|  |
| --- |
| 数据结构与算法分析：第一次作业 |
| 算法分析与排序问题 |

|  |
| --- |
| 课程 数据结构与算法分析  姓名 白靖  专业班级 软件52  学号 2151601024  邮箱 516267116@qq.com  提交日期 2016年10月7日 |

## 目录

[目录 0](#_Toc463632558)

[1时间复杂度判断问题 1](#_Toc463632559)

[1.1 作业题目 1](#_Toc463632560)

[1.2 问题分析 1](#_Toc463632561)

[2 递归方程的大O表示与证明 2](#_Toc463632562)

[2.1 作业题目 2](#_Toc463632563)

[2.2 问题解答 2](#_Toc463632564)

[3 几种排序算法的测试 3](#_Toc463632565)

[3.1 作业题目 3](#_Toc463632566)

数据结构与算法分析作业报告：第一次作业

算法分析与排序问题

此报告作为数据结构与算法课程的第一次作业报告，内容包括作业题的题目分析、程序实现、结果展示与算法分析。作业中的程序代码采用Java语言实现并在Windows环境下测试，运行环境参数如下：

|  |  |
| --- | --- |
| 处理器及内存 | CPU：Intel(R) Core(TM) i7-4710HQ @2.50GHz  RAM：16.00GB(DDR3L 1600MHz) |
| 运行操作系统 | MS Windows 10 Home x64 (Version: 10.0.14393) |
| Java运行环境 | Java(TM) SE Runtime Environment  (build 1.8.0\_102-b14) |

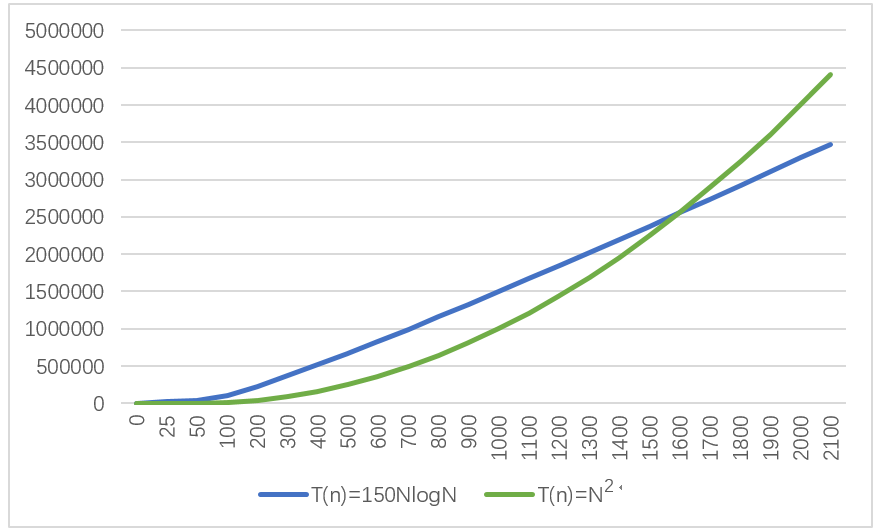
# 1时间复杂度判断问题

## 作业题目

|  |
| --- |
| *程序A和程序B经过分析发现其最坏情形运行时间分别不大于 150NlogN和 N2。如果可能，请回答下列问题：*  *A、对于 N 的大值（N>10000）,哪一个程序的运行时间有更好的保障？*  *B、对于 N 的小值（N<100），哪一个程序的运行时间有更好的保障？*  *C、对于 N=1000，哪一个程序平均运行得更快？*  *D、对于所有可能的输入，程序B是否总够比程序A运行得更快？* |

## 1.2 问题分析

对于增长率分别为150NlogN和N2的两种算法，下图分别对应了其输入规模（横轴）与时间开销（纵轴）的关系：



则由图可知，对于输入规模n较小的情形，增长率为N2的算法的时间开销相对较小;而对于输入规模n较大的情形，则增长率为150NlogN的算法时间开销较小。两种算法的具体时间开销的比较与输入规模有很大的关系，并且还与输入数据的情况（最佳、最差、平均）和数据的分布有关。

1. 对于N的大值（N>10000），程序B的运行时间有更好保障。
2. 对于N的小值（N<100），程序A的运行时间有更好保障。
3. 对于N=1000，两程序的运行速度无法比较，因为题目只给出了最坏情况的运行时间，无法得知平均情况下数据的分布情况，故无法做出判断。
4. 不能判断。两程序的运行时间不仅与数据的输入规模有关，还与输入数据的情况（最佳、最差、平均）和数据的分布有关。且题目只给出了最坏情况的运行时间，所以无法得出所有的输入情况的比较。

# 递归方程的大O表示与证明

## 作业题目

|  |
| --- |
| *考虑以下递归方程，定义函数T(n):*  *A、*  *B、* |
| *请给出A和B两种递归式的大O表示，并证明。* |

## 问题解答

A: 递归式A的大O表示为

以下用数学归纳法证明：

由迭代可得

对于基准情形*n*=1，对,

满足

设，下证对*k+*1也成立：

由

故递归式A的大O表示为。

B：递归式B的大O表示为

以下用数学归纳法证明：

由迭代可得

对于基准情形n=0，对，

满足，

设，下证对*k+*1也成立：

由

故递归式B的大O表示为.

# 几种排序算法的测试

## 作业题目

|  |
| --- |
| *实现直接插入排序、简单选择排序、希尔排序、快速排序和归并排序，以能够对给定数组的正序排序，并按照满足下列情形进行测试：*  *A、测试数组的大小为[100,200,300,…,10000]100种大小*  *B、测试数组中的元素分别为正序、逆序和随机序列*  *对测试的结果需要用图形的方式进行展示：*  *a）、展示每个排序算法在满足条件A和条件B情形下的运行时间趋势变化图，如图 1所示*    *图1 图2*  *b）、将所有排序算法在正序下、逆序下和随机序列下的运行时间的对比图，如图2所示* |

## 程序实现

只给出算法代码，完整代码详与完整测试数据详见百度云链接1

1 <http://pan.baidu.com/s/1kUZdg8n> 密码:iqtq

直接插入排序

//直接插入排序

**public** **class** InsertionSort{

**void** **sort**(**int** [] **array**){

**for**(**int** **curr** = 1; curr < array.length; curr++){

**int** **temp** = array[curr];

// 较大元素右移

**for**(**int** **next** = curr; next > 0 && array[next - 1] > array[next]; next--){

// 交换位置

array[next] = array[next - 1];

array[next - 1] = temp;

}

}

}

}

简单选择排序

//简单选择排序

**public** **class** SelectionSort{

**void** **sort**(**int** [] array){

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

// 记录最小值索引号及其值

**int** **low\_index** = i;

**int** **low\_value** = array[i];

// 遍历数组，寻找最小元素

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

**if**(array[j] < low\_value){

low\_value = array[j];

low\_index = j;

}

}

// 交换最小值与当前元素

**int** **temp** = array[low\_index];

array[low\_index] = array[i];

array[i] = temp;

}

}

}

希尔排序

//希尔排序

**public** **class** ShellSort{

**void** **sort**(**int** [] array){

// 选用已知最好的Marcin Ciura步长

**int** [] **interval** = {1750, 701, 301, 132, 57, 23, 10, 4, 1};

// 从大至小选用间隔

**for**(**int** **i** = 0; i < interval.length; i++){

// 对每个间隔进行直接插入排序

**for** (**int** **j** = interval[i]; j < array.length; j++ ) {

**int** **temp** = array[j];

**for**(**int** **k** = j; k >= interval[i] && array[k] > temp; k -= interval[i]){

array[k] = array[k - interval[i]];

}

array[j - interval[i]] = temp;

}

}

}

}

快速排序

//快速排序（递归实现）

**import** java.util.Random;

**public** **class** QuickSort{

**void** **sort**(**int** [] array){

// 调用递归

subSort(array, 0, array.length - 1);

}

**void** **subSort**(**int** [] array, **int** start, **int** end){

// 随机选取轴值索引pivot

Random **r** = **new** Random();

// 利用Random类的nextInt()函数生成超出数组下标的随机正整数

**int** **pivot** = r.nextInt(end);

// 交换轴值与最后一个元素

**int** **temp** = array[end];

array[end] = array[start + pivot];

array[start + pivot] = temp;

// 将大于轴值的元素放在数组的右半边

**int** **curr** = end;

**for** (**int** **i** = end - 1; i > start; i--) {

**if**(array[i] > array[end]){

curr--;

temp = array[curr];

array[curr] = array[i];

array[i] = temp;

}

}

// 将轴值居中

temp = array[end];

array[end] = array[curr];

array[curr] = temp;

// 递归调用对左右两部分排序

subSort(array, start, curr);

subSort(array, curr + 1, end);

}

}

归并排序

//归并排序

**public** **class** MergeSort {

**void** **sort**(**int**[] array) {

subSort(array, 0, array.length);

}

**void** subSort(**int**[] array, **int** l, **int** r) {

**int** mid = (l + r) / 2;

**if**(l == r) **return**; //只有一个元素时，返回

//左边

subSort(array, l, mid);

//右边

subSort(array, mid + 1, r);

//归并两个有序数组

**int**[] temp = **new** **int**[r - l + 1];

**int** i = l;// 左指针

**int** j = mid + 1;// 右指针

**int** k = 0;

// 把较小的数先移到新数组中

**while** (i <= mid && j <= r) {

**if** (array[i] < array[j]) {

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

} **else** {

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

}

}

// 把左边剩余的数移入数组

**while** (i <= mid) {

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

}

// 把右边边剩余的数移入数组

**while** (j <= r) {

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

}

// 把新数组中的数覆盖array数组

**for** (**int** k2 = 0; k2 < temp.length; k2++) {

array[k2 + l] = temp[k2];

}

}

## 测试结果