项目说明文档

数据结构课程设计

——8种排序算法的比较案例

作 者 姓 名： 杨宇琨

学 号： 2252843

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 分析

## 背景分析

## 软件开发人员而言，了解不同排序算法的性能特点对于选择合适的算法来解决实际问题至关重要。通过这个项目，可以对比不同排序算法在处理大规模数据时的表现，从而在实际开发中做出更好的算法选择。此外，在数据分析和处理领域，排序算法的性能也是一个重要的考量因素。通过该项目，可以更好地了解不同排序算法对数据处理效率的影响，在实际的数据处理过程中做出合适的决策。

## 1.2 功能分析

1. 用户输入随机数的个数：程序需要接受用户输入的随机数个数。

2. 生成随机数序列：根据输入的随机数个数，程序需要生成相应数量的随机数。

3. 实现快速排序算法：对生成的随机数序列进行快速排序，并统计排序时间、交换次数和比较次数。

4. 实现直接插入排序算法：对生成的随机数序列进行直接插入排序，并统计排序时间、交换次数和比较次数。

5. 实现冒泡排序算法：对生成的随机数序列进行冒泡排序，并统计排序时间、交换次数和比较次数。

6. 实现选择排序算法：对生成的随机数序列进行选择排序，并统计排序时间、交换次数和比较次数。

7. 实现希尔排序算法：对生成的随机数序列进行希尔排序，并统计排序时间、交换次数和比较次数。

8. 实现归并排序算法：对生成的随机数序列进行归并排序，并统计排序时间、交换次数和比较次数。

9. 实现堆排序算法：对生成的随机数序列进行堆排序，并统计排序时间、交换次数和比较次数。

10. 实现基数排序算法：对生成的随机数序列进行基数排序，并统计排序时间、交换次数和比较次数。

11. 显示排序结果和统计信息：将排序后的结果和每种排序方法所花费的排序时间、交换次数和比较次数展示给用户。

# 设计

1. 功能描述：

- 实现了一个排序算法比较工具，用户可以选择不同的排序算法对生成的随机数进行排序，并统计比较次数、交换次数和排序时间。

- 用户通过控制台界面进行交互，选择排序算法并查看每种排序算法的性能表现。

2. 程序结构设计：

- 用户交互部分：

- 主程序包含一个循环，允许用户多次执行排序操作，并提供退出程序的选项。

- 用户可以通过菜单选择不同的排序算法，根据输入的选择执行对应的排序算法，并展示排序结果和性能统计信息。

- 排序算法实现：

- 程序包含冒泡排序、选择排序、直接插入排序、希尔排序、快速排序、堆排序、归并排序和基数排序等排序算法的具体实现。

- 每个排序算法函数负责对输入数组进行排序，并统计比较次数、交换次数和排序时间。

- 工具函数和性能统计：

- 程序包含了暂停等待用户输入的函数`pause`，用于在某些情况下等待用户输入。

- 每种排序算法都有对应的性能统计函数，用于统计每种排序算法的比较次数、交换次数和排序时间，并将统计信息展示给用户。

3. 算法性能统计：

- 程序使用了`std::chrono`来统计每种排序算法的执行时间，通过高精度时钟记录排序算法的执行时间。

- 每种排序算法的比较次数和交换次数也被记录并展示给用户。

4. 随机数生成：

- 程序可以生成一定数量的随机数，并提供给用户选择的排序算法进行排序。

5. 界面交互和用户体验：

- 程序提供清晰的菜单界面，用户可以通过选择相应的数字来执行对应的排序算法，同时展示排序结果和性能统计信息。

- 用户体验较好，可以反复选择不同的排序算法进行比较，直到选择退出程序。

6. 循环执行逻辑：

- 程序允许用户多次执行排序操作，直到选择退出程序为止。

# 3 实现

## 3.1 希尔排序功能的实现

### 3.1.1 希尔排序功能流程

希尔排序（Shell Sort）是一种插入排序的改进版本，也称为缩小增量排序。其基本思想是将待排序的数组按照一定的间隔分为若干个子序列，对每个子序列进行插入排序；然后逐渐缩小间隔，重复进行插入排序操作，直到间隔为1，最后进行一次插入排序。希尔排序的逻辑流程如下：

1. 选择间隔序列： 首先选择一个间隔序列，通常使用希尔建议的间隔序列（1, 4, 13, 40, 121, ...）或者Hibbard间隔序列（1, 3, 7, 15, 31, ...）等。

2. 按照间隔分组： 将待排序的数组按照选定的间隔分为若干个子序列。

3.对每个子序列进行插入排序： 对每个子序列进行插入排序操作，即对第i个元素和第i+gap个元素进行比较和交换，其中gap为当前的间隔。

4. 缩小间隔，重复插入排序： 逐渐缩小间隔，重复进行插入排序操作，直到间隔为1。

5. 最终插入排序： 最后进行一次间隔为1的插入排序。

希尔排序的关键在于选择合适的间隔序列和不断缩小间隔的插入排序操作，通过这种方式可以在一定程度上提高插入排序的效率。

### 3.1.2 希尔排序功能核心代码

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function Name:shellSort

Function:To sort array by shellSort

Input Parameters:array to sort,size of array,comparisons,swaps

Return Value:void

Documentation：

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void shellSort(int arr[], int size, int& comparisons, int& swaps) {

//Gap from size/2 to 1

for (int gap = size / 2; gap > 0; gap /= 2) {

for (int i = gap; i < size; i++) {

int key = arr[i];//Store the element

int j = i;//record the index

//Insert sort

while (j >= gap && arr[j - gap] > key) {

comparisons++;

arr[j] = arr[j - gap];

swaps++;

j -= gap;

}

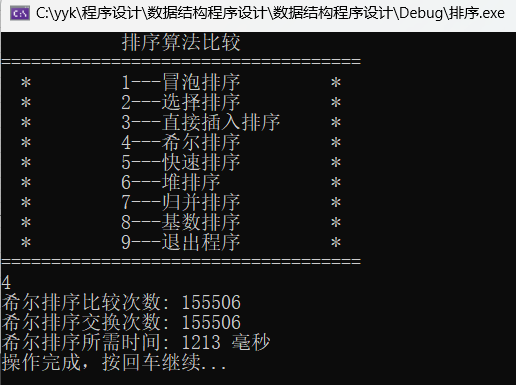
arr[j] = key;

}

}

}

3.1.3 插入功能截屏示例



## 3.2 堆排序功能的实现

### 3.2.1 堆排序功能流程

堆是一种特殊的树形数据结构，它满足堆属性：对于每个节点 i，父节点的值始终大于或小于其子节点的值（最大堆和最小堆）。堆排序利用堆的性质进行排序，具体的流程如下：

1. 将待排序的数组构建成一个最大堆。从最后一个非叶子节点开始（索引为 n/2-1），依次向前遍历，对每个节点执行下沉操作，使得以该节点为根的子树满足最大堆的性质。

2. 经过第一步的操作后，数组的根节点为最大值。将根节点与数组末尾元素交换，并将数组长度减一，相当于将最大值移动到数组末尾。

3. 对剩余的 n-1 个元素重新构建最大堆，重复步骤1和2，直到所有元素都被取出，此时数组就是有序的。

整个堆排序的逻辑流程可以总结为：

- 构建最大堆：从最后一个非叶子节点开始，依次对每个节点执行下沉操作，构建最大堆。

- 交换并重建堆：将根节点与数组末尾元素交换，并将数组长度减一，然后对剩余的元素重新构建最大堆。

- 重复交换并重建堆的操作，直到所有元素都被取出，数组排序完成。

### 3.2.2 堆排序功能核心代码

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function Name:heapify

Function:To make the number which the index of i is pointing is the largest number of its child

Input Parameters:array to sort,size of array,index i,comparisons,swaps

Return Value:void

Documentation：

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void heapify(int arr[], int size, int i, int& comparisons, int& swaps) {

int largest = i;//Parent index

int left = 2 \* i + 1;//Left child index

int right = 2 \* i + 2;//Right child index

//To maket the parent index is pointing to the largest number of its child

if (left < size && arr[left] > arr[largest]) {

comparisons++;

largest = left;

}

if (right < size && arr[right] > arr[largest]) {

comparisons++;

largest = right;

}

//If the largest number is already in the place,quit

if (largest != i) {

std::swap(arr[i], arr[largest]);//Swap the largest number to the i position

swaps++;

heapify(arr, size, largest, comparisons, swaps);//In case after switch,the largest position's number is not the largest number of its child

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function Name:heapSort

Function:To sort array by heapSort

Input Parameters:array to sort,size of array,comparisons,swaps

Return Value:void

Documentation：

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void heapSort(int arr[], int size, int& comparisons, int& swaps) {

//0 1 2 3 4 5 6 7 8 9(index)

//4 is the parent of 9

//3 is the parent of 7 8

//2 is the parent of 5 6

//1 is the parent of 3 4

//0 is the parent of 1 2

//Form the bottom to the top

for (int i = size / 2 - 1; i >= 0; i--) {

heapify(arr, size, i, comparisons, swaps);

}

//Make the Last element the largest number,and

for (int i = size - 1; i >= 0; i--) {

//arr[0]is the largest number,arr[i]is the last number of the unordered array

std::swap(arr[0], arr[i]);

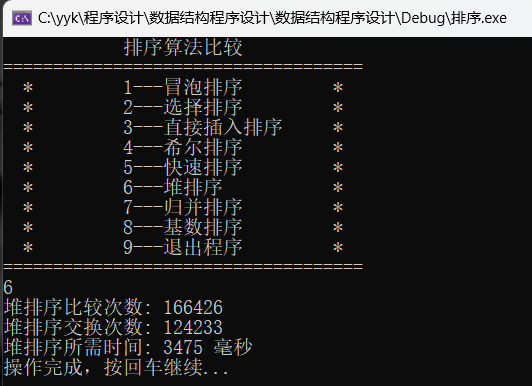
swaps++;

heapify(arr, i, 0, comparisons, swaps);//Make the arr[0] the largest number

}

}

### 3.2.3 堆排序功能截屏示例



## 3.3 归并排序功能的实现

### 3.3.1 归并排序功能流程

当进行归并排序时，需要将数组分割成较小的数组，然后不断地将这些较小的数组合并成较大的数组，直到最终合并成一个有序的数组。归并排序的基本流程如下：

1. 分割：将待排序的数组从中间分成两个子数组，然后对这两个子数组分别递归地进行分割，直到无法再分割为止。这样就得到了一系列长度较小的子数组。

2. 合并：递归地将这些较小的子数组两两合并，合并的过程中进行排序操作，直到最终合并成一个有序的数组。

具体的逻辑流程可以总结为：

- 分割：将数组从中间分成两个子数组，对每个子数组递归地进行分割，直到数组长度为1。

- 合并：将两个已经有序的子数组合并成一个有序的数组，合并过程中利用额外的空间来存储中间结果。

归并排序的核心思想在于分治和合并，通过不断地分割和合并数组，最终得到一个有序的数组。

### 3.3.2 归并排序功能核心代码

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function Name:merge

Function:To merge two parts

Input Parameters:array to sort,three index,comparisons,swaps

Return Value:void

Documentation：

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void merge(int arr[], int left, int mid, int right, int& comparisons, int& swaps)

{

int n1 = mid - left + 1;//Left part size

int n2 = right - mid;//Right part size

//To store the two parts

int\* L = new int[n1];//Left part

int\* R = new int[n2];//Right part

//Replicate the two parts

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];

}

int i = 0, j = 0, k = left;//Initialize three index

//If the index is in place

while (i < n1 && j < n2) {

//Choose the number that is smaller and move it to the right place in the array

if (L[i] <= R[j]) {

arr[k] = L[i];

i++;

}

else {

arr[k] = R[j];

j++;

}

swaps++;

comparisons++;

k++;

}

//If the right part has been checked over,Push all the left part

while (i < n1) {

arr[k] = L[i];

i++;

k++;

swaps++;

}

//If the left part has been checked over,Push all the right part

while (j < n2) {

arr[k] = R[j];

j++;

k++;

swaps++;

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function Name:mergeSort

Function:To sort array by mergeSort

Input Parameters:array to sort,left right index,comparisons,swaps

Return Value:void

Documentation：

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void mergeSort(int arr[], int left, int right, int& comparisons, int& swaps) {

if (left < right) {

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

mergeSort(arr, left, mid, comparisons, swaps);//Sort left part

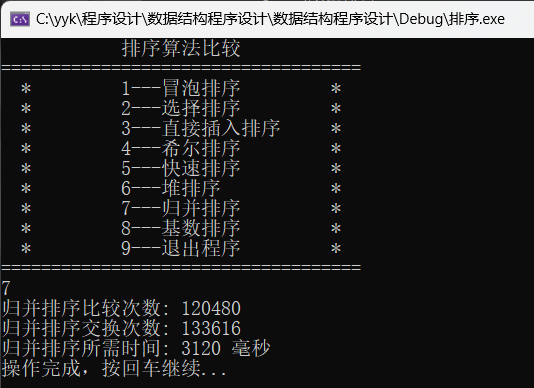
mergeSort(arr, mid + 1, right, comparisons, swaps);//Sort right part

merge(arr, left, mid, right, comparisons, swaps);//Merge the two part into an ordered array

}

}

3.3.3 归并排序功能截屏示例



## 3.4 基数排序功能的实现

### 3.4.1 基数排序功能实现流程

基数排序是一种线性时间复杂度的非比较排序算法，它利用数字的位数进行排序。下面是基数排序的流程：

1. 确定最大数的位数：首先找出待排序数组中最大的数，并确定其位数，假设最大数的位数为 d。

2. 从低位到高位，依次对每一位进行排序：从个位开始，对所有数字按照当前位数进行排序，可以使用计数排序或桶排序等算法来完成这一步骤。

3. 按照当前位数排序后，得到的数组顺序就是按照当前位数排好序的，然后继续对下一位数进行排序，直到对最高位数排序完成。

4. 经过 d 轮排序后，待排序数组就变成了一个有序序列。

基数排序的关键在于通过对数字的每一位进行排序，逐步得到最终有序的结果。整个过程中，每一轮排序都是基于上一轮排序的结果，直到所有位数都被排序完成。

希望这个基数排序的逻辑流程对你有所帮助！

### 3.4.2 基数排序功能核心代码

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function Name:getMax

Function:To get the max number of the array

Input Parameters:array,size

Return Value:void

Documentation：

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int getMax(int arr[], int size) {

int max = arr[0];

for (int i = 1; i < size; i++) {

if (arr[i] > max) {

max = arr[i];

}

}

return max;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function Name:countSort

Function:To sort array by countSort in exp

Input Parameters:array to sort,size of array,exp,comparisons,swaps

Return Value:void

Documentation：

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void countSort(int arr[],int size, int exp, int& comparisons, int& swaps)

{

int\* output = new int[size];//New a output array

int count[10] = { 0 };//There are ten position from 0 to 9 for all the numbers

for (int i = 0; i < size; i++) {

count[(arr[i] / exp) % 10]++;//(arr[i] / exp) % 10 is the key

}

//If count[1] is 10,and count[2] is 5

//So the first element in count[2] will be stored in array[14](5+10-1=14)

//Sum the count,as it will be used as an index

for (int i = 1; i < 10; i++) {

count[i] += count[i - 1];

}

for (int i = size - 1; i >= 0; i--) {

output[count[(arr[i] / exp) % 10] - 1] = arr[i];//Push the smallest number into the output

count[(arr[i] / exp) % 10]--;//Count--

}

for (int i = 0; i < size; i++) {

arr[i] = output[i];//Replicate to the array

swaps++;

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function Name:radixSort

Function:To sort array by radixSort

Input Parameters:array to sort,size of array,comparisons,swaps

Return Value:void

Documentation：

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void radixSort(int arr[], int size, int& comparisons, int& swaps) {

int max = getMax(arr, size);//Get the max

//Exp is the exponential,from 1 to the max / exp(highest exp of all number)

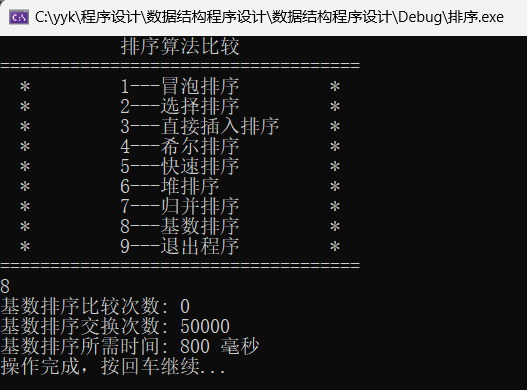
//Sort from unit's digit,ten's digit,hundred's digit...

for (int exp = 1; max / exp > 0; exp \*= 10) {

countSort(arr, size, exp, comparisons, swaps);

}

### }3.4.3 基数排序功能截屏示例



## 3.5 快速排序功能的实现

### 3.5.1 快速排序功能流程

当进行快速排序时，可以遵循以下流程：

1. 选择基准值：从待排序数组中选择一个基准值（pivot）。选择基准值的方式可以多样化，常见的方式包括选择第一个元素、最后一个元素、中间元素，甚至随机选择一个元素作为基准值。

2. 分区操作：将数组中的其他元素按照与基准值的大小关系分为两部分，一部分小于基准值，一部分大于基准值。分区操作可以通过双指针法或挖坑填数法来实现。具体步骤如下：

- 初始化两个指针：一个指向数组的起始位置，一个指向数组的结束位置。

- 从数组的起始位置开始，向后移动左指针，直到找到一个大于等于基准值的元素。

- 从数组的结束位置开始，向前移动右指针，直到找到一个小于等于基准值的元素。

- 如果左指针小于右指针，交换两个指针所指向的元素，使得大于基准值的元素位于右边，小于基准值的元素位于左边。

- 重复上述步骤，直到左指针大于等于右指针。

3. 递归操作：对分区后的两部分数组分别递归地应用快速排序算法。即对基准值左侧的子数组和右侧的子数组分别重复步骤1和步骤2，直到子数组长度为1或0时，表示子数组已经有序。

4. 合并：经过递归处理后，整个数组就变成了有序的。

这样，整个数组就被分割成了若干个子数组，每个子数组的基准值都处于正确的位置上，从而实现了排序。

3.5.2 快速排序核心代码

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function Name:quickSort

Function:To sort array by quicksort

Input Parameters:array to sort,left index,right index,comparisons,swaps

Return Value:void

Documentation：

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void quickSort(int arr[], int left, int right, int& comparisons, int& swaps)

{

int i = left, j = right;//I,j for left and right index

int pivot = arr[(left + right) / 2];//Set the middle number as the pivot

//If the left index is smaller than right index

while (i <= j) {

//The left index is to find the number that is greater than pivot

//From the left to the right

while (arr[i] < pivot) {

comparisons++;//Record every comparison

i++;//Move the index

}

//The left index is to find the number that is smaller than pivot

//From the right to the left

while (arr[j] > pivot) {

comparisons++;//Record every comparison

j--;//Move the index

}

//If the left index is smaller or equal to the right index after moving

if (i <= j) {

std::swap(arr[i], arr[j]);//Swap two number as we want the bigger in the right,smaller in the left

swaps++;//Record the swap

i++;//Move two indexes

j--;

}

}

//If the left(start index) is smaller than right index

if (left < j)

quickSort(arr, left, j, comparisons, swaps);//Go in to the left part

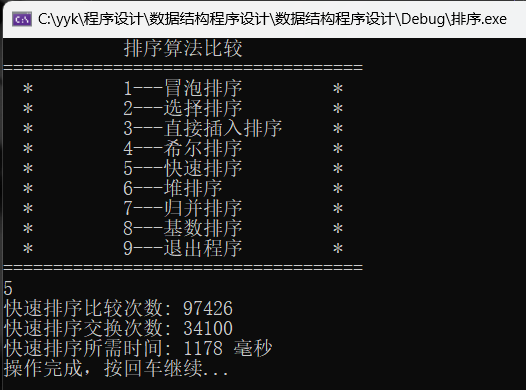
//If the right(start index) is bigger than left index

if (i < right)

quickSort(arr, i, right, comparisons, swaps);//Go in to the right part

### }

### 3.5.3 统计功能截屏示例



## 3. 6插入排序功能的实现

### 3.6.1 插入排序功能流程

当进行插入排序时，可以遵循以下流程：

1. 从第二个元素开始，将其视为当前需要插入的元素。

2. 将该元素与已经排序好的部分进行比较，从后往前逐个比较。

3. 如果当前元素小于已排序部分的某个元素，则将该元素向后移动一个位置，为当前元素腾出插入的位置。

4. 重复上述步骤，直到找到当前元素的插入位置或者已经比较到已排序部分的起始位置。

5. 将当前元素插入到找到的位置，此时已排序部分增加一个元素，继续下一个元素的插入操作。

6. 重复上述步骤，直到整个数组排序完成。

在流程中，通过逐个将元素插入到已排序部分的合适位置，实现了插入排序的过程。通过不断地将当前元素向前比较并移动，直到找到合适的插入位置，从而完成了插入排序的算法过程。

3.6.2 插入排序核心代码

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function Name:insertionSort

Function:To sort array by insertionSort

Input Parameters:array to sort,size of array,comparisons,swaps

Return Value:void

Documentation：

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void insertionSort(int arr[], int size, int& comparisons, int& swaps) {

for (int i = 1; i < size; ++i) {

int key = arr[i];//Set the number we are checking as the key(from the second number in the array)

int j = i - 1;//Set j next to i(we want to check all the number that is left to the key)

//Check all the number left to key until we find one that is smaller than the key

while (j >= 0 && arr[j] > key) {

comparisons++;//Record every comparison

arr[j + 1] = arr[j];//Move the number to the right(one step)

swaps++;//Record every swap

j--;//j index move to the left(one step)

}

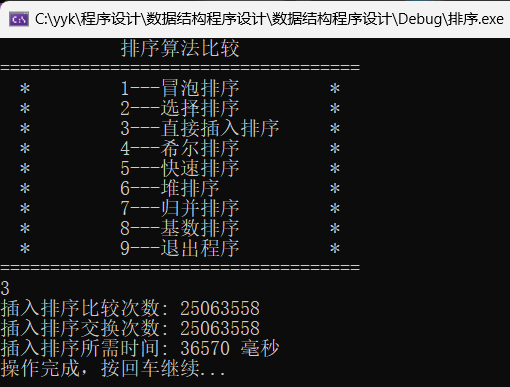
arr[j + 1] = key;//arr[j] is smaller than the key and the arr[j + 1] is the former place for the number bigger than the key

//Insert the key to the right position

}

### }

### 3.6.3 插入排序功能截屏示例



3. 7冒泡排序功能的实现

### 3.7.1 冒泡排序功能流程

当进行冒泡排序时，可以遵循以下流程：

1. 从第一个元素开始，依次比较相邻的两个元素，如果它们的顺序不正确（例如，对于升序排序，前一个元素大于后一个元素），则交换它们的位置。

2. 经过一轮比较之后，最大（或最小）的元素将“冒泡”到数组的末尾。

3. 重复进行以上的比较和交换步骤，直到没有任何一对相邻元素需要交换为止。

4. 经过一轮比较之后，最大（或最小）的元素将“冒泡”到数组的末尾。在一轮结束后，可以确定数组末尾的元素已经排好序，因此下一轮比较不需要考虑数组末尾的元素。

5. 重复上述步骤，每一轮冒泡将使得数组中下一个最大（或最小）的元素就位，直到整个数组排序完成。

在流程中，通过逐个比较相邻的元素并进行交换，实现了冒泡排序的过程。经过一轮比较之后，最大（或最小）的元素将“冒泡”到数组的末尾。

3.7.2 冒泡排序核心代码

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function Name:bubbleSort

Function:To sort array by bubbleSort

Input Parameters:array to sort,size of array,comparisons,swaps

Return Value:void

Documentation：

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void bubbleSort(int arr[], int size, int& comparisons, int& swaps) {

for (int i = 0; i < size - 1; ++i) {//Sort until end

for (int j = 0; j < size - i - 1; ++j) {//Sort until the number we are checking

comparisons++;//Record every comparison

if (arr[j] > arr[j + 1]) {

std::swap(arr[j], arr[j + 1]);//Greater in the right

swaps++;//Record every swap

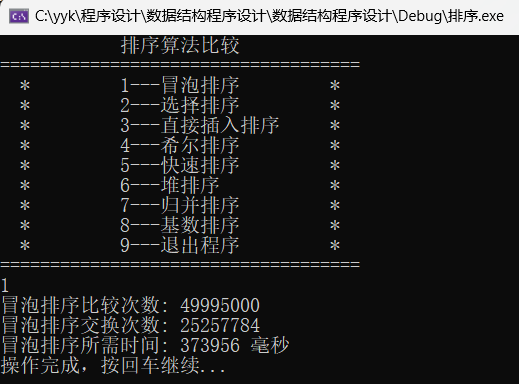
}

}

}

}

### 3.7.3 冒泡排序功能截屏示例



3. 8选择排序功能的实现

### 3.8.1 选择排序功能流程

当进行选择排序时，可以遵循以下流程：

1. 在未排序部分选择最小（或最大）的元素，将其与未排序部分的起始位置进行交换。

2. 经过一轮比较之后，最小（或最大）的元素将被放置在未排序部分的起始位置。

3. 下一轮比较将从未排序部分的下一个位置开始，并选择最小（或最大）的元素放置在未排序部分的起始位置。

4. 重复进行以上的比较和交换步骤，直到整个数组排序完成。```

在这个流程中，通过选择未排序部分的最小（或最大）的元素，将其放置在未排序部分的起始位置，从而逐步完成选择排序的过程。

3.8.2 选择排序核心代码

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function Name:selectionSort

Function:To sort array by bubbleSort

Input Parameters:array to sort,size of array,comparisons,swaps

Return Value:void

Documentation：

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void selectionSort(int arr[], int size, int& comparisons, int& swaps) {

for (int i = 0; i < size - 1; ++i) {

int minIndex = i;//Set the minindex as itself

for (int j = i + 1; j < size; ++j) {

comparisons++;//Record every comparisons

if (arr[j] < arr[minIndex])

minIndex = j;//Upgrade the minindex

}

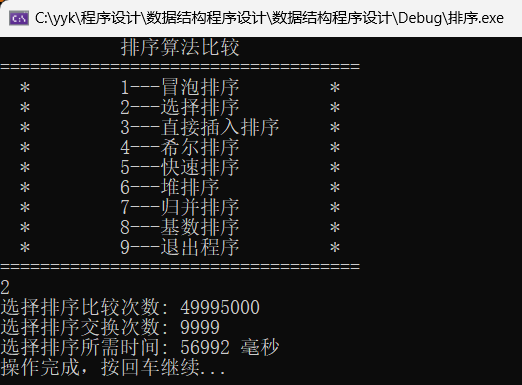
std::swap(arr[i], arr[minIndex]);//Always let the position store the minimun of all the number right to it

swaps++;//Record every swap

}

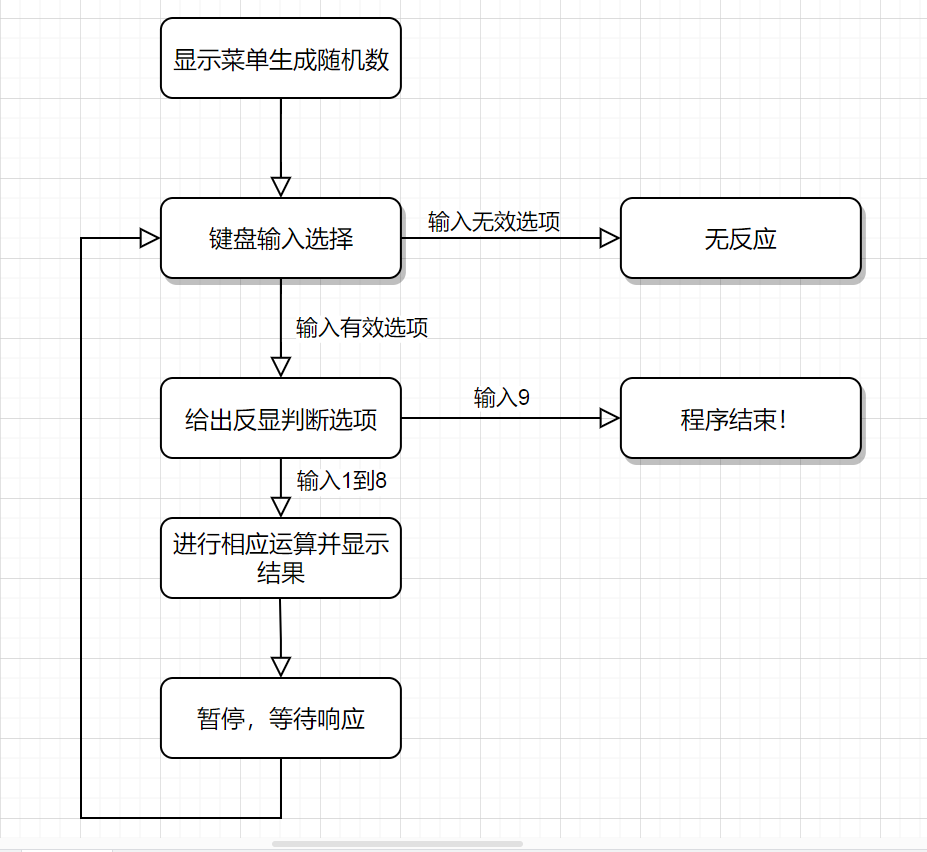
### }

### 3.8.3 选择排序功能截屏示例



## 3.9 总体系统的实现

### 3.9.1 总体系统流程图



### 3.6.2 总体系统核心代码

//The loop can let the user execute the process repeatedly

while (true) {

const int size = 10000;

int arr[size];

srand(static\_cast<unsigned>(time(0)));

generateRandomNumbers(arr, size);

int comparisons = 0, swaps = 0;

//Clear our screen

system("cls");

//Display our list

menuDisplay();

char controller;//Create a variable to hold the choice we input

while (true) {

controller = \_getch();//\_getch can avoid displaying undesired input on the screen

if (controller >= '1' && controller <= '9') {

std::cout << controller << std::endl;//Display the choice on the screen

switch (controller)

{

case '1':

bubbleCalculate(arr, size, comparisons, swaps);

std::cout << "操作完成，按回车继续..." << std::endl;

pause();

break;

case '2':

selectionCalculate(arr, size, comparisons, swaps);

std::cout << "操作完成，按回车继续..." << std::endl;

pause();

break;

case '3':

insertionCalculate(arr, size, comparisons, swaps);

std::cout << "操作完成，按回车继续..." << std::endl;

pause();

break;

case '4':

shellCalculate(arr, size, comparisons, swaps);

std::cout << "操作完成，按回车继续..." << std::endl;

pause();

break;

case '5':

quickCalculate(arr, size, comparisons, swaps);

std::cout << "操作完成，按回车继续..." << std::endl;

pause();

break;

case '6':

heapCalculate(arr, size, comparisons, swaps);

std::cout << "操作完成，按回车继续..." << std::endl;

pause();

break;

case '7':

mergeCalculate(arr, size, comparisons, swaps);

std::cout << "操作完成，按回车继续..." << std::endl;

pause();

break;

case '8':

radixCalculate(arr, size, comparisons, swaps);

std::cout << "操作完成，按回车继续..." << std::endl;

pause();

break;

case '9':

std::cout << "程序结束！" << std::endl;

return 0;

default:

continue;

}

break;

}//end of if

}//end of while

}//end of while

### return 0;

### 3.9.3 总体系统截屏示例

