SHANGHAI JIAO TONG UNIVERSITY

内部排序算法比较

吕艺

517021910745

December 19, 2018

1 需求分析

1. 本演示文件的主要目的是通过随机数据分析各算法的时间复杂度，并通过比较次数和关键字移动次数直观地分析。

2. 演示文件无需用户输入，程序自动生成数据并打印各算法的比较次数和关键字移动次数。

3. 程序执行的命令包括:
a. 随机生成数据的个数并按照个数生成随机数据
b. 分别对随机生成的数据调用六种内部排序算法
c. 打印各算法的比较次数和关键字的移动次数
d. 类似重复四轮

2 概要设计

本演示文件中的采取的排序算法分别为冒泡排序，直接插入排序，简单选择排序，快速排序，希尔排序和堆排序。1) 冒泡排序

void bubbleSort(int * data, int num);

操作条件：随即数据已生成

操作结果：对随机数据进行排序

2) 直接插入排序

student()state = 0;void simpleInsertSort(int * data, int num);

操作条件：随即数据已生成

操作结果：对随机数据进行排序

3) 简单选择排序

void simpleSelectSort(int * data, int num);

操作条件：随即数据已生成

操作结果：对随机数据进行排序

4) 快速排序

void quickSort(int * data, int num);

void quickSort(int * data, int low, int high,int & comp, int & moves);

int divide(int * data, int low, int high,int & comp, int & moves);

操作条件：随即数据已生成

操作结果：对随机数据进行排序

5) 希尔排序

void shellSort(int * data, int num);

操作条件：随即数据已生成

操作结果：对随机数据进行排序

6) 堆排序

void heapSort(int * data, int num);

void percolate(int * data, int hole, int num, int & comp, int & moves);

操作条件：随即数据已生成

操作结果：对随机数据进行排序

2. 本程序包括两个模块：

1) 主函数模块

int main() {

生成随机数个数

根据随机数个数生成随机数据

利用六种算法对随机数据进行排序并打印比较次数和移动次数

再重复四轮}

2) 排序 *Order* 单元模块--利用六种排序算法对随机数据进行排序

3 详细设计

1）*Order* 单元模块

```
void bubbleSort(int * data, int num)            //冒泡排序
{   int tmp;
    bool flag = true;
    int moves = 0;
    int comp = 0;
    int num1,num2;
    for(int i = 0;i < num; i++)
    {   flag = true;
        tmp = data[i];
        for(int j = 0; j< num - i; j++)
        {   comp ++;                            //从左到右冒泡
            if(data[j] > data[j+1])
            {   num1 = data[j];
                num2 = data[j+1];
                tmp = data[j];
                data[j] = data[j+1];
                data[j+1] = tmp;
                moves += 3;
                flag = false;
            }
        }
        if(flag) break;                         //若在一次起泡过程中没有出现交换现象则代表排序结束
    }
    cout<<"Bubble Sort       : compare times: "<<comp<<" moves: "<<moves<<endl;
}


void simpleInsertSort(int * data, int num)      //简单插入排序
{
    int tmp;
    int i,k;
    int moves = 0;
    int comp = 0;
```

```
    for(i = 1;i < num; i++)
    {
        tmp = data[i];                                  //按顺序将每个值插入到对应的位置
        for(k = i-1; tmp < data[k] && k >= 0; k--)
        {
            data[k+1] = data[k];
            moves ++;
            comp ++;
        }
        data[k] = tmp;
    }
    cout<<"Simple Insert Sort: compare times: "<<comp<<" moves: "<<moves<<endl;
}


void simpleSelectSort(int * data, int num)          //简单选择排序
{
    int tmp;
    int loc = -1;
    int val;
    int moves = 0;
    int comp = 0;
    for(int i = 0; i < num; i++)
    {
        tmp = data[i];                                  //每次选择最小的插入到对应位置
        for(int j = i + 1; j < num; j++)
        {   comp ++;
            if(data[j] < tmp)
            {
                tmp = data[j];
                loc = j;
            }
        }
        val = data[i];
        data[i] = tmp;
        data[loc] = val;
        if(loc != i) moves += 3;
    }
    cout<<"Simple Select Sort: compare times: "<<comp<<" moves: "<<moves<<endl;
}
```

```
int divide(int * data, int low, int high,int & comp, int & moves)
{
    int tmp = data[low];
    while(low < high)                //将数组元素按数组最低位对应的数值排序，左边比它小，右边比它大
    {
        while(low < high && data[high] >= tmp)
        { high --; comp ++;}
        comp ++;
        if(low < high)
        {
            data[low] = data[high];
            moves ++;
        }

        while(data[low] <= tmp && low < high)
        {low ++; comp ++;}
        if(low < high)
        {
            data[high] = data[low];
            moves ++;
        }
    }
    data[low] = tmp;
    moves ++;
    return low;
}


void quickSort(int * data, int low, int high,int & comp, int & moves)    //递归实现快速排序
{
    int mid;
    comp ++;
    if(low >= high) return;

    mid = divide(data, low, high, comp, moves);
    quickSort(data, mid + 1, high, comp, moves);
    quickSort(data, low, mid - 1, comp, moves);

}


void quickSort(int * data, int num)              //快速排序的外部接口
{   int comp = 0;
    int moves = 0;
    quickSort(data, 0 ,num - 1, comp, moves);
    cout<<"Quick Sort       : compare times: "<<comp<<" moves: "<<moves<<endl;
}


void shellSort(int * data, int num)              //希尔排序
```

```
{
    int tmp;
    int i,j;
    int comp = 0;
    int moves = 0;
    for(int step = num / 2; step > 0; step /= 2)        //变间隔排序
    {
        for(i = step; i < num; i++)
        {
            tmp = data[i];
            for(j = i - step; j > 0 && tmp > data[j]; j -= step)
            {
                data[j + step] = data[j];
                moves ++;
                comp ++;
            }
            data[j] = tmp;

        }
    }
    cout<<"Shell Sort........: compare times: "<<comp<<" moves: "<<moves<<endl;
}


void percolate(int * data, int hole, int num, int & comp, int & moves)
{
    int child;
    int tmp = data[hole];

    while(hole * 2 + 1 < num)           //保证堆顶元素是堆内最大的
    {
        child = 2 * hole + 1;
        comp += 2;
        if ( child < num - 1 && data[child + 1] > data[child])
        {
            child ++;
        }

        if(data[child] > tmp)
        {
            data[hole] = data[child];
            moves += 3;
        }
        else
            break;

        hole = child;
        moves ++;
    }
    data[hole] = tmp;
```

```
    moves ++;
}


void heapSort(int * data, int num)                //堆排序
{
    int tmp;
    int i;
    int comp = 0;
    int moves = 0;

    for(int i = num / 2 -1; i >= 0; i--)          //建堆
        percolate(data, i ,num, comp, moves);

    for(int j = num - 1; i > 0; i--)              //堆排序
    {
        tmp = data[0];
        data[0] = data[j];
        data[j] = tmp;
        percolate(data, 0, i, comp, moves);
        moves += 3;
    }

    cout<<"Heap Sort        : compare times: "<<comp<<" moves: "<<moves<<endl;
}
```

2）主函数模块

```
int main() {
    int num;                                  //随机数个数
    int ** data;                              //随机数据生成
    srand((unsigned)time(NULL));
    for(int k = 1; k <= 5; k++) {             //进行五次试验
        num =  rand() % 10000;                //生成随机数个数

        data = new int *[6];
        for(int i = 0; i < 6;i++)
            data[i] = new int [num];

        for (int i = 0; i < num; i++)
        { data[0][i] = rand(); }

        for(int i = 1;i < 6; i++)
            for(int j = 0;j < num; j++)       //为六种排序方式分别准备随机数据
                data[i][j] = data[0][j];

        cout<<endl;
        cout << "Round " << k << ": " << endl;
        cout << "Total num: " << num << endl;
        bubbleSort(data[0], num);             //冒泡排序
        simpleInsertSort(data[1],num);        //简单插入排序
```

```
        simpleSelectSort(data[2],num);          //简单选择排序
        quickSort(data[3],num);                 //快速排序
        shellSort(data[4],num);                 //希尔排序
        heapSort(data[5], num);                 //堆排序

        for(int k = 0;k < 6;k++)                //释放内存
            delete [] data[k];
        delete data;
    }


    return 0;
}
```

4 调试分析

1. 一开始本演示文件仅设计了一个指针来动态生成数据，然而因为指针的本身特性，在调用完第一个函数后，数据已被排序，使后几种排序算法的测试结果失效，因此之后改为生成二维数组来保留原始随机数据。

2. 算法的复杂度分析

1）时间复杂度

冒泡排序的主要思想为两两比较待排序数据元素的大小，发现两个数据元素的次序相反时即进行交换，直到没有反序的数据元素为止。对第 i 个元素需要进行的比较次数为 n-i，最差情况下交换次数为 3n-3i 次，对每个元素求和得冒泡排序的时间复杂度为 O(n^2)。

直接插入排序的主要思想为每次将一个待排序的数据元素，插入到前面已经排好序的数列中的适当位置，使数列依然有序；直到待排序数据元素全部插入完为止。对第 i 个元素最差情况下进行比较的次数为 i-1 次，交换 i-1 次，对每个元素的时间复杂度求和，得到直接插入排序的时间复杂度为 O(n^2)。

简单选择排序的主要思想为每一次从待排序的数据元素中选出最小（或最大）的一个元素，顺序放在已排好序的数列的最后，直到全部待排序的数据元素排完。对第 i 个元素的比较次数为 i-1 次，交换 1 次，对每个元素的时间复杂度求和，得到直接插入排序的时间复杂度为 O(n^2)。

快速排序的主要思想为在当前无序区 R[1..H] 中任取一个数据元素作为比较的" 基准"(不妨记为 X)，用此基准将当前无序区划分为左右两个较小的无序区：R[1..I-1] 和 R[I+1..H]，且左边的无序子区中数据元素均小于等于基准元素，右边的无序子区中数据元素均大于等于基准元素，而基准 X 则位于最终排序的位置上，即 R[1..I-1]X.KeyR[I+1..H](1IH)，当 R[1..I-1] 和 R[I+1..H] 均非空时，分别对它们进行上述的划分过程，直至所有无序子区中的数据元素均已排序为止。快速排序是种不稳定排序，时间复杂度为 O($n\log n$)。

希尔排序的主要思想为先取一个小于 n 的整数 d1 作为第一个增量，把文件的全部记录分成 d1 组。所有距离为 d1 的倍数的记录放在同一组中。先在各组内进行直接插入排序，然后取第二个增量 d2<d1 重复上述的分组和排序，直到所取的增量 dt=1，即所有记录放在同一组中进行直接插入排序为止。该方法实际上是一种分组插入方法。该算法是种不稳定的排序，最坏情况时间复杂度为 O(n^2)，平均时间复杂度为 O($n^{\frac{3}{2}}$)。

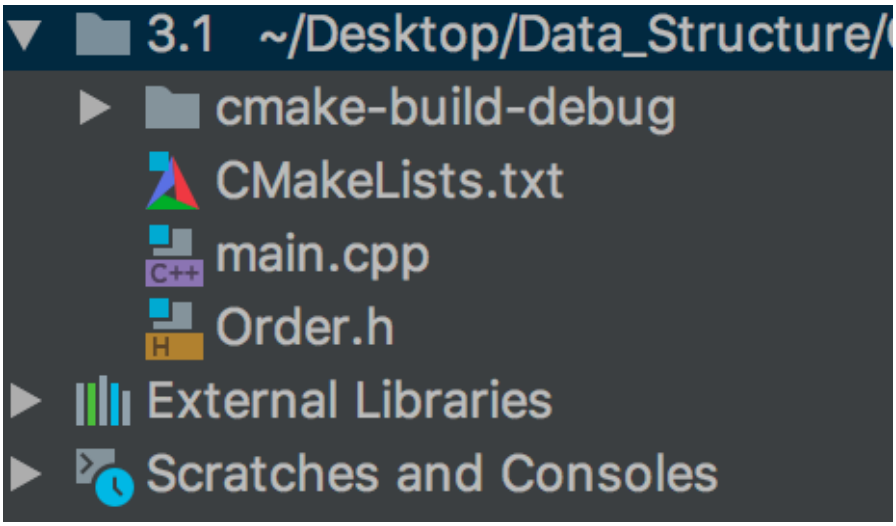
堆排序的主要思想为一树形选择排序，在排序过程中，将 R[1..N] 看成是一颗完全二叉树的顺序存储结构，利用完全二叉树中双亲结点和孩子结点之间的内在关系来选择最小的元素。该方法的时间复杂度为 O($n$log2n)。

2）空间复杂度

Order 排序模块中除了快速排序，其余的的空间复杂度为 $O(1)$，快速排序空间复杂度为 O(log2n) 主函数模块的复杂度取决于定义主函数作用域中的待排序元素个数，故空间复杂度也为 $O(n)$。

5 用户手册

1. 本程序以 Jetbrains Clion 2018.2.5, 采用 C++ 11 标准，程序以项目方式组织 (project)，如图 1 所示：



2. 依次点击菜单"Run"->build, 再点击"Run"，程序就执行六种排序算法

6 测试结果

程序生成随机数据后自动执行六种排序算法，并将五轮生成的随机数个数和各种排序算法的比较次数和移动次数打印在屏幕上。

```
Round 1:
Total num: 1427
Bubble Sort       : compare times: 1018878 moves: 1549761
Simple Insert Sort: compare times: 882220 moves: 882220
Simple Select Sort: compare times: 1017451 moves: 4275
Quick Sort        : compare times: 22369 moves: 6631
Shell Sort        : compare times: 346878 moves: 346878
Heap Sort         : compare times: 5532 moves: 10609
```

根据以上测试结果数据分析，可以得出模拟数据与理论时间复杂度相符。

7 附录

源程序文件名清单

main.cpp　　//主函数

Order.h　　//排序函数单元模块