《数据结构》课程实践报告

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 院、系 | 计算机学院 | | 年级专业 | 23软件工程 | 姓名 | 梅子羽 | 学号 | 2327406107 |
| 实验布置日期 | | 2024.12.17 | | 提交  日期 | 2024.1.5 | | 成绩 |  |

课程实践实验8：排序

## 一、问题描述及要求

排序的类定义

class Sort

{

public:

Sort(int r[ ], int n);

~Sort( );

void InsertSort( );

void ShellSort( );

void BubbleSort( );

void QuickSort(int first, int last);

void SelectSort( );

void HeapSort( );

void MergeSort1(int first, int last);

void MergeSort2( );

void Print( );

private:

int Partition(int first, int last);

void Sift(int k, int last);

void Merge(int first1, int last1, int last2);

void MergePass(int h);

int \*data;

int length;

};

为不失一般性，做如下约定：

（1）进行升序排序

（2）记录只有排序码一个数据项

（3）采用顺序存储

要求：

（1）编写程序创建一些整数文件用于排序。

创建的这些文件可以根据需要生成不同的长度，如长度分别为20，200和2000，以正序、逆序、随机顺序的方式创建这些文件，通过把所有这些测试数据保存在文件中（而不是每次在测试程序时用随机数生成），可以使用同样的数据去测试不同的方法，因此会更易于比较这些方法的性能。

（2）数据表采用顺序存储结构，实现插入排序，选择排序，希尔排序，归并排序，快速排序，堆排序等排序，并对这些算法的实现效率进行比较和分析。

（3）排序的指标包含：运行时间，比较次数，移动次数。

## 二、概要设计

#### （1）对实验内容的理解

本实验的目的是实现并测试常见的排序算法（如插入排序、冒泡排序、选择排序、希尔排序、堆排序、归并排序和快速排序），并对这些排序算法进行性能评估，包括比较次数和交换次数。为了进行实验测试，程序需要从外部文件读取不同排列顺序的数据（升序、降序、随机），并对这些数据进行排序操作。

排序算法的评估主要通过计算每个算法在执行过程中比较操作次数和交换（移动）操作次数。最终，程序会输出每个排序算法在不同输入数据上的执行时间、比较次数和交换次数，帮助了解不同算法在不同情况下的表现差异。

#### （2）系统功能列表

系统功能主要分为两个部分：文件操作与排序操作。

1. **文件操作**：
   * 创建文件：根据给定的文件名和数据长度，生成包含升序、降序、随机顺序的整数数据文件。
   * 读取文件：读取数据文件中的数据并存储到内存中。
2. **排序操作**：
   * 插入排序（InsertSort）：将数据从小到大排序，记录比较次数和交换次数。
   * 冒泡排序（BubbleSort）：通过重复交换相邻元素实现排序，记录比较次数和交换次数。
   * 选择排序（SelectSort）：通过选择未排序部分中的最小值与当前元素交换，记录比较次数和交换次数。
   * 希尔排序（ShellSort）：通过分组的方式逐步缩小排序间隔，提高排序效率。
   * 堆排序（HeapSort）：使用堆数据结构进行排序，记录比较次数和交换次数。
   * 归并排序（MergeSort1）：使用分治法递归地将数据分成两半，并通过合并操作排序。
   * 快速排序（QuickSort）：通过分区操作将数据分成左右两部分，递归排序两部分数据。
3. **性能评估与输出**：
   * 对每个排序算法执行时间、比较次数和交换次数进行记录和输出。

#### （3）程序运行的界面设计

程序的运行不涉及复杂的用户界面，采用命令行输入输出。用户通过选择数据文件和排序算法来测试不同的排序方法。程序的输入和输出流程如下：

1. 程序启动时，会提示用户选择要测试的文件类型（如：升序、降序、随机序列）。
2. 然后提示用户选择使用的排序算法（如：插入排序、冒泡排序、快速排序等）。
3. 程序根据用户的选择加载相应的文件，执行相应的排序算法，并输出排序时间、比较次数和交换次数。

#### （4）总体设计思路

在设计过程中，主要采用以下数据结构与算法：

* **数据结构**：
  + 使用 vector<int> 来存储待排序的数据，这种数据结构支持动态扩展，适合处理不同大小的排序问题。
  + 排序过程中用到的主要算法数据结构有堆（用于堆排序）和分治结构（用于归并排序和快速排序）。
* **类设计**：
  + 设计一个 Sort 类，负责实现所有排序算法。这个类包含：
    - 成员变量 data：一个指向整数数组的指针，用于存储待排序的数据。
    - 成员变量 length：数据的长度，用于循环和边界判断。
    - 成员函数：实现各个排序算法（如 InsertSort、BubbleSort、QuickSort 等），并计算比较次数和移动次数。
* **算法设计**：
  + **插入排序、冒泡排序、选择排序、希尔排序**：这些算法通过相邻元素交换或逐步缩小排序间隔的方式实现排序，适合中小规模数据排序。
  + **堆排序**：利用堆结构实现排序，适用于需要高效处理大量数据的情况。
  + **归并排序、快速排序**：通过分治法分解问题，适用于大规模数据排序。

#### （5）程序结构设计

程序的主要文件和功能模块如下：

1. **sort.h**：定义 Sort 类，包含排序算法和辅助函数的声明。
2. **sort.cpp**：实现 Sort 类的各个排序算法，包括插入排序、冒泡排序、选择排序等。
3. **main.cpp**：包含主函数，负责从文件读取数据，创建排序对象，调用排序算法，并输出结果。

程序结构的关系如下：

* **main.cpp** 调用 **sort.cpp** 中的排序函数，传入数据并输出排序结果。
* 数据存储和操作由 **Sort** 类管理，所有排序算法都在此类中实现。
* 文件读写操作与排序操作分离，保证程序模块化。

## 三、详细设计

#### 1. ****简单插入排序（InsertSort）****

**算法描述**：

* 简单插入排序通过将每一个未排序的元素插入到已排序部分的正确位置来逐步排序。每次插入时，需要将较大的元素依次向右移动，直到找到合适的位置。

**实现步骤**：

1. 从数组的第二个元素开始，假设第一个元素已排序。
2. 对每个元素，比较它与已排序部分的元素，从后往前找位置，将大于该元素的元素逐一向右移动。
3. 将当前元素插入到正确位置。

#### 2. ****冒泡排序（BubbleSort）****

**算法描述**：

* 冒泡排序通过重复交换相邻的元素，将较大的元素“冒泡”到数组的末尾。每次遍历结束后，最大的元素会被移到数组的最后一个位置。

**实现步骤**：

1. 进行 n−1n-1n−1 轮遍历。
2. 在每一轮遍历中，比较相邻的元素，如果前一个元素大于后一个，则交换它们。
3. 每轮遍历后，最大的元素会被放到正确位置。

#### 3. ****选择排序（SelectSort）****

**算法描述**：

* 选择排序通过每次从未排序部分选择最小的元素，放到已排序部分的末尾。这样每一轮都将一个最小的元素置于正确位置。

**实现步骤**：

1. 遍历未排序部分，找出最小的元素。
2. 将最小元素与未排序部分的第一个元素交换。
3. 重复步骤，直到所有元素排序完成。

#### 4. ****希尔排序（ShellSort）****

**算法描述**：

* 希尔排序是插入排序的一个改进版，通过选择一个增量序列，将数组分成多个子数组，分别对每个子数组进行插入排序。随着增量逐步减小，最终进行一次标准的插入排序。

**实现步骤**：

1. 选择一个增量序列（通常是数组长度的一半，然后不断减半）。
2. 对每一个增量进行排序，将数组按增量分为多个子数组，分别进行插入排序。
3. 当增量为1时，执行最终的插入排序。

#### 5. ****堆排序（HeapSort）****

**算法描述**：

* 堆排序利用堆这种数据结构来实现排序，首先将数组构建成一个最大堆，根节点为数组最大元素。然后交换根节点和末尾元素，再调整堆结构，重复此过程直至数组有序。

**实现步骤**：

1. 从数组的最后一个非叶子节点开始，逐步向上调整堆。
2. 交换堆顶元素与最后一个元素，然后重新调整堆。
3. 重复上述操作，直到整个数组有序。

#### 6. ****归并排序（MergeSort1）****

**算法描述**：

* 归并排序是典型的分治算法，首先将数组分成两个子数组，分别排序，然后将两个已排序的子数组合并为一个有序数组。每次分解直到每个子数组只有一个元素，再进行合并。

**实现步骤**：

1. 将数组递归地分为两个子数组，直到每个子数组只有一个元素。
2. 合并两个已排序的子数组。
3. 重复合并过程，直到得到最终的有序数组。

#### 7. ****快速排序（QuickSort）****

**算法描述**：

* 快速排序通过选择一个“基准”元素，将数组分成两部分，所有小于基准的元素在左边，大于基准的元素在右边，然后递归对两部分进行排序。

**实现步骤**：

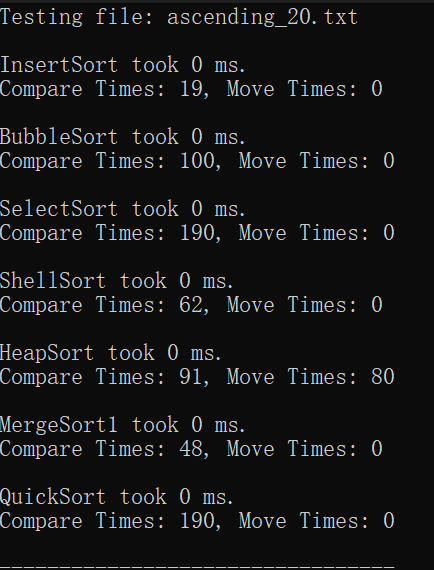
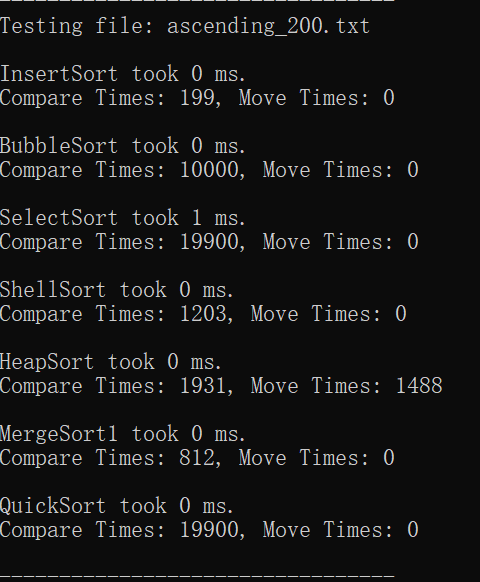
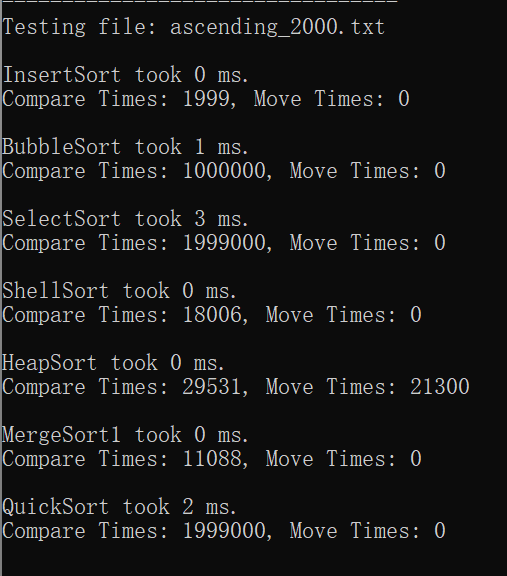
1. 选择一个基准元素。
2. 将数组分成两部分：一部分是小于基准的元素，另一部分是大于基准的元素。
3. 对分割后的两部分递归进行快速排序。

## 四、实验结果及分析

由于每次测试的结果太多，且本实验采取的数据生成本就具有普遍性，故这里只展示一次实验的结果，即对应个数20,200,2000分别产生完全顺序，完全逆序，随机顺序共9个样本数据，再对应7个排序算法进行测试，分析它们的比较次数，移动次数和运行时间。

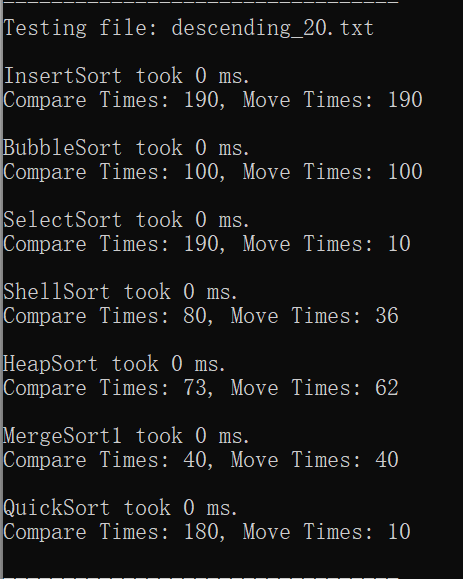
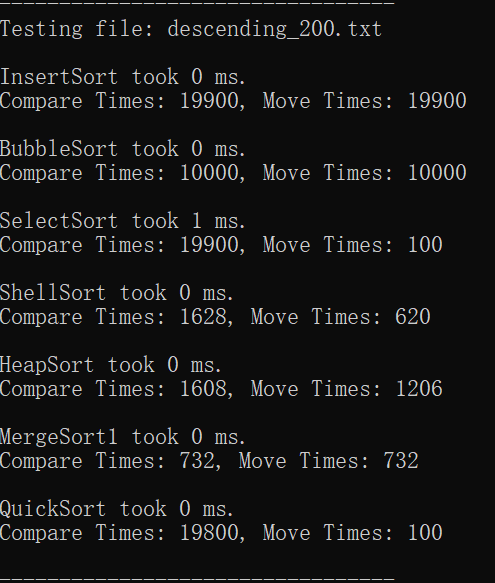
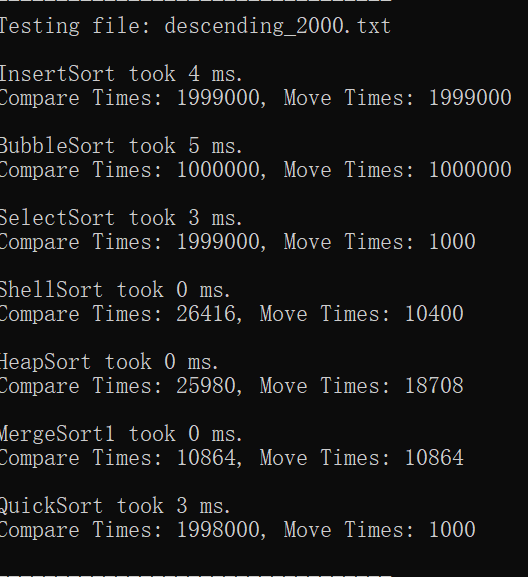
比较次数和移动次数不等价于时间复杂度，但能够反映后者。

首先看完全顺序，3个样本产生了21个数据输出，结果如下：

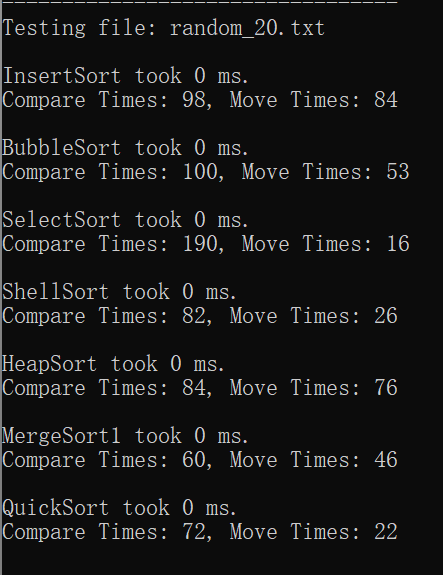
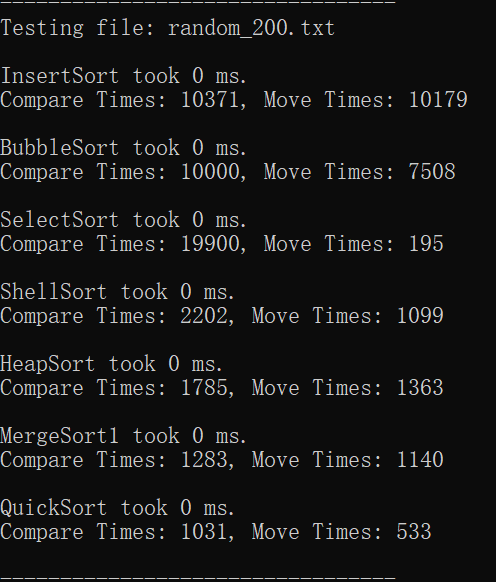
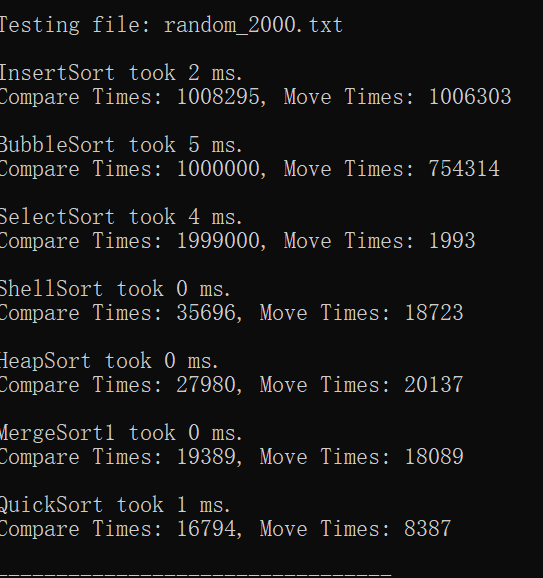
我们可以看到除堆排序以外所有排序算法的移动次数都为0，这符合实际情况；而堆排序由于采用的是完全二叉树，大根堆结构，即便对于完全有序的序列也需要进行建堆操作，从而产生了移动次数；比较特别的便是选择排序和快速排序，实际的数据比较次数是一致的。各个算法的运行时间最低且基本相同。

再者看完全逆序，3个样本产生了21个数据输出，结果如下：

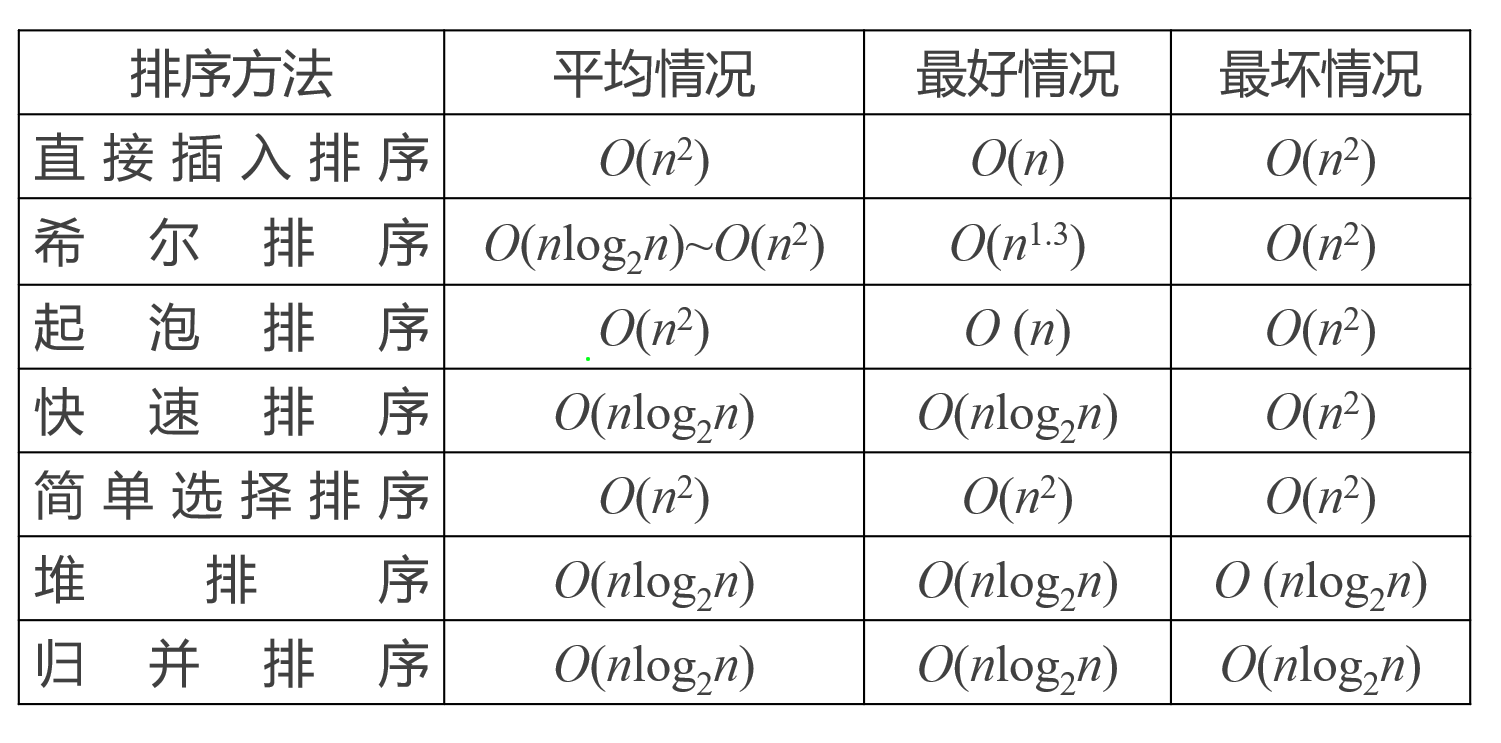
我们可以发现随着数据量的增大，除堆排序和归并排序以外，其他所有排序的比较和移动次数都有了显著增长，尤其是插入排序，冒泡排序和选择排序，选择排序移动次数少是其移动操作决定的，但比较次数依旧很大；按理来说希尔排序也应该呈现同样的次数增长，但表中呈现出来的结果并非如此，笔者是将插入排序的计数规则直接应用在希尔排序中的，大概是数据不够大的缘故；快速排序更是直接就完全退化成冒泡排序了。

最后看随机顺序，3个样本产生了21个数据输出，结果如下：

我们可以发现，7中排序算法呈现出符合它们各自平均情况时间复杂度大小的排序，这可以从数据的比较次数看出来，快速排序表现符合其名字，紧随其后的是归并排序，分治算法的有效性可见一斑；堆排序因为其特殊的二叉树结构也得到了良好的排序效果；传统三排序呈现稳定；而希尔排序从三种数据样本看来，也符合其浮动的平均情况时间复杂度。

综上，本次实验基本验证了如下的时间复杂度表格，笔者直观感受到了排序算法间的差距。



## 六、小结

在本次设计与实现过程中，我们围绕不同的排序算法进行了详细的实现和性能测试。通过实现插入排序、冒泡排序、选择排序、希尔排序、堆排序、归并排序和快速排序等经典排序算法，我们不仅巩固了对这些算法的理解，还通过对比分析，展示了各算法在不同数据集上的表现。以下是本次项目的总结与展望：

#### 1. ****完成的内容****：

* 实现了常见的排序算法：简单插入排序、冒泡排序、选择排序、希尔排序、堆排序、归并排序和快速排序，并且通过实际的测试对比了它们的性能。
* 完成了不同数据输入方式（升序、降序、随机）的测试案例生成，并通过测试文件验证了算法的效果。
* 在程序中设计了“计数比较次数”和“移动次数”的功能，帮助更好地理解排序算法的内部操作。
* 对每种排序算法进行了性能测试，计算并输出了每种排序算法的执行时间，比较了它们的效率。

#### 2. ****未完成的内容****：

* **优化部分**：尽管实现了各类排序算法，但对于某些算法的细节优化（如归并排序和快速排序的递归栈优化等）并没有深入。后续可以进一步研究如何优化递归深度，避免栈溢出。
* **更广泛的数据集**：目前的测试数据仅限于小规模的数组（20、200、2000个元素）。后续可以扩大测试规模，加入更大数据集的测试，来评估排序算法在海量数据处理中的表现。

#### 3. ****选做部分完成情况****：

* **部分文件输入/输出功能的扩展**：目前已经完成了从文件读取数据并进行排序的功能，但部分细节，例如文件格式的校验和数据处理，仍有进一步完善的空间。可以在未来实现更加通用的输入输出格式，支持多种数据源。
* **图形界面设计**：本次项目并没有实现图形界面部分，这可以作为后续扩展内容。如果需要更好的用户体验，可以设计图形界面，允许用户通过按钮选择排序算法并实时显示排序过程。

#### 4. ****程序的局限性****：

* **时间复杂度的评估**：虽然我们输出了每种排序算法的执行时间，但这些时间是受环境因素（如计算机硬件、操作系统等）的影响，因此在不同环境下结果可能有所不同。可以通过更多的重复测试来减少偶然误差。
* **算法优化**：某些排序算法（如插入排序和冒泡排序）在大规模数据时效率较低，尽管它们对小规模数据有较好的表现，但在实际应用中，通常会选择更高效的排序算法，如快速排序和堆排序。

#### 5. ****进一步学习和完善的方向****：

* **深入学习算法优化**：对于性能要求较高的应用，可以进一步研究排序算法的改进和优化，例如结合并行计算的快速排序或归并排序，以提高算法的效率。
* **扩展其他数据结构的排序算法**：例如基于堆的数据结构的排序可以进一步优化，或者使用其他高级数据结构（如红黑树、AVL树等）实现自平衡排序。
* **内存优化**：尤其是在处理大型数据集时，内存使用成为性能瓶颈。可以探索如何在有限内存环境下优化排序算法。

## 附录：源代码

**1、实验环境：Dev-C++ 5.11 C++11标准**

2、

（1）//sort.h

#ifndef SORT\_H

#define SORT\_H

#include <iostream>

#include <iostream>

#include <string>

#include <cmath>

#include <iomanip>

#include <ctime>

#include <vector>

#include <algorithm>

#include <stdio.h>

#include <cstdlib>

#include <map>

#include <fstream>

#include <sstream>

#include <ctime>

#include <queue>

#include <chrono>

using namespace std;

class Sort

{

private:

int \*data;

int length;

int Partition(int first, int last, int &compareTimes, int &moveTimes);

void Sift(int k, int last, int &compareTimes, int &moveTimes);

void Merge(int first1, int last1, int last2, int &compareTimes, int &moveTimes);

void MergePass(int h);

public:

Sort(int r[], int n);

void setSorter(int r[], int n);

void InsertSort(int &compareTimes, int &moveTimes);

void ShellSort(int &compareTimes, int &moveTimes);

void BubbleSort(int &compareTimes, int &moveTimes);

void SelectSort(int &compareTimes, int &moveTimes);

void HeapSort(int &compareTimes, int &moveTimes);

void QuickSort(int first, int last, int &compareTimes, int &moveTimes);

void MergeSort1(int first, int last, int &compareTimes, int &moveTimes);

void MergeSort2();

int getLength();

void Print();

};

#endif

（2）//sort.cpp

#include "sort.h"

#include <iostream>

#include <iostream>

#include <string>

#include <cmath>

#include <iomanip>

#include <ctime>

#include <vector>

#include <algorithm>

#include <stdio.h>

#include <cstdlib>

#include <map>

#include <fstream>

#include <sstream>

#include <ctime>

#include <queue>

#include <chrono>

using namespace std;

using namespace chrono;

Sort::Sort(int r[], int n) : data(r), length(n) {}

void Sort::InsertSort(int &compareTimes, int &moveTimes)

{

for(int i = 1; i < length; i ++)

{

int cur = data[i];

int pos = i;

while(data[pos - 1] > cur && pos > 0)

{

moveTimes ++;

compareTimes ++;

data[pos] = data[pos - 1];

pos --;

}

if(pos > 0) compareTimes ++;

data[pos] = cur;

}

}

void Sort::ShellSort(int &compareTimes, int &moveTimes)

{

for(int d = length / 2; d >= 1; d /= 2)

{

for(int i = d; i < length; i ++)

{

int cur = data[i];

int loc = i;

while(loc >= d && data[loc - d] > cur)

{

moveTimes ++;

compareTimes ++;

data[loc] = data[loc - d];

loc -= d;

}

if(loc >= d) compareTimes ++;

data[loc] = cur;

}

}

}

void Sort::BubbleSort(int &compareTimes, int &moveTimes)

{

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

{

for(int j = i; j < length - i - 1; j ++)

{

compareTimes ++;

if(data[j] > data[j + 1])

{

swap(data[j], data[j + 1]);

moveTimes ++;

}

}

}

}

int Sort::Partition(int first, int last, int &compareTimes, int &moveTimes)

{

int i = first, j = last;

while(i < j)

{

while(i < j && data[i] <= data[j])

{

j --;

compareTimes ++;

}

if(i < j)

{

swap(data[i], data[j]);

moveTimes ++;

i ++;

}

while(i < j && data[i] <= data[j])

{

compareTimes ++;

i ++;

}

if(i < j)

{

swap(data[i], data[j]);

moveTimes ++;

j --;

}

}

return i;

}

void Sort::QuickSort(int first, int last, int &compareTimes, int &moveTimes)

{

if(first >= last)

return;

int pivot = Partition(first, last, compareTimes, moveTimes);

QuickSort(first, pivot - 1, compareTimes, moveTimes);

QuickSort(pivot + 1, last, compareTimes, moveTimes);

}

void Sort::SelectSort(int &compareTimes, int &moveTimes)

{

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

{

int index = i;

for(int j = i + 1; j < length; j ++)

{

compareTimes ++;

if(data[j] < data[index])

index = j;

}

if(index != i)

{

moveTimes ++;

swap(data[index], data[i]);

}

}

}

void Sort::Sift(int k, int last, int &compareTimes, int &moveTimes)

{

int i, j;

i = k;

j = 2 \* i + 1; //ÕâÀïÖ¸µÄÊÇ×óº¢×Ó£¬ÒòÎªÊý×éÏÂ±êÊÇ´Ó0¿ªÊ¼µÄ

while(j <= last)

{

compareTimes ++;

if(j < last && data[j] < data[j + 1])

{

j ++;

compareTimes ++;

}

if(data[i] < data[j])

{

moveTimes ++;

swap(data[i], data[j]);

i = j;

j = 2 \* i + 1;

//¶ÑÊÇ´ÓºóÍùÇ°½¨Á¢µÄ£¬µ«Ç°ÃæÈç¹û·¢ÉúÁË±ä»¯£¬Ò²Ðí»áÆÆ»µÖ®Ç°µÄ¶Ñ

//ËùÒÔÒª¶ÔÕâ²¿·ÖÖØÐÂ½¨¶Ñ

}

else

break;

}

}

void Sort::HeapSort(int &compareTimes, int &moveTimes)

{

for(int i = ceil(length / 2) - 1; i >= 0; i --)

Sift(i, length - 1, compareTimes, moveTimes);

for(int i = 1; i < length; i ++)

{

swap(data[0], data[length - i]);

moveTimes ++;

Sift(0, length - i - 1, compareTimes, moveTimes);

}

}

void Sort::Merge(int first, int mid, int last, int &compareTimes, int &moveTimes)

{

int \*newData = new int[length];

int i = first, j = mid + 1, pos = first;

while(i <= mid && j <= last)

{

if(data[i] < data[j])

{

if(pos != i)

moveTimes ++;

newData[pos ++] = data[i ++];

}

else

{

if(pos != j)

moveTimes ++;

newData[pos ++] = data[j ++];

}

compareTimes ++;

}

while(i <= mid) newData[pos ++] = data[i ++];

while(j <= last) newData[pos ++] = data[j ++];

for(int i = first; i <= last; i ++)

{

data[i] = newData[i];

}

delete[] newData;

}

void Sort::MergeSort1(int first, int last, int &compareTimes, int &moveTimes)

{

if(first >= last)

return;

int mid = (first + last) / 2;

MergeSort1(first, mid, compareTimes, moveTimes);

MergeSort1(mid + 1, last, compareTimes, moveTimes);

Merge(first, mid, last, compareTimes, moveTimes);

}

int Sort::getLength()

{

return length;

}

void Sort::Print()

{

for(int i = 0; i < length; i ++)

cout << data[i] << " ";

cout << endl;

}

（3）//main.cpp

#include "sort.h"

#include <iostream>

#include <iostream>

#include <string>

#include <cmath>

#include <iomanip>

#include <ctime>

#include <vector>

#include <algorithm>

#include <stdio.h>

#include <cstdlib>

#include <map>

#include <fstream>

#include <sstream>

#include <ctime>

#include <queue>

#include <chrono>

using namespace std;

using namespace chrono;

void createFile(const string& filename, int length, const string& order)

{

ofstream file(filename);

vector<int> data(length);

if (order == "ascending")

{

for (int i = 0; i < length; i++)

{

data[i] = i + 1;

}

}

else if (order == "descending")

{

for (int i = 0; i < length; i++)

{

data[i] = length - i;

}

}

else if (order == "random")

{

srand(time(0));

for (int i = 0; i < length; i++)

{

data[i] = rand() % 10000;

}

}

for (int i = 0; i < length; i++)

{

file << data[i] << " ";

}

file.close();

}

vector<int> readFile(const string& filename)

{

ifstream file(filename);

vector<int> data;

int value;

while(file >> value)

data.push\_back(value);

file.close();

return data;

}

void sortAndTime1(Sort& sorter, void (Sort::\*sortFunction)(int&, int&), const string& sortName)

{

int compareTimes = 0, moveTimes = 0;

auto start = high\_resolution\_clock::now();

(sorter.\*sortFunction)(compareTimes, moveTimes);

auto end = high\_resolution\_clock::now();

auto duration = duration\_cast<milliseconds>(end - start).count();

cout << sortName << " took " << duration << " ms." << endl;

cout << "Compare Times: " << compareTimes << ", Move Times: " << moveTimes << endl;

cout << endl;

}

void sortAndTime2(Sort& sorter, void (Sort::\*sortFunction)(int, int, int&, int&), const string& sortName)

{

int compareTimes = 0, moveTimes = 0;

auto start = high\_resolution\_clock::now();

(sorter.\*sortFunction)(0, sorter.getLength() - 1, compareTimes, moveTimes);

auto end = high\_resolution\_clock::now();

auto duration = duration\_cast<milliseconds>(end - start).count();

cout << sortName << " took " << duration << " ms." << endl;

cout << "Compare Times: " << compareTimes << ", Move Times: " << moveTimes << endl;

cout << endl;

}

int main()

{

// ´´½¨²âÊÔÊý¾ÝÎÄ¼þ

createFile("ascending\_20.txt", 20, "ascending");

createFile("descending\_20.txt", 20, "descending");

createFile("random\_20.txt", 20, "random");

createFile("ascending\_200.txt", 200, "ascending");

createFile("descending\_200.txt", 200, "descending");

createFile("random\_200.txt", 200, "random");

createFile("ascending\_2000.txt", 2000, "ascending");

createFile("descending\_2000.txt", 2000, "descending");

createFile("random\_2000.txt", 2000, "random");

vector<int> data1, data2, data3, data4, data5, data6, data7;

vector<string> filenames{"ascending\_20.txt", "descending\_20.txt", "random\_20.txt",

"ascending\_200.txt", "descending\_200.txt", "random\_200.txt",

"ascending\_2000.txt", "descending\_2000.txt", "random\_2000.txt"};

for(const string& filename : filenames)

{

cout << "Testing file: " << filename << endl;

cout << endl;

ifstream file(filename);

vector<int> originalData;

int value;

while(file >> value)

originalData.push\_back(value);

file.close();

data1 = originalData;

Sort sorter1(data1.data(), data1.size());

sortAndTime1(sorter1, &Sort::InsertSort, "InsertSort");

data2 = originalData;

Sort sorter2(data2.data(), data2.size());

sortAndTime1(sorter2, &Sort::BubbleSort, "BubbleSort");

data3 = originalData;

Sort sorter3(data3.data(), data3.size());

sortAndTime1(sorter3, &Sort::SelectSort, "SelectSort");

data4 = originalData;

Sort sorter4(data4.data(), data4.size());

sortAndTime1(sorter4, &Sort::ShellSort, "ShellSort");

data5 = originalData;

Sort sorter5(data5.data(), data5.size());

sortAndTime1(sorter5, &Sort::HeapSort, "HeapSort");

data6 = originalData;

Sort sorter6(data6.data(), data6.size());

sortAndTime2(sorter6, &Sort::MergeSort1, "MergeSort1");

data7 = originalData;

Sort sorter7(data7.data(), data7.size());

sortAndTime2(sorter7, &Sort::QuickSort, "QuickSort");

cout << "---------------------------------" << endl;

}

return 0;

}