石 家 庄 铁 道 大 学

**实 验 报 告**

**实习六 常用排序算法的对比分析**

题 目： 排序对比分析

班 级： 信1901 –4

姓 名： 闫竞存

学 号：20194127

日 期：2020.11.12

**1. 实验题目**

常用排序算法的对比分析

[问题描述]

各种内部排序算法的时间复杂度分析结果只给出了算法执行时间的阶，或大概执行时间。试通过随机的数据比较各算法的关键字比较次数和关键字移动次数，以取得直观感受。

[基本要求]

对以下常用的内部排序算法进行比较：起泡排序、直接插入排序、简单选择排序、快速排序、希尔排序、堆排序。

[测试数据]

由随机产生器决定。

[实现提示]

待排序表的表长不少于 100；其中的数据要用伪随机数产生程序产生；至少要用 5 组不同的输入数据作比较；比较的指标为有关键字参加的比较次数和关键字移动次数（关键字交换计为 3 次移动）。

**2. 需求分析**

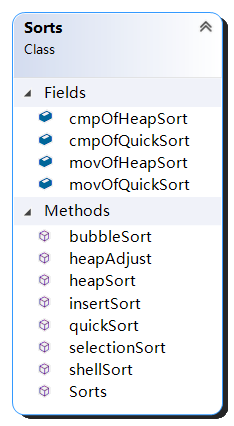
通过比较不同算法对的不同数据量进行排序的时，其关键字移动和比较的次数大小，来比较不同的算法在处理不同数据时候的性能

**3. 概要设计**

程序的主要函数如下

|  |  |  |
| --- | --- | --- |
| 函数名 | 作用 | 返回值 |
| vector<int> Sorts::bubbleSort() | 进行相应的排序，获取返回值 | 含有关键字移动和比较次数的向量 |
| vector<int> Sorts::insertSort() |
| vector<int> Sorts::selectionSort() |
| vector<int> Sorts::shellSort() |
| void Sorts::heapAdjust() | 调整堆 | 无 |
| void Sorts::heapSort() | 堆排序，获取相关的关键字比较和移动次数 | 无 |
| void Sorts::quickSort() | 快速排序，获取相关的关键字比较和移动次数 | 无 |
| vector<vector<int>> getData() | 获取对关键词比较和计算之后的最终数据 | 对比较数据进行计算的最终结果 |
| void main | 主函数 | 程序状态 |

程序的类图如下：

****

**4. 关键代码的详细设计**

关键字移动次数和比较测试均为取5次的平均值的结果。

在代码中，timeOfCompare 为关键字比较的次数， timeOfMove 为关键字移动次数

// 获取每次排序得到的结果取5次的平均的结果，其中包含每一种比较的相关的关键字移动次数和比较次数

vector<vector<int>> getData(int size) {

    srand(time(NULL));

    vector<vector<int>> allRst; // 存储所有比较的所有数据

    int\* nums = new int[size];

    for (int i = 0; i < 5; i++) {

        // 一共进行5次比较

        Sorts s;

        for (int i = 0; i < size; i++) nums[i] = rand() % 10000;

        allRst.push\_back(s.bubbleSort(nums, size));

        for (int i = 0; i < size; i++) nums[i] = rand() % 10000;

        allRst.push\_back(s.insertSort(nums, size));

        for (int i = 0; i < size; i++) nums[i] = rand() % 10000;

        allRst.push\_back(s.selectionSort(nums, size));

        for (int i = 0; i < size; i++) nums[i] = rand() % 10000;

        allRst.push\_back(s.shellSort(nums, size));

        // 由于快排涉及到了递归，所以用了“全局变量”来存储相关的数据

        vector<int> t;

        for (int i = 0; i < size; i++) nums[i] = rand() % 10000;

        s.quickSort(nums, 0, size);

        t.push\_back(s.cmpOfQuickSort);

        t.push\_back(s.movOfQuickSort);

        allRst.push\_back(t);

        // 先清除与快排相关的数据

        t.clear();

        for (int i = 0; i < size; i++) nums[i] = rand() % 10000;

        s.heapSort(nums, size);

        t.push\_back(s.cmpOfHeapSort);

        t.push\_back(s.movOfHeapSort);

        allRst.push\_back(t);

    } //获取所有比较的所有的5次数据

    vector<vector<int>> rst;

    int sumCmp[6] = { 0, 0, 0, 0, 0, 0 }; // 每一种比较的总的比较次数

    int sumMov[6] = { 0, 0, 0, 0, 0, 0 }; // 每一种比较的总的移动次数

    for (int i = 0; i < 5; i++) {

        for (int j = 0; j < 6; j++) {

            sumCmp[j] += allRst[j + i \* 5][0];

            sumMov[j] += allRst[j + i \* 5][1];

        }

    }

    for (int i = 0; i < 6; i++) {

        // 每一种比较的平均比较和移动次数

        vector<int> t;

        t.push\_back(sumCmp[i] / 5);

        t.push\_back(sumMov[i] / 5);

        rst.push\_back(t);

    }

    return rst;

}

// 冒泡排序

vector<int> Sorts::bubbleSort(int\* nums, int n) {

    int timeOfCompare = 0;

    int timeOfMove = 0;

    for (int i = 0; i < n; i++) {

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

            if (timeOfCompare++, nums[j - 1] > nums[j]) {

                timeOfMove += 3;

                swap(nums[j - 1], nums[j]);

            }

        }

    }

    vector<int> rst;

    rst.push\_back(timeOfCompare);

    rst.push\_back(timeOfMove);

    return rst;

}

//插入排序

vector<int> Sorts::insertSort(int\* nums, int n) {

    int timeOfCompare = 0;

    int timeOfMove = 0;

    int temp = 0;

    int i, j;

    if (n <= 2) return vector<int>();

    for(i = 1; i < n; i++){

        temp = nums[i];

        if(timeOfCompare++, nums[i] < nums[i - 1]){

            timeOfMove++;

            nums[i] = nums[i - 1];

        }

        for (j = i - 1; temp < nums[j]; j--) {

            timeOfMove++;

            nums[j + 1] = nums[j];

        }

        nums[j + 1] = temp;

    }

    vector<int> rst;

    rst.push\_back(timeOfCompare);

    rst.push\_back(timeOfMove);

    return rst;

}

// 选择排序

vector<int> Sorts::selectionSort(int\* nums, int n) {

    int timeOfCompare = 0;

    int timeOfMove = 0;

    int maxdir = 0;

    for (int i = 0; i < n; i++) {

        maxdir = 0;

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

            if (timeOfCompare++, nums[j] > nums[maxdir])

                maxdir = j;

        }

        swap(nums[maxdir], nums[n - i - 1]);

        timeOfMove += 3;

    }

    vector<int> rst;

    rst.push\_back(timeOfCompare);

    rst.push\_back(timeOfMove);

    return rst;

}

// 快速排序

void Sorts::quickSort(int\* nums, int first, int last) {

    int low = first;

    int high = last;

    if (low >= high) return;

    int key = nums[low];

    while (low < high) {

        while (this->cmpOfQuickSort++, low < high && nums[high] >= key) high--;

        nums[low] = nums[high];

        while (this->cmpOfQuickSort++, low < high && nums[low] <= key) low++;

        nums[high] = nums[low];

        this->movOfQuickSort += 2;

    }

    nums[low] = key;

    this->movOfQuickSort++;

    quickSort(nums, first, low - 1);

    quickSort(nums, low + 1, last);

}

// 希尔排序

vector<int> Sorts::shellSort(int\* nums, int length) {

    int timeOfCompare = 0;

    int timeOfMove = 0;

    int gap;    //gap是分组的步长

    int temp;   //希尔排序是在直接插入排序的基础上实现的,所以仍然需要哨兵

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

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

            for (int j = i + gap; j < length; j = j + gap) {    //单独一次的插入排序

                if (timeOfCompare++, nums[j] < nums[j - gap]) {

                    temp = nums[j]; //哨兵

                    int k = j - gap;

                    while (timeOfCompare++, k >= 0 && nums[k] > temp) {

                        nums[k + gap] = nums[k];

                        k = k - gap;

                        timeOfMove++;

                    }

                    nums[k + gap] = temp;

                }

            }

        }

    }

    vector<int> rst;

    rst.push\_back(timeOfCompare);

    rst.push\_back(timeOfMove);

    return rst;

}

// 堆排序

void Sorts::heapAdjust(int\* arr, int start, int end) {

    int dad = start;

    int son = dad \* 2 + 1;

    while (son <= end) {

        if (this->cmpOfHeapSort++, son + 1 <= end && arr[son] < arr[son + 1])

            son++;

        if (this->cmpOfHeapSort++, arr[dad] > arr[son])

            return;

        else {

            this->movOfHeapSort += 3;

            swap(arr[dad], arr[son]);

            dad = son;

            son = dad \* 2 + 1;

        }

    }

}

void Sorts::heapSort(int\* nums, int n) {

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

        this->heapAdjust(nums, i, n - 1);

    for (int i = n - 1; i > 0; i--) {

        swap(nums[0], nums[i]);

        this->movOfHeapSort += 3; // 交换看作3次移动

        this->heapAdjust(nums, 0, i - 1);

    }

}

**5. 遇到的问题以及解决方案**

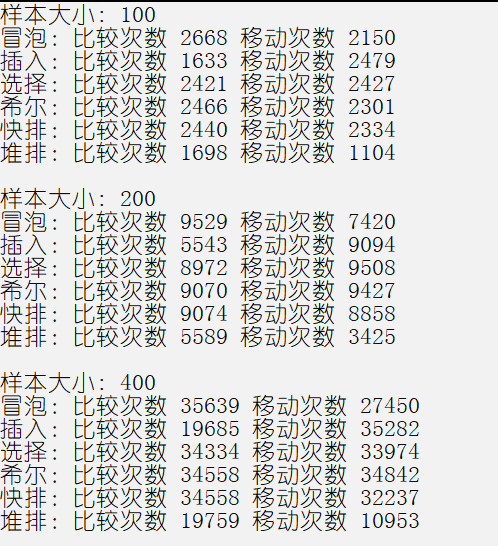
**问题描述**：在进行每一遍排序的时候，往往只有第一次的排序返回了移动次数和比较次数的相关数据，后面就不再返回正确的数据了

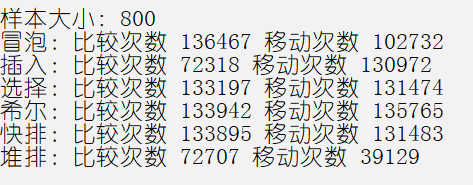
**原因分析**：因为是对数组本身进行排序，所以在每次排序之后再次排序的时候，数组本身就是有序的，因此在排序的时候只进行比较，而不进行移动了。

**解决方案**：在每一次排序完毕之后再次对数组进行赋值，重新对其进行赋予随机值。

**6. 用户使用说明**

直接打开程度即可查看运行结果，打开程序之后显示如下:

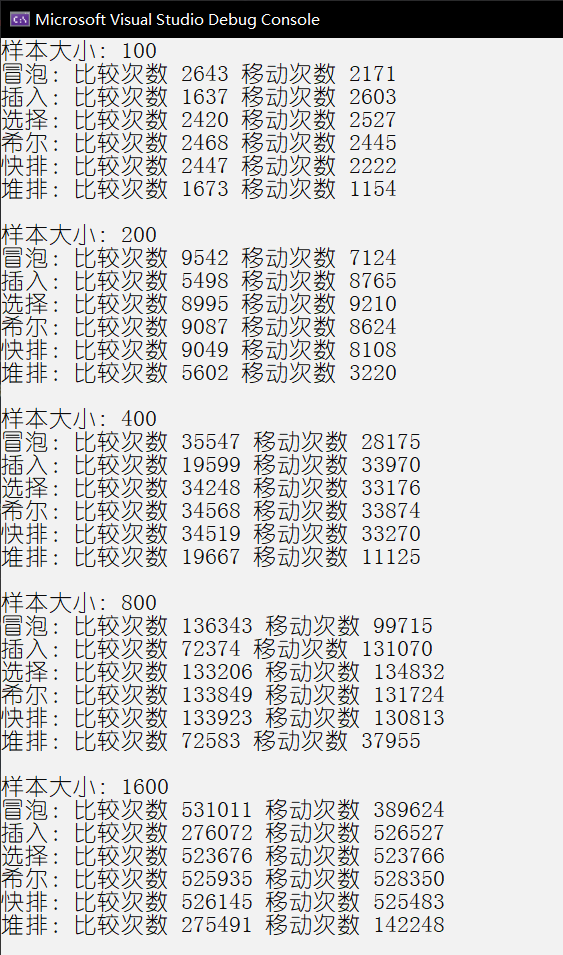


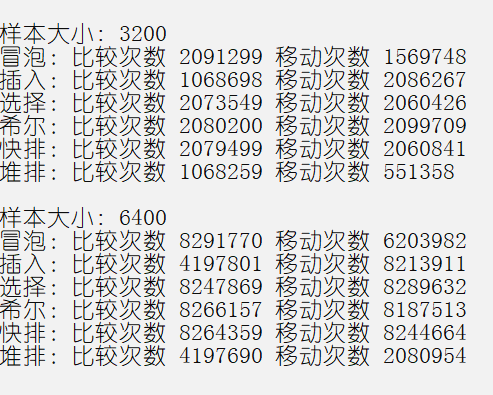




**7. 测试结果**

**第一次测试:**





**第二次测试：**



