



**实验报告： 三种排序算法的设计与分析**

**学院：计算机学院（国家示范性软件学院）**

**专业： 计算机科学与技术**

**班级： 2022211305**

**学号： 2022211683**

**姓名： 张晨阳**

**2024年11月2号**

**目录**

[1. 实验概述 1](#_Toc181478521)

[1.1. 实验目的 1](#_Toc181478522)

[1.2. 实验内容及要求 1](#_Toc181478523)

[1.3. 实验环境 1](#_Toc181478524)

[2. 算法设计与实现 2](#_Toc181478525)

[2.1. 堆排序 2](#_Toc181478526)

[2.1.1. 算法说明 2](#_Toc181478527)

[2.1.2. 实现思路与代码结构 2](#_Toc181478528)

[2.2. 归并排序 4](#_Toc181478529)

[2.2.1. 算法说明 4](#_Toc181478530)

[2.2.2. 实现思路与代码结构 4](#_Toc181478531)

[2.3. 快速排序 6](#_Toc181478532)

[2.3.1. 算法说明 6](#_Toc181478533)

[2.3.2. 实现思路与代码结构 6](#_Toc181478534)

[3. 测试设计 8](#_Toc181478535)

[3.1. 测试数据设计 8](#_Toc181478536)

[3.1.1. Small Random Array 8](#_Toc181478537)

[3.1.2. Small Sorted Array 8](#_Toc181478538)

[3.1.3. Small Reversed Array 8](#_Toc181478539)

[3.1.4. Large Random Array 8](#_Toc181478540)

[3.1.5. Large Sorted Array 9](#_Toc181478541)

[3.1.6. Large Reversed Array 9](#_Toc181478542)

[3.1.7. Repeated Array 9](#_Toc181478543)

[3.2. 测试程序说明 10](#_Toc181478544)

[3.3. 测试实例结果 11](#_Toc181478545)

[4. 结果分析与评估 12](#_Toc181478546)

[4.1. 时间复杂度分析 12](#_Toc181478547)

[4.2. 操作次数与性能评估 13](#_Toc181478548)

[4.3. 特殊表现分析 15](#_Toc181478549)

[4.3.1. 快速排序的退化 15](#_Toc181478550)

[4.3.2. 堆排序在重复数据上的特殊表现 15](#_Toc181478551)

[5. 总结心得 16](#_Toc181478552)

# 实验概述

## 实验目的

* 理解分治法的策略，掌握基于递归的分治算法的实现方法；
* 掌握基于分治法的合并排序、快速排序的实现方法；
* 理解并掌握在渐进意义下的算法复杂性的评价方法；
* 掌握算法测试的基本流程。

## 实验内容及要求

1. 算法的设计与实现

设计并实现堆排序、归并排序（合并排序）、快速排序算法，通过比较三种排序算法在不同数据量的情况下所需的移动次数、比较次数，分析算法在最差情况、平均情况下的算法复杂度。

1. 测试要求

设计测试数据集，编写测试程序，用于测试：

a) 正确性：所实现的三种算法的正确性；

b) 算法复杂性：三种排序算法中，设计测试数据集，评价各个算法在算法复杂性上的表现；（最差情况、平均情况）

c) 效率：在三种排序算法中，设计测试数据集，评价各个算法中比较的频率，移动的频率。

## 实验环境

* Visual Studio Code
* C++ 17

# 算法设计与实现

三种算法的效率、复杂度将在第4节详细展开说明，此处主要介绍算法核心思想。

## 堆排序

### 算法说明

堆排序是一种基于堆数据结构的排序算法，属于选择排序的一种，具有 的时间复杂度。堆排序通过将数组调整为一个**最大堆或最小堆**，不断将堆顶（最值）元素与末尾元素交换，实现排序的过程。

堆排序的主要步骤包括：

1. **构建最大堆**：从最后一个非叶子节点开始，逐步将子堆调整为最大堆，使每个父节点的值不小于其子节点的值。
2. **排序**：将堆顶元素（最大值）与当前未排序部分的末尾元素交换，将最大值“移出”堆，并对剩余部分重新调整为最大堆，重复这一过程直到所有元素有序。

堆排序的特点在于其空间复杂度为 ，因为排序在原数组上进行，不需要额外的存储空间。

### 实现思路与代码结构

在实现堆排序时，核心部分包括两个函数：heapSort 和 heapify。其中，heapSort 用于构建最大堆并执行排序，heapify 用于维护堆的性质。

**heapSort 函数**：该函数分为两个阶段：

1. **构建最大堆**：从第一个非叶子节点( )开始，向前遍历每个节点，对每个节点调用 heapify，确保数组形成一个最大堆。
2. **排序过程**：从堆顶元素开始，每次将其与未排序部分的末尾元素交换，并调用 heapify 调整堆结构，直到整个数组有序。

**heapify 函数**：heapify 是递归函数，用于调整堆的结构，使其符合最大堆性质。其实现流程如下：

1. 设定当前节点的索引为 i，则其左孩子为，右孩子为 。
2. 将当前节点与其左右孩子进行比较，找出三者中最大的元素索引 largest。
3. 若最大元素不在当前节点位置 i，则将当前节点与 largest 位置的元素交换，并递归调用 heapify 对剩余部分调整，直到堆的性质完全满足。

具体代码实现如下：

1. void Algorithms::heapSort(std::vector<int>& data, int& comparisons, int& moves){

2.     int n = data.size();

3.     // 构建最大堆

4.     for (int i = n / 2 - 1; i >= 0; i--)

5.         heapify(data, n, i, comparisons, moves);

6.

7.     for (int i = n - 1; i >= 0; i--) {

8.         std::swap(data[0], data[i]);

9.         moves++;

10.         heapify(data, i, 0, comparisons, moves);

11.     }

12. }

13.

14. void Algorithms::heapify(std::vector<int>& data, int n, int i, int& comparisons, int& moves) {

15.     int largest = i;  // 最大元素索引

16.     int left = 2 \* i + 1;

17.     int right = 2 \* i + 2;

18.

19.     if (left < n && data[left] > data[largest]) {

20.         largest = left;

21.         comparisons++;

22.     }

23.     if (right < n && data[right] > data[largest]) {

24.         largest = right;

25.         comparisons++;

26.     }

27.     if (largest != i) {

28.         std::swap(data[i], data[largest]);

29.         moves++;

30.         heapify(data, n, largest, comparisons, moves);  // 递归调整剩下的堆

31.     }

32. }

## 归并排序

### 算法说明

归并排序是一种**分治法**排序算法，具有稳定的时间复杂度 。

其基本思想是将数组分成两个子数组，分别对每个子数组进行排序，然后合并成一个有序数组。归并排序是一种稳定排序算法，即相同的元素在排序后相对位置不变。

归并排序的主要步骤包括：

1. **拆分**：递归地将数组分成两部分，直到每部分只包含一个元素。
2. **合并**：在递归返回的过程中，将两个有序的子数组合并成一个更大的有序数组。

归并排序在排序时需要额外的存储空间用于存放临时数据，但其时间复杂度始终是 ，在大多数情况下表现较为稳定。

### 实现思路与代码结构

归并排序的实现分为两个部分：mergeSort 和 merge。其中 mergeSort 实现分割和递归，merge 实现两个有序子数组的合并。

**mergeSort 函数**：递归函数，将数组分割为两个子数组分别排序。

1. 判断是否满足递归基准条件 left < right，当数组仅有一个元素或为空时，不再分割。
2. 计算中间位置 mid，将数组分为 [left, mid] 和 [mid + 1, right] 两部分。
3. 递归调用 mergeSort 对左半部分和右半部分分别排序。
4. 调用 merge 函数，将 [left, mid] 和 [mid + 1, right] 两部分合并成一个有序数组。

**merge 函数**：合并两个已排序的子数组，使其成为一个有序数组。

1. 分别创建两个临时数组 L 和 R，存放左子数组和右子数组的元素。
2. 使用两个指针 i 和 j 遍历 L 和 R，另一个指针 k 用于在 data 中放置合并结果。
3. 比较 L[i] 和 R[j]，将较小的元素放入 data[k] 中，并移动指针 i 或 j。
4. 若 L 或 R 中仍有剩余元素，直接将其拷贝到 data 中。

具体代码实现如下：

1. void Algorithms::mergeSort(std::vector<int>& data, int left, int right, int& comparisons, int& moves) {

2.     if (left < right) {

3.         int mid = left + (right - left) / 2;

4.         mergeSort(data, left, mid, comparisons, moves);

5.         mergeSort(data, mid + 1, right, comparisons, moves);

6.         merge(data, left, mid, right, comparisons, moves);

7.     }

8. }

9.

10. void Algorithms::merge(std::vector<int>& data, int left, int mid, int right, int& comparisons, int& moves) {

11.     int n1 = mid - left + 1;

12.     int n2 = right - mid;

13.     std::vector<int> L(n1), R(n2);

14.     for (int i = 0; i < n1; i++)

15.         L[i] = data[left + i];

16.     for (int j = 0; j < n2; j++)

17.         R[j] = data[mid + 1 + j];

18.

19.     int i = 0, j = 0, k = left;

20.     while (i < n1 && j < n2) {

21.         comparisons++;

22.         if (L[i] <= R[j]) {

23.             data[k++] = L[i++];

24.         } else {

25.             data[k++] = R[j++];

26.         }

27.         moves++;

28.     }

29.

30.     while (i < n1) {

31.         data[k++] = L[i++];

32.         moves++;

33.     }

34.     while (j < n2) {

35.         data[k++] = R[j++];

36.         moves++;

37.     }

38. }

## 快速排序

### 算法说明

快速排序是一种基于**分治法**的高效排序算法，平均时间复杂度为 ，在最坏情况下（如输入数据为有序时）为。

其基本思想是选择一个基准元素（pivot），通过**分区**操作将数组分成两部分，使得基准左侧的元素都小于基准，右侧的元素都大于或等于基准。然后对这两部分分别递归排序。

快速排序的主要步骤包括：

1. **选择基准**：通常选择数组的最后一个元素作为基准。
2. **分区**：将数组分成左右两部分，使得左侧元素小于基准，右侧元素大于或等于基准。
3. **递归排序**：分别对左、右两部分递归执行快速排序，直到所有部分有序。

快速排序具有较高的效率，但在最坏情况下（如选择的基准元素始终为最大或最小值）性能下降，因此可以通过选择中位数或随机基准来改善。

### 实现思路与代码结构

快速排序的实现分为两个部分：quickSort 和 partition。其中 quickSort 实现递归分治，partition 实现基于基准的分区。

**quickSort 函数**：快速排序的递归函数，用于分治操作。

1. 判断递归基准条件 low < high，当子数组仅含一个或零个元素时，自动视为有序。
2. 调用 partition 函数，对 [low, high] 区间进行分区，获取基准位置 pi。
3. 递归调用 quickSort 对基准左侧 [low, pi - 1] 和右侧 [pi + 1, high] 分区排序。

**partition 函数**：分区函数，将数组划分为两部分。

1. 选择区间最后一个元素 data[high] 作为基准。
2. 使用 i 指针表示小于基准的元素的下一个位置，j 指针用于遍历区间 [low, high - 1]。
3. 若 data[j] < pivot，将其与 data[i] 交换位置，并递增 i，同时增加移动计数。
4. 遍历结束后，将基准元素放置到正确位置，即与 data[i] 交换，返回 i 作为基准位置。

具体代码实现如下：

1. // 快速排序实现

2. void Algorithms::quickSort(std::vector<int>& data, int low, int high, int& comparisons, int& moves) {

3.     if (low < high) {

4.         int pi = partition(data, low, high, comparisons, moves);

5.         quickSort(data, low, pi - 1, comparisons, moves);

6.         quickSort(data, pi + 1, high, comparisons, moves);

7.     }

8. }

9.

10. // 分区函数

11. int Algorithms::partition(std::vector<int>& data, int low, int high, int& comparisons, int& moves) {

12.     int pivot = data[high];  // 基准

13.     int i = low;

14.

15.     for (int j = low; j < high; j++) {

16.         comparisons++;

17.         if (data[j] < pivot) {

18.             std::swap(data[i++], data[j]);

19.             moves++;

20.         }

21.     }

22.     std::swap(data[i], data[high]);

23.     moves++;

24.     return i;  // 返回分区点

25. }

# 测试设计

## 测试数据设计

在排序算法中，使用不同类型的测试数据可以更全面地评估算法的性能效率。

为此，本实验设计了 7 种测试数据，涵盖了不同规模和顺序特性的数组。每种测试数据的设计都有其特定的意义和测试价值，具体如下：

### Small Random Array

**内容描述**：包含 100 个随机值，取值范围较大。

**目的**：测试排序算法在小规模、无序数据下的表现。这种数据类型可以快速验证算法的基本功能和正确性。用于进行初步调试和性能对比。

### Small Sorted Array

**内容描述**：包含 100 个元素，且已按升序排列。

**目的**：测试算法在输入已排序数据时的表现。对于某些排序算法，如快速排序，已排序数据可能导致最坏情况，但对其他排序算法（如归并排序）则影响较小。

### Small Reversed Array

**内容描述**：包含 100 个元素，按降序排列。

**目的**：测试排序算法在输入完全逆序数据时的性能。逆序数据对于某些排序算法（如插入排序、快速排序）可能会导致更多的移动操作和比较次数。

### Large Random Array

**内容描述**：包含 10,000 个随机值，取值范围较大。

**目的**：测试排序算法在大规模、无序数据下的表现。大规模数据能够显现出不同算法的复杂度优势和劣势，从而验证算法的时间复杂度。

### Large Sorted Array

**内容描述**：包含 10,000 个元素，按升序排列。

**目的**：测试算法在大规模已排序数据上的表现，以验证算法在处理大规模最佳情况输入时的性能。

### Large Reversed Array

**内容描述**：包含 10,000 个元素，按降序排列。

**目的**：测试排序算法在大规模逆序数据上的表现。逆序数据对一些排序算法而言可能是最坏情况，例如快速排序需要多次交换才能完成排序。

### Repeated Array

**内容描述**：包含 100 个相同的元素（如全为 1）。

**目的**：测试算法在所有元素相同的情况下的表现。对于排序算法而言，这是一种特殊的情况，数据已经具有部分有序性。可以验证算法在极端情况下的稳定性和效率，观察是否有不必要的操作（如重复交换相同的元素），并确保算法在重复数据下的稳定性。

## 测试程序说明

为了全面验证排序算法在不同数据类型上的性能和正确性，我编写了测试数据生成程序 TestUtils，该程序提供了多种数据生成和验证工具函数。这些函数负责生成各种测试数据，并验证排序结果的正确性。

具体包括以下几个部分（具体代码见TestUtils.cpp）：

1. **随机数组生成函数 generateRandomArray**

**功能：**生成一个大小为 size 的随机数组，其中每个元素为 0 到 maxValue 之间的整数。

**实现：**使用 <random> 库生成随机数，std::uniform\_int\_distribution<> dis(0, maxValue) 设置整数范围，std::generate 填充数组。

1. **有序数组生成函数 generateSortedArray**

**功能：**生成一个大小为 size 的升序数组，元素值为从 0 开始的自然数序列。

**实现：**使用 std::iota 填充数组，将 data[i] 设置为 i。

1. **逆序数组生成函数 generateReversedArray**

**功能：**生成一个大小为 size 的降序数组，元素值从 size - 1 开始逐渐递减至 0。

**实现：**使用 std::iota 结合反向迭代器 rbegin 和 rend 生成降序排列的数组。

1. **重复数组生成函数 generateRepeatedArray**

**功能：**生成一个大小为 size 的数组，其中每个元素的值都相同，等于 value。

**实现：**使用 std::vector<int> data(size, value) 初始化一个所有元素均为 value 的数组。

1. **排序结果验证函数 validateSort**

**功能：**检查数组是否已按升序排列。

**实现：**使用标准库函数 std::is\_sorted，返回布尔值 true 表示数组有序，false 表示数组未排序。

## 测试实例结果

在main.cpp中，生成测试数据实例代码如下：

1.std::make\_pair("Small Random Array", TestUtils::generateRandomArray(100,100000))

2.std::make\_pair("Small Sorted Array", TestUtils::generateSortedArray(100))

3.std::make\_pair("Small Reversed Array", TestUtils::generateReversedArray(100))

4.std::make\_pair("Large Random Array",TestUtils::generateRandomArray(10000,100000))

5.std::make\_pair("Large Sorted Array", TestUtils::generateSortedArray(10000))

6.std::make\_pair("Large Reversed Array", TestUtils::generateReversedArray(10000))

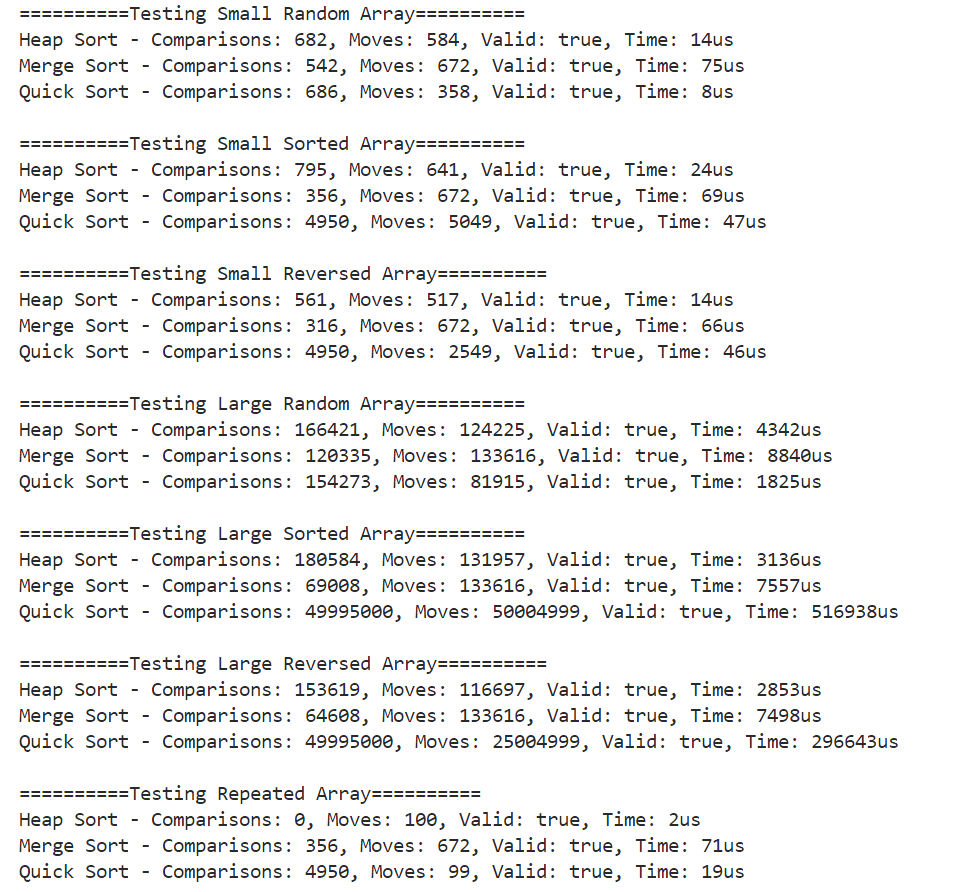
7.std::make\_pair("Repeated Array", TestUtils::generateRepeatedArray(100, 1))

编译文件及测试命令如下：

> g++ -o test main.cpp .\Algorithms.cpp .\TestUtils.cpp

> ./test

测试运行结果实例如下：



# 结果分析与评估

在本实验中，通过多种数据类型测试了堆排序、归并排序和快速排序三种算法的表现，比较它们的时间复杂度、操作次数（包括比较和移动次数），并分析在不同输入情况下的效率和性能。

以下是详细的分析和对比结果。

## 时间复杂度分析

1. **堆排序：**

**平均时间复杂度**：，在所有测试数据上表现稳定。

**最佳情况**：，在重复数组上时间较短。

**最坏情况**：，在大规模数据上时间逐渐增加，但始终较为稳定。

**说明**：堆排序的性能对输入数据分布不敏感，无论数据是否已排序或逆序，时间复杂度基本不变，因此适合处理大规模无序数据。

1. **归并排序：**

**平均时间复杂度**：，同样在所有测试数据上表现稳定。

**最佳情况**：，在所有数据类型上表现均匀。

**最坏情况**：，空间复杂度较高，需要额外存储空间。

**说明**：归并排序是稳定的排序算法，不受数据初始顺序的影响。适合对有序性较差的、尤其是链式结构的数据进行排序。

1. **快速排序：**

**平均时间复杂度**：，在随机数据上效率高。

**最佳情况**：，在随机数据上表现较好。

**最坏情况**：，在顺序和逆序数组上快速排序退化为最坏情况，时间显著增加。

**说明**：快速排序在随机数组上表现出色，但对已排序或逆序数据，操作次数增多，导致效率低下。通过改进基准选择方式（如三数取中法）可以减小退化风险。

## 操作次数与性能评估

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **数据类型** | **算法** | **比较次数** | **移动次数** | **时间** |
| Small Random Array | Heap Sort | 682 | 584 | 14us |
| Merge Sort | 542 | 672 | 75us |
| Quick Sort | 686 | 358 | 8us |
| Small Sorted Array | Heap Sort | 795 | 641 | 24us |
| Merge Sort | 356 | 672 | 69us |
| Quick Sort | 4950 | 5049 | 47us |
| Small Reversed Array | Heap Sort | 561 | 517 | 14us |
| Merge Sort | 316 | 672 | 66us |
| Quick Sort | 4950 | 2549 | 46us |
| Large Random Array | Heap Sort | 166421 | 124225 | 4342us |
| Merge Sort | 120335 | 133616 | 8840us |
| Quick Sort | 154273 | 81915 | 1825us |
| Large Sorted Array | Heap Sort | 180584 | 131957 | 3136us |
| Merge Sort | 69008 | 133616 | 7557us |
| Quick Sort | 49995000 | 50004999 | 516938us |
| Large Reversed Array | Heap Sort | 153619 | 116697 | 2853us |
| Merge Sort | 64608 | 133616 | 7498us |
| Quick Sort | 49995000 | 25004999 | 296643us |
| Repeated Array | Heap Sort | 0 | 100 | 2us |
| Merge Sort | 356 | 672 | 71us |
| Quick Sort | 4950 | 99 | 19us |

1. **随机数据的表现**

在随机数据测试中，快速排序耗时最少，操作次数最少，表现最佳；堆排序次之，归并排序耗时较长。

**说明**：快速排序在随机数据上性能较好，操作次数较少；堆排序时间较稳定，且不受数据分布影响；归并排序相对耗时，但依然在可接受范围内。

1. **顺序数据的表现**

快速排序在顺序数组上退化，导致比较和移动次数显著增加。

**说明**：顺序数据对快速排序影响较大，导致算法退化为 的最坏情况；堆排序和归并排序则受影响较小，表现依旧稳定。

1. **逆序数据的表现**

逆序数据的结果类似于顺序数据，快速排序出现了退化，时间显著增加。

**说明**：逆序数据对快速排序同样会引起最坏情况，归并排序和堆排序依旧表现稳定，尤其是归并排序在处理逆序数据时操作次数较少。

1. **重复数据的表现**

堆排序在处理重复数据时无比较操作，快速排序操作次数最低，归并排序略多。

**说明**：重复数据对堆排序和快速排序有利，特别是堆排序不涉及比较操作，而归并排序的操作次数稍高，因其需要递归合并过程。

## 特殊表现分析

### 快速排序的退化

快速排序在平均情况下的时间复杂度为 ，但在某些特殊数据分布下会退化为最坏情况 。

导致退化的原因主要在于**基准选择**：

* **基准选择问题：**理想情况下，基准将数组平分，使每次递归的数组规模减半，从而达到 的复杂度。
* **顺序/逆序数据：**在完全有序的数据中，若选择最左/右作为基准，分区会非常不平衡。此时，每次分区只能移除一个元素（即基准），剩余部分几乎保持完整，从而递归次，每次处理的子数组规模接近，最终复杂度退化为.
* **解决方法：**为避免退化，可以改进基准选择策略，如采用三数取中法（选择最左、中间、最右三者的中位数为基准），或随机选择基准，从而减少分区的不平衡情况，降低退化的概率。

在本实验的测试结果中，顺序和逆序数组的快速排序运行时间显著高于随机数据，验证了基准选择对快速排序性能的影响。

### 堆排序在重复数据上的特殊表现

在处理重复数据时，堆排序显示出了一些特殊的优势，这归因于其构建和调整堆的方式。

* **重复数据减少比较操作：**在构建最大堆的过程中，如果堆中存在大量相等的元素，调整堆结构时可以减少比较次数。例如，当左、右子节点值与父节点相等时，不需要进行交换和调整操作。
* **结果表现：**在本实验的“重复数组”测试中，堆排序的比较次数为 0。这是因为所有元素相等，堆排序在每次调整堆时无需比较，直接进行移动即可。
* **稳定的时间复杂度：**虽然堆排序在重复数据上表现优异，但在随机、顺序和逆序数组上的表现差异不大，这体现了堆排序的稳定性。堆排序对数据的初始排列和有序性不敏感，适合处理重复数据多、结构不规则的大规模数据集。

# 总结心得

本次实验通过实现并测试堆排序、归并排序和快速排序，深入理解了三种排序算法的设计思路、实现细节及其性能特征。我不仅掌握了分治法在排序算法中的应用，还通过测试数据设计和结果分析，进一步认识到不同排序算法在各种数据分布下的优劣。

在实现三种排序算法时，我体验了递归和迭代两种不同的编程思想。递归实现虽然代码简洁，但在实际应用中需要关注其对栈空间的需求和递归深度的限制。

尤其是快速排序的性能极大地依赖基准选择策略，通过改进基准选择，可以有效减小顺序和逆序数据上的退化风险。这次实验使我深刻认识到，优化排序算法需要平衡理论复杂度和实际实现中的细节。

除此之外，通过设计小型随机数组、大型逆序数组、重复数组等多种数据分布，我系统地分析了每种算法在不同场景中的表现。

实验结果验证了经典理论：快速排序在随机数组上表现优异，但在有序数据上会退化；堆排序的复杂度稳定，对数据分布不敏感；归并排序虽然需要额外空间，但在复杂度和稳定性方面表现出色。

此外，重复数据在堆排序中减少了比较操作，也展示出堆排序在重复数据上的特殊优势，这为我理解数据分布如何影响算法提供了实际的案例。

通过对比不同算法的测试结果，我深刻理解了编程实践中的算法优化和性能分析方法，为今后的学习和开发奠定了良好的基础。

总的来说，本次实验的整体过程涵盖了算法设计、实现、测试和分析的完整流程，使我对排序算法的核心思想有了更系统的理解。

无论是堆排序、归并排序还是快速排序，每种算法都各有优劣，实际应用中需根据数据特征和应用场景合理选择。