**《数据结构》课程实践报告**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 院、系 | 计算机学院 | | 年级专业 | 23软工 | 姓名 | 王天予 | 学号 | 2362401031 |
| 实验布置日期 | | 2024.12.17 | | 提交  日期 |  | | 成绩 |  |

课程实践实验8：排序

## 一、问题描述及要求

1. 创建长度为 1000 的随机整数数组，用于测试各种排序算法。
2. 实现以下排序算法，并对其性能进行比较：
   * 插入排序
   * 希尔排序
   * 冒泡排序
   * 快速排序
   * 选择排序
   * 堆排序
   * 归并排序（递归和非递归）
3. 统计排序算法的关键指标，包括比较次数、记录移动次数以及运行时间。
4. 对比分析不同算法的性能特点。

**实验要求：**

* 生成长度为 1000 的随机整数数组，作为测试数据。
* 实现上述排序算法，支持统计比较次数和移动次数。
* 输出各排序算法在测试数据上的性能指标，包括：
  + 比较次数
  + 移动次数
  + 总运行时间

## 二、概要设计

### 系统功能列表

* **数据生成：**
  + 生成随机整数数组（长度为 1000，范围为 [1, 10000]）。
* **排序算法实现：**
  + 插入排序
  + 希尔排序
  + 冒泡排序
  + 快速排序
  + 选择排序
  + 堆排序
  + 归并排序（递归和非递归）
* **性能统计：**
  + 统计比较次数。
  + 统计记录移动次数。
  + 记录运行时间。

## 三、实验结果

### 测试数据

* 输入：随机生成的长度为 1000 的整数数组，范围为 [1, 10000]。

### 测试结果

Insertion Sort - Comparisons: 249430, Movements: 250429

Time: 3.85267 ms

Shell Sort - Comparisons: 14939, Movements: 15461

Time: 0.487875 ms

Bubble Sort - Comparisons: 499500, Movements: 498860

Time: 10.5097 ms

Quick Sort - Comparisons: 10876, Movements: 16191

Time: 0.220625 ms

Selection Sort - Comparisons: 499500, Movements: 1984

Time: 3.32704 ms

Heap Sort - Comparisons: 8442, Movements: 12600

Time: 0.324375 ms

Merge Sort (Recursive) - Comparisons: 8689, Movements: 19952

Time: 0.924833 ms

Merge Sort (Non-Recursive) - Comparisons: 8716, Movements: 19968

Time: 0.83725 ms

## 四、实验分析与探讨

### 1. 性能比较

* **时间复杂度分析**：
  + 插入排序、冒泡排序、选择排序在输入数据较大时性能较差，时间复杂度为 。
  + 快速排序、希尔排序、堆排序和归并排序性能优异，时间复杂度接近 。
* **移动次数分析**：
  + 选择排序的移动次数显著高于其他排序算法，这是由于选择排序频繁交换元素。
  + 归并排序的移动次数也较多，但由于时间复杂度较低，整体性能仍优于插入和冒泡排序。

### 2. 问题与优化

* **冒泡排序劣势**：冒泡排序比较次数和移动次数均较多，不适合大规模数据。
* **快速排序优化**：对于小规模数据，可以采用插入排序优化递归调用。

## 五、总结与反思

* 快速排序在大多数情况下表现最佳，适合大规模数据。
* 归并排序稳定且时间复杂度固定为 ，适合需要稳定排序的场景。
* 希尔排序对中等规模数据具有良好性能。

**附录：源代码**

1. **实验环境**：

编译环境：TDM-GCC 4.9.2 64bit release

命令行参数：-static-libgcc -std=c++11

2、**目录结构**

（1）Solution.cpp

#include <iostream>

#include <vector>

#include <algorithm>

#include <chrono>

#include <random>

class Sort {

public:

Sort(const std::vector<int>& r) : data(r), length(r.size()), comparisons(0), movements(0) {}

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 resetCounters();

void displayMetrics(const std::string& algorithm);

void Print() const;

private:

std::vector<int> data;

int length;

int comparisons;

int movements;

int Partition(int first, int last);

void Sift(int k, int last);

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

void MergePass(int h);

};

// 重置比较次数与移动次数

void Sort::resetCounters() {

comparisons = 0;

movements = 0;

}

// 打印比较次数与移动次数

void Sort::displayMetrics(const std::string& algorithm) {

std::cout << algorithm << " - Comparisons: " << comparisons

<< ", Movements: " << movements << std::endl;

}

// 打印当前数组（仅用于调试）

void Sort::Print() const {

for (auto num : data) {

std::cout << num << " ";

}

std::cout << std::endl;

}

// 插入排序

void Sort::InsertSort() {

resetCounters();

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

int temp = data[i];

int j = i - 1;

while (j >= 0 && data[j] > temp) {

data[j + 1] = data[j];

--j;

comparisons++;

movements++;

}

data[j + 1] = temp;

movements++;

}

}

// 希尔排序（ShellSort）

void Sort::ShellSort() {

resetCounters();

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

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

int temp = data[i];

int j = i;

while (j >= gap) {

comparisons++;

if (data[j - gap] > temp) {

data[j] = data[j - gap];

movements++;

j -= gap;

} else {

break;

}

}

data[j] = temp;

movements++;

}

}

}

// 冒泡排序（BubbleSort）

void Sort::BubbleSort() {

resetCounters();

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

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

comparisons++;

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

// 拆解swap为两次赋值，分别记移动

int temp = data[j];

data[j] = data[j + 1];

movements++;

data[j + 1] = temp;

movements++;

}

}

}

}

// 快速排序辅助函数 Partition

int Sort::Partition(int first, int last) {

int pivot = data[last];

int i = first - 1;

for (int j = first; j < last; ++j) {

comparisons++;

if (data[j] <= pivot) {

i++;

// swap作为3次操作

int temp = data[i];

data[i] = data[j];

data[j] = temp;

movements += 3;

}

}

// pivot

int temp = data[i + 1];

data[i + 1] = data[last];

data[last] = temp;

movements += 3;

return i + 1;

}

// 快速排序

void Sort::QuickSort(int first, int last) {

if (first < last) {

int pivotIndex = Partition(first, last);

QuickSort(first, pivotIndex - 1);

QuickSort(pivotIndex + 1, last);

}

}

// 选择排序（SelectSort）

void Sort::SelectSort() {

resetCounters();

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

int minIndex = i;

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

comparisons++;

if (data[j] < data[minIndex]) {

minIndex = j;

}

}

if (minIndex != i) {

// 拆解swap为两次赋值

int temp = data[i];

data[i] = data[minIndex];

movements++;

data[minIndex] = temp;

movements++;

}

}

}

// 堆排序辅助函数 Sift

void Sort::Sift(int k, int last) {

int i = k;

int j = 2 \* i + 1;

int temp = data[i];

while (j <= last) {

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

j++;

}

comparisons++;

if (temp >= data[j]) {

break;

}

data[i] = data[j];

movements++;

i = j;

j = 2 \* i + 1;

}

data[i] = temp;

movements++;

}

// 堆排序

void Sort::HeapSort() {

resetCounters();

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

Sift(i, length - 1);

}

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

int temp = data[0];

data[0] = data[i];

data[i] = temp;

movements += 3;

Sift(0, i - 1);

}

}

// 归并辅助函数 Merge

void Sort::Merge(int first1, int last1, int last2) {

std::vector<int> temp(last2 - first1 + 1);

int i = first1, j = last1 + 1, k = 0;

while (i <= last1 && j <= last2) {

comparisons++;

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

temp[k++] = data[i++];

} else {

temp[k++] = data[j++];

}

movements++;

}

while (i <= last1) {

temp[k++] = data[i++];

movements++;

}

while (j <= last2) {

temp[k++] = data[j++];

movements++;

}

for (k = 0, i = first1; i <= last2; ++i, ++k) {

data[i] = temp[k];

movements++;

}

}

// 归并排序（递归）

void Sort::MergeSort1(int first, int last) {

if (first < last) {

int mid = (first + last) / 2;

MergeSort1(first, mid);

MergeSort1(mid + 1, last);

Merge(first, mid, last);

}

}

// 归并排序辅助函数 MergePass

void Sort::MergePass(int h) {

for (int i = 0; i + h < length; i += 2 \* h) {

int mid = i + h - 1;

int last = std::min(i + 2 \* h - 1, length - 1);

Merge(i, mid, last);

}

}

// 归并排序

void Sort::MergeSort2() {

resetCounters();

for (int h = 1; h < length; h \*= 2) {

MergePass(h);

}

}

int main() {

std::vector<int> data(1000);

std::random\_device rd;

std::mt19937 gen(rd());

std::uniform\_int\_distribution<> dis(1, 10000);

for (int &num : data) {

num = dis(gen);

}

{

Sort sorter(data);

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

sorter.InsertSort();

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

sorter.displayMetrics("Insertion Sort");

std::chrono::duration<double, std::milli> diff = end - start;

std::cout << "Time: " << diff.count() << " ms\n\n";

}

{

Sort sorter(data);

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

sorter.ShellSort();

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

sorter.displayMetrics("Shell Sort");

std::chrono::duration<double, std::milli> diff = end - start;

std::cout << "Time: " << diff.count() << " ms\n\n";

}

{

Sort sorter(data);

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

sorter.BubbleSort();

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

sorter.displayMetrics("Bubble Sort");

std::chrono::duration<double, std::milli> diff = end - start;

std::cout << "Time: " << diff.count() << " ms\n\n";

}

{

Sort sorter(data);

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

sorter.QuickSort(0, data.size() - 1);

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

sorter.displayMetrics("Quick Sort");

std::chrono::duration<double, std::milli> diff = end - start;

std::cout << "Time: " << diff.count() << " ms\n\n";

}

{

Sort sorter(data);

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

sorter.SelectSort();

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

sorter.displayMetrics("Selection Sort");

std::chrono::duration<double, std::milli> diff = end - start;

std::cout << "Time: " << diff.count() << " ms\n\n";

}

{

Sort sorter(data);

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

sorter.HeapSort();

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

sorter.displayMetrics("Heap Sort");

std::chrono::duration<double, std::milli> diff = end - start;

std::cout << "Time: " << diff.count() << " ms\n\n";

}

{

Sort sorter(data);

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

sorter.MergeSort1(0, data.size() - 1);

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

sorter.displayMetrics("Merge Sort (Recursive)");

std::chrono::duration<double, std::milli> diff = end - start;

std::cout << "Time: " << diff.count() << " ms\n\n";

}

{

Sort sorter(data);

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

sorter.MergeSort2();

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

sorter.displayMetrics("Merge Sort (Non-Recursive)");

std::chrono::duration<double, std::milli> diff = end - start;

std::cout << "Time: " << diff.count() << " ms\n\n";

}

return 0;

}