数据结构课程设计

项目说明文档

8种排序算法的比较案例

|  |  |
| --- | --- |
| 作者姓名： | 高逸轩 |
| 学 号： | 2053385 |
| 指导教师： | 张 颖 |
| 学院专业： | 软件学院 软件工程 |



同济大学

Tongji University

# 1项目分析

## 1.1 项目需求分析

针对于8种排序算法的比较这一系统，本项目在实现的过程中，考虑并且满足了以下的需求：

* 代码可读性强

本项目在实现过程中，将代码根据功能的不同划分为了不同的代码块，同时进行了合理封装。

* 健壮性

当用户输入的数据不合理时，系统应当给予相应的提示而非直接报错。

* 可视化

该系统通过输出当前正在执行的操作内容，使得用户可以直观的了解到当前操作内容。

## 1.2 项目要求

### 1.2.1 功能要求

用户指定数据规模，系统产生随机数，用以下排序算法：冒泡排序，选择排序，直接插入排序，希尔排序，快速排序，堆排序，归并排序、基数排序分别完成排序，并记录各类排序算法的所用时间、比较次数等信息。

### 1.3.2 输入格式

排序数据范围

### 1.3.3 输出格式

输出各排序算法的排序时间和比较次数。

### 1.3.4 项目示例

# 2 项目实现

本项目共实现了以下8种排序算法：

* 冒泡排序
* 选择排序
* 插入排序
* 希尔排序
* 堆排序
* 快速排序
* 归并排序
* 基数排序

下面将依次对本项目实现的这8种排序算法进行介绍。

# 2.1 冒泡排序

### 2.1.1 算法逻辑

冒泡排序算法是把较小的元素往前调或者把较大的元素往后调。这种方法主要是通过对相邻两个元素进行大小的比较，根据比较结果和算法规则对该二元素的位置进行交换，这样逐个依次进行比较和交换，就能达到排序目的。冒泡排序的基本思想是，首先将第1个和第2个记录的关键字比较大小，如果是逆序的，就将这两个记录进行交换，再对第2个和第3个记录的关键字进行比较，依次类推，重复进行上述计算，直至完成第(n-1)个和第n个记录的关键字之间的比较，此后，再按照上述过程进行第2次、第3次排序，直至整个序列有序为止。排序过程中要特别注意的是，当相邻两个元素大小一致时，这一步操作就不需要交换位置，因此也说明冒泡排序是一种严格的稳定排序算法，它不改变序列中相同元素之间的相对位置关系。

### 2.1.2 算法代码

// 冒泡排序

void BubbleSort(int a[], int begin, int end)

{

    // i记录当前有序的位置

    for (int i = end; i >= begin; i--)

    {

        // j记录当前比较元素

        for (int j = begin; j < i; j++)

        {   // 冒泡

            if (a[j] > a[j + 1])

            {

                swap(a[j], a[j + 1]);

            }

            cnt++;

        }

    }

}

### 2.1.3 算法分析

冒泡排序就是把小的元素往前调或者把大的元素往后调。比较是相邻的两个元素比较，交换也发生在这两个元素之间。所以，如果两个元素相等，是不会再交换的；如果两个相等的元素没有相邻，那么即使通过前面的两两交换把两个相邻起来，这时候也不会交换，所以相同元素的前后顺序并没有改变，所以冒泡排序是一种稳定排序算法。

## 2.2 选择排序

### 2.2.1 算法逻辑

选择排序算法的基本思路是为每一个位置选择当前最小的元素。选择排序的基本思想是，基于直接选择排序和堆排序这两种基本的简单排序方法。首先从第1个位置开始对全部元素进行选择，选出全部元素中最小的给该位置，再对第2个位置进行选择，在剩余元素中选择最小的给该位置即可；以此类推，重复进行“最小元素”的选择，直至完成第(n-1)个位置的元素选择，则第n个位置就只剩唯一的最大元素，此时不需再进行选择。

### 2.2.2 算法代码

// 选择排序

void SelectSort(int a[], int begin, int end)

{

    for (int i = begin; i < end; i++)

    {

        // temp记录要交换的位置

        int temp = i;

        for (int j = i + 1; j <= end; j++)

        {

            // 更新交换位置的记录

            if (a[j] < a[temp]) temp = j;

            cnt++;

        }

        // 交换

        swap(a[i], a[temp]);

    }

}

### 2.2.3 算法分析

选择排序是给每个位置选择当前元素最小的，比如给第一个位置选择最小的，在剩余元素里面给第二个元素选择第二小的，依次类推，直到第n-1个元素，第n个元素不用选择了，因为只剩下它一个最大的元素了。那么，在一趟选择，如果一个元素比当前元素小，而该小的元素又出现在一个和当前元素相等的元素后面，那么交换后稳定性就被破坏了。因此，选择排序是一个不稳定的排序算法。**最坏情况下为，此时待排序列为逆序，或者说接近逆序。最好情况下为，此时待排序列为升序，或者说接近升序。**

## 2.3 插入排序

### 2.3.1 算法逻辑

直接插入排序通过构建有序序列，对于未排序数据，在已排序序列中从后向前扫描，找到相应位置并插入，如此重复，直至完成序列排序。

### 2.3.2 算法代码

// 插入排序

void InsertSort(int a[], int begin, int end)

{

    for (int i = begin + 1; i <= end; i++)

    {

        for (int j = i; j > begin; j--)

        {

            // 从无序区向前比较

            if (a[j] < a[j - 1]) swap(a[j], a[j - 1]);

            cnt++;

        }

    }

}

### 2.3.3 算法分析

直接插入排序是稳定的算法，它满足稳定算法的定义，即：假设在数列中存在a[i]=a[j]，若在排序之前，a[i]在a[j]前面；并且排序之后，a[i]仍然在a[j]前面

## 2.5 希尔排序

### 2.5.1 算法逻辑

希尔排序实质上是一种分组插入方法。它的基本思想是：对于n个待排序的数列，取一个小于n的整数gap将待排序元素分成若干个组子序列，所有距离为gap的倍数的记录放在同一个组中；然后，对各组内的元素进行直接插入排序。 这一趟排序完成之后，每一个组的元素都是有序的。然后减小gap的值，并重复执行上述的分组和排序。重复这样的操作，当gap=1时，整个数列就是有序的。

### 2.5.2 算法代码

// 希尔排序

void ShellSort(int a[], int begin, int end)

{

    // gap为分块大小，初始为数据范围的1/2

    int gap = (end - begin + 1) / 2;

    while (gap)

    {

        for (int k = 0; k < gap; k++)

        {

            for (int i = begin + gap + k; i <= end; i += gap)

            {

                for (int j = i; j > begin + k; j -= gap)

                {

                    if (a[j] < a[j - gap]) swap(a[j], a[j - gap]);

                    cnt++;

                }

            }

        }

        // 缩小gap

        gap /= 2;

    }

}

### 2.5.3 算法分析

希尔排序是按照不同步长对元素进行插入排序，当刚开始元素很无序的时候，步长最大，所以插入排序的元素个数很少，速度很快；当元素基本有序了，步长很小，插入排序对于有序的序列效率很高。希尔排序的时间复杂度会比好一些。同时，希尔排序是不稳定的排序算法。

## 2.6 堆排序

### 2.6.1 算法逻辑

堆排序是指利用堆（Heap）这种数据结构所设计的一种排序算法。堆是一个近似完全二叉树的结构，并同时满足堆积的性质：即子结点的键值或索引总是小于（或者大于）它的父节点。

堆是一类特殊数据结构的简称，通常是一个可以被看做一棵树的的数组对象。堆满足下述两个性质：

1. 堆中某个节点的值总是不大于或者不小于其父节点的值；
2. 堆总是一棵完全二叉树。

在堆排序的过程中，我们利用堆的性质，可以构建大根堆，得到堆中最大的元素放在堆顶。排序时，首先需要**构造初始堆。将给定无序序列构造成一个大顶堆，然后将堆顶元素与末尾元素进行交换，使末尾元素最大。然后固定末尾元素不动，继续调整堆，再将堆顶元素与末尾元素交换，得到第二大元素。如此反复进行交换、重建、交换。n-1次交换后，自然形成了有序的序列。**

### 2.6.2 算法代码

// 堆排序调整函数

void HeapAdjust(int a[], int begin, int i, int length)

{

    // 取出当前元素i，暂存tmp

    int tmp = a[begin + i];

    // 从i结点的左子结点开始,即2i+1处开始

    for (int k = 2 \* i + 1; k < length; k = k \* 2 + 1)

    {

        // 如果左子结点小于右子结点，则k指向右子结点(2k+1表示左子结点，2k+2表示右子结点)

        if (k + 1 < length && a[begin + k] < a[begin + k + 1])  k++;

        cnt++;

        // 如果子节点大于父节点,将子节点值赋给父节点(不用进行交换)

        if (a[begin + k] > tmp)

        {

            cnt++;

            a[begin + i] = a[begin + k];

            i = k;// i指向k,即将较小值放在较大值下方

        }

        // 自下而上进行调整,保证下方子树已为大顶堆

        else break;

    }

    //当for循环结束后，将以i为父结点的树的最大值放在局部堆顶

    a[begin + i] = tmp;

}

// 堆排序

void HeapSort(int a[], int begin, int end)

{

    // 长度

    int length = end - begin + 1;

    // 从最后一个有儿子节点的位置(length/2-1)开始，调整关系

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

        HeapAdjust(a, begin, i, length);

    // 将大根堆顶与最后一个元素调整位置

    for (int j = length - 1; j > 0; j--)

    {

        swap(a[begin], a[begin + j]);// 将堆顶元素与末尾元素进行交换

        HeapAdjust(a, begin, 0, j);  // 固定尾部位置，重新对堆进行调整

    }

}

### 2.6.3 算法逻辑

堆排序是一种选择排序，整体主要由构建初始堆+交换堆顶元素和末尾元素并重建堆两部分组成。其中构建初始堆经推导复杂度为，在交换并重建堆的过程中，需交换n-1次，而重建堆的过程中，根据完全二叉树的性质，可以认为每次调整重建的时间复杂度为**，**所以堆排序时间复杂度一般认为就是级。堆排序是一种不稳定的排序算法。

## 2.7 快速排序

### 2.7.1 算法逻辑

快速排序的基本思想是:通过一趟排序算法把所需要排序的序列的元素分割成两大块，其中，一部分的元素都要小于或等于另外一部分的序列元素，然后仍根据该种方法对划分后的这两块序列的元素分别再次实行快速排序算法，排序实现的整个过程可以是递归的来进行调用，最终能够实现将所需排序的无序序列元素变为一个有序的序列。

### 2.7.2 算法代码

// 快速排序

void QuickSort(int a[], int begin, int end)

{

    if (begin > end) return;

    // 比较指针

    int i = begin, j = end;

    while (i != j)

    {

        // a[begin]设置为比较基准

        while (a[j] >= a[begin] && j > i) j--;

        while (a[i] <= a[begin] && j > i) i++;

        if (j > i)  swap(a[i], a[j]);

        cnt += 3;

    }

    // 交换指针最终位置和基准位置

    swap(a[i], a[begin]);

    // 以指针最终位置为划分，分为两部分向下递归

    QuickSort(a, begin, i - 1);

    QuickSort(a, i + 1, end);

}

### 2.7.3 算法分析

快速排序的平均时间复杂度也是。因此，该排序方法被认为是目前最好的一种内部排序方法。

从空间性能上看，尽管快速排序只需要一个元素的辅助空间，但快速排序需要一个栈空间来实现递归。最好的情况下，即快速排序的每一趟排序都将元素序列均匀地分割成长度相近的两个子表，所需栈的最大深度为log n+1。这样，快速排序的空间复杂度为

同时，快速排序是一个不稳定的排序算法。

## 2.8 归并排序

### 2.8.1 算法逻辑

归并排序算法就是把序列递归划分成为一个个短序列，以其中只有1个元素的直接序列或者只有2个元素的序列作为短序列的递归出口，再将全部有序的短序列按照一定的规则进行排序为长序列。归并排序融合了分治策略，即将含有n个记录的初始序列中的每个记录均视为长度为1的子序列，再将这n个子序列两两合并得到n/2个长度为2(当凡为奇数时会出现长度为l的情况)的有序子序列；将上述步骤重复操作，直至得到1个长度为n的有序长序列。需要注意的是，在进行元素比较和交换时，若两个元素大小相等则不必刻意交换位置，因此该算法不会破坏序列的稳定性，即归并排序也是稳定的排序算法。

### 2.8.2 算法代码

// 归并排序

void MergeSort(int a[], int begin, int end)

{

    if (begin == end) return;

    // 找到中点位置

    int mid = (begin + end) / 2;

    // 分为两部分向下递归排序

    MergeSort(a, begin, mid);

    MergeSort(a, mid + 1, end);

    // 定义指针

    int i = begin, j = mid + 1, k = begin;

    // temp用来暂时存储当前部分的元素

    int\* temp= new int[100001];

    // 只要两部分都没有走到结尾

    while (i <= mid && j <= end)

    {

        // 从两部分当前指针位置选择较小元素加入temp

        if (a[i] <= a[j]) temp[k++] = a[i++];

        else temp[k++] = a[j++];

        cnt++;

    }

    // 将未选择的元素加入temp

    while (i <= mid) temp[k++] = a[i++];

    while (j <= end) temp[k++] = a[j++];

    // 将排序后temp的信息传递给a

    for (int i = begin; i <= end; i++) a[i] = temp[i];

    delete[]temp;

}

### 2.8.3 算法分析

不管元素在什么情况下都要做这些步骤，所以花销的时间是不变的，所以该算法的最优时间复杂度和最差时间复杂度及平均时间复杂度都是一样的为：

归并的空间复杂度就是那个临时的数组和递归时压入栈的数据占用的空间：n + logn；所以空间复杂度为:

归并排序算法中，归并最后到底都是相邻元素之间的比较交换，并不会发生相同元素的相对位置发生变化，故是稳定性算法。

## 2.10 基数排序

### 2.10.1 算法逻辑

将所有待比较数值（这里以正整数为例）统一为同样的数位长度，数位较短的数前面补零。设一整数的十进制位数表示为k1,k2...kd，其中k1为高位，kd为低位。然后，从最低位开始，依次进行一次排序。这样从最低位排序一直到最高位排序完成以后, 数列就变成一个有序序列。即先从kd开始排序，再对kd-1进行排序，依次重复，直到对k1排序后便得到一个有序序列。

### 2.10.2 算法代码

// 基数排序子函数

void \_RadixSort(int a[], int begin, int end, int exp)

{

    int\* result = new int[100001];    //存放桶中收集数据后的临时数组

    int bucket[10] = { 0 };    //初始化10个桶

    // 遍历A，将数据出现的次数存储在桶bucket中

    for (int i = begin; i <= end; i++)

    {

        bucket[(a[i] / exp) % 10]++;

        cnt++;

    }

    // 调整bucket各元素的值，调整后的值就是A中元素在result中的位置

    for (int i = 1; i < 10; i++)

        bucket[i] = bucket[i] + bucket[i - 1];

    // 将a中的元素填充到result中（从后往前排，先入后出）

    for (int i = end; i >= begin; i--)

    {

        int iexp = (a[i] / exp) % 10;

        result[bucket[iexp] - 1] = a[i];

        bucket[iexp]--;

        cnt++;

    }

    // 将排序好的数组result复制回A中

    for (int i = begin; i <= end; i++)

    {

        a[i] = result[i - begin];

        cnt++;

    }

    delete[]result;

}

// 基数排序

void RadixSort(int a[], int begin, int end)

{

    // 获取数组中的最大值

    int infoMax = 0;

    for (int i = begin; i <= end; i++) infoMax = infoMax > a[i] ? infoMax : a[i];

    // 从个位开始，对数组a按位进行基数排序

    for (int iexp = 1; infoMax / iexp > 0; iexp \*= 10)

        \_RadixSort(a, begin, end, iexp);

}

# 3 各排序算法对比

通过对各种排序算法实现方法的过程分析，可以得出它们的时间复杂度、空间复杂度和稳定性，如下所示：

|  |  |  |  |
| --- | --- | --- | --- |
| **算法** | **时间复杂度** | **空间复杂度** | **稳定性** |
| 冒泡排序 |  |  | 是 |
| 选择排序 |  |  | 否 |
| 插入排序 |  |  | 是 |
| 希尔排序 |  |  | 否 |
| 堆排序 |  |  | 否 |
| 快速排序 |  |  | 否 |
| 归并排序 |  |  | 是 |
| 基数排序 |  |  | 是 |

# 4 项目测试

# 5 心得体会与拓展

在本次作业中，我掌握了多种排序的基础方法，并对它们的稳定性、时空复杂度进行了理论分析。

在实际应用中，排序的知识点也有很多，例如**冒泡排序可以把元素和交换过程设置为双向的，便成为了鸡尾酒排序**；**稳定的归并排序常常用来求解逆序对问题；快速排序如果设置不同的比较基准则会一定程度上避免退化……**

当然，除了以上提到的排序方式，还有像Timsort这种更加优秀的排序方法值得我们认真学习。当Timsort运行在部分排序好的数组里面的时候，需要的比较次数要远小于nlogn，也是远小于相同情况下的归并排序算法需要的比较次数。但是和其他的归并排序算法一样，最坏情况下的时间复杂度是的水平。但是在最坏的情况下，Timsort需要的临时存储空间只有n/2，在最好的情况下，需要的额外空间是常数级别的。从各个方面都能够击败需要空间和稳定时间的归并排序。具体介绍可见以下链接。

<https://blog.csdn.net/bryansun/article/details/105182778?ops_request_misc=%257B%2522request%255Fid%2522%253A%2522164029065016780274125315%2522%252C%2522scm%2522%253A%252220140713.130102334..%2522%257D&request_id=164029065016780274125315&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~sobaiduend~default-2-105182778.pc_search_em_sort&utm_term=%E6%9C%80%E5%BF%AB%E7%9A%84%E6%8E%92%E5%BA%8F&spm=1018.2226.3001.4187>

现在在学习中，我们通常用到的是快速排序。为避免其退化，我们常常会在每次比较中设置不同位置的比较基准。下面这段代码，由我在高中竞赛时期完成，在每次比较时所取**比较基准是序列中点而非两端**，可以避免退化：

void qsort(int l,int r)//应用二分思想

{

if(l>r)

return ;

int mid=a[(l+r)/2];//中间数

int i=l,j=r;

while(i<j)

{

while(a[i]<mid) i++;//查找左半部分比中间数大的数

while(a[j]>mid) j--;//查找右半部分比中间数小的数

if(i<=j)//如果有一组不满足排序条件（左小右大）的数

{

swap(a[i],a[j]);//交换

i++;

j--;

}

}

if(l<j) qsort(l,j);//递归搜索左半部分

if(i<r) qsort(i,r);//递归搜索右半部分

}

由于篇幅原因，报告内还有很多内容与解释没有展示，请老师和助教老师再移步源程序，在其中的注释写了每一步过程的详解。