[i_idx] < pivot:
            # 当前元素小于基准值，将其交换到小于区域
            lt += 1
            arr[lt], arr[i_idx] = arr[i_idx], arr[lt]
            i_idx += 1
```

```
            elif arr[i_idx] > pivot:
                # 当前元素大于基准值，将其交换到大于区域
                gt -= 1
                arr[gt], arr[i_idx] = arr[i_idx], arr[gt]
                # 注意这里 i_idx 不自增，因为交换过来的元素还未处理
            else:
                # 当前元素等于基准值，保持在等于区域
                i_idx += 1

        # 将基准值放到等于区域的左边界
        arr[l], arr[lt] = arr[lt], arr[l]

        # 递归排序小于区域和大于区域
        self.quick_sort2(arr, l, lt - 1)
        self.quick_sort2(arr, gt, r)

    # 方法 3: 快速选择算法（用于查找第 k 小元素）
    def quick_select(self, arr, l, r, k):
        """
        快速选择算法（用于查找第 k 小元素）
        与快速排序的区别：只处理包含目标元素的子数组
        平均时间复杂度：O(n)
        :param arr: 数组
        :param l: 当前处理区间的左边界（包含）
        :param r: 当前处理区间的右边界（包含）
        :param k: 目标元素在排序后数组中的索引位置
        :return: 第 k 小的元素值
        """
        # 递归终止条件：当区间只有一个元素时，就是要找的位置
        if l >= r:
            return arr[l]

        # 随机选择基准值
        i = random.randint(l, r)

        # 将基准值交换到末尾位置，便于后续分区操作
        arr[i], arr[r] = arr[r], arr[i]

        # 基准值
        pivot = arr[r]

        # 小于等于基准值区域的右边界（不包含）
        left = l
```

```
    # 遍历数组，将小于等于基准值的元素放到左侧
    for j in range(l, r):
        if arr[j] <= pivot:
            arr[left], arr[j] = arr[j], arr[left]
            left += 1

    # 将基准值放到正确位置
    arr[left], arr[r] = arr[r], arr[left]

    # 根据基准值位置决定下一步操作
    if left == k:
        # 如果基准值位置正好是目标位置，直接返回
        return arr[left]
    elif left < k:
        # 如果基准值位置小于目标位置，在右半部分继续查找
        return self.quick_select(arr, left + 1, r, k)
    else:
        # 如果基准值位置大于目标位置，在左半部分继续查找
        return self.quick_select(arr, l, left - 1, k)

# 方法 4: 堆排序实现（用于比较）
def heap_sort(self, arr):
    """
    堆排序实现（用于比较）
    :param arr: 待排序列表
    """
    def heapify(arr, n, i):
        """
        调整堆结构
        :param arr: 数组
        :param n: 堆大小
        :param i: 当前节点索引
        """
        largest = i
        left = 2 * i + 1
        right = 2 * i + 2

        # 找到最大值的索引
        if left < n and arr[left] > arr[largest]:
            largest = left

        if right < n and arr[right] > arr[largest]:
```

```
            largest = right

        # 如果最大值不是当前节点，则交换并继续调整
        if largest != i:
            arr[i], arr[largest] = arr[largest], arr[i]
            heapify(arr, n, largest)

    n = len(arr)

    # 构建最大堆
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)

    # 逐个从堆顶取出元素
    for i in range(n - 1, 0, -1):
        arr[0], arr[i] = arr[i], arr[0]
        heapify(arr, i, 0)

# 【优化版本1】小数组插入排序优化
# 当数组长度小于阈值时，使用插入排序，减少递归开销
def quick_sort_optimized(self, arr, l, r):
    """
    优化版本的快速排序（小数组使用插入排序）
    当数组长度小于阈值时，使用插入排序，减少递归开销
    :param arr: 待排序列表
    :param l: 排序区间的左边界（包含）
    :param r: 排序区间的右边界（包含）
    """
    # 小数组阈值，经验值为10-20
    INSERTION_SORT_THRESHOLD = 15

    # 对小数组使用插入排序
    if r - l <= INSERTION_SORT_THRESHOLD:
        self.insertion_sort(arr, l, r)
        return

    # 对大数组继续使用快速排序
    i = random.randint(l, r)
    arr[l], arr[i] = arr[i], arr[l]

    pivot = arr[l]
    lt = l       # arr[l+1...lt] < pivot
    gt = r + 1   # arr[gt...r] > pivot
```

```
    i_idx = l + 1  # arr[lt+1...i-1] == pivot

    while i_idx < gt:
        if arr[i_idx] < pivot:
            lt += 1
            arr[lt], arr[i_idx] = arr[i_idx], arr[lt]
            i_idx += 1
        elif arr[i_idx] > pivot:
            gt -= 1
            arr[gt], arr[i_idx] = arr[i_idx], arr[gt]
        else:
            i_idx += 1

    arr[l], arr[lt] = arr[lt], arr[l]

    self.quick_sort_optimized(arr, l, lt - 1)
    self.quick_sort_optimized(arr, gt, r)

# 插入排序算法，用于小数组优化
def insertion_sort(self, arr, l, r):
    """
    插入排序算法，用于小数组优化
    对小规模数组使用插入排序比快速排序更高效
    :param arr: 数组
    :param l: 排序区间的左边界（包含）
    :param r: 排序区间的右边界（包含）
    """
    # 从第二个元素开始，逐个插入到已排序序列中
    for i in range(l + 1, r + 1):
        key = arr[i]  # 当前要插入的元素
        j = i - 1     # 已排序序列的最后一个位置

        # 在已排序序列中找到合适的插入位置
        while j >= l and arr[j] > key:
            arr[j + 1] = arr[j]  # 元素后移
            j -= 1

        # 插入元素
        arr[j + 1] = key

# 【优化版本 2】尾递归优化
# 将尾递归转换为迭代，减少栈空间使用
def quick_sort_tail_recursive(self, arr, l, r):
```

```
    """
    尾递归优化版本的快速排序
    将尾递归转换为迭代，减少栈空间使用
    :param arr: 待排序列表
    :param l: 排序区间的左边界（包含）
    :param r: 排序区间的右边界（包含）
    """
    while l < r:
        # 随机选择基准值
        i = random.randint(l, r)
        arr[l], arr[i] = arr[i], arr[l]

        pivot = arr[l]
        lt = l       # arr[l+1...lt] < pivot
        gt = r + 1   # arr[gt...r] > pivot
        i_idx = l + 1  # arr[lt+1...i-1] == pivot

        while i_idx < gt:
            if arr[i_idx] < pivot:
                lt += 1
                arr[lt], arr[i_idx] = arr[i_idx], arr[lt]
                i_idx += 1
            elif arr[i_idx] > pivot:
                gt -= 1
                arr[gt], arr[i_idx] = arr[i_idx], arr[gt]
            else:
                i_idx += 1

        arr[l], arr[lt] = arr[lt], arr[l]

        # 优先处理较小的子数组，减少递归深度
        if lt - l < r - gt + 1:
            # 左边子数组较小，先递归处理左边
            self.quick_sort_tail_recursive(arr, l, lt - 1)
            # 尾递归优化：将右边子数组的处理转换为迭代
            l = gt
        else:
            # 右边子数组较小，先递归处理右边
            self.quick_sort_tail_recursive(arr, gt, r)
            # 尾递归优化：将左边子数组的处理转换为迭代
            r = lt - 1

# 【优化版本 3】三数取中法选择基准
```

```
# 选择左、中、右三个位置的中位数作为基准，提高最坏情况性能
def quick_sort_median_of_three(self, arr, l, r):
    """
    三数取中法优化的快速排序
    选择左、中、右三个位置的中位数作为基准，提高最坏情况性能
    :param arr: 待排序列表
    :param l: 排序区间的左边界（包含）
    :param r: 排序区间的右边界（包含）
    """
    # 递归终止条件
    if l >= r:
        return

    # 三数取中选择基准
    mid = l + (r - l) // 2
    self.median_of_three(arr, l, mid, r)

    pivot = arr[l]
    lt = l       # arr[l+1...lt] < pivot
    gt = r + 1   # arr[gt...r] > pivot
    i_idx = l + 1  # arr[lt+1...i-1] == pivot

    while i_idx < gt:
        if arr[i_idx] < pivot:
            lt += 1
            arr[lt], arr[i_idx] = arr[i_idx], arr[lt]
            i_idx += 1
        elif arr[i_idx] > pivot:
            gt -= 1
            arr[gt], arr[i_idx] = arr[i_idx], arr[gt]
        else:
            i_idx += 1

    arr[l], arr[lt] = arr[lt], arr[l]

    self.quick_sort_median_of_three(arr, l, lt - 1)
    self.quick_sort_median_of_three(arr, gt, r)

# 三数取中辅助方法
def median_of_three(self, arr, a, b, c):
    """
    三数取中辅助方法
    选择左、中、右三个位置的中位数作为基准，提高最坏情况性能
```

```
    :param arr: 数组
    :param a: 左边界索引
    :param b: 中间索引
    :param c: 右边界索引
    """
    # 确保 arr[a] <= arr[b] <= arr[c]
    if arr[a] > arr[b]:
        arr[a], arr[b] = arr[b], arr[a]
    if arr[b] > arr[c]:
        arr[b], arr[c] = arr[c], arr[b]
    if arr[a] > arr[b]:
        arr[a], arr[b] = arr[b], arr[a]
    # 将中位数放到位置 a（作为基准）
    arr[a], arr[b] = arr[b], arr[a]

# 【LeetCode 215 解法】数组中的第 K 个最大元素
def find_kth_largest(self, nums, k):
    """
    LeetCode 215 解法：数组中的第 K 个最大元素
    :param nums: 输入数组
    :param k: 第 k 大的元素
    :return: 第 k 大的元素值
    """
    # 第 K 大元素等价于第 len(nums)-k 小的元素
    # 深拷贝避免修改原数组
    nums_copy = nums.copy()
    return self.quick_select(nums_copy, 0, len(nums_copy) - 1, len(nums_copy) - k)

# 【剑指 Offer 40 解法】最小的 k 个数
def get_least_numbers(self, arr, k):
    """
    剑指 Offer 40 解法：最小的 k 个数
    :param arr: 输入数组
    :param k: 需要返回的最小元素个数
    :return: 包含最小 k 个数的数组
    """
    # 边界条件处理
    if k <= 0:
        return []
    if k >= len(arr):
        return arr.copy()

    # 深拷贝避免修改原数组
```

```
    nums_copy = arr.copy()

    # 使用快速选择找到第 k 小的元素
    self.quick_select(nums_copy, 0, len(nums_copy) - 1, k - 1)

    # 收集前 k 个最小元素
    return nums_copy[:k]

# 【LeetCode 75 解法】颜色分类（三路快排应用）
def sort_colors(self, nums):
    """
    LeetCode 75 解法：颜色分类（三路快排应用）
    :param nums: 包含 0、1、2 的数组，分别代表红、白、蓝三种颜色
    """
    # 三路快排思想：0 放左边，1 放中间，2 放右边
    zero = -1       # [0...zero] == 0
    two = len(nums)  # [two...n-1] == 2
    i = 0           # 当前处理的位置

    while i < two:
        if nums[i] == 0:
            # 当前元素为 0，交换到 0 区域的下一个位置
            zero += 1
            nums[zero], nums[i] = nums[i], nums[zero]
            i += 1
        elif nums[i] == 1:
            # 当前元素为 1，保持在中间区域
            i += 1
        else:  # nums[i] == 2
            # 当前元素为 2，交换到 2 区域的前一个位置
            two -= 1
            nums[i], nums[two] = nums[two], nums[i]

# 【LeetCode 283 解法】移动零（分区思想应用）
def move_zeroes(self, nums):
    """
    LeetCode 283 解法：移动零（分区思想应用）
    :param nums: 输入数组
    """
    # 双指针：将非零元素移动到数组前面
    non_zero_pos = 0  # 指向下一个非零元素应该放的位置

    # 第一次遍历：将所有非零元素移到前面
```

```
    for i in range(len(nums)):
        if nums[i] != 0:
            if i != non_zero_pos:
                nums[i], nums[non_zero_pos] = nums[non_zero_pos], nums[i]
            non_zero_pos += 1

# 【LeetCode 462 解法】最少移动次数使数组元素相等 II（中位数思想）
def min_moves2(self, nums):
    """
    LeetCode 462 解法：最少移动次数使数组元素相等 II（中位数思想）
    :param nums: 输入数组
    :return: 最少移动次数
    """
    n = len(nums)
    # 深拷贝避免修改原数组
    nums_copy = nums.copy()
    median = self.quick_select(nums_copy, 0, n - 1, n // 2)

    # 计算所有元素到中位数的距离和
    moves = 0
    for num in nums:
        moves += abs(num - median)
    return moves

# 【LeetCode 912 解法】排序数组（标准快速排序实现）
def sort_array(self, nums):
    """
    LeetCode 912 解法：排序数组（标准快速排序实现）
    :param nums: 待排序数组
    :return: 排序后的数组
    """
    # 深拷贝避免修改原数组
    result = nums.copy()
    self.quick_sort_optimized(result, 0, len(result) - 1)
    return result

# 【洛谷 P1177 解法】快速排序模板题
def luogu_p1177(self, arr):
    """
    洛谷 P1177 解法：快速排序模板题
    :param arr: 待排序数组
    """
    self.quick_sort_optimized(arr, 0, len(arr) - 1)
```

```
# 【POJ 2388 解法】Who's in the Middle（中位数问题）
def find_median(self, nums):
    """
    POJ 2388 解法：Who's in the Middle（中位数问题）
    :param nums: 输入数组
    :return: 中位数
    """
    n = len(nums)
    # 中位数就是第 (n-1)//2 小的元素（0-based 索引）
    # 深拷贝避免修改原数组
    nums_copy = nums.copy()
    return self.quick_select(nums_copy, 0, n - 1, (n - 1) // 2)

# 【异常处理】健壮的快速排序实现
def robust_quick_sort(self, nums):
    """
    异常处理：健壮的快速排序实现
    :param nums: 待排序数组
    """
    # 空数组或单元素数组检查
    if len(nums) <= 1:
        return

    # 执行排序
    self.quick_sort_optimized(nums, 0, len(nums) - 1)

# 【AtCoder ABC121C 解法】Energy Drink Collector（贪心+排序）
def energy_drink_collector(self, drinks, budget):
    """
    AtCoder ABC121C 解法：Energy Drink Collector（贪心+排序）
    :param drinks: 能量饮料信息，每行包含价格和数量
    :param budget: 预算
    :return: 能获得的最大能量
    """
    # 按价格升序排序
    sorted_drinks = sorted(drinks, key=lambda x: x[0])

    total_energy = 0
    remaining_budget = budget

    for price, amount in sorted_drinks:
        can_buy = min(amount, remaining_budget // price)
```

```
        total_energy += can_buy
        remaining_budget -= can_buy * price

        if remaining_budget < price:
            break

    return total_energy

# 【Codeforces 401C 解法】Team（贪心构造）
def construct_team(self, n, m):
    """
    Codeforces 401C 解法：Team（贪心构造）
    :param n: 0 的数量
    :param m: 1 的数量
    :return: 构造的 01 序列
    """
    # 构造一个 01 序列，满足特定约束条件
    result = []

    while n > 0 or m > 0:
        # 优先放置 0 的情况
        if n > m:
            # 检查是否可以放置 00
            if n >= 2 and m >= 1:
                result.append("001")
                n -= 2
                m -= 1
            else:
                result.append("0")
                n -= 1
        else:
            # 优先放置 1 的情况
            if m >= 2 and n >= 1:
                result.append("110")
                m -= 2
                n -= 1
            else:
                result.append("1")
                m -= 1

    return ''.join(result)

# 【调试辅助方法】打印数组内容
```

```
    def print_array(self, nums):
        """
        调试辅助方法：打印数组内容
        :param nums: 要打印的数组
        """
        print(' '.join(map(str, nums)))

    # 【测试验证方法】检查数组是否有序
    def is_sorted(self, nums):
        """
        测试验证方法：检查数组是否有序
        :param nums: 要检查的数组
        :return: 是否有序
        """
        for i in range(1, len(nums)):
            if nums[i] < nums[i-1]:
                return False
        return True

# 测试函数
def main():
    """
    主函数：测试各种快速排序实现
    """
    solution = QuickSortSolution()

    # 测试基础快速排序
    arr1 = [5, 2, 3, 1, 4]
    print("原始数组:", arr1)
    solution.quick_sort1(arr1, 0, len(arr1) - 1)
    print("排序后数组:", arr1)

    # 测试三路快速排序
    arr2 = [5, 2, 3, 1, 4, 2, 3]
    print("\n原始数组:", arr2)
    solution.quick_sort2(arr2, 0, len(arr2) - 1)
    print("三路快排后:", arr2)

    # 测试快速选择算法
    arr3 = [3, 2, 1, 5, 6, 4]
    k = 2  # 查找第2大的元素（即索引为4的元素）
    result = solution.quick_select(arr3, 0, len(arr3) - 1, len(arr3) - k)
    print("\n数组:", arr3)
```

```
print(f"第{k}大的元素是: {result}")

# 测试堆排序
arr4 = [5, 2, 3, 1, 4]
print("\n原始数组:", arr4)
solution.heap_sort(arr4)
print("堆排序后:", arr4)

# 测试优化版本的快速排序
arr5 = [9, 8, 7, 6, 5, 4, 3, 2, 1, 0, 5, 6, 7, 8, 9]
print("\n原始数组:", arr5)
solution.quick_sort_optimized(arr5, 0, len(arr5) - 1)
print("优化快排后:", arr5)
print(f"数组是否有序: {solution.is_sorted(arr5)}")

# 测试颜色分类
colors = [2, 0, 2, 1, 1, 0]
print("\n原始颜色数组:", colors)
solution.sort_colors(colors)
print("颜色分类后:", colors)

# 测试移动零
zeros = [0, 1, 0, 3, 12]
print("\n原始数组:", zeros)
solution.move_zeroes(zeros)
print("移动零后:", zeros)

# 测试寻找第K大元素
nums = [3, 2, 1, 5, 6, 4]
k = 2
print("\n数组:", nums)
print(f"第{k}大的元素是: {solution.find_kth_largest(nums, k)}")

# 测试最小的k个数
arr6 = [3, 2, 1, 5, 6, 4]
k = 3
print("\n数组:", arr6)
print(f"最小的{k}个数是: {solution.get_least_numbers(arr6, k)}")

# 测试中位数问题
arr7 = [1, 2, 3, 4, 5]
print("\n数组:", arr7)
print(f"中位数是: {solution.find_median(arr7)}")
```

```
    # 测试插入排序
    arr8 = [9, 8, 7, 6, 5]
    print("\n原始数组:", arr8)
    solution.insertion_sort(arr8, 0, len(arr8) - 1)
    print("插入排序后:", arr8)

    # 测试健壮性处理
    empty_arr = []
    single_arr = [5]
    solution.robust_quick_sort(empty_arr)
    solution.robust_quick_sort(single_arr)
    print("\n空数组排序后:", empty_arr)
    print("单元素数组排序后:", single_arr)

if __name__ == "__main__":
    main()
```

==========

文件: TestAll.java

==========

```
package class023;

import java.util.Arrays;

/**
 * 综合测试类，用于测试所有快速排序相关算法的正确性
 * 本类包含对基础快速排序、三路快速排序、快速选择算法和最小k个数算法的全面测试
 */
public class TestAll {

    /**
     * 程序入口点
     * @param args 命令行参数
     */
    public static void main(String[] args) {
        // 测试基础快速排序
        testBasicQuickSort();

        // 测试三路快速排序
        testThreeWayQuickSort();
```

```
        // 测试快速选择算法
        testQuickSelect();

        // 测试最小 k 个数
        testGetLeastNumbers();

        System.out.println("所有测试通过！");
    }

    /**
     * 测试基础快速排序
     * 包含多种测试场景：普通数组、包含重复元素的数组、已排序数组、逆序数组
     */
    private static void testBasicQuickSort() {
        System.out.println("=== 测试基础快速排序 ===");

        // 测试用例 1：普通数组
        int[] arr1 = {5, 2, 3, 1, 4};
        System.out.println("排序前: " + Arrays.toString(arr1));
        Code02_QuickSort.sortArray(arr1);
        System.out.println("排序后: " + Arrays.toString(arr1));
        assert isSorted(arr1) : "基础快速排序测试失败";

        // 测试用例 2：包含重复元素的数组
        int[] arr2 = {5, 2, 3, 1, 4, 2, 3};
        System.out.println("排序前: " + Arrays.toString(arr2));
        Code02_QuickSort.sortArray(arr2);
        System.out.println("排序后: " + Arrays.toString(arr2));
        assert isSorted(arr2) : "包含重复元素的数组排序测试失败";

        // 测试用例 3：已排序数组
        int[] arr3 = {1, 2, 3, 4, 5};
        System.out.println("排序前: " + Arrays.toString(arr3));
        Code02_QuickSort.sortArray(arr3);
        System.out.println("排序后: " + Arrays.toString(arr3));
        assert isSorted(arr3) : "已排序数组测试失败";

        // 测试用例 4：逆序数组
        int[] arr4 = {5, 4, 3, 2, 1};
        System.out.println("排序前: " + Arrays.toString(arr4));
        Code02_QuickSort.sortArray(arr4);
        System.out.println("排序后: " + Arrays.toString(arr4));
        assert isSorted(arr4) : "逆序数组测试失败";
```

```
        System.out.println("基础快速排序测试通过\n");
    }

    /**
     * 测试三路快速排序
     * 专门测试处理包含大量重复元素的数组场景
     */
    private static void testThreeWayQuickSort() {
        System.out.println("=== 测试三路快速排序 ===");

        // 测试用例：包含大量重复元素的数组
        int[] arr = {3, 2, 3, 1, 2, 4, 5, 5, 6, 3, 3, 3};
        System.out.println("排序前: " + Arrays.toString(arr));

        // 使用三路快排思想进行排序
        Code02_QuickSort solution = new Code02_QuickSort();
        if (arr.length > 1) {
            // 调用改进版快速排序算法（三路快排）
            solution.quickSort2(arr, 0, arr.length - 1);
        }

        System.out.println("排序后: " + Arrays.toString(arr));
        assert isSorted(arr) : "三路快速排序测试失败";

        System.out.println("三路快速排序测试通过\n");
    }

    /**
     * 测试快速选择算法
     * 用于验证在数组中查找第 k 大元素的正确性
     */
    private static void testQuickSelect() {
        System.out.println("=== 测试快速选择算法 ===");

        Code03_QuickSort_Leetcode215 solution = new Code03_QuickSort_Leetcode215();

        // 测试用例 1：查找第 2 大的元素
        int[] nums1 = {3, 2, 1, 5, 6, 4};
        int k1 = 2;
        int result1 = solution.findKthLargest(nums1, k1);
        System.out.println("数组: " + Arrays.toString(nums1) + ", k=" + k1 + ", 第" + k1 + "大的
元素是: " + result1);
```

```
        // 验证结果：数组排序后为[1,2,3,4,5,6]，第 2 大元素是 5
        assert result1 == 5 : "快速选择算法测试 1 失败";

        // 测试用例 2：包含重复元素的数组，查找第 4 大的元素
        int[] nums2 = {3, 2, 3, 1, 2, 4, 5, 5, 6};
        int k2 = 4;
        int result2 = solution.findKthLargest(nums2, k2);
        System.out.println("数组: " + Arrays.toString(nums2) + ", k=" + k2 + ", 第" + k2 + "大的
元素是: " + result2);
        // 验证结果：数组排序后为[1,2,2,3,3,4,5,5,6]，第 4 大元素是 4
        assert result2 == 4 : "快速选择算法测试 2 失败";

        System.out.println("快速选择算法测试通过\n");
    }

    /**
     * 测试最小 k 个数
     * 用于验证找出数组中最小的 k 个数的正确性
     */
    private static void testGetLeastNumbers() {
        System.out.println("=== 测试最小 k 个数 ===");

        Code04_QuickSort_JZ40 solution = new Code04_QuickSort_JZ40();

        // 测试用例 1：找出最小的 2 个数
        int[] arr1 = {3, 2, 1};
        int k1 = 2;
        int[] result1 = solution.getLeastNumbers(arr1, k1);
        System.out.println("数组: " + Arrays.toString(arr1) + ", k=" + k1 + ", 最小的" + k1 + "个
数是: " + Arrays.toString(result1));
        // 验证结果：数组排序后为[1,2,3]，最小的 2 个数是[1,2]
        assert result1.length == k1 && isSubArraySorted(result1) : "最小 k 个数测试 1 失败";

        // 测试用例 2：找出最小的 1 个数
        int[] arr2 = {0, 1, 2, 1};
        int k2 = 1;
        int[] result2 = solution.getLeastNumbers(arr2, k2);
        System.out.println("数组: " + Arrays.toString(arr2) + ", k=" + k2 + ", 最小的" + k2 + "个
数是: " + Arrays.toString(result2));
        // 验证结果：数组排序后为[0,1,1,2]，最小的 1 个数是[0]
        assert result2.length == k2 && isSubArraySorted(result2) : "最小 k 个数测试 2 失败";

        System.out.println("最小 k 个数测试通过\n");
```

```
    }

    /**
     * 检查数组是否已排序（升序）
     * @param arr 待检查的数组
     * @return 如果数组已排序返回 true，否则返回 false
     */
    private static boolean isSorted(int[] arr) {
        // 遍历数组，检查每个元素是否小于等于下一个元素
        for (int i = 1; i < arr.length; i++) {
            if (arr[i] < arr[i - 1]) {
                // 发现逆序对，数组未排序
                return false;
            }
        }
        // 没有发现逆序对，数组已排序
        return true;
    }

    /**
     * 检查数组子集是否已排序
     * 通过与排序后的数组比较来验证数组元素的正确性
     * @param arr 待检查的数组
     * @return 如果数组元素正确排序返回 true，否则返回 false
     */
    private static boolean isSubArraySorted(int[] arr) {
        // 创建数组副本并排序
        int[] sorted = arr.clone();
        Arrays.sort(sorted);

        // 比较原数组与排序后数组的元素
        for (int i = 0; i < arr.length; i++) {
            if (arr[i] != sorted[i]) {
                // 发现不匹配的元素
                return false;
            }
        }
        // 所有元素都匹配
        return true;
    }
}
```

==========