数据结构课程设计文档

题目：排序算法比较

姓名：赵卓冰

学号：2252750

专业：软件工程

年级：2023 级

指导教师：张颖

2024 年 12 月 12 日

1. 开发环境与编译运行环境

本项目可以在不同的开发环境和编译运行环境上运行

- Window操作系统：
  - 版本：Windows 10 x64
  - IDE： Visual Studio 2022 (Debug模式)
  - 编译器： MSVC 14.39.33519
- Linux操作系统：
  - 版本：Ubuntu 20.04.6 LTS
  - IDE： VS Code
  - 编译器：gcc version 9.4.0 (Ubuntu 9.4.0-1ubuntu1~20.04.2)

2. 项目背景及问题描述

排序是数据处理与分析中的基础操作，不同的排序算法在时间复杂度、空间复杂度、适用数据规模和数据分布上各有特点。本项目通过随机产生指定数量的随机数，并采用多种经典排序算法对这些数据进行排序，将详细记录每种算法在不同数据规模下的运行时间、比较次数（基础操作次数），以及总结这些算法的优势与劣势。

数据规模可由用户指定（例如100、1000、10000、100000个随机数），项目将使用以下八种排序算法对数据进行排序和评估：

1. 冒泡排序（Bubble Sort）
2. 选择排序（Selection Sort）
3. 直接插入排序（Insertion Sort）
4. 希尔排序（Shell Sort）
5. 快速排序（Quick Sort）

6. 堆排序（Heap Sort）
7. 归并排序（Merge Sort）
8. 基数排序（Radix Sort）

通过实验结果，我们可以：

- 对比不同算法在不同数据规模下的性能表现。
- 根据结果分析各算法的适用场景和优缺点。

3. 项目功能需求

1. **数据生成**：用户输入需要生成的随机数的数量（例如100、1000、10000、100000），程序使用随机函数生成相应数量的随机整数。
2. **多种排序算法比较**：程序实现了8种经典排序算法。用户可从菜单中选择其中任意一种进行排序。排序执行后，程序将输出该算法的运行时间和比较次数。
3. **性能统计与展示**：
   - **时间统计**：使用 `clock()` 函数记录排序前后的CPU时间，计算耗时。
   - **比较/操作次数统计**：在排序过程中计数关键比较或交换等基础操作次数。

   最终将结果以清晰的形式输出给用户。

4. 数据结构与代码架构

4.1. Vector类

`Vector`类是项目中的核心容器类，用于存储随机生成的整数数据。其主要功能包括：

- 动态扩容，支持在运行过程中存储更多元素。
- 提供下标访问和迭代器遍历，方便对数据进行操作。
- 基础的增删操作和内存管理。

4.2. 随机数生成模块

使用 C++ 标准库的随机数引擎 `std::mt19937` 和均匀分布 `std::uniform_int_distribution`，在指定范围内生成用户需要数量的随机整数。

5. 排序算法模块

项目实现了8种经典排序算法：

1. **冒泡排序（Bubble Sort）**：
   原理：
   冒泡排序通过多轮扫描相邻元素的方式进行排序。在每一趟扫描中，从数组的起始位置开始两两比较相邻元素，当发现前一个元素大于后一个元素时就进行交换。这样，一轮比较下

来，会将当前未排序部分中最大的元素“冒泡”到数组的末尾位置。然后在下一轮中对除了最后一个已排好位置的部分继续重复该过程，直到整个数组有序。

时间复杂度：平均和最坏情况均为 $O(n^2)$。

优点：实现简单，适合少量数据或教学入门。

缺点：数据量一大性能急剧下降。

代码示例（核心逻辑片段）：

```
for (int i = 0; i < n - 1; ++i) {
    for (int j = 0; j < n - 1 - i; ++j) {
        if (arr[j] > arr[j + 1]) {
            Swap(arr[j], arr[j + 1]);
        }
    }
}
```

2. **选择排序（Selection Sort）**：
原理：
选择排序在每一轮中从未排序部分中选取最小元素，将其放到已排序部分的末尾。第一轮找最小值放在第一个位置，第二轮在剩下的部分中找最小值放在第二个位置，以此类推，直到整个数组有序。

时间复杂度：平均和最坏情况均为 $O(n^2)$。

优点：交换次数少，实现简洁。

缺点：比较次数仍为 (O(n^2))，对大数据仍然效率不高。

代码示例（核心逻辑片段）：

```
for (int i = 0; i < n - 1; ++i) {
    int min_index = i;
    for (int j = i + 1; j < n; ++j) {
        if (arr[j] < arr[min_index]) {
            min_index = j;
        }
    }
    Swap(arr[i], arr[min_index]);
}
```

3. **直接插入排序（Insertion Sort）**：
原理：
将数组分为已排序和未排序两部分。每次从未排序中取出一个元素，插入到已排序部分的适当位置。已排序部分不断扩展，最终整个数组有序。

时间复杂度：平均和最坏情况为 $O(n^2)$。

优点：对部分有序数据或小规模数据较为高效，代码简单。

缺点：对随机大规模数据仍不够高效。

代码示例（核心逻辑片段）：

```
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--;
    }
    arr[j + 1] = key;
}
```

4. **希尔排序（Shell Sort）**：
   原理：
   希尔排序先定义一个增量（gap），按该增量将数组分组，并对每组进行插入排序。随着算法进行，不断缩小增量，直到增量为1，此时数组已接近有序，再进行插入排序可大幅减少操作。

   时间复杂度：约为 $O(n^{1.3})$，依赖增量序列。

   优点：比单纯的插入排序更高效，对中等数据有较好表现。

   缺点：性能不如 (O(n \log n)) 算法稳定，增量选择困难。

   代码示例（核心逻辑片段）：

```
for (int gap = n / 2; gap >= 1; gap /= 2) {
    for (int i = gap; i < n; ++i) {
        int key = arr[i], j = i;
        while (j - gap >= 0 && arr[j - gap] > key) {
            arr[j] = arr[j - gap];
            j -= gap;
        }
        arr[j] = key;
    }
}
```

5. **快速排序（Quick Sort）**：
   原理：
   快速排序选取枢轴元素，通过一次分区操作将数组分为小于枢轴和大于枢轴的两部分，然后递归地对左右子序列进行排序。平均情况下效率较高。

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

   优点：平均性能优异，常用的内部排序算法。

   缺点：最坏情况较差，可随机化枢轴改善；不稳定。

   代码示例（核心逻辑片段）：

```
int Partition(int arr[], int low, int high) {
```

```
    int pivot = arr[high];
    int i = low - 1;
    for (int j = low; j < high; ++j) {
        if (arr[j] < pivot) {
            i++;
            Swap(arr[i], arr[j]);
        }
    }
    Swap(arr[i + 1], arr[high]);
    return i + 1;
}

void QuickSort(int arr[], int low, int high) {
    if (low < high) {
        int pivot_index = Partition(arr, low, high);
        QuickSort(arr, low, pivot_index - 1);
        QuickSort(arr, pivot_index + 1, high);
    }
}
```

6. **堆排序（Heap Sort）**：

   原理：
   利用最大堆选出最大元素并放到数组末尾，然后排除最后位置重复构建最大堆，使得每轮都能将当前未排序部分的最大值移到正确位置。

   时间复杂度：始终 $O(n \log n)$。

   优点：稳定的上界性能，无需额外很大空间。

   缺点：常数因子较大，实际中略慢于快排，不稳定。

   代码示例（核心逻辑片段）：

```
void Heapify(int arr[], int n, int i) {
    int largest = i;
    int left = 2*i + 1;
    int right = 2*i + 2;

    if (left < n && arr[left] > arr[largest]) largest = left;
    if (right < n && arr[right] > arr[largest]) largest = right;

    if (largest != i) {
        Swap(arr[i], arr[largest]);
        Heapify(arr, n, largest);
    }
}

void HeapSort(int arr[], int n) {
    for (int i = n/2 - 1; i >= 0; --i) {
        Heapify(arr, n, i);
    }
    for (int i = n-1; i > 0; --i) {
```

```
        Swap(arr[0], arr[i]);
        Heapify(arr, i, 0);
    }
}
```

7. **归并排序（Merge Sort）**：
   原理：
   将数组分成两半，分别递归排序，再将两个有序子序列合并成一个有序序列。分治思想保证了较好的时间复杂度。

   时间复杂度：始终 $O(n \log n)$。

   优点：稳定、最坏情况也高效，适用于超大规模数据。

   缺点：需要额外空间。

   代码示例（核心逻辑片段）：

```
void Merge(int arr[], int left, int mid, int right) {
    // 将arr[left..mid]与arr[mid+1..right]合并
    // 略...
}

void MergeSort(int arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left)/2;
        MergeSort(arr, left, mid);
        MergeSort(arr, mid+1, right);
        Merge(arr, left, mid, right);
    }
}
```

8. **基数排序（Radix Sort）**：
   原理：
   基数排序按位（从最低位到最高位）对数据进行多次分配和收集（利用计数排序等稳定排序做子过程），最终达到整体有序。

   时间复杂度：$O(kn)$，其中 k 为最大数的位数。

   优点：对整数近似线性时间，高效。

   缺点：依赖数据特征，需要额外空间。

   代码示例（核心逻辑片段）：

```
void CountingSortForRadix(int arr[], int n, int exp) {
    int output[n];
    int count[10] = {0};

    for (int i = 0; i < n; ++i)
        count[(arr[i]/exp) % 10]++;
```

```
    for (int i = 1; i < 10; ++i)
        count[i] += count[i-1];

    for (int i = n-1; i >= 0; --i) {
        int index = (arr[i]/exp) % 10;
        output[count[index]-1] = arr[i];
        count[index]--;
    }

    for (int i = 0; i < n; ++i)
        arr[i] = output[i];
}

void RadixSort(int arr[], int n) {
    int max_val = GetMax(arr, n);
    for (int exp = 1; max_val/exp > 0; exp *= 10) {
        CountingSortForRadix(arr, n, exp);
    }
}
```

通过上述代码片段与原理介绍，读者可以对每种排序算法的实现有基本认识。这些代码展示了各个算法的核心思想和关键步骤，在实际工程中可根据项目需求和数据特性加以优化和扩展。

5.1. 性能统计模块

利用 `clock()` 函数测量排序前后CPU时间差，以秒为单位输出。
在排序过程中通过变量计数比较或交换操作的次数，输出基础操作次数。
用户可对比各算法在不同数据规模下的耗时与操作次数。

5.2. 用户交互模块

命令行交互界面：

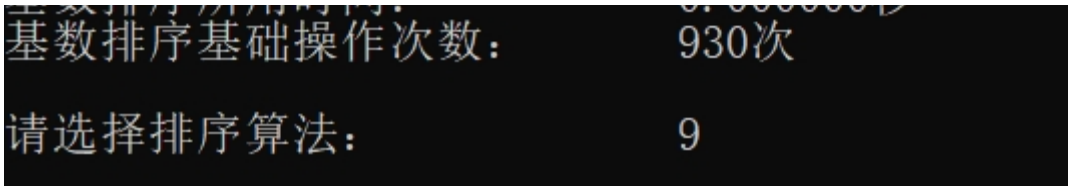
- 用户输入随机数数量。
- 用户选择排序算法（1～9中之一）。
- 程序输出该算法的时间与操作次数。
- 用户可多次选择不同算法进行对比，或选择"9"退出程序。

6. 实验结果与分析

在实际运行中，可以测试以下数据规模：100、1000、10000、100000，并收集各算法的时间和比较次数统计。下表给出示例（数据仅为参考，实际结果依赖运行环境和随机数据特性）：

- 数据规模100

```
-----------------------------------
|            排序算法比较          |
 ---------------------------------
|            1--冒泡排序           |
|            2--选择排序           |
|            3--直接插入排序       |
|            4--希尔排序           |
|            5--快速排序           |
|            6--堆排序             |
|            7--归并排序           |
|            8--基数排序           |
|            9--退出程序           |
-----------------------------------

请输入要产生的随机数的个数：100

请选择排序算法：                  1
冒泡排序所用时间：                0.000000秒
冒泡排序基础操作次数：            4950次

请选择排序算法：                  2
选择排序所用时间：                0.001000秒
选择排序基础操作次数：            4950次

请选择排序算法：                  3
直接插入排序所用时间：            0.001000秒
直接插入排序基础操作次数：        2549次

请选择排序算法：                  4
希尔排序所用时间：                0.000000秒
希尔排序基础操作次数：            489次

请选择排序算法：                  5
快速排序所用时间：                0.000000秒
快速排序基础操作次数：            671次

请选择排序算法：                  6
堆排序所用时间：                  0.000000秒
堆排序基础操作次数：              478次

请选择排序算法：                  7
归并排序所用时间：                0.000000秒
归并排序基础操作次数：            1344次

请选择排序算法：                  8
基数排序所用时间：                0.000000秒
```



- 数据规模1000

```
-----------------------------------
|            排序算法比较          |
 ---------------------------------
|            1--冒泡排序           |
|            2--选择排序           |
|            3--直接插入排序       |
|            4--希尔排序           |
|            5--快速排序           |
|            6--堆排序             |
|            7--归并排序           |
|            8--基数排序           |
|            9--退出程序           |
-----------------------------------

请输入要产生的随机数的个数：1000

请选择排序算法：                  1
冒泡排序所用时间：                0.022000秒
冒泡排序基础操作次数：            499500次

请选择排序算法：                  2
选择排序所用时间：                0.013000秒
选择排序基础操作次数：            499500次

请选择排序算法：                  3
直接插入排序所用时间：            0.009000秒
直接插入排序基础操作次数：        243107次

请选择排序算法：                  4
希尔排序所用时间：                0.000000秒
希尔排序基础操作次数：            7184次

请选择排序算法：                  5
快速排序所用时间：                0.000000秒
快速排序基础操作次数：            10488次

请选择排序算法：                  6
堆排序所用时间：                  0.001000秒
堆排序基础操作次数：              8070次

请选择排序算法：                  7
归并排序所用时间：                0.002000秒
归并排序基础操作次数：            19952次

请选择排序算法：                  8
基数排序所用时间：                0.000000秒
```

- 数据规模10000

```
-----------------------------------
|            排序算法比较          |
 ---------------------------------
|            1--冒泡排序           |
|            2--选择排序           |
|            3--直接插入排序       |
|            4--希尔排序           |
|            5--快速排序           |
|            6--堆排序             |
|            7--归并排序           |
|            8--基数排序           |
|            9--退出程序           |
-----------------------------------

请输入要产生的随机数的个数：10000

请选择排序算法：                  1
冒泡排序所用时间：                2.242000秒
冒泡排序基础操作次数：            49995000次

请选择排序算法：                  2
选择排序所用时间：                1.227000秒
选择排序基础操作次数：            49995000次

请选择排序算法：
3
直接插入排序所用时间：            0.898000秒
直接插入排序基础操作次数：        25117185次

请选择排序算法：                  4
希尔排序所用时间：                0.010000秒
希尔排序基础操作次数：            146318次

请选择排序算法：                  5
快速排序所用时间：                0.006000秒
快速排序基础操作次数：            153022次

请选择排序算法：                  6
堆排序所用时间：                  0.013000秒
堆排序基础操作次数：              114104次

请选择排序算法：                  7
归并排序所用时间：                0.016000秒
归并排序基础操作次数：            267232次

请选择排序算法：                  8
```

- 数据规模100000

```
------------------------------------
|            排序算法比较           |
 ----------------------------------
|            1--冒泡排序            |
|            2--选择排序            |
|            3--直接插入排序        |
|            4--希尔排序            |
|            5--快速排序            |
|            6--堆排序              |
|            7--归并排序            |
|            8--基数排序            |
|            9--退出程序            |
------------------------------------

请输入要产生的随机数的个数：100000

请选择排序算法：                  1
冒泡排序所用时间：                259.310000秒
冒泡排序基础操作次数：            4999950000次

请选择排序算法：                  2
选择排序所用时间：                134.372000秒
选择排序基础操作次数：            4999950000次

请选择排序算法：                  3
直接插入排序所用时间：            103.489000秒
直接插入排序基础操作次数：        2500562929次

请选择排序算法：                  4
希尔排序所用时间：                0.179000秒
希尔排序基础操作次数：            2973100次

请选择排序算法：                  5
快速排序所用时间：                0.073000秒
快速排序基础操作次数：            2140532次

请选择排序算法：                  6
堆排序所用时间：                  0.183000秒
堆排序基础操作次数：              1475100次

请选择排序算法：                  7
归并排序所用时间：                0.212000秒
归并排序基础操作次数：            3337856次

请选择排序算法：                  8
基数排序所用时间：                0.069000秒
基数排序基础操作次数              1800060次
```

基数排序基础操作次数：　　1800000次

- 数据规模100000

```
------------------------------------
|            排序算法比较           |
 ----------------------------------
|            1--冒泡排序            |
|            2--选择排序            |
|            3--直接插入排序        |
|            4--希尔排序            |
|            5--快速排序            |
|            6--堆排序              |
|            7--归并排序            |
|            8--基数排序            |
|            9--退出程序            |
------------------------------------

请输入要产生的随机数的个数：100000

请选择排序算法：                  1
冒泡排序所用时间：                259.310000秒
冒泡排序基础操作次数：            4999950000次

请选择排序算法：                  2
选择排序所用时间：                134.372000秒
选择排序基础操作次数：            4999950000次

请选择排序算法：                  3
直接插入排序所用时间：            103.489000秒
直接插入排序基础操作次数：        2500562929次

请选择排序算法：                  4
希尔排序所用时间：                0.179000秒
希尔排序基础操作次数：            2973100次

请选择排序算法：                  5
快速排序所用时间：                0.073000秒
快速排序基础操作次数：            2140532次

请选择排序算法：                  6
堆排序所用时间：                  0.183000秒
堆排序基础操作次数：              1475100次

请选择排序算法：                  7
归并排序所用时间：                0.212000秒
归并排序基础操作次数：            3337856次

请选择排序算法：                  8
基数排序所用时间：                0.069000秒
基数排序基础操作次数              1800060次
```

基数排序基础操作次数：　　1800000次

从数据可以看出：

- 对于大规模数据（如10000以上），**冒泡、选择、直接插入**等 $O(n^2)$ 算法耗时明显过长，不适合大数据集。
- **希尔排序**比插入排序快，但仍不及 $O(n \log n)$ 的算法。
- **快速排序**、**堆排序**、**归并排序**在大数据集下表现均良好，其中快速排序平均最快，但最坏情况可能较差；归并排序和堆排序表现稳定。
- **基数排序**在对整数数据排序时有特别的优势，可能在某些场景下表现优于比较排序算法。
- 综合来看，**快速排序**的效率最高。

7. 存储开销与时间开销分析

- **空间开销**：
  - 冒泡、选择、直接插入、希尔、快速、堆排序基本为原地排序，额外空间需求较少。
  - 归并排序需要分配额外的辅助数组空间，空间复杂度为 $O(n)$。
  - 基数排序需要额外的计数数组和输出数组，对空间要求也较高。
- **时间开销**：
  随着数据规模的增大， $O(n^2)$ 算法（冒泡、选择、直接插入）耗时迅速上升。
  $O(n \log n)$ 算法（快速、堆、归并）在大数据下更具可扩展性。
  基数排序的时间复杂度与数据的位数有关，对特定数据类型（如整数）可能非常高效。

8. 各算法的优缺点总结

- **冒泡排序**：实现简单，但性能极差，不建议用于大数据。
- **选择排序**：简单易实现，交换次数少，但仍是 $O(n^2)$，大数据表现不佳。
- **直接插入排序**：对小规模或部分有序数据有较好的表现，大数据仍不理想。
- **希尔排序**：比插入排序快，对中等规模数据有效，但不如 $O(n \log n)$ 算法。
- **快速排序**：平均性能优异，适合大数据集，最广泛使用的内部排序算法之一。
- **堆排序**：性能稳定，始终 $O(n \log n)$，最坏情况也有保证。
- **归并排序**： $O(n \log n)$ 且性能稳定，需要额外空间，对超大数据稳定高效。
- **基数排序**：非比较排序，对于整数数据可能优于比较类排序，但依赖数据特性和额外空间。

9. 结论

通过本项目的实验对比可知：

- 对小数据量：任意算法都能较快完成，但插入、希尔等简单改进算法较为省事。

- 对大数据量：应首选快速、堆或归并等 $O(n \log n)$ 算法，基数排序在特定数据类型下表现突出。

10. 心得体会

通过本次项目的开发与测试，我对多种经典排序算法有了更加深刻、直观的理解。以下是我在这个过程中获得的一些体会：

1. **理论与实践结合**：
   在数据结构与算法的学习中，我们往往先学习算法的原理和时间复杂度。然而，只有将这些算法付诸实践，运行在真实的数据上，才能更直观地体会到算法的优缺点和效率差异。通过本项目的实验测试，不同规模的数据、不同算法的耗时和比较次数等指标让我对时间复杂度与实际性能的联系有了更为切实的认识。
2. **选择算法要考虑场景**：
   学习算法的最大目的在于解决实际问题。实验结果表明，并非所有算法都适用于任意数据规模和数据特征。在小数据集下，简单的算法（如直接插入、冒泡、选择）或许已经足够，且实现简单；但对于大规模数据，快速、堆、归并这样的 $O(n \log n)$ 算法才更符合实际需求。此外，基数排序在特定场景（如整数序列）下展现出的高效性能更让我明白，算法选择需要在理解数据特性与需求的基础上进行权衡。
3. **优化和变通**：
   快速排序的最坏情况为 $O(n^2)$，但通过随机选取枢轴或三数取中等变通方法，可以大大减少这种情况的出现。归并排序虽然需要额外空间，但却保证了最坏情况下的高效性和稳定性。堆排序和希尔排序的性能则体现了在特定策略（如构建最大堆或使用不同增量序列）下的优化潜力。实践让我更明确地认识到，算法的精髓在于不断优化和调优。
4. **稳定性和内存开销的重要性**：
   实际应用中有时并不仅仅关注速度。一些场景下，对稳定性的要求较高（如涉及数据库记录排序时需要保持相同关键字的记录顺序），则归并或直接插入排序等稳定算法可能更具吸引力。此外，内存开销也是选择算法时的重要因素。归并和基数排序需要额外的存储空间，而快速排序、堆排序则是原地进行，对内存资源有限的场合更有优势。

通过本项目的实践，我深刻地感受到，算法的选择和优化不仅仅是追求时间复杂度的最佳值，更是根据实际问题的规模、数据特征、硬件条件和稳定性要求等多维度因素来做出合理决策。这样有针对性的分析和测试能力，在未来的工程实践中将会非常有用。