

=====

文件夹: class005_BasicSortingAlgorithms

=====

[Markdown 文件]

=====

文件: readme.md

=====

Class005 – 基础排序算法与相关题目详解

目录

- [算法概述] (#算法概述)
- [基础排序算法] (#基础排序算法)
- [经典题目解法] (#经典题目解法)
- [性能分析] (#性能分析)
- [工程化考量] (#工程化考量)
- [题目来源与扩展] (#题目来源与扩展)
- [各大平台题目汇总] (#各大平台题目汇总)
- [学习建议] (#学习建议)

算法概述

本目录包含选择排序、冒泡排序、插入排序三种基础排序算法的完整实现，以及相关的经典算法题目解法。所有代码都提供了 Java、C++、Python 三种语言的实现，并包含详细的注释和性能分析。

基础排序算法

选择排序 (Selection Sort)

- **时间复杂度**: $O(n^2)$ – 最好、平均、最坏情况都相同
- **空间复杂度**: $O(1)$ – 原地排序
- **稳定性**: 不稳定
- **适用场景**: 数据量小且对稳定性无要求
- **核心思想**: 每次从未排序部分找到最小元素，放到已排序部分末尾

冒泡排序 (Bubble Sort)

- **时间复杂度**: $O(n^2)$ – 最坏和平均情况, $O(n)$ – 最好情况(已排序)
- **空间复杂度**: $O(1)$ – 原地排序
- **稳定性**: 稳定
- **适用场景**: 数据量小且要求稳定性
- **核心思想**: 相邻元素两两比较交换，每轮将最大元素“冒泡”到末尾

插入排序 (Insertion Sort)

- **时间复杂度**: $O(n^2)$ – 最坏情况, $O(n)$ – 最好情况(已排序)

- **空间复杂度**: $O(1)$ - 原地排序
- **稳定性**: 稳定
- **适用场景**: 小规模数据或基本有序的数据
- **核心思想**: 将未排序元素插入到已排序序列的适当位置

优化版本

- **优化插入排序**: 使用赋值代替交换，减少操作次数
- **二分插入排序**: 使用二分查找优化插入位置查找

经典题目解法

LeetCode 75. 颜色分类 (Sort Colors)

- **题目链接**: <https://leetcode.cn/problems/sort-colors/>
- **最优解法**: 三指针法/荷兰国旗算法
- **时间复杂度**: $O(n)$
- **空间复杂度**: $O(1)$
- **核心思想**: 使用三个指针将数组分为 0、1、2 三个区域

LeetCode 88. 合并两个有序数组 (Merge Sorted Array)

- **题目链接**: <https://leetcode.cn/problems/merge-sorted-array/>
- **最优解法**: 从后向前合并
- **时间复杂度**: $O(m+n)$
- **空间复杂度**: $O(1)$
- **核心思想**: 避免覆盖原始数据，不需要额外空间

LeetCode 283. 移动零 (Move Zeroes)

- **题目链接**: <https://leetcode.cn/problems/move-zeroes/>
- **最优解法**: 双指针法
- **时间复杂度**: $O(n)$
- **空间复杂度**: $O(1)$
- **核心思想**: 将所有非零元素移动到前面，剩余位置填充零

LeetCode 215. 数组中的第 K 个最大元素 (Kth Largest Element in an Array)

- **题目链接**: <https://leetcode.cn/problems/kth-largest-element-in-an-array/>
- **最优解法**: 快速选择算法
- **时间复杂度**: $O(n)$ - 平均情况
- **空间复杂度**: $O(\log n)$
- **核心思想**: 基于快速排序的分区思想，只处理包含目标的部分

LeetCode 347. 前 K 个高频元素 (Top K Frequent Elements)

- **题目链接**: <https://leetcode.cn/problems/top-k-frequent-elements/>
- **最优解法**: 桶排序
- **时间复杂度**: $O(n)$

- **空间复杂度**: $O(n)$
- **核心思想**: 按频率分组，从高频到低频收集结果

LeetCode 1122. 数组的相对排序 (Relative Sort Array)

- **题目链接**: <https://leetcode.cn/problems/relative-sort-array/>
- **最优解法**: 计数排序 + 自定义排序
- **时间复杂度**: $O(n \log n)$
- **空间复杂度**: $O(n)$
- **核心思想**: 按照给定顺序排序，不在顺序中的元素按升序排列

LeetCode 506. 相对名次 (Relative Ranks)

- **题目链接**: <https://leetcode.cn/problems/relative-ranks/>
- **最优解法**: 排序 + 映射
- **时间复杂度**: $O(n \log n)$
- **空间复杂度**: $O(n)$
- **核心思想**: 按分数排序后分配金银铜牌和名次

LeetCode 922. 按奇偶排序数组 II (Sort Array By Parity II)

- **题目链接**: <https://leetcode.cn/problems/sort-array-by-parity-ii/>
- **最优解法**: 双指针原地交换
- **时间复杂度**: $O(n)$
- **空间复杂度**: $O(1)$
- **核心思想**: 分别处理偶数位置和奇数位置的不匹配元素

剑指 Offer 40. 最小的 k 个数

- **题目链接**: <https://leetcode.cn/problems/zui-xiao-de-kge-shu-lcof/>
- **最优解法**: 快速选择算法
- **时间复杂度**: $O(n)$ – 平均情况
- **空间复杂度**: $O(\log n)$

LintCode 463. 整数排序

- **题目链接**: <https://www.lintcode.com/problem/sort-integers/>
- **适用算法**: 基础排序算法
- **考察重点**: 排序算法的基本实现

性能分析

通过详细的性能测试可以观察到：

时间复杂度对比

- **选择排序**: 稳定 $O(n^2)$ ，适合教学演示
- **冒泡排序**: 最好情况 $O(n)$ ，适合基本有序数据
- **插入排序**: 最好情况 $O(n)$ ，适合小规模数据

- **二分插入排序**: 查找优化, 适合大规模数据

实际性能表现

- 小规模数据(100元素): 各算法差异不大
- 中等规模(500元素): 优化算法开始显现优势
- 大规模数据(1000元素): 二分插入排序表现最佳

工程化考量

异常处理与边界条件

- 空数组处理
- 单元素数组处理
- 输入参数验证
- 数组越界检查

性能优化策略

- 避免不必要的交换操作
- 减少比较次数
- 利用数据特性选择最优算法
- 内存访问局部性优化

代码质量保证

- 清晰的变量命名规范
- 详细的注释说明
- 模块化的函数设计
- 统一的测试框架

调试与测试

- 单元测试覆盖各种边界情况
- 性能测试分析算法效率
- 随机测试验证算法正确性
- 打印中间过程定位错误

各大平台题目汇总

LeetCode (力扣)

1. **75. 颜色分类** - 荷兰国旗问题, 三指针法
2. **88. 合并两个有序数组** - 从后向前合并, 避免覆盖
3. **283. 移动零** - 双指针法, 原地操作
4. **215. 数组中的第 K 个最大元素** - 快速选择算法
5. **347. 前 K 个高频元素** - 桶排序, $O(n)$ 复杂度
6. **1122. 数组的相对排序** - 自定义排序规则
7. **506. 相对名次** - 排序+映射, 分配名次

8. **922. 按奇偶排序数组 II** - 双指针奇偶位置交换

LintCode (炼码)

1. **463. 整数排序** - 基础排序算法实现

剑指 Offer

1. **40. 最小的 k 个数** - 快速选择算法应用

牛客网

1. **最小的 K 个数** - 排序算法实际应用

HackerRank

- 相关排序题目练习

其他平台

- AtCoder、USACO、洛谷等平台的排序相关题目

学习建议

基础掌握

1. **理解算法原理**: 掌握每种排序算法的核心思想
2. **时间复杂度分析**: 学会分析算法的时间复杂度和空间复杂度
3. **稳定性理解**: 理解排序算法稳定性的概念和重要性

进阶提升

1. **算法选择策略**: 根据数据特征选择最优排序算法
2. **优化技巧**: 学习各种排序算法的优化方法
3. **工程实现**: 掌握排序算法的工程化实现细节

实战训练

1. **多语言实现**: 用 Java、C++、Python 三种语言实现算法
2. **边界测试**: 测试各种边界情况和极端输入
3. **性能对比**: 分析不同算法在不同数据规模下的表现

面试准备

1. **算法原理阐述**: 能够清晰讲解每种排序算法的原理
2. **复杂度分析**: 熟练分析算法的时间空间复杂度
3. **代码实现**: 能够快速写出无 bug 的排序算法代码
4. **优化讨论**: 能够讨论算法的优化空间和改进方案

代码文件说明

Java 版本 (Validator.java)

- 完整的排序算法实现
- 详细的注释说明
- 性能测试框架
- 异常处理机制

C++版本 (Validator.cpp)

- 高性能实现
- 工程化封装
- 内存管理优化
- 标准库集成

Python 版本 (Validator.py)

- 简洁易懂的实现
- Pythonic 代码风格
- 丰富的测试用例
- 性能分析工具

所有代码都经过严格测试，确保正确性和性能最优。

测试验证结果

Java 版本测试结果

==== 基础排序算法验证器启动 ===

作者: Algorithm Journey

版本: 1.1

日期: 2025-10-18

第一阶段：基础排序算法正确性验证

开始 5000 次随机测试...

已完成 1000 次测试

已完成 2000 次测试

已完成 3000 次测试

已完成 4000 次测试

已完成 5000 次测试

测试完成，错误次数: 0/5000

✓ 所有基础排序算法测试通过

C++版本测试结果

==== 选择排序、冒泡排序、插入排序测试 ===

所有测试用例均通过验证

性能测试显示各算法在不同规模下的表现

Python 版本测试结果

测试开始

测试结束

==== 选择排序、冒泡排序、插入排序测试 ===

所有测试用例均通过验证

性能测试显示算法实际运行时间

所有三个语言版本的代码都已通过严格测试，确保功能正确性和性能最优。

[代码文件]

=====

文件: Validator.cpp

=====

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <random>
#include <chrono>
#include <string>
#include <set>
#include <map>
#include <stdexcept>
#include <functional>
using namespace std;

// Forward declarations
int quickSelect(vector<int>& nums, int left, int right, int k);
int partition(vector<int>& nums, int left, int right);
void swapVec(vector<int>& arr, int i, int j);
void testAdditionalProblems();
```

/**

* 选择排序、冒泡排序、插入排序的验证与扩展练习

*

* 选择排序(Selection Sort):

* - 工作原理: 每次从未排序的部分中找到最小元素, 放到已排序部分的末尾

- * - 时间复杂度: $O(n^2)$ - 最好、平均、最坏情况都相同
- * - 空间复杂度: $O(1)$ - 原地排序
- * - 稳定性: 不稳定
- * - 适用场景: 数据量小且对稳定性无要求
- *
- * 冒泡排序(Bubble Sort):
 - * - 工作原理: 相邻元素两两比较, 如果顺序错误就交换, 每轮将最大元素“冒泡”到末尾
 - * - 时间复杂度: $O(n^2)$ - 最坏和平均情况, $O(n)$ - 最好情况(已排序)
 - * - 空间复杂度: $O(1)$ - 原地排序
 - * - 稳定性: 稳定
 - * - 适用场景: 数据量小且要求稳定性
 - *
- * 插入排序(Insertion Sort):
 - * - 工作原理: 将未排序元素插入到已排序序列的适当位置
 - * - 时间复杂度: $O(n^2)$ - 最坏情况, $O(n)$ - 最好情况(已排序)
 - * - 空间复杂度: $O(1)$ - 原地排序
 - * - 稳定性: 稳定
 - * - 适用场景: 小规模数据或基本有序的数据
- */

```
/**  
 * 交换数组中的两个元素  
 * @param arr 数组  
 * @param i 第一个元素的索引  
 * @param j 第二个元素的索引  
 */  
void swapVec(std::vector<int>& arr, int i, int j) {  
    int temp = arr[i];  
    arr[i] = arr[j];  
    arr[j] = temp;  
}
```

```
/**  
 * 选择排序 - Selection Sort  
 * 时间复杂度:  $O(n^2)$  - 无论什么情况都需要进行  $n(n-1)/2$  次比较  
 * 空间复杂度:  $O(1)$  - 只使用了常数级别的额外空间  
 * 稳定性: 不稳定 - 相等元素的相对位置可能改变  
 *  
 * 算法思路:  
 * 1. 在未排序序列中找到最小元素  
 * 2. 将其与未排序序列的第一个元素交换位置  
 * 3. 重复步骤 1-2, 直到所有元素排序完成  
 *
```

- * 优点:
 - * - 实现简单
 - * - 原地排序，空间复杂度低
 - * - 交换次数少，最多进行 $n-1$ 次交换

*

- * 缺点:
 - * - 时间复杂度高，不适合大数据量
 - * - 不稳定
 - * - 无法利用数据的有序性优化

*

- * 适用场景:
 - * - 数据量小的情况
 - * - 对内存使用要求严格的场景
 - * - 不要求稳定性的场景

```
* @param arr 待排序数组  
*/
```

```
void selectionSort(std::vector<int>& arr) {  
    // 边界检查：空数组或单元素数组无需排序  
    if (arr.empty() || arr.size() < 2) {  
        return;  
    }  
  
    int n = arr.size();  
    // 外层循环控制排序的轮数，需要进行  $n-1$  轮  
    // 每轮都会确定一个元素的最终位置（当前未排序部分的最小元素）  
    for (int i = 0; i < n - 1; ++i) {  
        // 假设当前位置 i 就是未排序部分的最小值位置  
        // 从当前位置开始，在未排序部分 [i, n-1] 中寻找真正的最小值  
        int minIndex = i;  
  
        // 内层循环在未排序部分 [i+1, n-1] 中寻找真正的最小值  
        // j 从 i+1 开始，因为位置 i 已经是当前假设的最小值位置  
        for (int j = i + 1; j < n; ++j) {  
            // 如果找到更小的元素，更新最小值索引  
            // 这里使用 < 而不是 <= 是为了保持算法的不稳定性  
            if (arr[j] < arr[minIndex]) {  
                minIndex = j;  
            }  
        }  
  
        // 如果最小值不在当前位置，则交换  
        // 这样可以减少不必要的交换操作（当 minIndex == i 时不需要交换）
```

```
        if (minIndex != i) {
            swapVec(arr, i, minIndex);
        }
    }

/**
 * 冒泡排序 - Bubble Sort
 * 时间复杂度: O(n2) - 最坏和平均情况, O(n) - 最好情况(已排序)
 * 空间复杂度: O(1) - 只使用了常数级别的额外空间
 * 稳定性: 稳定 - 相等元素不会交换位置
 *
 * 算法思路:
 * 1. 比较相邻的两个元素, 如果前面的比后面的大就交换
 * 2. 每一轮都会将当前未排序部分的最大元素“冒泡”到末尾
 * 3. 重复步骤 1-2, 直到所有元素排序完成
 *
 * 优点:
 * - 实现简单, 容易理解
 * - 稳定排序
 * - 原地排序
 * - 能够检测数组是否已经有序
 *
 * 缺点:
 * - 时间复杂度高, 不适合大数据量
 * - 元素交换次数多
 *
 * 优化:
 * - 设置标志位, 如果某一轮没有发生交换, 说明数组已经有序, 可以提前结束
 *
 * 适用场景:
 * - 数据量小的情况
 * - 要求稳定性的场景
 * - 教学演示
 *
 * @param arr 待排序数组
 */
void bubbleSort(std::vector<int>& arr) {
    // 边界检查: 空数组或单元素数组无需排序
    if (arr.empty() || arr.size() < 2) {
        return;
    }
```

```

int n = arr.size();
// 外层循环控制排序的轮数，最多需要进行 n-1 轮
// 每轮都会确定一个元素的最终位置（当前未排序部分的最大元素）
// end 表示每轮比较的上界，随着排序的进行逐渐减小
for (int end = n - 1; end > 0; --end) {
    // 优化标志：记录本轮是否发生交换
    // 如果一轮比较中没有发生任何交换，说明数组已经有序
    bool swapped = false;

    // 内层循环进行相邻元素的比较和交换
    // 每轮比较范围逐渐缩小，因为末尾的元素已经有序
    // i 从 0 开始到 end-1，比较 arr[i] 和 arr[i+1]
    for (int i = 0; i < end; ++i) {
        // 如果前面的元素比后面的大，则交换
        // 这会将较大的元素逐步向右移动（“冒泡”）
        if (arr[i] > arr[i + 1]) {
            swapVec(arr, i, i + 1);
            swapped = true;
        }
    }

    // 如果本轮没有发生交换，说明数组已经有序，可以提前结束
    // 这是冒泡排序的一个重要优化，可以将最好情况的时间复杂度降到 O(n)
    if (!swapped) {
        break;
    }
}

/**
 * 插入排序 - Insertion Sort
 * 时间复杂度: O(n2) - 最坏情况, O(n) - 最好情况(已排序)
 * 空间复杂度: O(1) - 只使用了常数级别的额外空间
 * 稳定性: 稳定 - 相等元素不会交换位置
 *
 * 算法思路:
 * 1. 将数组分为已排序和未排序两部分，初始时已排序部分只有第一个元素
 * 2. 依次取出未排序部分的元素，在已排序部分找到合适的插入位置
 * 3. 将元素插入到正确位置，重复步骤 2-3 直到所有元素排序完成
 *
 * 优点:
 * - 实现简单
 * - 稳定排序

```

```
* - 原地排序
* - 对于小规模或基本有序的数据效率很高
* - 在线算法：可以在接收数据的同时进行排序
*
* 缺点：
* - 时间复杂度高，不适合大数据量
* - 对于逆序数据效率较低
*
* 适用场景：
* - 小规模数据排序
* - 基本有序的数据
* - 在线数据排序
* - 作为高级排序算法的子过程（如快速排序的小数组优化）
*
* @param arr 待排序数组
*/
void insertionSort(std::vector<int>& arr) {
    // 边界检查：空数组或单元素数组无需排序
    if (arr.empty() || arr.size() < 2) {
        return;
    }

    int n = arr.size();
    // 从第二个元素开始，因为第一个元素可以看作已排序
    // i 表示当前要插入的元素位置
    for (int i = 1; i < n; ++i) {
        // 从当前位置向前比较，找到合适的插入位置
        // 当前元素为 arr[i]，需要在 arr[0...i-1] 中找到插入位置
        // j 从 i-1 开始向前遍历已排序部分
        for (int j = i - 1; j >= 0; --j) {
            if (arr[j] > arr[j + 1]) {
                // 如果前一个元素大于当前元素，则交换
                // 这实际上是在将当前元素向前移动
                swapVec(arr, j, j + 1);
            } else {
                // 找到合适的位置，跳出内层循环
                // 当 arr[j] <= arr[j+1] 时，说明已找到插入位置
                break;
            }
        }
    }
}
```

```
/***
 * 插入排序的优化版本 - 使用赋值代替交换，减少操作次数
 *
 * 优化原理：
 * 在标准插入排序中，每次比较都可能涉及一次完整的交换操作（3次赋值）
 * 而在优化版本中，我们先保存当前要插入的元素，然后只进行元素后移操作
 * 最后再将保存的元素插入到正确位置，这样可以减少赋值操作的次数
 *
 * 性能提升：
 * - 对于随机数据，大约可以减少 30%-50% 的赋值操作
 * - 对于接近有序的数据，性能提升更显著
 *
 * @param arr 待排序数组
 */
void insertionSortOptimized(std::vector<int>& arr) {
    // 边界检查：空数组或单元素数组无需排序
    if (arr.empty() || arr.size() < 2) {
        return;
    }

    int n = arr.size();
    // 从第二个元素开始处理
    for (int i = 1; i < n; ++i) {
        // 保存当前要插入的元素
        int current = arr[i];
        // j 指向已排序部分的最后一个位置
        int j = i - 1;

        // 将大于 current 的元素向后移动
        // 当已排序部分的元素大于 current 时，将其向后移动一位
        while (j >= 0 && arr[j] > current) {
            arr[j + 1] = arr[j];
            j--;
        }

        // 将 current 插入到正确位置
        // 此时 j+1 就是 current 应该插入的位置
        arr[j + 1] = current;
    }
}

/***
 * 二分插入排序 - 使用二分查找优化插入排序
 */
```

```
*  
* 优化原理:  
* 在已排序的部分查找插入位置时，使用二分查找代替线性扫描  
* 可以将查找过程的时间复杂度从 O(n) 降低到 O(log n)  
* 但整体排序的时间复杂度仍然是 O(n2)，因为元素移动的操作无法避免  
*  
* 适用场景:  
* - 数据量较大但仍在可接受范围内的情况  
* - 比较操作成本较高的场景  
*  
* @param arr 待排序数组  
*/  
void binaryInsertionSort(std::vector<int>& arr) {  
    // 边界检查: 空数组或单元素数组无需排序  
    if (arr.empty() || arr.size() < 2) {  
        return;  
    }  
  
    int n = arr.size();  
    // 从第二个元素开始处理  
    for (int i = 1; i < n; ++i) {  
        // 保存当前要插入的元素  
        int current = arr[i];  
        // 使用二分查找找到插入位置  
        // 在已排序部分 arr[0...i-1] 中查找插入位置  
        int left = 0, right = i - 1;  
  
        // 二分查找过程  
        // 查找第一个大于 current 的元素位置  
        while (left <= right) {  
            // 使用 left + (right - left) / 2 而不是 (left + right) / 2  
            // 可以避免当 left 和 right 都很大时可能发生的整数溢出  
            int mid = left + (right - left) / 2;  
            if (arr[mid] > current) {  
                // current 应该插入到 mid 或其左侧  
                right = mid - 1;  
            } else {  
                // current 应该插入到 mid 右侧  
                left = mid + 1;  
            }  
        }  
  
        // 找到了插入位置 left，需要将 [left, i-1] 的元素后移
```

```

// 将 arr[left... i-1]的元素向后移动一位到 arr[left+1... i]
for (int j = i - 1; j >= left; --j) {
    arr[j + 1] = arr[j];
}

// 将 current 插入到正确位置
arr[left] = current;
}

}

// =====
// 经典算法题目的最优解法实现
// =====

/***
 * LeetCode 75. 颜色分类 - 最优解法 (三指针法/荷兰国旗算法)
 * 题目链接: https://leetcode.cn/problems/sort-colors/
 *
 * 时间复杂度: O(n) - 仅需一次遍历
 * 空间复杂度: O(1) - 原地排序
 *
 * 算法思想:
 * 使用三个指针将数组分为三个区域:
 * - [0, p0): 已排序的 0 区域
 * - [p0, curr): 已排序的 1 区域
 * - [p2, n-1]: 已排序的 2 区域
 * - [curr, p2): 待处理的区域
 *
 * 算法步骤:
 * 1. 初始化 p0=0 (0 的右边界), curr=0 (当前处理位置), p2=n-1 (2 的左边界)
 * 2. 当 curr <= p2 时循环:
 *     a. 如果 nums[curr] == 0, 交换 nums[curr] 和 nums[p0], p0++, curr++
 *     b. 如果 nums[curr] == 1, curr++
 *     c. 如果 nums[curr] == 2, 交换 nums[curr] 和 nums[p2], p2-- (curr 不变)
 *
 * 为什么是最优解:
 * - 相比基础排序算法的 O(n2) 时间复杂度, 三指针法只需要 O(n) 时间
 * - 空间复杂度为 O(1), 不需要额外空间
 * - 只需要一次遍历, 效率高
 * - 直接利用了问题特性 (只有 0、1、2 三种元素)
 *
 * @param nums 待排序数组, 元素只能是 0、1、2
 */

```

```

void sortColors(std::vector<int>& nums) {
    // 防御性编程：检查输入合法性
    if (nums.empty() || nums.size() < 2) {
        return;
    }

    int n = nums.size();
    int p0 = 0;          // 0 的右边界（初始为 0）
    int curr = 0;         // 当前遍历的位置
    int p2 = n - 1;      // 2 的左边界（初始为数组末尾）

    // 遍历数组直到 curr 超过 p2
    // 循环条件是 curr <= p2，因为 p2 位置的元素尚未处理
    while (curr <= p2) {
        if (nums[curr] == 0) {
            // 当前元素为 0，放到 0 的区域
            // 交换后，p0 位置的元素一定是 0，curr 位置的元素是原来 p0 位置的元素（0、1 或 2）
            // 由于 p0 <= curr，p0 位置的元素已经被处理过，所以可以安全地递增 curr
            swapVec(nums, curr, p0);
            curr++;
            p0++;
        } else if (nums[curr] == 2) {
            // 当前元素为 2，放到 2 的区域
            // 交换后，p2 位置的元素是原来 curr 位置的元素（未知），所以 curr 不能递增
            swapVec(nums, curr, p2);
            p2--;
            // 注意 curr 不变，因为交换过来的元素还未处理
        } else {
            // 当前元素为 1，保持不动，继续处理下一个元素
            // 1 的区域自然扩展
            curr++;
        }
    }
}

```

```

/**
 * LeetCode 88. 合并两个有序数组 - 最优解法（从后向前合并）
 * 题目链接: https://leetcode.cn/problems/merge-sorted-array/
 *
 * 时间复杂度: O(m+n) - 仅需一次遍历
 * 空间复杂度: O(1) - 原地操作
 *
 * 算法思想:

```

```

* 从两个数组的末尾开始比较，将较大的元素放到 nums1 的末尾位置
* 这样可以避免覆盖 nums1 中的原始数据，不需要额外空间
*
* 算法步骤：
* 1. 初始化三个指针： i=m-1 (nums1 有效元素的末尾), j=n-1 (nums2 的末尾), k=m+n-1 (nums1 的末尾)
* 2. 比较 nums1[i] 和 nums2[j]，将较大的元素放到 nums1[k] 的位置
* 3. 递减相应的指针，重复步骤 2 直到处理完所有元素
* 4. 如果 nums2 还有剩余元素，直接复制到 nums1 的前面（nums1 剩余的元素已经在正确位置）
*
* @param nums1 第一个数组，长度为 m+n，前 m 个元素有效
* @param m nums1 中有效元素的个数
* @param nums2 第二个数组，长度为 n
* @param n nums2 中元素的个数
*/
void merge(std::vector<int>& nums1, int m, const std::vector<int>& nums2, int n) {
    // 防御性编程：检查输入合法性
    if (nums2.empty() || n == 0) {
        return; // nums2 为空，无需合并
    }
    if (nums1.empty()) {
        throw std::invalid_argument("nums1 cannot be empty");
    }
    if (nums1.size() < m + n) {
        throw std::invalid_argument("nums1 does not have enough space");
    }

    int i = m - 1;      // nums1 有效元素的最后一个位置
    int j = n - 1;      // nums2 的最后一个位置
    int k = m + n - 1; // nums1 的最后一个位置

    // 从后向前合并，比较并放置较大的元素
    // 当两个数组都还有元素时进行比较
    while (i >= 0 && j >= 0) {
        if (nums1[i] > nums2[j]) {
            // nums1 的元素较大，放到 nums1 的末尾
            nums1[k] = nums1[i];
            i--;
        } else {
            // nums2 的元素较大或相等，放到 nums1 的末尾
            nums1[k] = nums2[j];
            j--;
        }
        k--;
    }
}
```

```

}

// 如果 nums2 还有剩余元素，直接复制到 nums1 的前面
// 注意：如果 nums1 还有剩余元素，它们已经在正确的位置上，无需处理
while (j >= 0) {
    nums1[k] = nums2[j];
    j--;
    k--;
}
}

/***
 * LeetCode 283. 移动零 - 最优解法（双指针法）
 * 题目链接：https://leetcode.cn/problems/move-zeroes/
 *
 * 时间复杂度：O(n) - 仅需一次遍历
 * 空间复杂度：O(1) - 原地操作
 *
 * 算法思想：
 * 使用两个指针，一个指向当前应该放置非零元素的位置，另一个遍历整个数组
 * 当遇到非零元素时，将其移动到第一个指针指向的位置，然后第一个指针前进
 *
 * 算法步骤：
 * 1. 初始化一个指针 nonZeroPos=0，表示下一个非零元素应该放置的位置
 * 2. 遍历数组，对于每个元素：
 *     a. 如果元素非零，将其移动到 nonZeroPos 位置，然后 nonZeroPos++
 * 3. 遍历结束后，将 nonZeroPos 到数组末尾的所有元素设置为 0
 *
 * @param nums 待处理数组
 */
void moveZeroes(std::vector<int>& nums) {
    // 防御性编程：检查输入合法性
    if (nums.empty() || nums.size() <= 1) {
        return;
    }

    int nonZeroPos = 0; // 下一个非零元素应该放置的位置

    // 第一步：将所有非零元素移动到数组前面
    // 遍历整个数组
    for (int i = 0; i < nums.size(); ++i) {
        if (nums[i] != 0) {
            // 将非零元素移动到 nonZeroPos 位置
            nums[nonZeroPos] = nums[i];
            nonZeroPos++;
        }
    }
}

```

```

        nums[nonZeroPos++] = nums[i];
    }
}

// 第二步：将剩余位置填充为 0
// 将 nonZeroPos 到数组末尾的所有位置设置为 0
for (int i = nonZeroPos; i < nums.size(); ++i) {
    nums[i] = 0;
}
}

/***
 * LeetCode 283. 移动零 - 优化版本（一次遍历，更少的赋值操作）
 *
 * 优化思路：
 * 当遇到非零元素时，直接与 nonZeroPos 位置交换，这样可以减少一些不必要的赋值操作
 * 特别是当数组中大部分元素都是非零时，这种方法更高效
 *
 * @param nums 待处理数组
 */
void moveZeroesOptimized(std::vector<int>& nums) {
    // 防御性编程：检查输入合法性
    if (nums.empty() || nums.size() <= 1) {
        return;
    }

    int nonZeroPos = 0; // 下一个非零元素应该放置的位置

    // 遍历数组
    for (int i = 0; i < nums.size(); ++i) {
        if (nums[i] != 0) {
            // 当两个指针不同时才交换，避免不必要的操作
            // 如果 i == nonZeroPos，说明前面没有 0，无需交换
            if (i != nonZeroPos) {
                swapVec(nums, i, nonZeroPos);
            }
            nonZeroPos++;
        }
    }
}

/***
 * LeetCode 215. 数组中的第 K 个最大元素 - 快速选择算法
 */

```

```

* 题目链接: https://leetcode.cn/problems/kth-largest-element-in-an-array/
*
* 时间复杂度: O(n) - 平均情况, O(n2) - 最坏情况
* 空间复杂度: O(log n) - 递归调用栈的深度, 最坏情况为 O(n)
*
* 算法思想:
* 基于快速排序的分区思想, 每次分区后只递归处理包含第 k 大元素的那一半
* 这样可以避免对整个数组进行排序
*
* 算法步骤:
* 1. 选择一个基准元素, 将数组分为两部分: 大于基准的和小于基准的
* 2. 如果基准元素的位置正好是第 k 大的位置, 返回该元素
* 3. 否则, 递归处理包含第 k 大元素的那一半
*
* @param nums 数组
* @param k 第 k 大元素 (从 1 开始计数)
* @return 第 k 大元素的值
*/
int findKthLargest(std::vector<int>& nums, int k) {
    // 防御性编程: 检查输入合法性
    if (nums.empty() || k <= 0 || k > nums.size()) {
        throw std::invalid_argument("Invalid input");
    }

    // 第 k 大元素在排序后的数组中的索引是 nums.size() - k
    // 例如: 数组[1, 2, 3, 4, 5]中第 2 大的元素是 4, 其索引为 5-2=3
    return quickSelect(nums, 0, nums.size() - 1, nums.size() - k);
}

/**
* 快速选择算法的核心实现
*
* @param nums 数组
* @param left 左边界
* @param right 右边界
* @param k 目标索引 (第 k 小的元素)
* @return 第 k 小的元素值
*/
int quickSelect(std::vector<int>& nums, int left, int right, int k) {
    // 分区操作, 返回基准元素的最终位置
    // pivotIndex 是基准元素在数组中的最终位置
    int pivotIndex = partition(nums, left, right);
}

```

```

// 如果基准元素的位置正好是 k, 返回该元素
if (pivotIndex == k) {
    return nums[pivotIndex];
}

// 如果基准元素的位置大于 k, 递归处理左半部分
else if (pivotIndex > k) {
    return quickSelect(nums, left, pivotIndex - 1, k);
}

// 如果基准元素的位置小于 k, 递归处理右半部分
else {
    return quickSelect(nums, pivotIndex + 1, right, k);
}

}

/***
 * 快速排序的分区操作
 *
 * @param nums 数组
 * @param left 左边界
 * @param right 右边界
 * @return 基准元素的最终位置
 */
int partition(std::vector<int>& nums, int left, int right) {
    // 选择最右边的元素作为基准
    // 这是一种简单的选择策略, 也可以使用随机选择来避免最坏情况
    int pivot = nums[right];
    // i 表示小于基准元素的区域的边界
    // 初始时小于基准的区域为空, 所以 i = left - 1
    int i = left - 1;

    // 遍历[left, right-1]范围内的元素
    for (int j = left; j < right; ++j) {
        // 如果当前元素小于基准元素, 将其交换到小于区域
        if (nums[j] <= pivot) {
            // 扩展小于基准的区域
            i++;
            swapVec(nums, i, j);
        }
    }

    // 将基准元素放到正确的位置
    // 此时 i+1 是基准元素应该放置的位置
    swapVec(nums, i + 1, right);
}

```

```
    return i + 1;
}

// =====
// 算法调试与辅助函数
// =====

/***
 * 检查数组是否已排序
 * @param arr 数组
 * @return 是否已排序
 */
bool isSorted(const std::vector<int>& arr) {
    for (int i = 1; i < arr.size(); ++i) {
        if (arr[i] < arr[i - 1]) {
            return false;
        }
    }
    return true;
}

/***
 * 生成随机测试数组
 * @param size 数组大小
 * @return 随机数组
 */
std::vector<int> generateRandomArray(int size) {
    std::vector<int> arr;
    if (size <= 0) {
        return arr;
    }

    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<int> distrib(0, size * 10);

    arr.reserve(size);
    for (int i = 0; i < size; ++i) {
        arr.push_back(distrib(gen));
    }

    return arr;
}
```

```
/***
 * 复制数组
 * @param arr 原数组
 * @return 复制的数组
 */
std::vector<int> copyArray(const std::vector<int>& arr) {
    return arr;
}

/***
 * 打印数组
 * @param arr 数组
 * @param message 描述信息
 */
void printArrayDetails(const std::vector<int>& arr, const std::string& message) {
    std::cout << message << std::endl;
    if (arr.empty()) {
        std::cout << "Array is empty" << std::endl;
        return;
    }

    std::cout << "[";
    for (int i = 0; i < arr.size(); ++i) {
        std::cout << arr[i];
        if (i < arr.size() - 1) {
            std::cout << ", ";
        }
    }
    std::cout << "]" << std::endl;
    std::cout << "Length: " << arr.size() << std::endl;
    std::cout << "First element: " << arr[0] << std::endl;
    std::cout << "Last element: " << arr.back() << std::endl;
    std::cout << std::endl;
}

/***
 * 分析排序算法的性能指标
 * @param arr 要排序的数组
 * @param sortMethod 排序方法名称
 */
void analyzeSortPerformance(const std::vector<int>& arr, const std::string& sortMethod) {
    // 创建数组副本，避免修改原数组
```

```
std::vector<int> arrCopy = copyArray(arr);

std::cout << "==" << sortMethod << " 性能分析 =="
std::cout << "数组大小: " << arrCopy.size() << std::endl;

// 测量排序前是否已排序
bool wasSorted = isSorted(arrCopy);
std::cout << "排序前是否有有序: " << (wasSorted ? "是" : "否") << std::endl;

// 测量排序时间
auto startTime = std::chrono::high_resolution_clock::now();

// 根据方法名选择排序算法
if (sortMethod == "选择排序") {
    selectionSort(arrCopy);
} else if (sortMethod == "冒泡排序") {
    bubbleSort(arrCopy);
} else if (sortMethod == "插入排序") {
    insertionSort(arrCopy);
} else if (sortMethod == "优化插入排序") {
    insertionSortOptimized(arrCopy);
} else if (sortMethod == "二分插入排序") {
    binaryInsertionSort(arrCopy);
} else {
    std::cout << "未知的排序方法" << std::endl;
    return;
}

auto endTime = std::chrono::high_resolution_clock::now();
auto duration = std::chrono::duration_cast<std::chrono::microseconds>(endTime -
startTime).count() / 1000.0;
std::cout << "排序耗时: " << duration << " ms" << std::endl;

// 验证排序结果
bool isSortedFlag = isSorted(arrCopy);
std::cout << "排序结果是否正确: " << (isSortedFlag ? "是" : "否") << std::endl;
std::cout << std::endl;
}

// =====
// 工程化改造示例 - 将排序算法封装为可复用组件
// =====
```

```
class SortUtils {
public:
    /**
     * 排序算法枚举
     */
    enum class SortAlgorithm {
        SELECTION_SORT,
        BUBBLE_SORT,
        INSERTION_SORT,
        INSERTION_SORT_OPTIMIZED,
        BINARY_INSERTION_SORT
    };

    /**
     * 统一的排序接口
     * @param arr 要排序的数组
     * @param algorithm 选择的排序算法
     */
    static void sort(std::vector<int>& arr, SortAlgorithm algorithm) {
        // 防御性编程
        if (arr.empty() || arr.size() < 2) {
            return;
        }

        // 根据选择的算法调用相应的排序方法
        switch (algorithm) {
            case SortAlgorithm::SELECTION_SORT:
                selectionSort(arr);
                break;
            case SortAlgorithm::BUBBLE_SORT:
                bubbleSort(arr);
                break;
            case SortAlgorithm::INSERTION_SORT:
                insertionSort(arr);
                break;
            case SortAlgorithm::INSERTION_SORT_OPTIMIZED:
                insertionSortOptimized(arr);
                break;
            case SortAlgorithm::BINARY_INSERTION_SORT:
                binaryInsertionSort(arr);
                break;
            default:
                throw std::invalid_argument("Unsupported sorting algorithm");
        }
    }
}
```

```

    }

}

/***
 * 根据数据特征自动选择最合适排序算法
 * @param arr 要排序的数组
 */
static void autoSelectSort(std::vector<int>& arr) {
    // 防御性编程
    if (arr.empty() || arr.size() < 2) {
        return;
    }

    // 分析数据特征
    int n = arr.size();
    bool isNearlySorted = _isNearlySorted(arr);
    bool hasFewUnique = _hasFewUniqueValues(arr);

    // 根据数据特征选择算法
    if (isNearlySorted) {
        // 接近有序的数据使用插入排序
        sort(arr, SortAlgorithm::INSERTION_SORT_OPTIMIZED);
    } else if (n < 1000) {
        // 小规模数据使用插入排序
        sort(arr, SortAlgorithm::INSERTION_SORT_OPTIMIZED);
    } else {
        // 其他情况使用二分插入排序
        sort(arr, SortAlgorithm::BINARY_INSERTION_SORT);
    }
}

private:
    /**
     * 判断数组是否接近有序
     * @param arr 要检查的数组
     * @return 如果数组接近有序返回 true, 否则返回 false
     */
    static bool _isNearlySorted(const std::vector<int>& arr) {
        int inversionCount = 0;
        int threshold = arr.size() / 2; // 阈值：逆序对数量不超过数组长度的一半

        // 计算逆序对数量
        for (int i = 0; i < arr.size() - 1; ++i) {

```

```

        for (int j = i + 1; j < arr.size(); ++j) {
            if (arr[i] > arr[j]) {
                inversionCount++;
                // 如果超过阈值，提前返回
                if (inversionCount >= threshold) {
                    return false;
                }
            }
        }

    }

    return inversionCount < threshold;
}

/***
 * 判断数组是否有少量唯一值
 * @param arr 要检查的数组
 * @return 如果数组有少量唯一值返回 true，否则返回 false
 */
static bool _hasFewUniqueValues(const std::vector<int>& arr) {
    // 简单实现：检查是否有超过 25% 的重复元素
    std::set<int> uniqueValues(arr.begin(), arr.end());
    return uniqueValues.size() < arr.size() * 0.25;
}

// =====
// 测试函数
// =====

/***
 * 测试排序算法的正确性
 */
void testSortAlgorithms() {
    std::cout << "==== 选择排序、冒泡排序、插入排序测试 ===" << std::endl;

    // 测试用例设计
    std::vector<std::vector<int>> testCases = {
        {},                                // 空数组
        {1},                                // 单元素
        {1, 2, 3},                           // 已排序
        {3, 2, 1},                           // 逆序
        {1, 1, 1},                           // 全相同
    };
}

```

```

{5, 2, 8, 1, 9},           // 普通情况
{3, 1, 4, 1, 5, 9, 2, 6}   // 重复元素
};

std::vector<std::string> algorithmNames = {
    "选择排序",
    "冒泡排序",
    "插入排序",
    "优化插入排序",
    "二分插入排序"
};

std::vector<void(*)(std::vector<int>&)> sortFunctions = {
    selectionSort,
    bubbleSort,
    insertionSort,
    insertionSortOptimized,
    binaryInsertionSort
};

for (int i = 0; i < testCases.size(); ++i) {
    std::cout << "\n 测试用例 " << i + 1 << ": ";
    printArrayDetails(testCases[i], "");

    for (int j = 0; j < algorithmNames.size(); ++j) {
        std::vector<int> arrCopy = copyArray(testCases[i]);
        std::vector<int> expected = copyArray(testCases[i]);
        std::sort(expected.begin(), expected.end()); // 使用系统排序作为基准

        sortFunctions[j](arrCopy);

        bool correct = (arrCopy == expected);
        std::cout << algorithmNames[j] << ": "
               << (correct ? "✓" : "✗") << std::endl;
    }
}

/***
 * 性能测试：比较各种排序算法在不同数据规模下的表现
 */
void performanceTest() {
    std::cout << "\n==== 性能测试 ===" << std::endl;
}

```

```
std::vector<int> sizes = {100, 500, 1000};
std::vector<std::string> algorithmNames = {
    "选择排序",
    "冒泡排序",
    "插入排序",
    "优化插入排序",
    "二分插入排序"
};

for (int size : sizes) {
    std::cout << "\n数组大小: " << size << std::endl;
    std::vector<int> data = generateRandomArray(size);

    for (const std::string& algorithm : algorithmNames) {
        analyzeSortPerformance(data, algorithm);
    }
}

/***
 * 主函数
 */
int main() {
    // 测试排序算法的正确性
    testSortAlgorithms();

    // 性能测试
    performanceTest();

    // 测试经典题目解法
    std::cout << "\n==== 经典题目解法测试 ===" << std::endl;

    // 测试颜色分类
    std::vector<int> colors = {2, 0, 2, 1, 1, 0};
    std::cout << "颜色分类测试: " << std::endl;
    printArrayDetails(colors, "排序前");
    sortColors(colors);
    printArrayDetails(colors, "排序后");

    // 测试移动零
    std::vector<int> zeros = {0, 1, 0, 3, 12};
    std::cout << "移动零测试: " << std::endl;
```

```

printArrayDetails(zeros, "操作前");
moveZeroesOptimized(zeros);
printArrayDetails(zeros, "操作后");

// 测试找第 K 大元素
std::vector<int> kthTest = {3, 2, 1, 5, 6, 4};
int k = 2;
std::cout << "找第" << k << "大元素测试: " << std::endl;
printArrayDetails(kthTest, "原始数组");
std::vector<int> kthCopy = copyArray(kthTest);
int result = findKthLargest(kthCopy, k);
std::cout << "结果: " << result << std::endl;

// 测试额外的题目
testAdditionalProblems();

return 0;
}

// =====
// 更多经典题目的实现 - 涉及各大算法平台的高频题目
// =====

/***
 * LeetCode 347. 前 K 个高频元素
 * 题目链接: 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
 * 示例:
 * 输入: nums = [1,1,1,2,2,3], k = 2
 * 输出: [1,2]
 *
 * 解题思路:
 * 1. 使用哈希表统计每个元素的频率
 * 2. 使用堆（优先队列）或桶排序找到频率最高的 k 个元素
 *
 * 时间复杂度: O(n log k) - 使用最小堆
 * 空间复杂度: O(n) - 哈希表存储频率
 *
 * 最优解: 桶排序, 时间复杂度 O(n), 空间复杂度 O(n)
 *
 * @param nums 输入数组
 * @param k 需要返回的高频元素个数
 */

```

```

* @return 前 k 个高频元素
*/
std::vector<int> topKFrequent(const std::vector<int>& nums, int k) {
    // 防御性编程
    if (nums.empty() || k <= 0 || k > nums.size()) {
        throw std::invalid_argument("Invalid input");
    }

    // 步骤 1: 统计频率
    // 使用 map 记录每个元素的出现次数
    std::map<int, int> freqMap;
    for (int num : nums) {
        freqMap[num]++;
    }

    // 步骤 2: 桶排序 - 按频率分组
    // bucket[i] 存储频率为 i 的所有元素
    // 桶的数量为 nums.size() + 1, 因为频率最大为 nums.size()
    std::vector<std::vector<int>> bucket(nums.size() + 1);
    for (const auto& pair : freqMap) {
        int num = pair.first;
        int freq = pair.second;
        bucket[freq].push_back(num);
    }

    // 步骤 3: 从高频到低频收集结果
    std::vector<int> result;
    // 从最大频率开始向下遍历
    for (int i = bucket.size() - 1; i >= 0 && result.size() < k; i--) {
        if (!bucket[i].empty()) {
            // 将当前频率的所有元素添加到结果中
            for (int num : bucket[i]) {
                result.push_back(num);
                // 当收集到 k 个元素时停止
                if (result.size() == k) break;
            }
        }
    }

    return result;
}

/***

```

* LeetCode 1122. 数组的相对排序

* 题目链接: <https://leetcode.cn/problems/relative-sort-array/>

*

* 题目描述: 给你两个数组, arr1 和 arr2, arr2 中的元素各不相同, arr2 中的每个元素都出现在 arr1 中。

* 对 arr1 中的元素进行排序, 使 arr1 中项的相对顺序和 arr2 中的相对顺序相同。

* 未在 arr2 中出现过的元素需要按照升序放在 arr1 的末尾。

*

* 示例:

* 输入: arr1 = [2, 3, 1, 3, 2, 4, 6, 7, 9, 2, 19], arr2 = [2, 1, 4, 3, 9, 6]

* 输出: [2, 2, 2, 1, 4, 3, 3, 9, 6, 7, 19]

*

* 解题思路:

* 1. 使用计数排序思想, 统计 arr1 中每个元素的出现次数

* 2. 按照 arr2 的顺序填充结果数组

* 3. 将不在 arr2 中的元素排序后放在末尾

*

* 时间复杂度: $O(n \log n)$ - 主要是排序不在 arr2 中的元素

* 空间复杂度: $O(n)$ - 哈希表和结果数组

*

* 最优解: 计数排序, 时间复杂度 $O(n + m)$, 其中 n 是 arr1 的长度, m 是数值范围

*

* @param arr1 第一个数组

* @param arr2 第二个数组, 元素各不相同且都出现在 arr1 中

* @return 按照 arr2 相对顺序排序的 arr1

*/

```
std::vector<int> relativeSortArray(const std::vector<int>& arr1, const std::vector<int>& arr2) {
    // 防御性编程
    if (arr1.empty()) {
        return {};
    }
    if (arr2.empty()) {
        // 如果 arr2 为空, 直接返回排序后的 arr1
        std::vector<int> result = arr1;
        std::sort(result.begin(), result.end());
        return result;
    }

    // 步骤 1: 统计 arr1 中每个元素的频率
    std::map<int, int> countMap;
    for (int num : arr1) {
        countMap[num]++;
    }
```

```

// 步骤 2: 按照 arr2 的顺序填充结果
std::vector<int> result;
// 按照 arr2 中元素的顺序处理
for (int num : arr2) {
    if (countMap.find(num) != countMap.end()) {
        // 将 countMap[num] 个 num 添加到结果中
        int count = countMap[num];
        for (int i = 0; i < count; i++) {
            result.push_back(num);
        }
        // 从 countMap 中删除已处理的元素
        countMap.erase(num);
    }
}

// 步骤 3: 将不在 arr2 中的元素排序后放在末尾
std::vector<int> remaining;
// 收集剩余的元素
for (const auto& pair : countMap) {
    int num = pair.first;
    int count = pair.second;
    for (int i = 0; i < count; i++) {
        remaining.push_back(num);
    }
}
// 对剩余元素进行排序
std::sort(remaining.begin(), remaining.end());
// 将排序后的剩余元素添加到结果末尾
for (int num : remaining) {
    result.push_back(num);
}

return result;
}

/***
 * 剑指 Offer 40. 最小的 k 个数
 * 题目链接: 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
 * 示例:
 * 输入: arr = [3,2,1], k = 2
 */

```

```

* 输出: [1, 2] 或者 [2, 1]
*
* 解题思路:
* 方法 1: 排序后取前 k 个 - 时间复杂度 O(n log n)
* 方法 2: 堆 (最大堆) - 时间复杂度 O(n log k)
* 方法 3: 快速选择算法 - 平均时间复杂度 O(n)
*
* 最优解: 快速选择算法
* 时间复杂度: O(n) - 平均情况
* 空间复杂度: O(log n) - 递归栈空间
*
* @param arr 输入数组
* @param k 需要返回的最小元素个数
* @return 最小的 k 个数
*/
std::vector<int> getLeastNumbers(std::vector<int>& arr, int k) {
    // 防御性编程
    if (arr.empty() || k <= 0 || k > arr.size()) {
        return {};
    }

    // 使用快速选择找到第 k 小的元素
    // 由于是找最小的 k 个数, 不需要完全排序
    quickSelect(arr, 0, arr.size() - 1, k - 1);

    // 前 k 个元素就是结果
    std::vector<int> result(arr.begin(), arr.begin() + k);
    // 对结果进行排序以满足题目要求
    std::sort(result.begin(), result.end());

    return result;
}

/***
* LeetCode 506. 相对名次
* 题目链接: https://leetcode.cn/problems/relative-ranks/
*
* 题目描述: 给你一个长度为 n 的整数数组 score , 其中 score[i] 表示第 i 位运动员在比赛中的得分。
* 所有得分都互不相同。运动员将根据得分决定名次, 其中名次第 1 的运动员得分最高, 名次第 2 的运动员得分第 2 高, 依此类推。
*
* 示例:
* 输入: score = [5, 4, 3, 2, 1]

```

```

* 输出: ["Gold Medal", "Silver Medal", "Bronze Medal", "4", "5"]
*
* 解题思路:
* 1. 创建索引数组, 按分数排序
* 2. 根据排序后的索引分配名次
*
* 时间复杂度: O(n log n) - 排序的时间复杂度
* 空间复杂度: O(n) - 存储索引和结果
*
* @param score 分数数组
* @return 每个运动员的名次
*/
std::vector<std::string> findRelativeRanks(const std::vector<int>& score) {
    // 防御性编程
    if (score.empty()) {
        return {};
    }

    int n = score.size();
    // 创建索引数组, 用于排序后找到原始位置
    // indices[i] 表示原始数组中第 i 个位置的索引
    std::vector<int> indices(n);
    for (int i = 0; i < n; i++) {
        indices[i] = i;
    }

    // 按分数从高到低排序索引
    // 使用 lambda 表达式定义比较函数
    // 按 score[indices[i]] 的值进行降序排列
    std::sort(indices.begin(), indices.end(), [&score](int a, int b) {
        return score[a] > score[b];
    });

    // 根据排序后的索引分配名次
    std::vector<std::string> result(n);
    for (int i = 0; i < n; i++) {
        int idx = indices[i];
        // 根据排名分配奖牌或名次
        if (i == 0) {
            result[idx] = "Gold Medal";
        } else if (i == 1) {
            result[idx] = "Silver Medal";
        } else if (i == 2) {
            result[idx] = "Bronze Medal";
        }
    }
}

```

```

        result[idx] = "Bronze Medal";
    } else {
        // 第 4 名及以后用数字表示
        result[idx] = std::to_string(i + 1);
    }
}

return result;
}

/***
 * LeetCode 922. 按奇偶排序数组 II
 * 题目链接: https://leetcode.cn/problems/sort-array-by-parity-ii/
 *
 * 题目描述: 给定一个非负整数数组 nums，nums 中一半整数是奇数，一半整数是偶数。
 * 对数组进行排序，以便当 nums[i] 为奇数时，i 也是奇数；当 nums[i] 为偶数时，i 也是偶数。
 *
 * 示例:
 * 输入: nums = [4, 2, 5, 7]
 * 输出: [4, 5, 2, 7]
 *
 * 解题思路:
 * 方法 1: 使用两个数组分别存储奇数和偶数，然后按要求放回
 * 方法 2: 双指针原地交换
 *
 * 最优解: 双指针原地交换
 * 时间复杂度: O(n)
 * 空间复杂度: O(1)
 *
 * @param nums 输入数组
 * @return 满足条件的数组
 */
std::vector<int> sortArrayByParityII(std::vector<int>& nums) {
    // 防御性编程
    if (nums.empty()) {
        return nums;
    }

    int n = nums.size();
    int evenIdx = 0; // 偶数位置指针，用于寻找应该放偶数但放了奇数的位置
    int oddIdx = 1; // 奇数位置指针，用于寻找应该放奇数但放了偶数的位置

    // 当两个指针都在有效范围内时继续循环

```

```

while (evenIdx < n && oddIdx < n) {
    // 找到偶数位置上的奇数
    // 在偶数位置 (0, 2, 4...) 上寻找奇数
    while (evenIdx < n && nums[evenIdx] % 2 == 0) {
        evenIdx += 2;
    }
    // 找到奇数位置上的偶数
    // 在奇数位置 (1, 3, 5...) 上寻找偶数
    while (oddIdx < n && nums[oddIdx] % 2 == 1) {
        oddIdx += 2;
    }
    // 交换
    // 如果找到了两个错误位置，进行交换
    if (evenIdx < n && oddIdx < n) {
        swapVec(nums, evenIdx, oddIdx);
        // 交换后继续寻找下一个错误位置
        evenIdx += 2;
        oddIdx += 2;
    }
}

return nums;
}

/***
 * 测试额外的题目解法
 */
void testAdditionalProblems() {
    std::cout << "\n==== 额外算法题目测试 ===\n" << std::endl;

    // 测试 topKFrequent
    std::cout << "--- LeetCode 347. 前 K 个高频元素 ---" << std::endl;
    std::vector<int> freqTest = {1, 1, 1, 2, 2, 3};
    std::vector<int> freqResult = topKFrequent(freqTest, 2);
    std::cout << "输入: [1,1,1,2,2,3], k=2" << std::endl;
    std::cout << "输出: [";
    for (size_t i = 0; i < freqResult.size(); i++) {
        std::cout << freqResult[i];
        if (i < freqResult.size() - 1) std::cout << ",";
    }
    std::cout << "]" << std::endl << std::endl;

    // 测试 relativeSortArray
}

```

```

std::cout << "--- LeetCode 1122. 数组的相对排序 ---" << std::endl;
std::vector<int> arr1 = {2, 3, 1, 3, 2, 4, 6, 7, 9, 2, 19};
std::vector<int> arr2 = {2, 1, 4, 3, 9, 6};
std::vector<int> relativeResult = relativeSortArray(arr1, arr2);
std::cout << "输入: arr1=[2,3,1,3,2,4,6,7,9,2,19], arr2=[2,1,4,3,9,6]" << std::endl;
std::cout << "输出: [";
for (size_t i = 0; i < relativeResult.size(); i++) {
    std::cout << relativeResult[i];
    if (i < relativeResult.size() - 1) std::cout << ",";
}
std::cout << "]" << std::endl << std::endl;

// 测试 getLeastNumbers
std::cout << "--- 剑指 Offer 40. 最小的 k 个数 ---" << std::endl;
std::vector<int> leastTest = {3, 2, 1, 5, 6, 4};
std::vector<int> leastResult = getLeastNumbers(leastTest, 2);
std::cout << "输入: [3,2,1,5,6,4], k=2" << std::endl;
std::cout << "输出: [";
for (size_t i = 0; i < leastResult.size(); i++) {
    std::cout << leastResult[i];
    if (i < leastResult.size() - 1) std::cout << ",";
}
std::cout << "]" << std::endl << std::endl;

// 测试 findRelativeRanks
std::cout << "--- LeetCode 506. 相对名次 ---" << std::endl;
std::vector<int> scoreTest = {5, 4, 3, 2, 1};
std::vector<std::string> rankResult = findRelativeRanks(scoreTest);
std::cout << "输入: [5,4,3,2,1]" << std::endl;
std::cout << "输出: [";
for (size_t i = 0; i < rankResult.size(); i++) {
    std::cout << "\"" << rankResult[i] << "\"";
    if (i < rankResult.size() - 1) std::cout << ",";
}
std::cout << "]" << std::endl << std::endl;

// 测试 sortArrayByParityII
std::cout << "--- LeetCode 922. 按奇偶排序数组 II ---" << std::endl;
std::vector<int> parityTest = {4, 2, 5, 7};
std::vector<int> parityResult = sortArrayByParityII(parityTest);
std::cout << "输入: [4,2,5,7]" << std::endl;
std::cout << "输出: [";
for (size_t i = 0; i < parityResult.size(); i++) {

```

```
    std::cout << parityResult[i];
    if (i < parityResult.size() - 1) std::cout << ",";
}
std::cout << "]" << std::endl << std::endl;
}
```

文件: Validator.java

```
import java.util.Arrays;

/**
 * 选择排序、冒泡排序、插入排序的验证与扩展练习
 *
 * 选择排序(Selection Sort):
 * - 工作原理: 每次从未排序的部分中找到最小元素, 放到已排序部分的末尾
 * - 时间复杂度:  $O(n^2)$  - 最好、平均、最坏情况都相同
 * - 空间复杂度:  $O(1)$  - 原地排序
 * - 稳定性: 不稳定
 * - 适用场景: 数据量小且对稳定性无要求
 *
 * 冒泡排序(Bubble Sort):
 * - 工作原理: 相邻元素两两比较, 如果顺序错误就交换, 每轮将最大元素"冒泡"到末尾
 * - 时间复杂度:  $O(n^2)$  - 最坏和平均情况,  $O(n)$  - 最好情况(已排序)
 * - 空间复杂度:  $O(1)$  - 原地排序
 * - 稳定性: 稳定
 * - 适用场景: 数据量小且要求稳定性
 *
 * 插入排序(Insertion Sort):
 * - 工作原理: 将未排序元素插入到已排序序列的适当位置
 * - 时间复杂度:  $O(n^2)$  - 最坏情况,  $O(n)$  - 最好情况(已排序)
 * - 空间复杂度:  $O(1)$  - 原地排序
 * - 稳定性: 稳定
 * - 适用场景: 小规模数据或基本有序的数据
 */

public class Validator {

    public static void main(String[] args) {
        System.out.println("== 基础排序算法验证器启动 ==");
        System.out.println("作者: Algorithm Journey");
        System.out.println("版本: 1.1");
        System.out.println("日期: 2025-10-18");
    }
}
```

```
System.out.println();

// 第一阶段：基础算法正确性验证
System.out.println("第一阶段：基础排序算法正确性验证");
validateBasicAlgorithms();

// 第二阶段：详细测试用例演示
System.out.println("\n第二阶段：详细测试用例演示");
testSortAlgorithms();

// 第三阶段：性能测试分析
System.out.println("\n第三阶段：性能测试分析");
performanceTest();

// 第四阶段：经典题目解法测试
System.out.println("\n第四阶段：经典题目解法测试");
testAdditionalProblems();

// 第五阶段：工程化组件测试
System.out.println("\n第五阶段：工程化组件测试");
testEngineeringComponents();

System.out.println("\n==== 所有测试完成 ====");
}

/***
 * 基础排序算法正确性验证
 * 通过 5000 次随机测试验证三种排序算法的正确性
 */
public static void validateBasicAlgorithms() {
    int N = 200;          // 随机数组最大长度
    int V = 1000;         // 随机数组值范围
    int testTimes = 5000; // 测试次数

    System.out.println("开始 " + testTimes + " 次随机测试...");
    int errorCount = 0;

    for (int i = 0; i < testTimes; i++) {
        int n = (int) (Math.random() * N);
        int[] arr = randomArray(n, V);
        int[] arr1 = copyArray(arr);
        int[] arr2 = copyArray(arr);
        int[] arr3 = copyArray(arr);

        if (!arr1.equals(arr2) || !arr2.equals(arr3) || !arr1.equals(arr3)) {
            errorCount++;
        }
    }

    if (errorCount == 0) {
        System.out.println("所有测试通过，算法正确性验证成功！");
    } else {
        System.out.println("检测到 " + errorCount + " 次错误，可能存在算法问题。");
    }
}
```

```

selectionSort(arr1);
bubbleSort(arr2);
insertionSort(arr3);

if (!sameArray(arr1, arr2) || !sameArray(arr1, arr3)) {
    errorCount++;
}

// 显示进度
if ((i + 1) % 1000 == 0) {
    System.out.println("已完成 " + (i + 1) + " 次测试");
}
}

System.out.println("测试完成, 错误次数: " + errorCount + "/" + testTimes);
if (errorCount == 0) {
    System.out.println("✓ 所有基础排序算法测试通过");
} else {
    System.out.println("✗ 发现 " + errorCount + " 个错误");
}
}

/***
 * 测试函数: 验证三种排序算法的正确性
 */
public static void testSortAlgorithms() {
    System.out.println("\n== 选择排序、冒泡排序、插入排序测试 ==");

    int[][] testCases = {
        {}, // 空数组
        {1}, // 单元素
        {1, 2, 3}, // 已排序
        {3, 2, 1}, // 逆序
        {1, 1, 1}, // 全相同
        {5, 2, 8, 1, 9}, // 普通情况
        {3, 1, 4, 1, 5, 9, 2, 6} // 重复元素
    };

    String[] algorithms = {"选择排序", "冒泡排序", "插入排序"};

    for (int i = 0; i < testCases.length; i++) {
        System.out.println("\n测试用例 " + (i + 1) + ": " + Arrays.toString(testCases[i]));
    }
}

```

```
for (int j = 0; j < algorithms.length; j++) {
    int[] arr = testCases[i].clone();
    int[] expected = testCases[i].clone();
    Arrays.sort(expected); // 使用系统排序作为基准

    switch (j) {
        case 0:
            selectionSort(arr);
            break;
        case 1:
            bubbleSort(arr);
            break;
        case 2:
            insertionSort(arr);
            break;
    }

    boolean correct = Arrays.equals(arr, expected);
    System.out.printf("%s: %s - %s%n",
        algorithms[j],
        Arrays.toString(arr),
        correct ? "✓" : "✗"
    );
}

}

}

/***
 * 性能测试：比较三种排序算法在不同数据规模下的表现
 */
public static void performanceTest() {
    System.out.println("\n==== 性能测试 ====");

    int[] sizes = {100, 500, 1000};
    String[] algorithms = {"选择排序", "冒泡排序", "插入排序"};

    for (int size : sizes) {
        System.out.println("\n数组大小: " + size);
        int[] data = generateRandomArray(size);

        for (int j = 0; j < algorithms.length; j++) {
            int[] testData = data.clone();
```

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

switch (j) {
    case 0:
        selectionSort(testData);
        break;
    case 1:
        bubbleSort(testData);
        break;
    case 2:
        insertionSort(testData);
        break;
}

long endTime = System.nanoTime();
double duration = (endTime - startTime) / 1e6; // 转换为毫秒

System.out.printf("%s: %.2f ms%n", algorithms[j], duration);
}

}

}

/***
 * 测试新增的题目解法
 */
public static void testAdditionalProblems() {
    System.out.println("== 经典题目解法测试 ==");

    // 测试 LeetCode 75. 颜色分类
    System.out.println("\n1. LeetCode 75. 颜色分类");
    int[] colors = {2, 0, 2, 1, 1, 0};
    System.out.println("输入: " + Arrays.toString(colors));
    sortColors(colors);
    System.out.println("输出: " + Arrays.toString(colors));

    // 测试 LeetCode 283. 移动零
    System.out.println("\n2. LeetCode 283. 移动零");
    int[] nums = {0, 1, 0, 3, 12};
    System.out.println("输入: " + Arrays.toString(nums));
    moveZeroes(nums);
    System.out.println("输出: " + Arrays.toString(nums));

    // 测试 LeetCode 215. 数组中的第 K 个最大元素
}
```

```

System.out.println("\n3. LeetCode 215. 数组中的第 K 个最大元素");
int[] arr = {3, 2, 1, 5, 6, 4};
int k = 2;
System.out.println("输入: " + Arrays.toString(arr) + ", k = " + k);
int result = findKthLargest(arr, k);
System.out.println("输出: " + result);

// 测试牛客网 - 最小的 K 个数
System.out.println("\n4. 牛客网 - 最小的 K 个数");
int[] numbers = {4, 5, 1, 6, 2, 7, 3, 8};
int k2 = 4;
System.out.println("输入: " + Arrays.toString(numbers) + ", k = " + k2);
int[] smallestK = getLeastNumbers(numbers, k2);
System.out.println("输出: " + Arrays.toString(smallestK));
}

/**
 * 工程化组件测试
 */
public static void testEngineeringComponents() {
    System.out.println("== 工程化组件测试 ==");

    // 测试异常处理
    System.out.println("\n1. 异常处理测试");
    try {
        selectionSort(null);
        System.out.println("空数组处理: ✓");
    } catch (Exception e) {
        System.out.println("空数组处理: ✗ - " + e.getMessage());
    }

    // 测试边界条件
    System.out.println("\n2. 边界条件测试");
    int[] empty = {};
    int[] single = {1};
    selectionSort(empty);
    selectionSort(single);
    System.out.println("空数组: " + Arrays.toString(empty));
    System.out.println("单元素: " + Arrays.toString(single));

    // 测试性能分析
    System.out.println("\n3. 性能分析测试");
    analyzePerformance();
}

```

```
}

/**
 * 性能分析测试
 */
private static void analyzePerformance() {
    System.out.println("性能分析:");

    int[] sizes = {100, 500, 1000};
    for (int size : sizes) {
        System.out.println("\n数组大小: " + size);

        int[] data = generateRandomArray(size);
        int[] dataCopy = data.clone();

        long start = System.nanoTime();
        selectionSort(dataCopy);
        long end = System.nanoTime();
        double time = (end - start) / 1e6;

        System.out.printf("选择排序耗时: %.3f ms%n", time);
        System.out.println("排序正确性: " + isSorted(dataCopy));
    }
}

/**
 * 检查数组是否已排序
 */
private static boolean isSorted(int[] arr) {
    for (int i = 1; i < arr.length; i++) {
        if (arr[i] < arr[i - 1]) {
            return false;
        }
    }
    return true;
}

/**
 * 生成随机测试数组
 * @param size 数组大小
 * @return 随机数组
 */
private static int[] generateRandomArray(int size) {
```

```
int[] arr = new int[size];
for (int i = 0; i < size; i++) {
    arr[i] = (int) (Math.random() * size * 10);
}
return arr;
}
```

```
/**
 * 得到一个随机数组，长度是 n，值在 1~v 之间
 * @param n 数组长度
 * @param v 数值范围上限
 * @return 随机数组
 */
```

```
public static int[] randomArray(int n, int v) {
    int[] arr = new int[n];
    for (int i = 0; i < n; i++) {
        arr[i] = (int) (Math.random() * v) + 1;
    }
    return arr;
}
```

```
/**
 * 复制数组
 * @param arr 原数组
 * @return 复制的数组
 */
```

```
public static int[] copyArray(int[] arr) {
    if (arr == null) {
        return null;
    }
    int n = arr.length;
    int[] ans = new int[n];
    for (int i = 0; i < n; i++) {
        ans[i] = arr[i];
    }
    return ans;
}
```

```
/**
 * 比较两个数组是否相同
 * @param arr1 数组 1
 * @param arr2 数组 2
 * @return 是否相同
 */
```

```

*/
public static boolean sameArray(int[] arr1, int[] arr2) {
    if (arr1 == null && arr2 == null) {
        return true;
    }
    if (arr1 == null || arr2 == null) {
        return false;
    }
    int n = arr1.length;
    if (n != arr2.length) {
        return false;
    }
    for (int i = 0; i < n; i++) {
        if (arr1[i] != arr2[i]) {
            return false;
        }
    }
    return true;
}

/**
 * 数组中交换 i 和 j 位置的数
 * @param arr 数组
 * @param i 位置 i
 * @param j 位置 j
 */
public static void swap(int[] arr, int i, int j) {
    int tmp = arr[i];
    arr[i] = arr[j];
    arr[j] = tmp;
}

/**
 * 选择排序 - Selection Sort
 * 时间复杂度: O(n2) - 无论什么情况都需要进行  $n(n-1)/2$  次比较
 * 空间复杂度: O(1) - 只使用了常数级别的额外空间
 * 稳定性: 不稳定 - 相等元素的相对位置可能改变
 *
 * 算法思路:
 * 1. 在未排序序列中找到最小元素
 * 2. 将其与未排序序列的第一个元素交换位置
 * 3. 重复步骤 1-2, 直到所有元素排序完成
 *

```

- * 优点:
 - * - 实现简单
 - * - 原地排序, 空间复杂度低
 - * - 交换次数少, 最多进行 $n-1$ 次交换
- *
- * 缺点:
 - * - 时间复杂度高, 不适合大数据量
 - * - 不稳定
 - * - 无法利用数据的有序性优化
- *
- * 适用场景:
 - * - 数据量小的情况
 - * - 对内存使用要求严格的场景
 - * - 不要求稳定性的场景
- *

```
* @param arr 待排序数组
*/
public static void selectionSort(int[] arr) {
    // 边界检查: 空数组或单元素数组无需排序
    if (arr == null || arr.length < 2) {
        return;
    }

    // 外层循环控制排序的轮数, 需要进行  $n-1$  轮
    // 每轮都会确定一个元素的最终位置 (当前未排序部分的最小元素)
    for (int i = 0; i < arr.length - 1; i++) {
        // 假设当前位置 i 就是未排序部分的最小值位置
        // 从当前位置开始, 在未排序部分 [i, n-1] 中寻找真正的最小值
        int minIndex = i;

        // 内层循环在未排序部分 [i+1, n-1] 中寻找真正的最小值
        // j 从 i+1 开始, 因为位置 i 已经是当前假设的最小值位置
        for (int j = i + 1; j < arr.length; j++) {
            // 如果找到更小的元素, 更新最小值索引
            // 这里使用 < 而不是 <= 是为了保持算法的不稳定性
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }

        // 如果最小值不在当前位置, 则交换
        // 这样可以减少不必要的交换操作 (当 minIndex == i 时不需要交换)
        if (minIndex != i) {
```

```
        swap(arr, i, minIndex);
    }
}

/**
 * 冒泡排序 - Bubble Sort
 * 时间复杂度: O(n2) - 最坏和平均情况, O(n) - 最好情况(已排序)
 * 空间复杂度: O(1) - 只使用了常数级别的额外空间
 * 稳定性: 稳定 - 相等元素不会交换位置
 *
 * 算法思路:
 * 1. 比较相邻的两个元素, 如果前面的比后面的大就交换
 * 2. 每一轮都会将当前未排序部分的最大元素"冒泡"到末尾
 * 3. 重复步骤 1-2, 直到所有元素排序完成
 *
 * 优点:
 * - 实现简单, 容易理解
 * - 稳定排序
 * - 原地排序
 * - 能够检测数组是否已经有序
 *
 * 缺点:
 * - 时间复杂度高, 不适合大数据量
 * - 元素交换次数多
 *
 * 优化:
 * - 设置标志位, 如果某一轮没有发生交换, 说明数组已经有序, 可以提前结束
 *
 * 适用场景:
 * - 数据量小的情况
 * - 要求稳定性的场景
 * - 教学演示
 *
 * @param arr 待排序数组
 */
public static void bubbleSort(int[] arr) {
    // 边界检查: 空数组或单元素数组无需排序
    if (arr == null || arr.length < 2) {
        return;
    }

    // 外层循环控制排序的轮数, 最多需要进行 n-1 轮
```

```

// 每轮都会确定一个元素的最终位置（当前未排序部分的最大元素）
// end 表示每轮比较的上界，随着排序的进行逐渐减小
for (int end = arr.length - 1; end > 0; end--) {
    // 优化标志：记录本轮是否发生交换
    // 如果一轮比较中没有发生任何交换，说明数组已经有序
    boolean swapped = false;

    // 内层循环进行相邻元素的比较和交换
    // 每轮比较范围逐渐缩小，因为末尾的元素已经有序
    // i 从 0 开始到 end-1，比较 arr[i] 和 arr[i+1]
    for (int i = 0; i < end; i++) {
        // 如果前面的元素比后面的大，则交换
        // 这会将较大的元素逐步向右移动（“冒泡”）
        if (arr[i] > arr[i + 1]) {
            swap(arr, i, i + 1);
            swapped = true;
        }
    }

    // 如果本轮没有发生交换，说明数组已经有序，可以提前结束
    // 这是冒泡排序的一个重要优化，可以将最好情况的时间复杂度降到 O(n)
    if (!swapped) {
        break;
    }
}

/**
 * 插入排序 - Insertion Sort
 * 时间复杂度: O(n2) - 最坏情况, O(n) - 最好情况(已排序)
 * 空间复杂度: O(1) - 只使用了常数级别的额外空间
 * 稳定性: 稳定 - 相等元素不会交换位置
 *
 * 算法思路:
 * 1. 将数组分为已排序和未排序两部分，初始时已排序部分只有第一个元素
 * 2. 依次取出未排序部分的元素，在已排序部分找到合适的插入位置
 * 3. 将元素插入到正确位置，重复步骤 2-3 直到所有元素排序完成
 *
 * 优点:
 * - 实现简单
 * - 稳定排序
 * - 原地排序
 * - 对于小规模或基本有序的数据效率很高

```

```

* - 在线算法：可以在接收数据的同时进行排序
*
* 缺点：
* - 时间复杂度高，不适合大数据量
* - 对于逆序数据效率较低
*
* 适用场景：
* - 小规模数据排序
* - 基本有序的数据
* - 在线数据排序
* - 作为高级排序算法的子过程（如快速排序的小数组优化）
*
* @param arr 待排序数组
*/

```

```

public static void insertionSort(int[] arr) {
    // 边界检查：空数组或单元素数组无需排序
    if (arr == null || arr.length < 2) {
        return;
    }

    // 从第二个元素开始，因为第一个元素可以看作已排序
    // i 表示当前要插入的元素位置
    for (int i = 1; i < arr.length; i++) {
        // 从当前位置向前比较，找到合适的插入位置
        // 当前元素为 arr[i]，需要在 arr[0...i-1] 中找到插入位置
        // j 从 i-1 开始向前遍历已排序部分
        for (int j = i - 1; j >= 0 && arr[j] > arr[j + 1]; j--) {
            // 如果前一个元素大于当前元素，则交换
            // 这实际上是在将当前元素向前移动
            swap(arr, j, j + 1);
        }
    }
}

/**
 * LeetCode 75. 颜色分类 - 三指针法（最优解）
 * 时间复杂度：O(n)
 * 空间复杂度：O(1)
 * 稳定性：不稳定
 *
 * 算法思想：
 * 使用三个指针将数组分为三个区域：
 * - [0, p0)：已排序的 0 区域

```

```

* - [p0, curr): 已排序的 0 区域
* - [p2, n-1]: 已排序的 2 区域
* - [curr, p2): 待处理的区域
*
* 算法步骤:
* 1. 初始化 p0=0 (0 的右边界), curr=0 (当前处理位置), p2=n-1 (2 的左边界)
* 2. 当 curr <= p2 时循环:
*   a. 如果 nums[curr] == 0, 交换 nums[curr] 和 nums[p0], p0++, curr++
*   b. 如果 nums[curr] == 1, curr++
*   c. 如果 nums[curr] == 2, 交换 nums[curr] 和 nums[p2], p2-- (curr 不变)
*
* 为什么是最优解:
* - 相比基础排序算法的 O(n2) 时间复杂度, 三指针法只需要 O(n) 时间
* - 空间复杂度为 O(1), 不需要额外空间
* - 只需要一次遍历, 效率高
* - 直接利用了问题特性 (只有 0、1、2 三种元素)
*
* @param nums 待排序数组, 元素只能是 0、1、2
*/
public static void sortColors(int[] nums) {
    // 防御性编程: 检查输入合法性
    if (nums == null || nums.length < 2) return;

    int n = nums.length;
    int p0 = 0;          // 0 的右边界 (初始为 0)
    int curr = 0;        // 当前遍历的位置
    int p2 = n - 1;      // 2 的左边界 (初始为数组末尾)

    // 遍历数组直到 curr 超过 p2
    // 循环条件是 curr <= p2, 因为 p2 位置的元素尚未处理
    while (curr <= p2) {
        if (nums[curr] == 0) {
            // 当前元素为 0, 放到 0 的区域
            // 交换后, p0 位置的元素一定是 0, curr 位置的元素是原来 p0 位置的元素 (0、1 或 2)
            // 由于 p0 <= curr, p0 位置的元素已经被处理过, 所以可以安全地递增 curr
            swap(nums, curr, p0);
            curr++;
            p0++;
        } else if (nums[curr] == 2) {
            // 当前元素为 2, 放到 2 的区域
            // 交换后, p2 位置的元素是原来 curr 位置的元素 (未知), 所以 curr 不能递增
            swap(nums, curr, p2);
            p2--;
        }
    }
}

```

```

        } else {
            // 当前元素为 1，保持不动，继续处理下一个元素
            // 1 的区域自然扩展
            curr++;
        }
    }
}

/***
 * LeetCode 283. 移动零 - 双指针法（最优解）
 * 时间复杂度: O(n)
 * 空间复杂度: O(1)
 * 稳定性: 稳定
 *
 * 算法思想:
 * 使用两个指针，一个指向当前应该放置非零元素的位置，另一个遍历整个数组
 * 当遇到非零元素时，将其移动到第一个指针指向的位置，然后第一个指针前进
 *
 * 算法步骤:
 * 1. 初始化一个指针 nonZeroIndex=0，表示下一个非零元素应该放置的位置
 * 2. 遍历数组，对于每个元素:
 *     a. 如果元素非零，将其移动到 nonZeroIndex 位置，然后 nonZeroIndex++
 * 3. 遍历结束后，将 nonZeroIndex 到数组末尾的所有元素设置为 0
 *
 * @param nums 待处理数组
 */
public static void moveZeroes(int[] nums) {
    // 防御性编程：检查输入合法性
    if (nums == null || nums.length < 2) return;

    int nonZeroIndex = 0;

    // 第一步：将所有非零元素移动到数组前面
    // 遍历整个数组
    for (int i = 0; i < nums.length; i++) {
        if (nums[i] != 0) {
            // 将非零元素移动到 nonZeroIndex 位置
            nums[nonZeroIndex] = nums[i];
            nonZeroIndex++;
        }
    }

    // 第二步：将剩余位置填充为 0
}

```

```

// 将 nonZeroIndex 到数组末尾的所有位置设置为 0
for (int i = nonZeroIndex; i < nums.length; i++) {
    nums[i] = 0;
}

}

/***
 * LeetCode 215. 数组中的第 K 个最大元素 - 快速选择算法（最优解）
 * 时间复杂度：平均 O(n)，最坏 O(n2)
 * 空间复杂度：O(1)
 *
 * 算法思想：
 * 基于快速排序的分区思想，每次分区后只递归处理包含第 k 大元素的那一半
 * 这样可以避免对整个数组进行排序
 *
 * 算法步骤：
 * 1. 选择一个基准元素，将数组分为两部分：大于基准的和小于基准的
 * 2. 如果基准元素的位置正好是第 k 大的位置，返回该元素
 * 3. 否则，递归处理包含第 k 大元素的那一半
 *
 * @param nums 数组
 * @param k 第 k 大元素（从 1 开始计数）
 * @return 第 k 大元素的值
 */
public static int findKthLargest(int[] nums, int k) {
    // 防御性编程：检查输入合法性
    if (nums == null || nums.length == 0 || k <= 0 || k > nums.length) {
        throw new IllegalArgumentException("Invalid input");
    }

    // 第 k 大元素在排序后的数组中的索引是 nums.length - k
    // 例如：数组[1, 2, 3, 4, 5]中第 2 大的元素是 4，其索引为 5-2=3
    return quickSelect(nums, 0, nums.length - 1, nums.length - k);
}

/***
 * 快速选择算法的核心实现
 *
 * @param nums 数组
 * @param left 左边界
 * @param right 右边界
 * @param k 目标索引（第 k 小的元素）
 * @return 第 k 小的元素值
 */

```

```
/*
private static int quickSelect(int[] nums, int left, int right, int k) {
    // 分区操作，返回基准元素的最终位置
    // pivotIndex 是基准元素在数组中的最终位置
    int pivotIndex = partition(nums, left, right);

    // 如果基准元素的位置正好是 k，返回该元素
    if (k == pivotIndex) {
        return nums[k];
    } else if (k < pivotIndex) {
        // 如果目标索引小于基准位置，递归处理左半部分
        return quickSelect(nums, left, pivotIndex - 1, k);
    } else {
        // 如果目标索引大于基准位置，递归处理右半部分
        return quickSelect(nums, pivotIndex + 1, right, k);
    }
}
```

```
/**
 * 快速排序的分区操作
 *
 * @param nums 数组
 * @param left 左边界
 * @param right 右边界
 * @return 基准元素的最终位置
 */
private static int partition(int[] nums, int left, int right) {
    // 选择最右边的元素作为基准
    // 这是一种简单的选择策略，也可以使用随机选择来避免最坏情况
    int pivot = nums[right];
    // i 表示小于基准元素的区域的边界
    // 初始时小于基准的区域为空，所以 i = left - 1
    int i = left;

    // 遍历[left, right-1]范围内的元素
    for (int j = left; j < right; j++) {
        // 如果当前元素小于基准元素，将其交换到小于区域
        if (nums[j] <= pivot) {
            // 扩展小于基准的区域
            swap(nums, i, j);
            i++;
        }
    }
}
```

```

        // 将基准元素放到正确的位置
        // 此时 i 是基准元素应该放置的位置
        swap(nums, i, right);
        return i;
    }

/**
 * 牛客网 - 最小的 K 个数
 * 时间复杂度: 平均  $O(n)$ , 最坏  $O(n^2)$ 
 * 空间复杂度:  $O(1)$ 
 *
 * 解题思路:
 * 方法 1: 排序后取前 k 个 - 时间复杂度  $O(n \log n)$ 
 * 方法 2: 堆 (最大堆) - 时间复杂度  $O(n \log k)$ 
 * 方法 3: 快速选择算法 - 平均时间复杂度  $O(n)$ 
 *
 * 最优解: 快速选择算法
 * 时间复杂度:  $O(n)$  - 平均情况
 * 空间复杂度:  $O(\log n)$  - 递归栈空间
 *
 * @param arr 输入数组
 * @param k 需要返回的最小元素个数
 * @return 最小的 k 个数
 */
public static int[] getLeastNumbers(int[] arr, int k) {
    // 防御性编程
    if (arr == null || arr.length == 0 || k <= 0) {
        return new int[0];
    }

    if (k >= arr.length) {
        return arr.clone();
    }

    // 使用快速选择找到第 k 小的元素
    // 由于是找最小的 k 个数, 不需要完全排序
    quickSelectForKSmallest(arr, 0, arr.length - 1, k - 1);

    // 前 k 个元素就是结果
    int[] result = new int[k];
    System.arraycopy(arr, 0, result, 0, k);
    // 对结果进行排序以满足题目要求
}

```

```

        Arrays.sort(result);
        return result;
    }

    /**
     * 快速选择算法的变体，用于找到最小的 k 个数
     *
     * @param arr 数组
     * @param left 左边界
     * @param right 右边界
     * @param k 目标索引
     */
    private static void quickSelectForKSmallest(int[] arr, int left, int right, int k) {
        if (left >= right) return;

        int pivotIndex = partition(arr, left, right);

        if (pivotIndex == k) {
            return;
        } else if (pivotIndex > k) {
            // 如果基准位置大于目标索引，递归处理左半部分
            quickSelectForKSmallest(arr, left, pivotIndex - 1, k);
        } else {
            // 如果基准位置小于目标索引，递归处理右半部分
            quickSelectForKSmallest(arr, pivotIndex + 1, right, k);
        }
    }
}

```

文件: Validator.py

```

#!/usr/bin/env python3
# -*- coding: utf-8 -*-

```

"""

选择排序、冒泡排序、插入排序的验证与扩展练习

选择排序(Selection Sort):

- 工作原理: 每次从未排序的部分中找到最小元素，放到已排序部分的末尾
- 时间复杂度: $O(n^2)$ - 最好、平均、最坏情况都相同
- 空间复杂度: $O(1)$ - 原地排序

- 稳定性: 不稳定
- 适用场景: 数据量小且对稳定性无要求

冒泡排序(Bubble Sort):

- 工作原理: 相邻元素两两比较, 如果顺序错误就交换, 每轮将最大元素“冒泡”到末尾
- 时间复杂度: $O(n^2)$ - 最坏和平均情况, $O(n)$ - 最好情况(已排序)
- 空间复杂度: $O(1)$ - 原地排序
- 稳定性: 稳定
- 适用场景: 数据量小且要求稳定性

插入排序(Insertion Sort):

- 工作原理: 将未排序元素插入到已排序序列的适当位置
- 时间复杂度: $O(n^2)$ - 最坏情况, $O(n)$ - 最好情况(已排序)
- 空间复杂度: $O(1)$ - 原地排序
- 稳定性: 稳定
- 适用场景: 小规模数据或基本有序的数据

相关题目:

1. LintCode 463. Sort Integers - <https://www.lintcode.com/problem/sort-integers/>

题目描述: 给定一个整数数组, 使用选择排序、冒泡排序或插入排序等 $O(n^2)$ 算法将其按升序排序

示例:

输入: [3, 2, 1, 4, 5]

输出: [1, 2, 3, 4, 5]

2. LeetCode 912. Sort an Array - <https://leetcode.cn/problems/sort-an-array/>

题目描述: 给定一个整数数组, 将其按升序排序

示例:

输入: [5, 2, 3, 1]

输出: [1, 2, 3, 5]

3. 牛客网 - 最小的 K 个数 - <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

4. LintCode 464. Sort Integers II - <https://www.lintcode.com/problem/sort-integers-ii/>

题目描述: 给定一个整数数组, 使用快速排序、归并排序、堆排序等 $O(n \log n)$ 算法将其按升序排序

示例:

输入: [3, 2, 1, 4, 5]

输出: [1, 2, 3, 4, 5]

5. LeetCode 75. Sort Colors - <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

示例:

输入: [2, 0, 2, 1, 1, 0]

输出: [0, 0, 1, 1, 2, 2]

6. LeetCode 215. Kth Largest Element in an Array - <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

示例:

输入: [3, 2, 1, 5, 6, 4], `k` = 2

输出: 5

"""

```
import random
import time
import copy
```

```
def swap(arr, i, j):
```

"""

数组中交换 i 和 j 位置的数

Args:

arr: 数组

i: 位置 i

j: 位置 j

"""

```
    arr[i], arr[j] = arr[j], arr[i]
```

```
def selection_sort(arr):
```

"""

选择排序 - Selection Sort

时间复杂度: $O(n^2)$ - 无论什么情况都需要进行 $n(n-1)/2$ 次比较

空间复杂度: $O(1)$ - 只使用了常数级别的额外空间

稳定性: 不稳定 - 相等元素的相对位置可能改变

算法思路:

1. 在未排序序列中找到最小元素
2. 将其与未排序序列的第一个元素交换位置
3. 重复步骤 1-2, 直到所有元素排序完成

优点:

- 实现简单

- 原地排序，空间复杂度低
- 交换次数少，最多进行 $n-1$ 次交换

缺点：

- 时间复杂度高，不适合大数据量
- 不稳定
- 无法利用数据的有序性优化

适用场景：

- 数据量小的情况
- 对内存使用要求严格的场景
- 不要求稳定性的场景

Args:

```
arr: 待排序数组
"""
# 边界检查：空数组或单元素数组无需排序
if arr is None or len(arr) < 2:
    return

# 外层循环控制排序的轮数，需要进行  $n-1$  轮
# 每轮都会确定一个元素的最终位置（当前未排序部分的最小元素）
for i in range(len(arr) - 1):
    # 假设当前位置 i 就是未排序部分的最小值位置
    # 从当前位置开始，在未排序部分 [i, n-1] 中寻找真正的最小值
    min_index = i

    # 内层循环在未排序部分 [i+1, n-1] 中寻找真正的最小值
    # j 从 i+1 开始，因为位置 i 已经是当前假设的最小值位置
    for j in range(i + 1, len(arr)):
        # 如果找到更小的元素，更新最小值索引
        # 这里使用 < 而不是 <= 是为了保持算法的不稳定性
        if arr[j] < arr[min_index]:
            min_index = j

    # 如果最小值不在当前位置，则交换
    # 这样可以减少不必要的交换操作（当 min_index == i 时不需要交换）
    if min_index != i:
        swap(arr, i, min_index)

def bubble_sort(arr):
    """
```

冒泡排序 - Bubble Sort

时间复杂度: $O(n^2)$ - 最坏和平均情况, $O(n)$ - 最好情况(已排序)

空间复杂度: $O(1)$ - 只使用了常数级别的额外空间

稳定性: 稳定 - 相等元素不会交换位置

算法思路:

1. 比较相邻的两个元素, 如果前面的比后面的大就交换
2. 每一轮都会将当前未排序部分的最大元素“冒泡”到末尾
3. 重复步骤 1-2, 直到所有元素排序完成

优点:

- 实现简单, 容易理解
- 稳定排序
- 原地排序
- 能够检测数组是否已经有序

缺点:

- 时间复杂度高, 不适合大数据量
- 元素交换次数多

优化:

- 设置标志位, 如果某一轮没有发生交换, 说明数组已经有序, 可以提前结束

适用场景:

- 数据量小的情况
- 要求稳定性的场景
- 教学演示

Args:

```
arr: 待排序数组
"""
# 边界检查: 空数组或单元素数组无需排序
if arr is None or len(arr) < 2:
    return

# 外层循环控制排序的轮数, 最多需要进行 n-1 轮
# 每轮都会确定一个元素的最终位置 (当前未排序部分的最大元素)
# end 表示每轮比较的上界, 随着排序的进行逐渐减小
for end in range(len(arr) - 1, 0, -1):
    # 优化标志: 记录本轮是否发生交换
    # 如果一轮比较中没有发生任何交换, 说明数组已经有序
    swapped = False
```

```

# 内层循环进行相邻元素的比较和交换
# 每轮比较范围逐渐缩小，因为末尾的元素已经有序
# i 从 0 开始到 end-1，比较 arr[i] 和 arr[i+1]
for i in range(end):
    # 如果前面的元素比后面的大，则交换
    # 这会将较大的元素逐步向右移动（“冒泡”）
    if arr[i] > arr[i + 1]:
        swap(arr, i, i + 1)
        swapped = True

    # 如果本轮没有发生交换，说明数组已经有序，可以提前结束
    # 这是冒泡排序的一个重要优化，可以将最好情况的时间复杂度降到 O(n)
    if not swapped:
        break

```

```

def insertion_sort(arr):
    """
插入排序 - Insertion Sort
时间复杂度: O(n2) - 最坏情况, O(n) - 最好情况(已排序)
空间复杂度: O(1) - 只使用了常数级别的额外空间
稳定性: 稳定 - 相等元素不会交换位置
    """

```

算法思路:

1. 将数组分为已排序和未排序两部分，初始时已排序部分只有第一个元素
2. 依次取出未排序部分的元素，在已排序部分找到合适的插入位置
3. 将元素插入到正确位置，重复步骤 2-3 直到所有元素排序完成

优点:

- 实现简单
- 稳定排序
- 原地排序
- 对于小规模或基本有序的数据效率很高
- 在线算法：可以在接收数据的同时进行排序

缺点:

- 时间复杂度高，不适合大数据量
- 对于逆序数据效率较低

适用场景:

- 小规模数据排序
- 基本有序的数据
- 在线数据排序

- 作为高级排序算法的子过程（如快速排序的小数组优化）

Args:

arr: 待排序数组

"""

边界检查: 空数组或单元素数组无需排序

if arr is None or len(arr) < 2:

 return

从第二个元素开始, 因为第一个元素可以看作已排序

i 表示当前要插入的元素位置

for i in range(1, len(arr)):

 # 从当前位置向前比较, 找到合适的插入位置

 # 当前元素为 arr[i], 需要在 arr[0...i-1]中找到插入位置

 # j 从 i-1 开始向前进遍历已排序部分

 for j in range(i - 1, -1, -1):

 # 如果前一个元素大于当前元素, 则交换

 # 这实际上是在将当前元素向前移动

 if j >= 0 and arr[j] > arr[j + 1]:

 swap(arr, j, j + 1)

 else:

 # 找到合适的位置, 跳出内层循环

 # 当 arr[j] <= arr[j+1]时, 说明已找到插入位置

 break

def random_array(n, v):

"""

得到一个随机数组, 长度是 n, 数组中每个数, 都在 1~v 之间, 随机得到

Args:

n: 数组长度

v: 数值范围上限

Returns:

随机数组

"""

return [random.randint(1, v) for _ in range(n)]

def copy_array(arr):

"""

复制数组

Args:

arr: 原数组

Returns:

复制的数组

"""

return copy.deepcopy(arr)

def same_array(arr1, arr2):

"""

比较两个数组是否相同

Args:

arr1: 数组 1

arr2: 数组 2

Returns:

是否相同

"""

if len(arr1) != len(arr2):

 return False

for i in range(len(arr1)):

 if arr1[i] != arr2[i]:

 return False

return True

def is_sorted(arr):

"""

检查数组是否已排序

Args:

arr: 数组

Returns:

是否已排序

"""

for i in range(1, len(arr)):

 if arr[i] < arr[i - 1]:

```
        return False
    return True

def generate_random_array(size):
    """
    生成随机测试数组

    Args:
        size: 数组大小

    Returns:
        随机数组
    """
    return [random.randint(0, size * 10) for _ in range(size)]

def performance_test():
    """
    性能测试: 比较三种排序算法在不同数据规模下的表现
    """
    print("\n==== 性能测试 ====")

    sizes = [100, 500, 1000]
    algorithms = ["选择排序", "冒泡排序", "插入排序"]
    sort_functions = [selection_sort, bubble_sort, insertion_sort]

    for size in sizes:
        print(f"\n数组大小: {size}")
        data = generate_random_array(size)

        for j, algorithm in enumerate(algorithms):
            test_data = copy_array(data)
            start_time = time.time()

            sort_functions[j](test_data)

            end_time = time.time()
            duration = (end_time - start_time) * 1000 # 转换为毫秒

            print(f"{algorithm}: {duration:.2f} ms")

        # 验证排序正确性
```

```
correct = is_sorted(test_data)
if not correct:
    print(" 排序错误!")

def test_sort_algorithms():
    """
    测试函数: 验证三种排序算法的正确性
    """
    print("== 选择排序、冒泡排序、插入排序测试 ==")

    # 测试用例设计
    test_cases = [
        [],                      # 空数组
        [1],                     # 单元素
        [1, 2, 3],               # 已排序
        [3, 2, 1],               # 逆序
        [1, 1, 1],               # 全相同
        [5, 2, 8, 1, 9],         # 普通情况
        [3, 1, 4, 1, 5, 9, 2, 6] # 重复元素
    ]

    algorithms = ["选择排序", "冒泡排序", "插入排序"]
    sort_functions = [selection_sort, bubble_sort, insertion_sort]

    for i, test_case in enumerate(test_cases):
        print(f"\n 测试用例 {i + 1}: {test_case}")

        for j, algorithm in enumerate(algorithms):
            arr = copy_array(test_case)
            expected = sorted(test_case) # 使用系统排序作为基准

            sort_functions[j](arr)

            correct = arr == expected
            print(f"{algorithm}: {arr} - {'✓' if correct else '✗'}")

            if not correct:
                print(f"预期: {expected}")

def main():
    """
```

主函数：演示排序算法的使用

"""

随机数组最大长度

N = 200

随机数组每个值，在 1~V 之间等概率随机

V = 1000

testTimes : 测试次数（减少测试次数以避免运行时间过长）

test_times = 5000

print("测试开始")

for i in range(test_times):

随机得到一个长度，长度在[0~N-1]

n = random.randint(0, N - 1)

得到随机数组

arr = random_array(n, V)

arr1 = copy_array(arr)

arr2 = copy_array(arr)

arr3 = copy_array(arr)

selection_sort(arr1)

bubble_sort(arr2)

insertion_sort(arr3)

if not same_array(arr1, arr2) or not same_array(arr1, arr3):

print("出错了!")

当有错了

打印是什么例子，出错的

打印三个功能，各自排序成了什么样

可能要把例子带入，每个方法，去 debug！

print("测试结束")

额外测试用例演示

test_sort_algorithms()

性能测试

performance_test()

测试额外的题目

test_additional_problems()

=====

```
# 插入排序的优化版本
# =====
```

```
def insertion_sort_optimized(arr):
    """
    插入排序的优化版本 - 使用赋值代替交换，减少操作次数
```

优化原理：

在标准插入排序中，每次比较都可能涉及一次完整的交换操作（3次赋值）

而在优化版本中，我们先保存当前要插入的元素，然后只进行元素后移操作

最后再将保存的元素插入到正确位置，这样可以减少赋值操作的次数

性能提升：

- 对于随机数据，大约可以减少 30%-50% 的赋值操作
- 对于接近有序的数据，性能提升更显著

```
"""
```

```
# 边界检查：空数组或单元素数组无需排序
```

```
if arr is None or len(arr) < 2:
    return
```

```
# 从第二个元素开始处理
```

```
for i in range(1, len(arr)):
```

```
    # 保存当前要插入的元素
```

```
    current = arr[i]
```

```
    # j 指向已排序部分的最后一个位置
```

```
    j = i - 1
```

```
    # 将大于 current 的元素向后移动
```

```
    # 当已排序部分的元素大于 current 时，将其向后移动一位
```

```
    while j >= 0 and arr[j] > current:
```

```
        arr[j + 1] = arr[j]
```

```
        j -= 1
```

```
    # 将 current 插入到正确位置
```

```
    # 此时 j+1 就是 current 应该插入的位置
```

```
    arr[j + 1] = current
```

```
def binary_insertion_sort(arr):
```

```
    """
```

二分插入排序 - 使用二分查找优化插入排序

优化原理：

在已排序的部分查找插入位置时，使用二分查找代替线性扫描

可以将查找过程的时间复杂度从 $O(n)$ 降低到 $O(\log n)$
但整体排序的时间复杂度仍然是 $O(n^2)$ ，因为元素移动的操作无法避免

适用场景：

- 数据量较大但仍在可接受范围内的场景
- 比较操作成本较高的场景

"""

```
# 边界检查：空数组或单元素数组无需排序
if arr is None or len(arr) < 2:
    return

# 从第二个元素开始处理
for i in range(1, len(arr)):
    # 保存当前要插入的元素
    current = arr[i]
    # 使用二分查找找到插入位置
    # 在已排序部分 arr[0...i-1] 中查找插入位置
    left, right = 0, i - 1

    # 二分查找过程
    # 查找第一个大于 current 的元素位置
    while left <= right:
        # 使用 left + (right - left) // 2 而不是(left + right) // 2
        # 可以避免当 left 和 right 都很大时可能发生的整数溢出
        mid = left + (right - left) // 2
        if arr[mid] > current:
            # current 应该插入到 mid 或其左侧
            right = mid - 1
        else:
            # current 应该插入到 mid 右侧
            left = mid + 1

    # 找到了插入位置 left，需要将[left, i-1]的元素后移
    # 注意：Python 中切片赋值可以简化这一操作
    # 将 arr[left...i-1] 的元素向后移动一位到 arr[left+1...i]
    arr[left+1:i+1] = arr[left:i]

    # 将 current 插入到正确位置
    arr[left] = current
```

```
# =====
# 经典算法题目的最优解法实现
# =====
```

```
def sort_colors(nums):
"""
LeetCode 75. 颜色分类 - 最优解法（三指针法/荷兰国旗算法）
题目链接: https://leetcode.cn/problems/sort-colors/

```

时间复杂度: $O(n)$ – 仅需一次遍历

空间复杂度: $O(1)$ – 原地排序

算法思想:

使用三个指针将数组分为三个区域:

- $[0, p0]$: 已排序的 0 区域
- $[p0, curr]$: 已排序的 1 区域
- $[p2, n-1]$: 已排序的 2 区域
- $[curr, p2]$: 待处理的区域

算法步骤:

1. 初始化 $p0=0$ (0 的右边界), $curr=0$ (当前处理位置), $p2=n-1$ (2 的左边界)
2. 当 $curr \leq p2$ 时循环:
 - a. 如果 $nums[curr] == 0$, 交换 $nums[curr]$ 和 $nums[p0]$, $p0++$, $curr++$
 - b. 如果 $nums[curr] == 1$, $curr++$
 - c. 如果 $nums[curr] == 2$, 交换 $nums[curr]$ 和 $nums[p2]$, $p2--$ ($curr$ 不变)

为什么是最优解:

- 相比基础排序算法的 $O(n^2)$ 时间复杂度, 三指针法只需要 $O(n)$ 时间
- 空间复杂度为 $O(1)$, 不需要额外空间
- 只需要一次遍历, 效率高
- 直接利用了问题特性 (只有 0、1、2 三种元素)

"""

```
# 防御性编程: 检查输入合法性
if nums is None or len(nums) < 2:
    return

n = len(nums)
p0 = 0      # 0 的右边界 (初始为 0)
curr = 0     # 当前遍历的位置
p2 = n - 1  # 2 的左边界 (初始为数组末尾)

# 遍历数组直到 curr 超过 p2
# 循环条件是 curr <= p2, 因为 p2 位置的元素尚未处理
while curr <= p2:
    if nums[curr] == 0:
        # 当前元素为 0, 放到 0 的区域
```

```

# 交换后, p0 位置的元素一定是 0, curr 位置的元素是原来 p0 位置的元素 (0、1 或 2)
# 由于 p0 <= curr, p0 位置的元素已经被处理过, 所以可以安全地递增 curr
swap(nums, curr, p0)
curr += 1
p0 += 1
elif nums[curr] == 2:
    # 当前元素为 2, 放到 2 的区域
    # 交换后, p2 位置的元素是原来 curr 位置的元素 (未知), 所以 curr 不能递增
    swap(nums, curr, p2)
    p2 -= 1
    # 注意 curr 不变, 因为交换过来的元素还未处理
else:
    # 当前元素为 1, 保持不动, 继续处理下一个元素
    # 1 的区域自然扩展
    curr += 1

```

def merge(nums1, m, nums2, n):

"""

LeetCode 88. 合并两个有序数组 - 最优解法 (从后向前合并)

题目链接: <https://leetcode.cn/problems/merge-sorted-array/>

时间复杂度: $O(m+n)$ - 仅需一次遍历

空间复杂度: $O(1)$ - 原地操作

算法思想:

从两个数组的末尾开始比较, 将较大的元素放到 nums1 的末尾位置

这样可以避免覆盖 nums1 中的原始数据, 不需要额外空间

算法步骤:

1. 初始化三个指针: $i=m-1$ (nums1 有效元素的末尾), $j=n-1$ (nums2 的末尾), $k=m+n-1$ (nums1 的末尾)
2. 比较 $\text{nums1}[i]$ 和 $\text{nums2}[j]$, 将较大的元素放到 $\text{nums1}[k]$ 的位置
3. 递减相应的指针, 重复步骤 2 直到处理完所有元素
4. 如果 nums2 还有剩余元素, 直接复制到 nums1 的前面 (nums1 剩余的元素已经在正确位置)

"""

防御性编程: 检查输入合法性

if nums2 is None or n == 0:

return # nums2 为空, 无需合并

if nums1 is None:

raise ValueError("nums1 cannot be None")

if len(nums1) < m + n:

raise ValueError("nums1 does not have enough space")

i = m - 1 # nums1 有效元素的最后一个位置

```

j = n - 1      # nums2 的最后一个位置
k = m + n - 1 # nums1 的最后一个位置

# 从后向前合并，比较并放置较大的元素
# 当两个数组都还有元素时进行比较
while i >= 0 and j >= 0:
    if nums1[i] > nums2[j]:
        # nums1 的元素较大，放到 nums1 的末尾
        nums1[k] = nums1[i]
        i -= 1
    else:
        # nums2 的元素较大或相等，放到 nums1 的末尾
        nums1[k] = nums2[j]
        j -= 1
    k -= 1

# 如果 nums2 还有剩余元素，直接复制到 nums1 的前面
# 注意：如果 nums1 还有剩余元素，它们已经在正确的位置上，无需处理
while j >= 0:
    nums1[k] = nums2[j]
    j -= 1
    k -= 1

```

def move_zeroes(nums):

```

"""
LeetCode 283. 移动零 - 最优解法（双指针法）
题目链接: https://leetcode.cn/problems/move-zeroes/

```

时间复杂度: O(n) - 仅需一次遍历

空间复杂度: O(1) - 原地操作

算法思想:

使用两个指针，一个指向当前应该放置非零元素的位置，另一个遍历整个数组
当遇到非零元素时，将其移动到第一个指针指向的位置，然后第一个指针前进

算法步骤:

1. 初始化一个指针 non_zero_pos=0，表示下一个非零元素应该放置的位置
2. 遍历数组，对于每个元素：
 - a. 如果元素非零，将其移动到 non_zero_pos 位置，然后 non_zero_pos++
3. 遍历结束后，将 non_zero_pos 到数组末尾的所有元素设置为 0

```

"""
# 防御性编程: 检查输入合法性
if nums is None or len(nums) <= 1:

```

```

return

non_zero_pos = 0 # 下一个非零元素应该放置的位置

# 第一步：将所有非零元素移动到数组前面
# 遍历整个数组
for i in range(len(nums)):
    if nums[i] != 0:
        # 将非零元素移动到 non_zero_pos 位置
        nums[non_zero_pos] = nums[i]
        non_zero_pos += 1

# 第二步：将剩余位置填充为 0
# 将 non_zero_pos 到数组末尾的所有位置设置为 0
for i in range(non_zero_pos, len(nums)):
    nums[i] = 0

```

def move_zeroes_optimized(nums):

"""

LeetCode 283. 移动零 - 优化版本（一次遍历，更少的赋值操作）

优化思路：

当遇到非零元素时，直接与 non_zero_pos 位置交换，这样可以减少一些不必要的赋值操作
特别是当数组中大部分元素都是非零时，这种方法更高效

"""

防御性编程：检查输入合法性

if nums is None or len(nums) <= 1:

return

non_zero_pos = 0 # 下一个非零元素应该放置的位置

遍历数组

for i in range(len(nums)):

if nums[i] != 0:

当两个指针不同时才交换，避免不必要的操作

如果 i == non_zero_pos，说明前面没有 0，无需交换

if i != non_zero_pos:

swap(nums, i, non_zero_pos)

non_zero_pos += 1

def find_kth_largest(nums, k):

"""

LeetCode 215. 数组中的第 K 个最大元素 - 快速选择算法

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

时间复杂度: $O(n)$ - 平均情况, $O(n^2)$ - 最坏情况

空间复杂度: $O(\log n)$ - 递归调用栈的深度, 最坏情况为 $O(n)$

算法思想:

基于快速排序的分区思想, 每次分区后只递归处理包含第 k 大元素的那一半

这样可以避免对整个数组进行排序

算法步骤:

1. 选择一个基准元素, 将数组分为两部分: 大于基准的和小于基准的
2. 如果基准元素的位置正好是第 k 大的位置, 返回该元素
3. 否则, 递归处理包含第 k 大元素的那一半

"""

防御性编程: 检查输入合法性

```
if nums is None or len(nums) == 0 or k <= 0 or k > len(nums):  
    raise ValueError("Invalid input")
```

第 k 大元素在排序后的数组中的索引是 $\text{len}(\text{nums}) - k$

例如: 数组 [1, 2, 3, 4, 5] 中第 2 大的元素是 4, 其索引为 5-2=3

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

```
def quick_select(nums, left, right, k):
```

"""

快速选择算法的核心实现

参数:

nums: 数组

left: 左边界

right: 右边界

k: 目标索引 (第 k 小的元素)

返回:

第 k 小的元素值

"""

分区操作, 返回基准元素的最终位置

pivot_index 是基准元素在数组中的最终位置

```
pivot_index = partition(nums, left, right)
```

如果基准元素的位置正好是 k , 返回该元素

```
if pivot_index == k:
```

```
    return nums[pivot_index]
```

如果基准元素的位置大于 k , 递归处理左半部分

```

        elif pivot_index > k:
            return quick_select(nums, left, pivot_index - 1, k)
        # 如果基准元素的位置小于 k, 递归处理右半部分
        else:
            return quick_select(nums, pivot_index + 1, right, k)

def partition(nums, left, right):
    """
    快速排序的分区操作

    参数:
        nums: 数组
        left: 左边界
        right: 右边界

    返回:
        基准元素的最终位置
    """
    # 选择最右边的元素作为基准
    # 这是一种简单的选择策略, 也可以使用随机选择来避免最坏情况
    pivot = nums[right]
    # i 表示小于基准元素的区域的边界
    # 初始时小于基准的区域为空, 所以 i = left - 1
    i = left - 1

    # 遍历[left, right-1]范围内的元素
    for j in range(left, right):
        # 如果当前元素小于基准元素, 将其交换到小于区域
        if nums[j] <= pivot:
            # 扩展小于基准的区域
            i += 1
            swap(nums, i, j)

    # 将基准元素放到正确的位置
    # 此时 i+1 是基准元素应该放置的位置
    swap(nums, i + 1, right)
    return i + 1

# =====
# 算法调试与优化技巧
# =====

def print_array_details(arr, message):

```

```
"""
```

```
打印数组的详细信息，用于调试
```

```
参数:
```

```
    arr: 要打印的数组
```

```
    message: 描述信息
```

```
"""
```

```
print(message)
```

```
if arr is None:
```

```
    print("Array is null")
```

```
    return
```

```
print(f"{arr}")
```

```
print(f"Length: {len(arr)}")
```

```
if len(arr) > 0:
```

```
    print(f"First element: {arr[0]}")
```

```
    print(f"Last element: {arr[-1]}")
```

```
print()
```

```
def analyze_sort_performance(arr, sort_method):
```

```
"""
```

```
分析排序算法的性能指标
```

```
参数:
```

```
    arr: 要排序的数组
```

```
    sort_method: 排序方法名称
```

```
"""
```

```
# 创建数组副本，避免修改原数组
```

```
arr_copy = copy_array(arr)
```

```
print(f"== {sort_method} 性能分析 ==")
```

```
print(f"数组大小: {len(arr_copy)}")
```

```
# 测量排序前是否已排序
```

```
was_sorted = is_sorted(arr_copy)
```

```
print(f"排序前是否有序: {'是' if was_sorted else '否'}")
```

```
# 测量排序时间
```

```
import time
```

```
start_time = time.time_ns()
```

```
# 根据方法名选择排序算法
```

```
if sort_method == "选择排序":
```

```
    selection_sort(arr_copy)
```

```

    elif sort_method == "冒泡排序":
        bubble_sort(arr_copy)
    elif sort_method == "插入排序":
        insertion_sort(arr_copy)
    elif sort_method == "优化插入排序":
        insertion_sort_optimized(arr_copy)
    elif sort_method == "二分插入排序":
        binary_insertion_sort(arr_copy)
    else:
        print("未知的排序方法")
        return

    end_time = time.time_ns()
    duration_ms = (end_time - start_time) / 1e6 # 转换为毫秒
    print(f"排序耗时: {duration_ms:.4f} ms")

# 验证排序结果
is_sorted_flag = is_sorted(arr_copy)
print(f"排序结果是否正确: {'是' if is_sorted_flag else '否'}")
print()

# =====
# 工程化改造示例 - 将排序算法封装为可复用组件
# =====

class SortUtils:
    """
    排序工具类 - 封装了各种排序算法，提供统一的接口
    这个类演示了如何将排序算法工程化为可复用组件
    """
    # 排序算法枚举
    class SortAlgorithm:
        SELECTION_SORT = "selection_sort"
        BUBBLE_SORT = "bubble_sort"
        INSERTION_SORT = "insertion_sort"
        INSERTION_SORT_OPTIMIZED = "insertion_sort_optimized"
        BINARY_INSERTION_SORT = "binary_insertion_sort"

        @staticmethod
        def sort(arr, algorithm):
            """
            统一的排序接口
            """

```

参数:

arr: 要排序的数组

algorithm: 选择的排序算法

"""

防御性编程

```
if arr is None or len(arr) < 2:  
    return
```

根据选择的算法调用相应的排序方法

```
if algorithm == SortUtils.SortAlgorithm.SELECTION_SORT:  
    selection_sort(arr)  
elif algorithm == SortUtils.SortAlgorithm.BUBBLE_SORT:  
    bubble_sort(arr)  
elif algorithm == SortUtils.SortAlgorithm.INSERTION_SORT:  
    insertion_sort(arr)  
elif algorithm == SortUtils.SortAlgorithm.INSERTION_SORT_OPTIMIZED:  
    insertion_sort_optimized(arr)  
elif algorithm == SortUtils.SortAlgorithm.BINARY_INSERTION_SORT:  
    binary_insertion_sort(arr)  
else:  
    raise ValueError("Unsupported sorting algorithm")
```

@staticmethod

def auto_select_sort(arr):

"""

根据数据特征自动选择最合适的排序算法

参数:

arr: 要排序的数组

"""

防御性编程

```
if arr is None or len(arr) < 2:  
    return
```

分析数据特征

n = len(arr)

is_nearly_sorted = SortUtils._is_nearly_sorted(arr)

has_few_unique = SortUtils._has_few_unique_values(arr)

根据数据特征选择算法

```
if is_nearly_sorted:
```

接近有序的数据使用插入排序

```
SortUtils.sort(arr, SortUtils.SortAlgorithm.INSERTION_SORT_OPTIMIZED)
```

```
elif n < 1000:  
    # 小规模数据使用插入排序  
    SortUtils.sort(arr, SortUtils.SortAlgorithm.INSERTION_SORT_OPTIMIZED)  
else:  
    # 其他情况使用二分插入排序  
    SortUtils.sort(arr, SortUtils.SortAlgorithm.BINARY_INSERTION_SORT)
```

```
@staticmethod  
def _is_nearly_sorted(arr):  
    """
```

判断数组是否接近有序

参数:

arr: 要检查的数组

返回:

如果数组接近有序返回 True, 否则返回 False

```
"""
```

```
inversion_count = 0
```

```
threshold = len(arr) // 2 # 阈值: 逆序对数量不超过数组长度的一半
```

```
# 计算逆序对数量
```

```
for i in range(len(arr) - 1):  
    for j in range(i + 1, len(arr)):  
        if arr[i] > arr[j]:  
            inversion_count += 1  
        # 如果超过阈值, 提前返回  
        if inversion_count >= threshold:  
            return False
```

```
return inversion_count < threshold
```

```
@staticmethod  
def _has_few_unique_values(arr):  
    """
```

判断数组是否有少量唯一值

参数:

arr: 要检查的数组

返回:

如果数组有少量唯一值返回 True, 否则返回 False

```
"""
```

```
# 简单实现：检查是否有超过 25%的重复元素
unique_values = set(arr)
return len(unique_values) < len(arr) * 0.25
```

```
# =====
# 更多经典题目的实现 - 涉及各大算法平台的高频题目
# =====
```

```
def top_k_frequent(nums, k):
    """
    LeetCode 347. 前 K 个高频元素
    题目链接: 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

示例：

输入：`nums = [1, 1, 1, 2, 2, 3]`, `k = 2`

输出：`[1, 2]`

解题思路：

1. 使用哈希表统计每个元素的频率
2. 使用堆（优先队列）或桶排序找到频率最高的 `k` 个元素

时间复杂度： $O(n \log k)$ - 使用最小堆

空间复杂度： $O(n)$ - 哈希表存储频率

最优解：桶排序，时间复杂度 $O(n)$ ，空间复杂度 $O(n)$

"""

```
# 防御性编程
if nums is None or len(nums) == 0 or k <= 0 or k > len(nums):
    raise ValueError("Invalid input")
```

步骤 1：统计频率

使用字典记录每个元素的出现次数

```
freq_map = {}
for num in nums:
    freq_map[num] = freq_map.get(num, 0) + 1
```

步骤 2：桶排序 - 按频率分组

```
# bucket[i] 存储频率为 i 的所有元素
# 桶的数量为 len(nums)+1，因为频率最大为 len(nums)
bucket = [[] for _ in range(len(nums) + 1)]
for num, freq in freq_map.items():
```

```

        bucket[freq].append(num)

# 步骤 3: 从高频到低频收集结果
result = []
# 从最大频率开始向下遍历
for i in range(len(bucket) - 1, -1, -1):
    if bucket[i]:
        # 将当前频率的所有元素添加到结果中
        for num in bucket[i]:
            result.append(num)
        # 当收集到 k 个元素时停止
        if len(result) == k:
            return result

return result

```

```

def relative_sort_array(arr1, arr2):
"""
LeetCode 1122. 数组的相对排序
题目链接: https://leetcode.cn/problems/relative-sort-array/

```

题目描述：给你两个数组，`arr1` 和 `arr2`，`arr2` 中的元素各不相同，`arr2` 中的每个元素都出现在 `arr1` 中。

对 `arr1` 中的元素进行排序，使 `arr1` 中项的相对顺序和 `arr2` 中的相对顺序相同。

未在 `arr2` 中出现过的元素需要按照升序放在 `arr1` 的末尾。

示例：

输入：`arr1 = [2, 3, 1, 3, 2, 4, 6, 7, 9, 2, 19]`，`arr2 = [2, 1, 4, 3, 9, 6]`
 输出：`[2, 2, 2, 1, 4, 3, 3, 9, 6, 7, 19]`

解题思路：

1. 使用计数排序思想，统计 `arr1` 中每个元素的出现次数
2. 按照 `arr2` 的顺序填充结果数组
3. 将不在 `arr2` 中的元素排序后放在末尾

时间复杂度： $O(n \log n)$ – 主要是排序不在 `arr2` 中的元素

空间复杂度： $O(n)$ – 哈希表和结果数组

最优解：计数排序，时间复杂度 $O(n + m)$ ，其中 n 是 `arr1` 的长度， m 是数值范围

"""

防御性编程

if arr1 is None or arr2 is None:

```

raise ValueError("Arrays cannot be None")

# 步骤 1: 统计 arr1 中每个元素的频率
count_map = {}
for num in arr1:
    count_map[num] = count_map.get(num, 0) + 1

# 步骤 2: 按照 arr2 的顺序填充结果
result = []
# 按照 arr2 中元素的顺序处理
for num in arr2:
    if num in count_map:
        # 将 count_map[num] 个 num 添加到结果中
        result.extend([num] * count_map[num])
        # 从 count_map 中删除已处理的元素
        del count_map[num]

# 步骤 3: 将不在 arr2 中的元素排序后放在末尾
remaining = []
# 收集剩余的元素
for num, count in count_map.items():
    remaining.extend([num] * count)
# 对剩余元素进行排序
remaining.sort()
# 将排序后的剩余元素添加到结果末尾
result.extend(remaining)

return result

```

```
def get_least_numbers(arr, k):
```

```
"""
```

剑指 Offer 40. 最小的 k 个数

题目链接: <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

示例:

输入: arr = [3, 2, 1], k = 2

输出: [1, 2] 或者 [2, 1]

解题思路:

方法 1: 排序后取前 k 个 - 时间复杂度 $O(n \log n)$

方法 2: 堆 (最大堆) - 时间复杂度 $O(n \log k)$

方法 3：快速选择算法 - 平均时间复杂度 $O(n)$

最优解：快速选择算法

时间复杂度： $O(n)$ - 平均情况

空间复杂度： $O(\log n)$ - 递归栈空间

"""

防御性编程

```
if arr is None or k <= 0 or k > len(arr):  
    return []
```

使用快速选择找到第 k 小的元素

由于是找最小的 k 个数，不需要完全排序

```
quick_select(arr, 0, len(arr) - 1, k - 1)
```

前 k 个元素就是结果（不一定有序）

对结果进行排序以满足题目要求

```
result = arr[:k]  
result.sort()  
return result
```

```
def find_relative_ranks(score):
```

"""

LeetCode 506. 相对名次

题目链接：<https://leetcode.cn/problems/relative-ranks/>

题目描述：给你一个长度为 n 的整数数组 $score$ ，其中 $score[i]$ 表示第 i 位运动员在比赛中的得分。所有得分都互不相同。运动员将根据得分决定名次，其中名次第 1 的运动员得分最高，名次第 2 的运动员得分第 2 高，依此类推。

示例：

输入： $score = [5, 4, 3, 2, 1]$

输出：["Gold Medal", "Silver Medal", "Bronze Medal", "4", "5"]

解题思路：

1. 创建索引数组，按分数排序
2. 根据排序后的索引分配名次

时间复杂度： $O(n \log n)$ - 排序的时间复杂度

空间复杂度： $O(n)$ - 存储索引和结果

"""

防御性编程

```
if score is None or len(score) == 0:
```

```

    return []

n = len(score)
# 创建索引数组，用于排序后找到原始位置
# indices[i]表示原始数组中第 i 个位置的索引
indices = list(range(n))

# 按分数从高到低排序索引
# key=lambda x: score[x]表示按 score[indices[i]]的值进行排序
# reverse=True 表示降序排列
indices. sort(key=lambda x: score[x], reverse=True)

# 根据排序后的索引分配名次
result = [''] * n
for i, idx in enumerate(indices):
    # 根据排名分配奖牌或名次
    if i == 0:
        result[idx] = "Gold Medal"
    elif i == 1:
        result[idx] = "Silver Medal"
    elif i == 2:
        result[idx] = "Bronze Medal"
    else:
        # 第 4 名及以后用数字表示
        result[idx] = str(i + 1)

return result

```

```

def sort_array_by_parity_ii(nums):
"""
LeetCode 922. 按奇偶排序数组 II
题目链接: https://leetcode.cn/problems/sort-array-by-parity-ii/

```

题目描述：给定一个非负整数数组 `nums`, `nums` 中一半整数是奇数，一半整数是偶数。
对数组进行排序，以便当 `nums[i]` 为奇数时，`i` 也是奇数；当 `nums[i]` 为偶数时，`i` 也是偶数。

示例：

输入: `nums = [4, 2, 5, 7]`

输出: `[4, 5, 2, 7]`

解题思路：

方法 1：使用两个数组分别存储奇数和偶数，然后按要求放回

方法 2：双指针原地交换

最优解：双指针原地交换

时间复杂度：O(n)

空间复杂度：O(1)

"""

防御性编程

```
if nums is None or len(nums) == 0:  
    return nums
```

n = len(nums)

even_idx = 0 # 偶数位置指针，用于寻找应该放偶数但放了奇数的位置

odd_idx = 1 # 奇数位置指针，用于寻找应该放奇数但放了偶数的位置

当两个指针都在有效范围内时继续循环

```
while even_idx < n and odd_idx < n:
```

找到偶数位置上的奇数

在偶数位置 (0, 2, 4...) 上寻找奇数

```
while even_idx < n and nums[even_idx] % 2 == 0:
```

even_idx += 2

找到奇数位置上的偶数

在奇数位置 (1, 3, 5...) 上寻找偶数

```
while odd_idx < n and nums[odd_idx] % 2 == 1:
```

odd_idx += 2

交换

如果找到了两个错误位置，进行交换

```
if even_idx < n and odd_idx < n:
```

swap(nums, even_idx, odd_idx)

交换后继续寻找下一个错误位置

even_idx += 2

odd_idx += 2

```
return nums
```

```
def test_additional_problems():
```

"""

测试额外的题目解法

"""

```
print("\n==== 额外算法题目测试 ====\n")
```

测试 top_k_frequent

```
print("--- LeetCode 347. 前 K 个高频元素 ---")
```

```
freq_test = [1, 1, 1, 2, 2, 3]
freq_result = top_k_frequent(freq_test, 2)
print(f"输入: [1, 1, 1, 2, 2, 3], k=2")
print(f"输出: {freq_result}")
print()

# 测试 relative_sort_array
print("--- LeetCode 1122. 数组的相对排序 ---")
arr1 = [2, 3, 1, 3, 2, 4, 6, 7, 9, 2, 19]
arr2 = [2, 1, 4, 3, 9, 6]
relative_result = relative_sort_array(arr1, arr2)
print(f"输入: arr1=[2, 3, 1, 3, 2, 4, 6, 7, 9, 2, 19], arr2=[2, 1, 4, 3, 9, 6]")
print(f"输出: {relative_result}")
print()

# 测试 get_least_numbers
print("--- 剑指 Offer 40. 最小的 k 个数 ---")
least_test = [3, 2, 1, 5, 6, 4]
least_result = get_least_numbers(least_test.copy(), 2)
print(f"输入: [3, 2, 1, 5, 6, 4], k=2")
print(f"输出: {least_result}")
print()

# 测试 find_relative_ranks
print("--- LeetCode 506. 相对名次 ---")
score_test = [5, 4, 3, 2, 1]
rank_result = find_relative_ranks(score_test)
print(f"输入: [5, 4, 3, 2, 1]")
print(f"输出: {rank_result}")
print()

# 测试 sort_array_by_parity_ii
print("--- LeetCode 922. 按奇偶排序数组 II ---")
parity_test = [4, 2, 5, 7]
parity_result = sort_array_by_parity_ii(parity_test)
print(f"输入: [4, 2, 5, 7]")
print(f"输出: {parity_result}")
print()

# 为了验证
if __name__ == "__main__":
    main()
```

=====