项目说明文档

数据结构课程设计

——8 种排序算法比较

作 者 姓 名：陆诚彬

学　　号：2254321

指 导 教 师：张颖

学院、 专业：软件学院 软件工程

同济大学

Tongji University

目录

1 项目背景

排序算法是计算机科学中最基本和重要的算法之一。它们在数据处理和分析中扮演着关键角色，用于组织和管理大量数据。有效的排序是数据库管理、搜索引擎、数据分析等领域的核心。

每种排序算法都有其独特的特点和适用场景。例如，快速排序因其平均情况下的高效率而广受欢迎，而归并排序则因其稳定性和对大数据集的有效处理而著称。了解不同算法的性能对于选择合适的排序策略至关重要。

在现实世界的应用中，选择正确的排序算法可以显著影响程序的性能。针对特定类型的数据或特定的应用场景选择最优算法是软件开发和数据科学领域的一个重要挑战。

2 项目需求分析

2.1 功能需求

1) **数据集生成与输入：**用户能够定义随机数的个数。系统能够基于用户输入生成指定数量的随机数。
2) **排序算法实现：**实现以下排序算法：快速排序、直接插入排序、冒泡排序、选择排序、希尔排序、堆排序、归并排序、基数排序。每种算法都应能处理相同的数据集。
3) **性能比较与统计：**计算并展示每种排序算法完成排序的时间。记录并展示每种排序算法的数据交换次数和比较次数。
4) **结果展示：**为用户提供一种方式查看和比较不同排序算法的性能指标。结果应包括排序时间、交换次数和比较次数等。
5) **用户交互界面：**提供一个简单的用户界面，允许用户输入随机数的数量并选择查看不同排序算法的性能结果。

2.2 非功能需求

1) **性能：**系统应能快速生成随机数。排序算法应优化以确保最佳性能。
2) **可用性：**用户界面应直观易用，适合非专业用户。结果展示应清晰、易于理解。
3) **扩展性：**系统设计应允许未来轻松添加更多排序算法。应能方便地引入新的性能评估指标。

2.3 项目输入输出需求

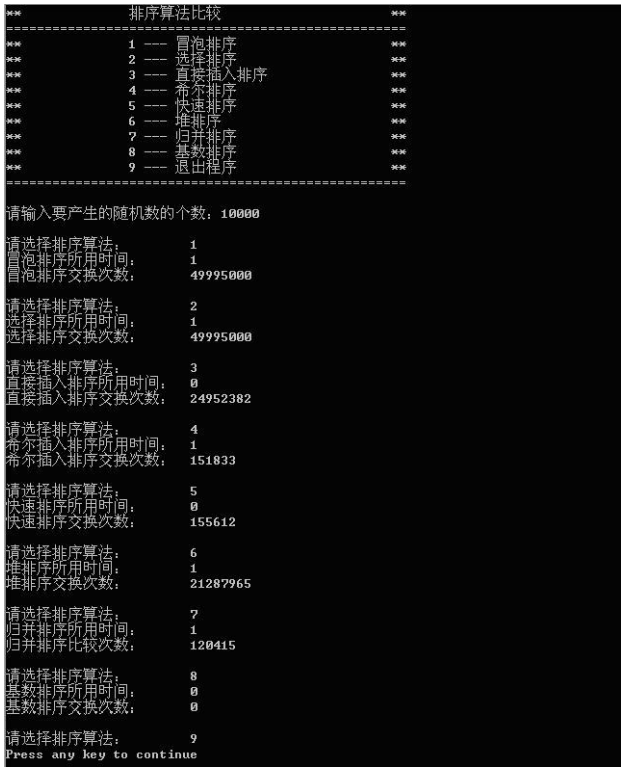
2.3.1 输入格式

输入相应的操作数，以及数据数字。

2.3.2 输出格式

输出相应的提示信息及排序结果统计。

2.3.3 项目示例



3 项目设计

3.1 数据结构设计

3.1.1 数组数据结构

目的：存储待排序的随机数。用于各排序算法的输入。

类型：int* - 动态分配的整数数组。

实现细节：数组的大小由用户输入决定，动态分配以适应不同的数据集大小。数组中的元素为随机生成的整数。

3.1.2 时间测量

目的：测量每种排序算法的执行时间。

类型：std::chrono::duration<double, std::milli> 记录以毫秒为单位的时间间隔。

实现细节：使用 std::chrono::high_resolution_clock 记录排序操作的开始和结束时间。

4 项目实现

由于本项目关键在于 8 种排序算法的实现，故下面会分析每种排序算法的特点：

4.1 冒泡排序

- **最好情况时间复杂度:** O(n)（已排序）
- **最差情况时间复杂度:** O(n²)（逆序）
- **平均时间复杂度:** O(n²)
- **特点:** 简单但效率低，适用于小数据集。

```
void bubbleSort(int* arr, int n, int& comparisons, int& swaps) {
    comparisons = 0;
    swaps = 0;
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            comparisons++;
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
                swaps++;
            }
        }
    }
}
```

4.2 选择排序

- **最好、最差和平均时间复杂度:** O(n²)
- **特点:** 不稳定排序，无论什么数据都是 O(n²)，适用于小数据集。

```
void selectionSort(int* arr, int n, int& comparisons, int& swaps) {
    comparisons = 0;
    swaps = 0;

```

```
        for (int i = 0; i < n - 1; i++) {
            // 找到最小元素的索引
            int min_index = i;
            for (int j = i + 1; j < n; j++) {
                comparisons++;
                if (arr[j] < arr[min_index]) {
                    min_index = j;
                }
            }

            // 将找到的最小元素与第 i 个元素交换
            if (min_index != i) {
                swap(arr[i], arr[min_index]);
                swaps++;
            }
        }
    }
```

4.3 插入排序

- **最好情况时间复杂度:** O(n)（已排序）
- **最差情况时间复杂度:** O(n²)（逆序）
- **平均时间复杂度:** O(n²)
- **特点:** 稳定排序，效率高于冒泡和选择，适用于部分排序好的小数据集。

```
    void insertionSort(int* arr, int n, int& comparisons, int& swaps) {
        comparisons = 0;
        swaps = 0;
        for (int i = 1; i < n; i++) {
            int key = arr[i];
            int j = i - 1;
            while (j >= 0 && arr[j] > key) {
                arr[j + 1] = arr[j];
                j = j - 1;
                comparisons++;
                swaps++;
            }
            arr[j + 1] = key;
            if (i != j + 1) {
                swaps++;
            }
        }
    }
```

4.4 希尔排序

- **最好情况时间复杂度:** O(n log n)（部分排序）
- **最差情况时间复杂度:** 依赖于间隔序列，最差可达 $O(n^2)$
- **平均时间复杂度:** 依赖于间隔序列
- **特点:** 非稳定排序，改进版的插入排序，对中等大小的数据集效果良好。

```
void shellSort(int* arr, int n, int& comparisons, int& swaps) {
    comparisons = 0;
    swaps = 0;
    // 开始时的间隔设为数组长度的一半，并逐渐减半
    for (int gap = n / 2; gap > 0; gap /= 2) {
        // 对于每个间隔，进行插入排序
        for (int i = gap; i < n; i += 1) {
            int temp = arr[i];
            int j;
            for (j = i; j >= gap && arr[j - gap] > temp; j -= gap) {
                arr[j] = arr[j - gap];
                comparisons++; // 对比较次数进行记录
                swaps++; // 这里的赋值也视为一次交换
            }
            arr[j] = temp;
            if (j != i) {
                swaps++; // 插入操作也视为一次交换
            }
        }
    }
}
```

4.5 快速排序

- **最好情况时间复杂度:** O(n log n)
- **最差情况时间复杂度:** $O(n^2)$（逆序或有序）
- **平均时间复杂度:** O(n log n)
- **特点:** 非稳定排序，效率高，适用于大数据集，但最差情况下效率低。

```
int partition(int* arr, int low, int high, int& comparisons, int& swaps) {
    int pivot = arr[high];
    int i = low - 1;

    for (int j = low; j <= high - 1; j++) {
        comparisons++; // 比较次数增加
```

```
            if (arr[j] < pivot) {
                i++;
                swap(arr[i], arr[j]);
                swaps++; // 交换次数增加
            }
        }
        swap(arr[i + 1], arr[high]);
        swaps++; // 交换次数增加
        return (i + 1);
    }

    void quickSort(int* arr, int low, int high, int& comparisons, int& swaps) {

        if (low < high) {
            int pi = partition(arr, low, high, comparisons, swaps);

            quickSort(arr, low, pi - 1, comparisons, swaps);
            quickSort(arr, pi + 1, high, comparisons, swaps);
        }
    }
```

4.6 堆排序

- **最好、最差和平均时间复杂度:** O(n log n)
- **特点:** 非稳定排序，适用于大数据集，效率相对稳定。

```
    void heapify(int* arr, int n, int i, int& comparisons, int& swaps) {
        int largest = i; // 初始化最大元素为根
        int left = 2 * i + 1; // 左孩子
        int right = 2 * i + 2; // 右孩子
        // 如果左孩子比根大
        if (left < n) {
            comparisons++; // 比较次数增加
            if (arr[left] > arr[largest]) {
                largest = left;
            }
        }
        // 如果右孩子存在且比目前最大的还大
        if (right < n) {
            comparisons++; // 比较次数增加
            if (arr[right] > arr[largest]) {
                largest = right;
            }
        }
```

```
        // 如果最大元素不是根
        if (largest != i) {
            swap(arr[i], arr[largest]);
            swaps++; // 交换次数增加

            // 递归地定义子堆
            heapify(arr, n, largest, comparisons, swaps);
        }
    }
    void heapSort(int* arr, int n, int& comparisons, int& swaps) {
        // 构建堆（重新排列数组）
        for (int i = n / 2 - 1; i >= 0; i--) {
            heapify(arr, n, i, comparisons, swaps);
        }
        // 一个个从堆顶取出元素
        for (int i = n - 1; i > 0; i--) {
            swap(arr[0], arr[i]);
            swaps++; // 交换次数增加
            // 调用 heapify 减少堆的大小并维护堆的性质
            heapify(arr, i, 0, comparisons, swaps);
        }
    }
```

4.7 归并排序

- **最好、最差和平均时间复杂度:** O(n log n)
- **特点:** 稳定排序，适用于任何大小的数据集，尤其是大数据集，需要额外的内存空间。

```
    void merge(int* arr, int left, int mid, int right, int& comparisons, int& sw
aps) {
        int n1 = mid - left + 1;
        int n2 = right - mid;
        int* L = new int[n1];
        int* R = new int[n2];
        // 拷贝数据到临时数组
        for (int i = 0; i < n1; i++)
            L[i] = arr[left + i];
        for (int j = 0; j < n2; j++)
            R[j] = arr[mid + 1 + j];
        // 合并临时数组
        int i = 0; // 初始索引第一个子数组
        int j = 0; // 初始索引第二个子数组
        int k = left; // 初始索引合并的子数组
```

```
15.         while (i < n1 && j < n2) {
16.             comparisons++; // 记录比较
17.             if (L[i] <= R[j]) {
18.                 arr[k] = L[i];
19.                 i++;
20.             }
21.             else {
22.                 arr[k] = R[j];
23.                 j++;
24.             }
25.             swaps++; // 记录交换
26.             k++;
27.         }
28.         // 拷贝 L[] 的剩余元素
29.         while (i < n1) {
30.             arr[k] = L[i];
31.             i++;
32.             k++;
33.             swaps++; // 记录交换
34.         }
35.         // 拷贝 R[] 的剩余元素
36.         while (j < n2) {
37.             arr[k] = R[j];
38.             j++;
39.             k++;
40.             swaps++; // 记录交换
41.         }
42.         delete[] L;
43.         delete[] R;
44.     }
45.     void mergeSort(int* arr, int left, int right, int& comparisons, int& swaps)
    {
46.         if (left < right) {
47.             int mid = left + (right - left) / 2;
48.             // 递归地对左右两半进行排序
49.             mergeSort(arr, left, mid, comparisons, swaps);
50.             mergeSort(arr, mid + 1, right, comparisons, swaps);
51.             // 合并两半
52.             merge(arr, left, mid, right, comparisons, swaps);
53.         }
54.     }
```

4.8 基数排序

- **最好、最差和平均时间复杂度:** O(nk)（k 是最大数的位数）
- **特点:** 稳定排序，适用于非负整数排序，效率高于比较排序算法，适用于大数据集。

```
void radixSort(int* arr, int n, int& comparisons, int& swaps) {
    int max = arr[0];
    for (int i = 1; i < n; i++) {
        comparisons++; // 记录比较
        if (arr[i] > max) {
            max = arr[i];
        }
    }
    // 对每一位进行计数排序
    for (int exp = 1; max / exp > 0; exp *= 10) {
        int* output = new int[n]; // 输出数组
        int count[10] = { 0 }; // 计数数组
        // 统计每个桶中的元素个数
        for (int i = 0; i < n; i++) {
            count[(arr[i] / exp) % 10]++;
        }
        // 将 count[i] 更新为包含 i 的元素的总数
        for (int i = 1; i < 10; i++) {
            count[i] += count[i - 1];
        }
        // 按照 count[i] 将 arr[i] 放到输出数组中
        for (int i = n - 1; i >= 0; i--) {
            output[count[(arr[i] / exp) % 10] - 1] = arr[i];
            count[(arr[i] / exp) % 10]--;
            swaps++; // 记录交换
        }
        // 将输出数组复制到 arr[] 中，现在 arr[] 包含从当前位排序的数字
        for (int i = 0; i < n; i++) {
            arr[i] = output[i];
        }
        delete[] output;
    }
}
```

5 设计小结

5.1 数据结构和类设计

数组数据结构: 项目采用动态分配的整数数组来存储待排序的随机数。这种简单的数据结构有效地满足了各种排序算法的基本需求，易于管理和访问。

时间测量: 使用 std::chrono 库来测量每种排序算法的执行时间。这提供了精确的时间测量，允许对不同算法的性能进行有效比较。

5.2 功能与非功能需求

功能需求: 成功实现了数据集的生成与输入、多种排序算法的实现、性能比较与统计，以及结果展示。这些功能为用户提供了全面的排序算法评估工具。

非功能需求: 通过优化排序算法和提供直观的用户界面，项目满足了性能和可用性要求。同时，设计的可扩展性确保了未来的维护和升级的便利性。

5.3 各排序算法的特点与实现

冒泡、选择、插入排序: 这些基本排序算法虽然简单，但效率较低，适用于小数据集。

希尔排序: 作为插入排序的改进版，它在中等大小的数据集上表现良好。

快速、堆、归并排序: 这些算法在大数据集上表现出色，但归并排序需要额外的内存空间。

基数排序: 特别适用于非负整数的排序，其性能在大数据集上超越传统的比较排序算法。

5.4 性能比较与统计的重要性

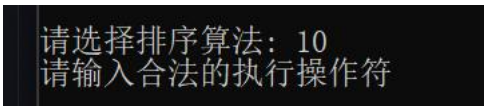
通过记录每种排序算法的数据交换次数和比较次数，项目有效地展示了不同算法在处理相同数据集时的性能差异。这种比较对于理解各算法的适用场景和优势至关重要。

6 软件测试

6.1 输入测试

6.1.1 正常输入

6.1.2 输入超界/非法



结论：符合输入逻辑判断

6.2 输出测试

结论：符合输出逻辑，且正确。