

北京邮电大学

实验报告



题目： 实验三：排序性能分析实验

班 级： 2023211XXX

学 号： 2023XXXXXXXX

姓 名： Yokumi

学 院： 计算机学院

2024 年 12 月 12 日

一、实验目的

- 1、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）；
- 2、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

操作系统为 MacOS 14.5 (23F79)，处理器架构为 arm64。C 标准采用 c17，编译器为 clang-1500.3.9.4。

```
▶(base) yokumi@YokumideMacBook-Air ~ ➤ sw_vers
ProductName:      macOS
ProductVersion:   14.5
BuildVersion:     23F79
```

图 1 操作系统

```
▶(base) yokumi@YokumideMacBook-Air ~ ➤ clang --version
Apple clang version 15.0.0 (clang-1500.3.9.4)
Target: arm64-apple-darwin23.5.0
Thread model: posix
InstalledDir: /Library/Developer/CommandLineTools/usr/bin
```

图 2 编译器环境

工作目录结构如下：

```
├─DS
│   └─Lab3
│       ├──lab3.c    // 源代码
│       └─stock_data.csv // 原始数据表
```

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作稳定性检查）。

3. **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）

4.1 高层数据结构设计

4.1.1 顺序表

本实验中，原始数据存储在数组中，排序直接对该数组进行操作，结果也存在于该数组中并保存至文件。其结构体定义如下：

```
1. typedef double Key_Open_Type;
2. typedef double Key_Close_Type;
3. typedef double Key_High_Type;
4. typedef double Key_Low_Type;
5. typedef long Key_Volume_Type;
6.
7. typedef struct {
8.     int Id;
9.     char Date[12];
10.    Key_Open_Type Open;
11.    Key_Close_Type Close;
12.    Key_High_Type High;
13.    Key_Low_Type Low;
14.    Key_Volume_Type Volume;
15. } StockType;
16.
17. typedef struct {
18.    StockType r[MAXSIZE + 1]; //r[0]闲置或作哨兵
19.    int length;
20. } SqList;
21.
22. typedef struct SqList HeapType; // 完全二叉堆，也用顺序表存储
23.
```

原始数据除 Name 之外的数据项均被存储在 StockType 结构体中。堆排序中堆也用顺序表来储存，因为其完全二叉堆的性质，孩子节点可以方便表示。

4.2 函数/高层算法设计

4.2.1 文件 I/O 操作

实验所需文件 `stock_data.csv`, `drop sample_1k_data.csv` 存放在源代码的工作目录下。对于 `csv` 文件, 每一行的不同数据项用“,”分隔, 在进行读取时, 可以按行读取, 按“,”分隔, 注意跳过表头。

```
1. // 读取 CSV 文件并存入 SqlList
2. void load_csv(const char *filename, SqlList *list) {
3.     FILE *file = fopen(filename, "r");
4.     if (!file) {
5.         perror("无法打开文件");
6.         return;
7.     }
8.
9.     char line[256];
10.    int line_count = 0;
11.
12.    // 跳过表头
13.    fgets(line, sizeof(line), file);
14.
15.    while (fgets(line, sizeof(line), file)) {
16.        if (line_count >= MAXSIZE) {
17.            printf("文件数据超出限制(%d)。\\n", MAXSIZE);
18.            break;
19.        }
20.
21.        // 解析一行数据
22.        StockType stock;
23.        char *token = strtok(line, ",");
24.        stock.Id = atoi(token);
25.
26.        token = strtok(NULL, ",");
27.        strncpy(stock.Date, token, sizeof(stock.Date) - 1);
28.        stock.Date[sizeof(stock.Date) - 1] = '\\0';
29.
30.        token = strtok(NULL, ",");
31.        stock.Open = atof(token);
32.
33.        token = strtok(NULL, ",");
34.        stock.High = atof(token);
35.
36.        token = strtok(NULL, ",");
37.        stock.Low = atof(token);
```

```

38.
39.     token = strtok(NULL, ",");
40.     stock.Close = atof(token);
41.
42.     token = strtok(NULL, ",");
43.     stock.Volume = atol(token);
44.
45.     // 将数据存入 SqlList
46.     list->r[++line_count] = stock;
47. }
48.
49. list->length = line_count;
50.
51. fclose(file);
52. printf("CSV 数据加载完成, 共加载 %d 条记录.\n", list->length);
53. }

```

该实验中, 排序后的结果需要存放至 csv 文件中, 编写函数如下, 其中, Name 数据项直接全部赋为“AABA”, 参数 start 用于区分结果全部输出和部分输出 (附加要求), 若全部输出, start = 1;

```

1. // 保存排序结果到文件
2. void save_to_file(const char *filename, int start, const SqlList *list) {
3.     FILE *file = fopen(filename, "w");
4.     if (!file) {
5.         perror("无法创建文件");
6.         return;
7.     }
8.
9.     // 写入表头
10.    fprintf(file, "Id,Date,Open,High,Low,Close,Volume,Name\n");
11.
12.    // 写入数据
13.    for (int i = start; i <= list->length; i++) {
14.        const StockType *stock = &list->r[i];
15.        fprintf(file, "%d,%s,%.2f,%.2f,%.2f,%.2f,%ld,%s\n",
16.            stock->Id, stock->Date, stock->Open, stock->High, stock->Low,
17.            stock->Close, stock->Volume, "AABA");
18.    }
19.    fclose(file);
20. }

```

4.2.2 排序依据

实验需要对多个 Key 分别进行排序，为增强代码可复用能力，设计关键词比较函数如下：

```
1. // 比较函数
2. int compare_open(const StockType *a, const StockType *b) {
3.     return (a->Open > b->Open) - (a->Open < b->Open);
4. }
5. int compare_close(const StockType *a, const StockType *b) {
6.     return (a->Close > b->Close) - (a->Close < b->Close);
7. }
8. int compare_high(const StockType *a, const StockType *b) {
9.     return (a->High > b->High) - (a->High < b->High);
10. }
11. int compare_low(const StockType *a, const StockType *b) {
12.     return (a->Low > b->Low) - (a->Low < b->Low);
13. }
14. int compare_volume(const StockType *a, const StockType *b) {
15.     return (a->Volume > b->Volume) - (a->Volume < b->Volume);
16. }
```

4.2.3 排序性能记录

由于需要记录不同排序所用的比较次数和移动次数，为避免排序函数的参数过多，将两者设置为全局变量，在每次调用排序函数进行排序前初始化为 0。

```
1. // 计数器，设置为全局变量，省去通过参数传递
2. long compare_count = 0;
3. long move_count = 0;
```

用时用 C 语言的 `time.h` 库函数进行计算。

稳定性是指若序列中任意两个记录的关键字值相同，即 $K_i=K_j$ (i 不等于 j)，排序之后两者相对位置保持不变。由于本实验中，原始数据元素按照 `Id` 升序排序，我们可以直接根据 `Id` 来判断其相对位置是否改变，函数设计如下：

```
1. // 检查稳定性
2. int check_stability(const SqList *list, int (*compare)(const StockType *, const
StockType *)) {
3.     for (int i = 1; i < list->length; i++) {
4.         if (compare(&list->r[i], &list->r[i + 1]) == 0 && list->r[i].Id > list->r[i +
1].Id) {
5.             return 0; // 不稳定
6.         }
7.     }
```

```
8.     return 1; // 稳定
9. }
```

对成交量排序时重复关键字极少但存在：

| | | | | | | | |
|------|-----------|-------|-------|-------|-------|----------|------|
| 1330 | 4/14/2011 | 16.55 | 16.82 | 16.43 | 16.69 | 16599422 | AABA |
| 2524 | 1/13/2016 | 30.89 | 31.17 | 29.32 | 29.44 | 16593732 | AABA |
| 2479 | 11/6/2015 | 34.94 | 35.2 | 33.46 | 34.2 | 16593732 | AABA |
| 122 | 6/27/2006 | 31.85 | 32.22 | 31.32 | 31.51 | 16589398 | AABA |

4.2.4 希尔排序实现

希尔排序通过不断减小步长实现对直接插入排序的优化，对希尔排序的优化主要包括步长序列等，本实验中使用 Sedgewick's increments 作为步长序列，实现代码如下：

```
1. // 希尔排序函数
2. void ShellSort(SqList *list, int (*compare)(const StockType *, const StockType *)) {
3.     // 给定 Sedgewick's 增量
4.     int gaps[] = {1, 5, 19, 41, 109, 209, 505, 929};
5.     int gap_size = 8;
6.
7.     compare_count = 0;
8.     move_count = 0;
9.
10.    for (int g = gap_size - 1; g >= 0; g--) {
11.        int gap = gaps[g];
12.        for (int i = gap + 1; i <= list->length; i++) {
13.            StockType temp = list->r[i];
14.            move_count++;
15.
16.            int j;
17.            for (j = i - gap; j > 0; j -= gap) {
18.                compare_count++;
19.                if (compare(&temp, &list->r[j]) >= 0) break;
20.                list->r[j + gap] = list->r[j];
21.                move_count++;
22.            }
23.            list->r[j + gap] = temp;
24.            move_count++;
25.        }
26.    }
27. }
```

4.2.5 快速排序实现

快速排序的核心思想是对序列的各部分不断划分，直到整个序列按关键码有序，在代码实现上，可以方便地用递归函数实现支点左右子序列的进一步划分：

```
1. // 划分
2. int Partition(Sqlist *list, int low, int high, int (*compare)(const StockType *,
const StockType *)) {
3.     list->r[0] = list->r[low];
4.     move_count++;
5.     StockType pivot = list->r[low]; // 使用临时变量存储支点值
6.     while (low < high) {
7.         while (low < high && compare(&pivot, &list->r[high]) <= 0) {
8.             compare_count++; // 比较次数++
9.             high--;
10.        }
11.        list->r[low] = list->r[high]; // 将较小值移到左侧
12.        move_count++; // 移动次数++
13.        while (low < high && compare(&pivot, &list->r[low]) >= 0) {
14.            compare_count++; // 比较次数++
15.            low++;
16.        }
17.        list->r[high] = list->r[low]; // 将较大值移到右侧
18.        move_count++; // 移动次数++
19.    }
20.    list->r[low] = list->r[0]; // 将支点值放入最终位置
21.    move_count++; // 移动次数++
22.    return low;
23. }
24.
25. // 快速排序递归函数
26. void Qsort(Sqlist *list, int low, int high, int (*compare)(const StockType *, const
StockType *)) {
27.     while (low < high) {
28.         int pivotloc = Partition(list, low, high, compare);
29.         Qsort(list, low, pivotloc - 1, compare); // 先处理左子数组
30.         low = pivotloc + 1; // 尾递归优化：右子数组的递归替换为循环
31.     }
32. }
33.
34. void QuickSort(Sqlist *list, int (*compare)(const StockType *, const StockType *)) {
35.     compare_count = 0;
36.     move_count = 0; // 初始化计数器
37.     Qsort(list, 1, list->length, compare);
38. }
```


4.2.6 堆排序实现

堆排序的算法思想是：

1. 取出根元素，放在 $R[n]$ ；
2. 将剩下的 $n-1$ 个元素重新调整为堆；
3. 再取出根元素，放在 $R[n-1]$ ，将剩下的元素再调整为堆
4. 如此反复，直到取尽堆中所有元素；

主要的问题是，取出根元素后，还需要保持堆的性质，维持性质的代码如下：

```
1. // 维持堆的性质
2. void HeapAdjust(HeapType *H, int root, int length, int (*compare)(const StockType *,
const StockType *)) {
3.     StockType r = H->r[root];
4.
5.     for (int j = 2 * root; j <= length; j *= 2) {
6.         if (j < length && compare(&H->r[j], &H->r[j + 1]) < 0) {
7.             // 比较左右子节点
8.             compare_count++;
9.             j++; // 选择右子节点
10.        }
11.
12.        // 如果根节点已经大于等于子节点，则退出
13.        compare_count++;
14.        if (compare(&r, &H->r[j]) >= 0)
15.            break;
16.
17.        // 将子节点移动到根节点位置
18.        H->r[root] = H->r[j];
19.        move_count++; // 移动次数计数
20.        root = j; // 更新当前根节点
21.    }
22.
23.    H->r[root] = r; // 最后将根节点放入正确位置
24.    move_count++; // 移动次数计数
25. }
```

堆排序的实现代码如下：

```
1. // 堆排序
2. void HeapSort(HeapType *H, int (*compare)(const StockType *, const StockType *)) {
3.     compare_count = 0;
4.     move_count = 0; // 初始化计数器
5.
6.     // 构建初始大顶堆
7.     for (int i = H->length / 2; i > 0; i--) {
```

```

8.         HeapAdjust(H, i, H->length, compare);
9.     }
10.
11.    // 进行排序操作
12.    for (int j = H->length; j > 1; j--) {
13.        // 交换堆顶和最后一个元素
14.        StockType temp = H->r[1];
15.        H->r[1] = H->r[j];
16.        H->r[j] = temp;
17.        move_count += 3; // 一次完整的交换计为三次移动
18.
19.        // 调整剩余元素使其重新满足堆性质
20.        HeapAdjust(H, 1, j - 1, compare);
21.    }
22. }

```

4.2.7 归并排序实现

归并排序的思想是将几个相邻的有序表合并成一个总的有序表。本实验中，采用倍增法实现归并排序，代码如下：

```

1. // 将两张有序表归并为一张有序表
2. void Merge(StockType SR[], StockType TR[], int left, int mid, int right, int
(*compare)(const StockType *, const StockType *)) {
3.     int i = left, j = mid + 1, k = left;
4.
5.     while (i <= mid && j <= right) {
6.         compare_count++;
7.         if (compare(&SR[i], &SR[j]) <= 0) {
8.             TR[k++] = SR[i++];
9.         } else {
10.            TR[k++] = SR[j++];
11.        }
12.        move_count++;
13.    }
14.
15.    while (i <= mid) {
16.        TR[k++] = SR[i++];
17.        move_count++;
18.    }
19.    while (j <= right) {
20.        TR[k++] = SR[j++];
21.        move_count++;
22.    }
23. }

```

```

24.
25. // 归并排序
26. void MergeSort(SqlList *list, int (*compare)(const StockType *, const StockType *)) {
27.     int len = list->length;
28.     StockType *TR = (StockType *)malloc((len + 1) * sizeof(StockType));
29.     if (TR == NULL) {
30.         perror("内存分配失败");
31.         exit(EXIT_FAILURE);
32.     }
33.
34.     for (int step = 1; step < len; step *= 2) { // 每次子序列长度加倍
35.         for (int left = 1; left <= len - step; left += 2 * step) {
36.             int mid = left + step;           // 左子序列的结束位置
37.             int right = (left + 2 * step > len) // 防止越界
38.                 ? len
39.                 : (left + 2 * step);
40.
41.             Merge(list->r, TR, left, mid, right, compare);
42.         }
43.
44.         // 每轮完成后将结果复制回原数组
45.         for (int i = 1; i <= len; i++) {
46.             list->r[i] = TR[i];
47.         }
48.     }
49.
50.     free(TR); // 释放辅助数组
51.     TR = NULL;
52. }

```

4.3 具体实验内容实现及运行结果

4.3.1 排序结果

按照实验要求，分别用希尔排序、快速排序、堆排序和归并排序四种算法，得到排序结果分别存放在以下文件中：

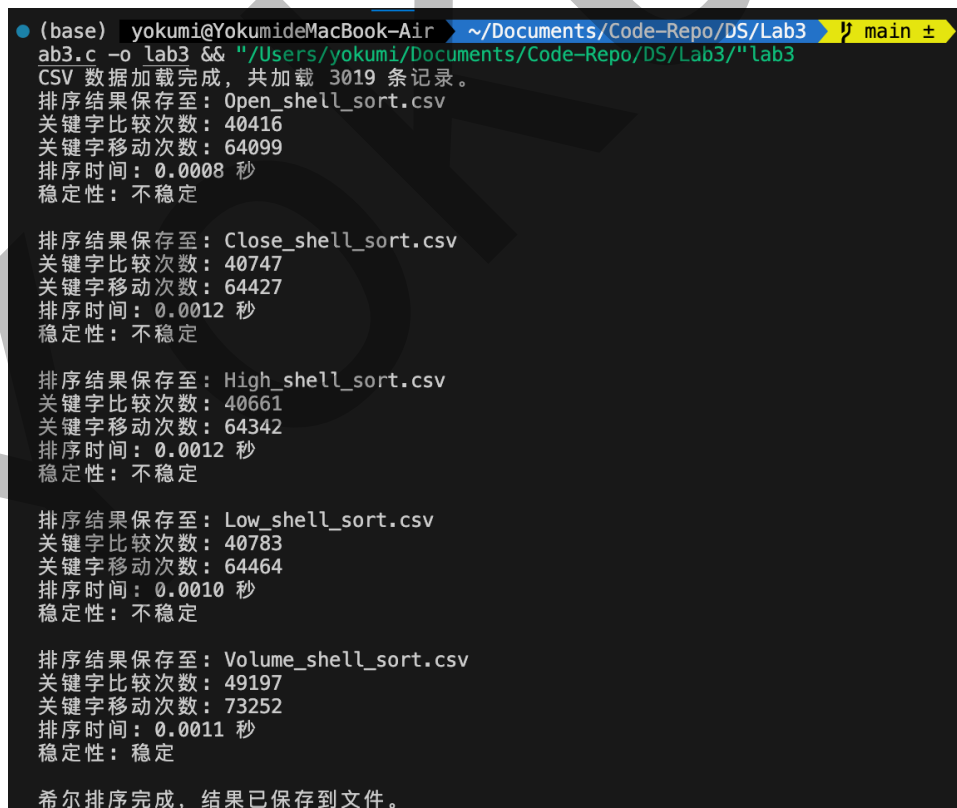
// 希尔排序结果

- ├Open_shell_sort.csv
- ├Close_shell_sort.csv
- ├High_shell_sort.csv
- ├Low_shell_sort.csv
- └Volume_shell_sort.csv

// 快速排序结果

```
└─Open_quick_sort.csv
└─Close_quick_sort.csv
└─High_quick_sort.csv
└─Low_quick_sort.csv
└─Volume_quick_sort.csv
// 堆排序结果
└─Open_heap_sort.csv
└─Close_heap_sort.csv
└─High_heap_sort.csv
└─Low_heap_sort.csv
└─Volume_heap_sort.csv
// 归并排序结果
└─Open_merge_sort.csv
└─Close_merge_sort.csv
└─High_merge_sort.csv
└─Low_merge_sort.csv
└─Volume_merge_sort.csv
```

部分过程截图展示:



```
(base) yokumi@YokumideMacBook-Air ~/Documents/Code-Repo/DS/Lab3 main ±
ab3.c -o lab3 && "/Users/yokumi/Documents/Code-Repo/DS/Lab3/"lab3
CSV 数据加载完成, 共加载 3019 条记录。
排序结果保存至: Open_shell_sort.csv
关键字比较次数: 40416
关键字移动次数: 64099
排序时间: 0.0008 秒
稳定性: 不稳定

排序结果保存至: Close_shell_sort.csv
关键字比较次数: 40747
关键字移动次数: 64427
排序时间: 0.0012 秒
稳定性: 不稳定

排序结果保存至: High_shell_sort.csv
关键字比较次数: 40661
关键字移动次数: 64342
排序时间: 0.0012 秒
稳定性: 不稳定

排序结果保存至: Low_shell_sort.csv
关键字比较次数: 40783
关键字移动次数: 64464
排序时间: 0.0010 秒
稳定性: 不稳定

排序结果保存至: Volume_shell_sort.csv
关键字比较次数: 49197
关键字移动次数: 73252
排序时间: 0.0011 秒
稳定性: 稳定

希尔排序完成, 结果已保存到文件。
```

4.3.2 排序性能比较

我们应用所有排序并按照上图所示的内容输出如下：

```
排序结果保存至：Open_shell_sort.csv
关键字比较次数：40416
关键字移动次数：64099
排序时间：0.0008 秒
稳定性：不稳定

排序结果保存至：Close_shell_sort.csv
关键字比较次数：40747
关键字移动次数：64427
排序时间：0.0012 秒
稳定性：不稳定

排序结果保存至：High_shell_sort.csv
关键字比较次数：40661
关键字移动次数：64342
排序时间：0.0012 秒
稳定性：不稳定

排序结果保存至：Low_shell_sort.csv
关键字比较次数：40783
关键字移动次数：64464
排序时间：0.0010 秒
稳定性：不稳定

排序结果保存至：Volume_shell_sort.csv
关键字比较次数：49197
关键字移动次数：73252
排序时间：0.0011 秒
稳定性：稳定

希尔排序完成，结果已保存到文件。
```

```
排序结果保存至：Open_heap_sort.csv
关键字比较次数：44528
关键字移动次数：42848
排序时间：0.0012 秒
稳定性：不稳定

排序结果保存至：Close_heap_sort.csv
关键字比较次数：44503
关键字移动次数：42831
排序时间：0.0010 秒
稳定性：不稳定

排序结果保存至：High_heap_sort.csv
关键字比较次数：44481
关键字移动次数：42827
排序时间：0.0010 秒
稳定性：不稳定

排序结果保存至：Low_heap_sort.csv
关键字比较次数：44501
关键字移动次数：42761
排序时间：0.0008 秒
稳定性：不稳定

排序结果保存至：Volume_heap_sort.csv
关键字比较次数：44687
关键字移动次数：42372
排序时间：0.0008 秒
稳定性：不稳定

堆排序完成，结果已保存到文件。
```

```
排序结果保存至：Open_quick_sort.csv
关键字比较次数：47528
关键字移动次数：19248
排序时间：0.0008 秒
稳定性：不稳定

排序结果保存至：Close_quick_sort.csv
关键字比较次数：52809
关键字移动次数：19184
排序时间：0.0007 秒
稳定性：不稳定

排序结果保存至：High_quick_sort.csv
关键字比较次数：48528
关键字移动次数：19338
排序时间：0.0009 秒
稳定性：不稳定

排序结果保存至：Low_quick_sort.csv
关键字比较次数：50493
关键字移动次数：19176
排序时间：0.0008 秒
稳定性：不稳定

排序结果保存至：Volume_quick_sort.csv
关键字比较次数：40530
关键字移动次数：20834
排序时间：0.0007 秒
稳定性：不稳定

快速排序完成，结果已保存到文件。
```

```
排序结果保存至：Open_merge_sort.csv
关键字比较次数：27244
关键字移动次数：38242
排序时间：0.0012 秒
稳定性：稳定

排序结果保存至：Close_merge_sort.csv
关键字比较次数：27199
关键字移动次数：38242
排序时间：0.0010 秒
稳定性：稳定

排序结果保存至：High_merge_sort.csv
关键字比较次数：27221
关键字移动次数：38242
排序时间：0.0009 秒
稳定性：稳定

排序结果保存至：Low_merge_sort.csv
关键字比较次数：27174
关键字移动次数：38242
排序时间：0.0009 秒
稳定性：稳定

排序结果保存至：Volume_merge_sort.csv
关键字比较次数：31336
关键字移动次数：38242
排序时间：0.0008 秒
稳定性：稳定

归并排序完成，结果已保存到文件。
```

将结果汇总形成如下表格：

| 排序算法 | 排序依据 | 关键字比较次数 | 关键字移动次数 | 排序时间（秒） | 稳定性 |
|------|--------|---------|---------|---------|-----|
| 希尔排序 | Open | 40416 | 64099 | 0.0008 | 不稳定 |
| | Close | 40747 | 64427 | 0.0012 | 不稳定 |
| | High | 40661 | 64342 | 0.0012 | 不稳定 |
| | Low | 40783 | 64464 | 0.0010 | 不稳定 |
| | Volume | 49197 | 73252 | 0.0011 | 稳定 |

| | | | | | |
|------|--------|-------|-------|--------|-----|
| 快速排序 | Open | 47528 | 19248 | 0.0008 | 不稳定 |
| | Close | 52809 | 19184 | 0.0007 | 不稳定 |
| | High | 48528 | 19338 | 0.0009 | 不稳定 |
| | Low | 50493 | 19176 | 0.0008 | 不稳定 |
| | Volume | 40530 | 20834 | 0.0007 | 不稳定 |
| 堆排序 | Open | 44528 | 42848 | 0.0012 | 不稳定 |
| | Close | 44503 | 42831 | 0.0010 | 不稳定 |
| | High | 44481 | 42827 | 0.0010 | 不稳定 |
| | Low | 44501 | 42761 | 0.0008 | 不稳定 |
| | Volume | 44687 | 42372 | 0.0008 | 不稳定 |
| 归并排序 | Open | 27244 | 38242 | 0.0012 | 稳定 |
| | Close | 27199 | 38242 | 0.0019 | 稳定 |
| | High | 27211 | 38242 | 0.0009 | 稳定 |
| | Low | 27174 | 38242 | 0.0009 | 稳定 |
| | Volume | 31336 | 38242 | 0.0008 | 稳定 |

首先分析每种排序的比较次数、交换次数以及时间复杂度：

1. 希尔排序：最好情况需要 $n-1$ (3018) 次比较，0 次移动，最坏情况需要 $1/2(n^2 - n)$ (4555671) 次比较， $3/2(n^2 - n)$ (13667013) 次移动，时间复杂度在 $O(n) \sim O(n^2)$ 之间，且属于不稳定排序（实验中出现稳定的情况是由于 Volume 中那两个相同关键码的元素恰好没有改变相对位置）。本实验中，原始数据有序性较好；空间存储开销为 $O(1)$ ，哨兵用。

2. 快速排序：每次划分约有 n 次关键码比较，最好情况需要排序趟数为 $\log n$ 次，最坏情况需要 n 次，所以时间复杂度介于 $O(n \log n) \sim O(n^2)$ 之间，同样也是不稳定排序；空间效率上，实验中使用了递归实现，递归调用层次数与二叉树的深度一致，存储开销为 $O(\log n) \sim O(n)$ 之间；

3. 堆排序：堆排序基于完全二叉树，由其性质可知，从根到叶子的筛选，关键码比较次数最多为 $2(k-1)$ 次，交换记录至多 k 次，其中 k 为树高（本实验中为

12)，所以时间复杂度为 $O(n\log n)$ ，实验结果与之接近。空间存储开销为 $O(1)$ ，哨兵用。排序为不稳定排序；

4. 归并排序：对 n 个元素的表，将 n 个元素看作叶子结点，若将两两归并生成的子表看作它们的父结点，则归并过程对应由叶向根生成一棵二叉树的过程，所以归并趟数约等于二叉树的高度，即 $O(\log n)$ ，每趟归并需移动记录 n 次，故时间复杂度为 $O(n\log n)$ ，实验结果与其接近。空间复杂度上，使用了辅助数组 TR，空间存储开销为 $O(n)$ 。且为稳定排序；

4.3.3 附加要求：不完全排序得到成交量 top100 和 top500 清单

要求不完全排序得到根据 Volume 关键字排序的前 100 名和前 500 名，根据上述四种排序算法，一种思路可以改进快速排序：每次仅对第一个子序列划分，直至子序列长度小于等于 k ；长度不足 k ，则再对其后的子序列划分出补足的 length 即可。不过问题是，改进后仅实现了快速选择，如果需要得到顺序排列还要再进行一次排序。

本实验通过改进堆排序来实现，虽然堆排序是完全排序，但是我们可以让它在筛选 k 次后停止。由于堆排序每次筛选的都是序列中的最大值，那么，顺序表中最后 k 项，即为从第 k 个最大的数据元素到最大的数据元素的顺序排列。代码如下：

```
1. // 堆排序得到 top100 和 top500
2. void HeapSortTopK(HeapType *H, int K, int (*compare)(const StockType *, const
StockType *)) {
3.     compare_count = 0;
4.     move_count = 0; // 初始化计数器
5.
6.     // 构建初始大顶堆
7.     for (int i = H->length / 2; i > 0; i--) {
8.         HeapAdjust(H, i, H->length, compare);
9.     }
10.
11.     // 对堆进行调整，只保留前 K 大元素
12.     for (int j = H->length; j > H->length - K; j--) {
13.         StockType temp = H->r[1];
14.         H->r[1] = H->r[j];
15.         H->r[j] = temp;
16.         move_count += 3; // 一次交换计三次移动
17.
18.         // 调整堆以维持堆性质
19.         HeapAdjust(H, 1, K, compare);
20.     }
```

运行截图如下：

```
排序结果保存至：volume_top100.csv  
关键字比较次数：4840  
关键字移动次数：4134  
排序时间：0.0002 秒
```

```
排序结果保存至：volume_top500.csv  
关键字比较次数：9824  
关键字移动次数：8941  
排序时间：0.0003 秒
```

top100 和 top500 清单结果存放在：

```
├─volume_top100.csv  
├─volume_top500.csv
```

五、实验分析（10 分）

实验时整体比较顺利，遇到的问题主要出现在归并排序时，一开始遇到如下问题：

```
lab3(8311,0x1f9c50c00) malloc: Incorrect checksum for freed object 0x132704528:  
probably modified after being freed.
```

```
Corrupt value: 0x4041399999999999a
```

```
lab3(8311,0x1f9c50c00) malloc: *** set a breakpoint in malloc_error_break to debug
```

阅读可知，报错是由于在释放内存后，程序尝试修改已释放的内存地址，或者在释放后内存被非法访问或意外修改所导致的。

通过以下命令进行 Debug：

```
clang -fsanitize=address -g -o lab3 lab3.c  
./lab3
```

发现是因为在归并时开的辅助数组 TR2，在每次 MSort 函数结束后释放 TR2 的动态内存，但是我的归并排序采用递归实现，可能在递归调用中释放了 TR2，而父级函数仍尝试使用该内存导致报错。改为非递归实现后正确运行。

六、实验总结（10 分）

这次实验加深了我对各种排序方法的理解，特别是对于希尔排序、快速排序、堆排序、归并排序的过程清晰明了了很多，对于他们的代码实现也熟悉了不少。特别是对不同排序方法的比较和交换次数有了直观的理解，对各种排序方法的优缺点、性能有了直观的比较。收获最大的还是 Debug 的过程，排序的代码基于板子优化，并且在此时实验中，编写的排序函数以及文件读写、性能分析函数都具有较高的可复用性。