**Pp**



**实验报告**



**题目： 实验三：排序性能分析实验**

**班 级： 2023211314**

**学 号： 2023211304**

**姓 名： 唐术洋**

**学 院： 计算机学院**

**2024 年 12 月 15 日**

1. 实验目的；
2. 掌握常见排序算法的实现、并分析算法的关键度量指标（关键字比较次数、移动次数、执行时间、是否是稳定排序）；
3. 学习自己查找相关资料以解决实际问题的能力。

二、实验环境

Windows，Vscode，gcc

三、实验内容

**场景设定：**

附件文件stock\_data.csv是某上市公司2006年至2016年的每日股价的开盘价（Open）、收盘价（Close）、最高价（High）、最低价（Low)和成交量（Volume）。请在此场景设定下完成以下要求。

**要求：**

1. 请至少使用4种不同的排序方法，分别对原始数据进行开盘价（Open）、收盘价（Close）、最高价（High）、最低价（Low)和成交量（Volume）五项指标进行非递减排序，并将排序后的数据分别保存成五个文件。命名规则为指标名称\_排序方法.csv。例如open\_heap\_sort.csv表示使用堆排序对开盘价进行的排序结果。
2. 算法性能测量

记录每组排序的关键字比较次数、移动次数、执行时间，以及稳定性检查（可通过检查排序后关键字相同的记录id是否逆序判定，对成交量排序时由于重复关键字极少可不做稳定性检查）。

1. **附加要求：**在不进行完全排序的前提下，分别给出成交量top100和top500清单，保存为volume\_top100.csv和volume\_top500.csv。并给出相应的比较次数、移动次数和执行时间。

四、实验步骤（80分+5分）

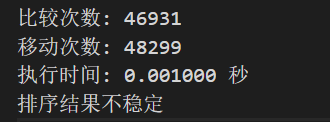
**PS：如下代码算移动三次：**

**{ node temp = arr[i];**

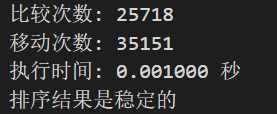
**arr[i] = arr[j];**

**arr[j] = temp; }**

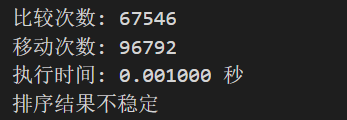
**关于close（希尔排序）：**



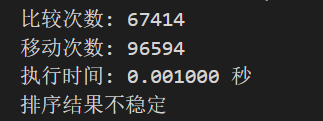
**关于open（归并排序）：**



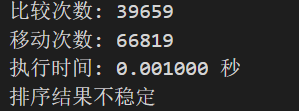
**关于high（堆排序）：**



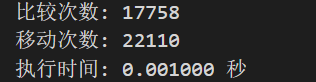
**关于low（堆排序）：**



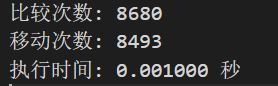
**关于volume（快速排序）：**



**Volume\_top500(堆排序的不完全排序):**



**Volume\_top100（堆排序的不完全排序）：**



**堆排序算法函数编写步骤：**

heapify 函数：

1. 初始化最大值为根节点。
2. 计算左子节点和右子节点的索引
3. 比较左子节点和当前最大值，更新最大值索引。
4. 比较右子节点和当前最大值，更新最大值索引。
5. 如果最大值索引不是根节点，则交换元素并递归调用 heapify。

heapSort 函数：

1. 构建最大堆：从最后一个非叶子节点开始，依次调用 heapify。
2. 提取元素并重建堆：从最后一个元素开始，与堆顶元素交换，并调用 heapify 重 建堆，直到处理完元素。

**快速排序算法函数编写步骤：**

partition函数：

1. 选择基准值为数组第一个元素。
2. 使用i记录小于等于基准值的元素位置
3. 遍历数组，比较每个元素与基准值，若小于等于则交换位置并更新i。
4. 最后将基准值放回到正确位置并更新索引

quickSort函数：

1. 检查low是否小于high，确保数组长度大于1。
2. 调用partition函数获取基准值位置pi。
3. 递归调用quickSort对基准值左右两边的子数组进行排序。

**归并排序算法函数编写步骤：**

merge函数：

1. 计算左右子数组的长度。
2. 创建临时数组存储左右子数组。
3. 将数据从原数组拷贝到临时数组。
4. 比较并合并两个临时数组，按顺序放入原数组。
5. 处理剩余未合并的元素。

mergeSort函数：

1. 判断是否需要继续分割数组。
2. 找到中间点，递归排序左半部分和右半部分。
3. 调用 merge 函数合并已排序的子数组。

**希尔排序算法函数编写步骤：**

Hillsort函数：

1. 初始化步长：将步长 gap 初始化为数组长度的一半。
2. 外层循环：使用 for 循环不断减小步长，直到 gap 为 0。
3. 内层循环：从当前步长 gap 开始遍历数组，保存当前元素到临时变量 temp。
4. 插入排序：通过内部 for 循环，比较并移动元素，确保子序列有序。
5. 放置元素：将临时变量 temp 放置到正确位置，并记录移动次数。

**如何实现不完全排序便能得到前100/500：**

利用堆排序的性质，大根堆每次调整完后的根节点都是当前序列中的最大值，于是可以从最大的关键字开始往下排，在循环时设置100/500次退出即可。

五、实验分析（10分）

**工作思路：**

先读取csv文件中必须的内容（所以name可以不要），随后编写排序算法代码，调用，最后得出结果。其中算法性能测量部分用全局变量进行记录，没进行一次相应操作便加1。

**设想与实际效果：**

根据我之前对快速排序的理解，我的分区函数是这样的：（源自pta上的一个作业题）

int Partition ( SqList L, int low, int high )

{

int i=low;

int j=high;

KeyType temp=L.elem[low];

L.elem[0]=L.elem[low];

while(i!=j)

{

while(L.elem[j]>temp&&i<j)

j--;

L.elem[i]=L.elem[j];

while(L.elem[i]<temp&&i<j)

i++;

L.elem[j]=L.elem[i];

}

L.elem[i]=L.elem[0];

return i;

}

从原理上，这个分区函数是对的，但是其运行效率在数据量较大且乱序程度高时效率特别低下，以至于我在第一次运行实验代码的时候以为是写了无穷递归导致一直没有结果。后面我通过分析，发现其时间主要消耗在双指针的移动上，次数太多了，于是我便改写了代码：

int partition(node arr[], int low, int high) {

int pivot = arr[low].volume;

int i = low;

for (int j = low + 1; j <= high; j++) {

if (arr[j].volume <= pivot) {

i++;

node temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

}

node temp = arr[i];

arr[i] = arr[low];

arr[low] = temp;

return i;

}

移动次数的减少使得算法效率明显提升，于是再次运行代码便很快得到了答案。

六、实验总结（10分）

由于我在进行本次实验之前完全不会写除快速排序算法外的代码（其余只是简单的理解了其原理与过程），所以在成功读取csv文件中必备的内容后陷入了茫然，于是回过头便再看了几遍ppt，才写出了实验用的代码。

希望在堆排序和归并排序处加强练习。

**源程序：**

（排序算法部分仅提交程序之间不同的部分，以volume为例，共同部分在最后）

**堆排序：**

void heapify(node arr[], int n, int i) {

int largest = i; // 初始化最大值为根节点

int left = 2 \* i + 1; // 左子节点

int right = 2 \* i + 2; // 右子节点

if (left < n && arr[left].volume > arr[largest].volume) {

largest = left;

}

compare\_count++;

if (right < n && arr[right].volume> arr[largest].volume) {

largest = right;

}

compare\_count++;

if (largest != i) {

node temp = arr[i];

arr[i] = arr[largest];

arr[largest] = temp;

move\_count += 3;

heapify(arr, n, largest);

//附加要求此处增加代码

/\*

if(i<=n-100)//500同理

break;

\*/

}

}

// 堆排序函数

void heapSort(node arr[], int n,FILE \*fp) {

// 构建大根堆

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

heapify(arr, n, i);

}

// 提取元素并重建堆

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

node temp = arr[0];

arr[0] = arr[i];

arr[i] = temp;

if(i>=n-500)

{

fprintf(fp,"%d,%s,%.2f,%.2f,%.2f,%.2f,%d\n",arr[i].id,arr[i].date,arr[i].open,arr[i].high,arr[i].low,arr[i].close,arr[i].volume);

}

move\_count += 3; // 三次移动

heapify(arr, i, 0);

}

}

**快速排序：**

int partition(node arr[], int low, int high) {

int pivot = arr[low].volume;

int i = low;

for (int j = low + 1; j <= high; j++) {

compare\_count++;

if (arr[j].volume <= pivot) {

i++;

node temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

move\_count += 3; // 三次移动

}

}

node temp = arr[i];

arr[i] = arr[low];

arr[low] = temp;

move\_count += 3; // 三次移动

return i;

}

void quickSort(node arr[], int low, int high) {

if (low < high) {

int pi = partition(arr, low, high);

quickSort(arr, low, pi - 1);

quickSort(arr, pi + 1, high);

}

}

**归并排序：**

void merge(node arr[], int l, int m, int r)

{

int i, j, k;

int n1 = m - l + 1;

int n2 = r - m;

// 创建临时数组

node L[n1], R[n2];

//拷贝数据到临时数组 L[] 和 R[]

for (i = 0; i < n1; i++)

L[i] = arr[l + i];

for (j = 0; j < n2; j++)

R[j] = arr[m + 1 + j];

i = 0;

j = 0;

k = l;

while (i < n1 && j < n2)

{

compare\_count++;

if (L[i].volume <= R[j].volume)

{

arr[k++] = L[i++];

}

else

{

arr[k++] = R[j++];

}

move\_count++;

}

// 拷贝 L[] 中剩余元素

while (i < n1)

{

arr[k++] = L[i++];

move\_count++;

}

//拷贝 R[] 中剩余元素

while (j < n2)

{

arr[k++] = R[j++];

move\_count++;

}

}

// 归并排序函数

void mergeSort(node arr[], int l, int r)

{

if (l < r)

{

// 找到中间点

int m = l + (r - l) / 2;

// 排序前半部分和后半部分

mergeSort(arr, l, m);

mergeSort(arr, m + 1, r);

merge(arr, l, m, r);

}

}

**希尔排序：**

void HillSort(node arr[], int n) {

// 初始化步长 gap 为数组长度的一半

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

// 内层循环，从 gap 开始遍历数组

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

node temp = arr[i];

int j;

// 插入排序的内部循环，比较并移动元素

for (j = i; j >= gap && arr[j - gap].volume > temp.volume; j -= gap) {

// 将 arr[j - gap] 移动到 arr[j]

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

move\_count++;

compare\_count++;

}

// 如果 j >= gap，记录比较次数

if (j >= gap) {

compare\_count++;

}

// 将 temp 放置到正确位置

arr[j] = temp;

move\_count++;

}

}

}

**共同部分：**

#include<stdio.h>

#include<string.h>

#include<stdlib.h>

#include<time.h>

#include<math.h>

typedef struct {

char date[16];

float open, close, high, low;

int id, volume;

} node;

// 全局变量记录比较次数和移动次数

int compare\_count = 0;

int move\_count = 0;

int main() {

char name[5];

FILE \*fp = fopen("stock\_data.csv", "r");

if (fp == NULL) {

printf("Error: can't open file\n");

return 1;

}

node\* total = (node\*)malloc(sizeof(node) \* 3020);

char line[100];

int i = 0;

fgets(line, 60, fp); // 读取文件头

while (fgets(line, 60, fp) != NULL) {

total[i].id = atoi(strtok(line, ","));

strcpy(total[i].date, strtok(NULL, ","));

total[i].open = atof(strtok(NULL, ","));

total[i].high = atof(strtok(NULL, ","));

total[i].low = atof(strtok(NULL, ","));

total[i].close = atof(strtok(NULL, ","));

total[i++].volume = atoi(strtok(NULL, ","));

// strcpy(name, strtok(NULL, ","));

}

fclose(fp);

// 记录排序开始时间

clock\_t start\_time = clock();

/\*堆排序产生的附加要求输出文件

FILE \*fp3= fopen("volume\_top500.csv", "w+");

fprintf(fp3,"id,date,open,high,low,close,volume\n");

heapSort(total, i,fp3);

\*/

HillSort(total, i);//排序算法调用部分

// 记录排序结束时间

clock\_t end\_time = clock();

double execution\_time = ((double)(end\_time - start\_time)) / CLOCKS\_PER\_SEC;

FILE \*fp2 = fopen("volume\_hill\_sort.csv", "w+");

if (fp2 == NULL) {

printf("error");

free(total);

return 1;

}

// 输出统计信息

fprintf(fp2, "比较次数: %d\n", compare\_count);

fprintf(fp2, "移动次数: %d\n", move\_count);

fprintf(fp2, "执行时间: %f 秒\n", execution\_time);

// 检查排序稳定性

if (isStable(total, i)) {

fprintf(fp2, "排序结果是稳定的\n");

} else {

fprintf(fp2, "排序结果不稳定\n");

}

fprintf(fp2, "id,date,open,high,low,close,volume\n");

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

fprintf(fp2, "%d,%s, %.2f, %.2f,%.2f, %.2f, %d\n",

total[j].id, total[j].date, total[j].open,

total[j].high, total[j].low, total[j].close,

total[j].volume);

}

//fclose(fp3);

fclose(fp2);

free(total); // 释放内存

return 0;

}