数据结构课程设计

项目说明文档

8种排序算法的比较案例

|  |  |
| --- | --- |
| 作者姓名： | 杨烜赫 |
| 学 号： | 2252709 |
| 指导教师： | 张 颖 |
| 学院专业： | 软件学院 软件工程 |



同济大学

Tongji University

1. **题目简介**

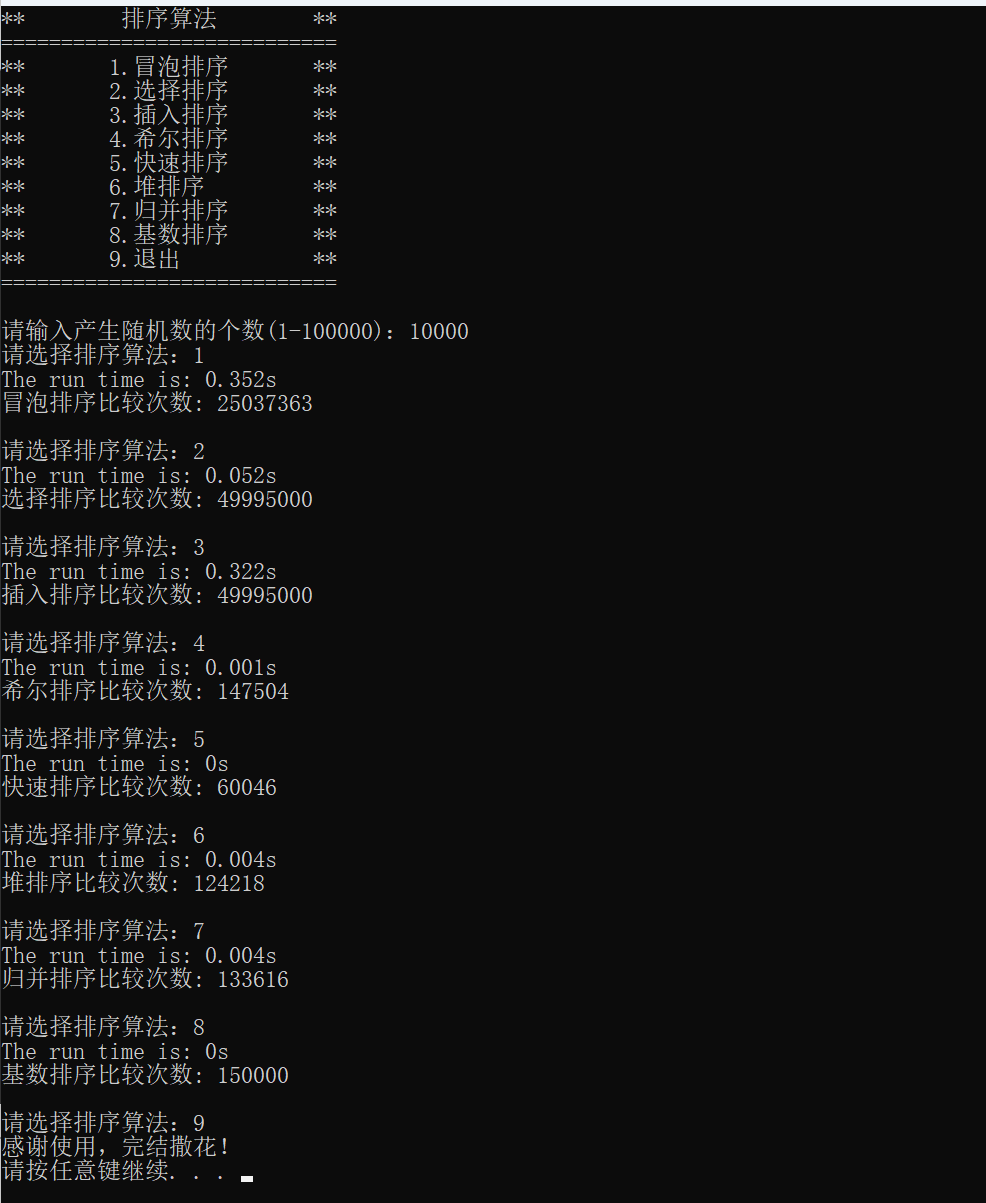
**1.实验背景**

随机函数产生10000个随机数，用快速排序，直接插入排序，冒泡排序，选择排序的排序方法排序，并统计每种排序所花费的排序时间和交换次数。其中，随机数的个数由用户定义，系统产生随机书。并且显示他们的比较次数。

**2.实验功能**

使用8种不同的比较函数来对产生的随机数进行比较。

**3.项目示例**

****

1. **项目设计**

**1.数据结构设计**

数组：用于存储待排序的数据。

计时器：使用clock\_t类型变量start\_time和end\_time来记录排序操作的开始和结束时间。

**2.类结构设计**

**类 sort**

作用：封装了不同的排序算法。

**成员变量：**

long long int num：用于记录排序过程中的比较或交换次数。

**成员函数：**

void bubblesort(int arr[], int start, int end)：实现冒泡排序。

void select\_sort(int arr[], int start, int end)：实现选择排序。

void insert\_sort(int arr[], int start, int end)：实现插入排序。

void shell\_sort(int arr[], int start, int end)：实现希尔排序。

void quick\_sort(int array[], int start, int finish)：实现快速排序。

void heapify(int array[], int start, int i, int n)：辅助函数，用于堆排序中调整堆。

void heap\_sort(int array[], int start, int finish)：实现堆排序。

void merge(int array[], int const left, int const mid, int const right)：辅助函数，用于归并排序中合并数组。

void merge\_sort(int array[], int begin, int end)：实现归并排序。

void custom\_radix\_sort\_step(int data[], int start\_idx, int end\_idx, int digit)：辅助函数，用于基数排序中按位排序。

void radix\_sort(int a[], int begin, int end)：实现基数排序。

**3.成员与操作设计**

**排序操作**：main函数中包含用户界面，用户可以选择不同的排序算法进行排序操作。

**数组初始化**：在每次选择排序算法前，使用随机数生成器填充数组。

**计时操作**：每次排序操作前后记录时间，计算排序耗时。

**用户输入验证**：确保用户输入有效的数组大小和排序算法选项。

**排序效率统计**：每种排序算法执行后输出所需时间和比较/交换次数。

## 4.算法设计

本程序的算法设计涵盖了多种经典的排序算法，包括冒泡排序、选择排序、插入排序、希尔排序、快速排序、堆排序、归并排序和基数排序。每种排序算法都被封装在sort类中，以成员函数的形式实现。冒泡排序通过重复遍历数组，比较并交换相邻元素来排序；选择排序逐个选出最小元素并将其放置在正确位置；插入排序通过构建有序序列，对未排序数据进行逐步插入；希尔排序是插入排序的一种高效版，通过引入间隔序列来改进；快速排序使用分治法，选择一个基准元素并围绕它进行排序；堆排序将数组视为二叉堆结构，进行上浮和下沉调整；归并排序也是一种分治法，它将数组分为两半，分别排序然后合并；基数排序则是按照低位先排序，逐步到高位的非比较排序方法。这些算法在程序中通过递归或迭代的方式实现，充分体现了各种排序策略在处理大量数据时的效率和特点。

1. **项目实现**

**3.1. 冒泡排序**

1. **void** sort::bubblesort(**int** arr[], **int** start, **int** end) {
2. num = 0; // 重置交换次数计数器
3. **for** (**int** i = end; i >= start; i--) {
4. **for** (**int** j = start; j < i; j++) {
5. **if** (arr[j] > arr[j + 1]) {
6. swap(arr[j], arr[j + 1]);
7. num++; // 仅在交换发生时增加计数
8. }
9. }
10. }
11. }

bubblesort 方法实现了冒泡排序算法，它通过重复遍历数组并交换相邻元素来实现排序。具体来说，该方法从数组的一端开始，比较相邻的元素，并在必要时交换它们，以确保每一轮遍历后最大的元素被移动到其最终位置。该过程重复进行，直到整个数组被排序。此外，该方法还计数每次发生的交换，从而提供对排序过程效率的衡量指标。

**3.2. 选择排序**

1. **void** sort::select\_sort(**int** arr[], **int** start, **int** end)
2. {
3. **for**(**int** i=start;i<end;i++)
4. {
5. **int** tem = i;
6. **for**(**int** j=i+1;j<=end;j++)
7. {
8. **if** (arr[j] < arr[tem])
9. tem = j;
10. num++;
11. }
12. swap(arr[i], arr[tem]);
13. }
14. }

select\_sort 方法实现了选择排序算法，其主要功能是通过两层循环逐步选择数组中的最小元素，并将其放置在正确的位置。在每次外层循环中，方法首先假设当前位置的元素是最小的，然后通过内层循环遍历剩余的元素，寻找实际的最小元素。一旦找到最小元素，就将其与当前位置的元素交换。这个过程从数组的起始位置开始，一直进行到数组的末尾，从而实现整个数组的排序。在此过程中，还记录了进行比较的次数，以评估算法的效率。

**3.3. 插入排序**

1. **void** sort::insert\_sort(**int** arr[], **int** start, **int** end)
2. {
3. **for** (**int** i = start + 1; i <= end; i++)
4. {
5. **for** (**int** j = i; j > start; j--)
6. {
7. // 从无序区向前比较
8. **if** (arr[j] < arr[j - 1]) swap(arr[j], arr[j - 1]);
9. num++;
10. }
11. }
12. }

insert\_sort 方法实现了插入排序算法，其功能是逐步构建有序序列。该方法从数组的第二个元素开始，将每个元素与它之前的所有元素依次比较，如果当前元素比比较的元素小，则将它们交换。这个过程重复进行，直到当前元素大于或等于它之前的元素，或者已经比较到数组的起始位置。通过这种方式，该方法逐渐将每个新元素插入到已排序部分的适当位置，从而在遍历完所有元素后完成整个数组的排序。在排序过程中，还记录了进行的比较次数，以便评估算法的效率。**3.4. 希尔排序**

1. **void** sort::shell\_sort(**int** arr[], **int** start, **int** end)
2. {
3. **int** range = (end - start + 1);
4. **for** (**int** gap = range / 2; gap > 0; gap /= 2) {
5. **for** (**int** i = start + gap; i <= end; i++) {
6. **int** temp = arr[i];
7. **int** j;
8. **for** (j = i; j >= start + gap && arr[j - gap] > temp; j -= gap) {
9. arr[j] = arr[j - gap];
10. num++;
11. }
12. arr[j] = temp;
13. }
14. }
15. }

这段代码实现了希尔排序，一种改进的插入排序算法，它通过首先比较距离较远的元素来提高排序速度。它工作的方式是首先确定一个“间隔”值，然后对数组中相隔这一间隔的元素进行插入排序，随后逐渐减少间隔值，重复排序过程，直至间隔为1，此时算法执行最后一轮普通插入排序，确保整个数组有序。这种方法的优点在于它可以快速减少大量的初期乱序，从而减少后期插入排序的工作量，提高整体效率。

**3.5. 快速排序**

1. **void** sort::quick\_sort(**int** array[], **int** start, **int** finish)
2. {
3. **if** (start < finish) {
4. **int** pivot = array[start];
5. **int** i = start, j = finish;
6. **while** (i < j) {
7. **while** (i < j && array[j] >= pivot) j--;
8. **if** (i < j) {
9. array[i++] = array[j];
10. num++;  // 交换次数增加
11. }
12. **while** (i < j && array[i] < pivot) i++;
13. **if** (i < j) {
14. array[j--] = array[i];
15. num++;  // 交换次数增加
16. }
17. }
18. array[i] = pivot;
19. num++;  // 将pivot放在其最终位置也算一次交换
21. quick\_sort(array, start, i - 1);
22. quick\_sort(array, i + 1, finish);
23. }
24. }

这段代码是快速排序算法的实现，它是一种高效的排序方法，用于对整数数组进行排序。该算法首先选择数组中的一个元素作为基准（pivot），然后重新排列数组，使得所有小于基准的元素都移到基准的左边，所有大于基准的元素都移到其右边。这个过程称为划分操作。完成划分后，基准元素就处于其最终位置。然后，算法递归地对基准左右两边的子数组进行同样的操作，直到每个子数组只包含一个元素，此时数组变为完全有序。快速排序的效率在于它能够快速减少大规模数据的排序问题规模，每次划分操作都能确保至少一个元素（基准）被放置在其最终位置。

**3.6. 堆排序**

1. void sort::heapify(**int** array[], **int** start, **int** i, **int** n)
2. {
3. **int** largest = i;
4. **int** left = 2 \* i + 1;
5. **int** right = 2 \* i + 2;
7. **if** (left < n && array[start + left] > array[start + largest]) {
8. largest = left;
9. }
10. **if** (right < n && array[start + right] > array[start + largest]) {
11. largest = right;
12. }
13. **if** (largest != i) {
14. swap(array[start + i], array[start + largest]);
15. num++;  // 增加交换次数
16. heapify(array, start, largest, n);
17. }
19. }
21. **void** sort::heap\_sort(**int** array[], **int** start, **int** finish)
22. {
23. **int** n = finish - start + 1;
24. **for** (**int** i = n / 2 - 1; i >= 0; i--) {
25. heapify(array, start, i, n);
26. }
27. **for** (**int** i = n - 1; i >= 0; i--) {
28. swap(array[start], array[start + i]);
29. num++;  // 增加交换次数
30. heapify(array, start, 0, i);
31. }
32. }

这段代码实现了堆排序算法，用于对整数数组进行有效排序。堆排序首先通过heapify函数将数组转换成一个最大堆结构，确保每个父节点的值都大于其子节点。这个过程从非叶子节点开始，向上调整堆。一旦建立了最大堆，heap\_sort函数接着将堆顶元素（即最大值）与数组末尾元素交换，然后减小堆的大小，并重新调整剩下的堆，保持最大堆的特性。这个过程重复进行，每次都从堆中移除最大元素并放置到数组的末尾，最终得到一个有序数组。通过这种方式，堆排序能有效地利用堆数据结构的性质，高效地对数组进行排序。

**3.7. 归并排序**

1. **void** sort::merge(**int** array[], **int** **const** left, **int** **const** mid, **int** **const** right)
2. {
3. **int** sub\_array\_one = mid - left + 1;
4. **int** sub\_array\_two = right - mid;
5. **int**\* left\_array = **new** **int**[sub\_array\_one];
6. **int**\* right\_array = **new** **int**[sub\_array\_two];
8. **for** (**int** i = 0; i < sub\_array\_one; i++)
9. left\_array[i] = array[left + i];
10. **for** (**int** j = 0; j < sub\_array\_two; j++)
11. right\_array[j] = array[mid + 1 + j];
13. **int** index\_one = 0, index\_two = 0;
14. **int** merged\_index = left;
16. **while** (index\_one < sub\_array\_one && index\_two < sub\_array\_two) {
17. **if** (left\_array[index\_one] <= right\_array[index\_two]) {
18. array[merged\_index] = left\_array[index\_one++];
19. }
20. **else** {
21. array[merged\_index] = right\_array[index\_two++];
22. }
23. merged\_index++;
24. num++;  // 每次复制增加交换次数
25. }
26. **while** (index\_one < sub\_array\_one) {
27. array[merged\_index++] = left\_array[index\_one++];
28. num++;  // 每次复制增加交换次数
29. }
30. **while** (index\_two < sub\_array\_two) {
31. array[merged\_index++] = right\_array[index\_two++];
32. num++;  // 每次复制增加交换次数
33. }
35. **delete**[] left\_array;
36. **delete**[] right\_array;
37. }
39. **void** sort::merge\_sort(**int** array[], **int** begin, **int** end)
40. {
41. **if** (begin >= end) **return**;
42. **int** mid = begin + (end - begin) / 2;
43. merge\_sort(array, begin, mid);
44. merge\_sort(array, mid + 1, end);
45. merge(array, begin, mid, end);
46. }

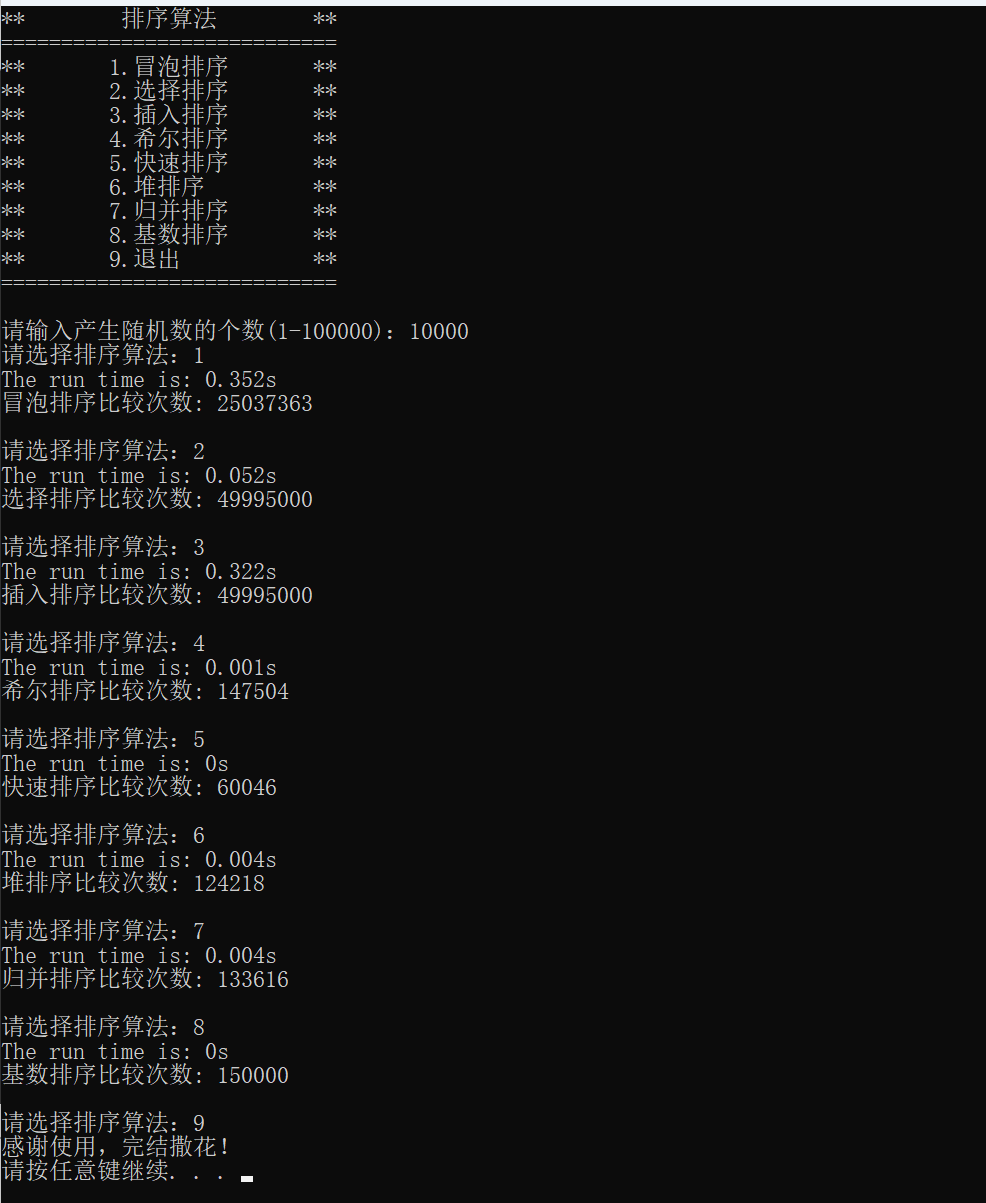
这段代码是归并排序算法的实现，它是一种高效的数组排序方法。归并排序的核心思想是将数组分成两半，分别对这两半递归地进行排序，然后将这两个有序的子数组合并成一个完整的有序数组。具体来说，merge\_sort函数首先将数组分成两部分，直到每个部分只有一个元素（或没有元素），此时认为每个部分都是有序的。接着，merge函数被用来将两个有序子数组合并成一个有序数组。它通过比较两个子数组的元素，逐一选择较小的元素放入原数组中，直到所有的元素都被合并。归并排序的效率在于它总是将问题分解成小问题来解决，然后合并小问题的解以得到原问题的解，这种分治法的策略使得归并排序在多种情况下都能保持良好的性能。

**3.8. 基数排序**

1. **void** sort::custom\_radix\_sort\_step(**int** data[], **int** start\_idx, **int** end\_idx, **int** digit) {
2. **int** sorted\_data[100001]; // 存放排序后数据的临时数组
3. **int** bucket[10] = { 0 };    // 初始化桶数组
5. // 遍历数据，计算每个桶的元素数量
6. **for** (**int** i = start\_idx; i <= end\_idx; i++) {
7. **int** index = (data[i] / digit) % 10;
8. bucket[index]++;
9. num++; // 直接使用类的成员变量
10. }
12. // 调整桶数组，计算在sorted\_data中的位置
13. **for** (**int** i = 1; i < 10; i++) {
14. bucket[i] += bucket[i - 1];
15. }
17. // 根据桶的信息，排序数据到sorted\_data数组
18. **for** (**int** i = end\_idx; i >= start\_idx; i--) {
19. **int** index = (data[i] / digit) % 10;
20. sorted\_data[--bucket[index]] = data[i];
21. num++;
22. }
24. // 将sorted\_data中的数据复制回原数组
25. **for** (**int** i = start\_idx; i <= end\_idx; i++) {
26. data[i] = sorted\_data[i - start\_idx];
27. num++;
28. }
29. }
31. **void** sort::radix\_sort(**int** a[], **int** begin, **int** end)
32. {
33. num = 0; // 重置计数器
34. // 找出数组中的最大值
35. **int** max\_value = a[begin];
36. **for** (**int** i = begin + 1; i <= end; i++) {
37. **if** (a[i] > max\_value) {
38. max\_value = a[i];
39. }
40. }
41. // 对每一位进行排序处理
42. **for** (**int** digit = 1; max\_value / digit > 0; digit \*= 10) {
43. custom\_radix\_sort\_step(a, begin, end, digit); // 不需要传递num
44. }
45. }

这段代码实现了基数排序算法，用于对整数数组进行高效排序。基数排序是一种非比较型整数排序算法，其基本思想是按照低位先排序，然后收集；再按照高位排序，然后再收集，以此类推，直到最高位。在这个实现中，radix\_sort函数首先找出数组中的最大值以确定排序的最大位数。接着，它对每一位数字（从低位到高位）执行custom\_radix\_sort\_step函数。在每一轮排序中，custom\_radix\_sort\_step函数使用桶排序的方式，首先统计每个桶（即每个数字位可能的值0-9）中有多少个元素，然后调整桶中元素的位置，最后根据这些桶的信息将元素按照当前位的值排序到临时数组sorted\_data中，最后将临时数组的内容复制回原数组。这种方法能够有效地对数字进行排序，尤其是在处理大量数据时，基数排序的速度通常远超传统的比较排序方法。

**健全性测试：**

****

**健全性体现**

多种排序算法：程序包含多种排序算法（冒泡、选择、插入、希尔、快速、堆、归并、基数排序），提供了多样的排序选择。

计数器使用：在类sort中有一个num成员变量，用于统计在排序过程中发生的比较或交换次数，这有助于性能分析。

用户交互：程序通过命令行交互，让用户选择排序算法和输入随机数的个数，提高了用户体验。

时间测量：使用clock\_t来记录排序操作的开始和结束时间，可以测量不同排序算法的执行时间。

**错误处理体现**

输入验证：程序在接收用户输入（如随机数个数、排序算法选择）时进行了检查，确保输入有效。

防止输入错误：当用户输入非法值时（如超出预定范围的数字），程序会提示错误并要求重新输入，防止程序因错误输入而崩溃。

临时数组的动态分配：在归并排序中，使用动态分配的临时数组来存储子数组，这有助于处理不同大小的数据集。

内存管理：在归并排序中，使用new和delete进行动态内存的分配和释放，避免了内存泄漏。

1. **项目创新点**

本排序程序的创新点在于它综合了多种经典排序算法（包括冒泡、选择、插入、希尔、快速、堆、归并和基数排序），提供了一个全面的排序解决方案。特别地，通过类sort的设计，程序实现了算法性能监控，借助成员变量num来统计每种排序算法的比较或交换次数，这为性能分析提供了便利。此外，程序具备良好的用户交互界面，允许用户选择不同的排序算法和输入数据规模，增加了实用性和灵活性。对于时间测量，使用clock\_t记录排序操作的执行时间，使用户能够直观地比较不同算法的效率。

**五、实验小结**

在实验中，本程序展现了各种排序算法在不同数据规模下的性能表现。实验结果显示，不同算法针对特定数据集有其优势和局限性，如快速排序在大多数情况下效率较高，而冒泡排序则相对较慢。通过记录的比较和交换次数，我们能够深入理解每种算法的内部机制及其效率。此外，程序在用户输入验证和错误处理方面表现良好，增强了程序的鲁棒性。然而，程序在高级错误处理和内存管理方面还有改进的空间，例如处理动态内存分配的异常情况。总的来说，这个实验提供了宝贵的实践经验，有助于更深入地理解和比较不同排序算法的特性和应用场景。