项目说明文档

数据结构课程设计

——8种排序算法的比较案例

作 者 姓 名： 何慧琳

学 号： 2152343

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 分析

## 1.1 背景分析

在生活中的各个领域都经常用到排序，如学校的学生信息系统有时要将学生信息按考号排列，有时要按名字拼音排列，又比如购物软件上各种排行榜按销量、好评率等排列……此外，排序也是很多算法的重要组成部分，由此，一个好的排序算法显得尤为重要。

## 1.2 功能分析

此项目将对现有常见8种内排序算法（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）进行比较。算法的分析一般要考虑时间复杂度和空间复杂度，其中排序算法时间复杂度主要看比较次数和交换次数。本项目将产生若干随机数（由用户输入随机数数量），进行用户所选择的排序算法，输出排序用时和交换次数。

# 2 设计

## 2.1 数据结构设计

如上功能分析所述，该系统首先需要一个数据表保存产生的随机数，然后还要有8种排序算法的函数。

## 2.2 成员与操作设计

**数据表结点类（element）**

**私有成员：**

int key;//关键码，此项目中key即生成随机数的数值

int link;//后面一个元素的序号

**公有操作：**

element() :key(0),link(0) {};

int getKey(int i = 0); //提取关键码

void setKey(const int x) { key = x; }//设置元素关键码

void setLink(int l) { link = l; }//设置后一个元素的序号

int getLink() { return link; }//取下一个元素序号

element\* operator =(element& x) { this->key = x.key; this->link = x.link; return this; }

**数据表类（datalist）**

**私有成员：**

element\* vector;//存储向量

int MaxSize, CurrentSize;//最大与当前个数

**公有操作：**

datalist(int Maxsz=DefaultSize):MaxSize(Maxsz),CurrentSize(Maxsz)

{//构造函数

vector = new element[Maxsz];

}

void swap(element& x, element& y)

{//交换

element temp = x;

x = y;

y = temp;

}

## void setdata(int i, int k) { vector[i].key = k; }//初始化数据表中第i个元素

**8个排序算法函数：**

void BubbleExchange(datalist& list, const int i, int& exchange);

void BubbleSort(datalist& list);//冒泡排序

int SelectExchange(datalist& list, const int i);

void SelectSort(datalist& list);//选择排序

int Insert(datalist& list, int i);

void InsertSort(datalist& list);//直接插入排序

int ShellInsert(datalist& list, const int gap);

void ShellSort(datalist& list);//希尔排序

int Partition(datalist& list, int low, int high, int& change\_time);

int QuickSort(datalist& list, int left, int right)//快速排序

int FilterDown(datalist& heap, const int i, const int EndOfHeap);

void HeapSort(datalist& list);//堆排序

int merge(datalist& initlist, datalist& mergedlist, const int m1, const int m2, const int n);

int MergePass(datalist& initlist, datalist& mergedlist, const int len);

void MergeSort(datalist& list);//归并排序

void RadixSort(datalist& list, const int d, const int radix);//基数排序

## 2.3 系统设计

系统首先在main()函数中完成对屏幕的初始化，提示用户输入产生随机数的数量和选择排序算法，根据用户的输入产生相应数量的随机数调用相应的函数对生成的随机数进行排序，输出此次排序用时和交换次数。

# 3 实验现象及分析

## 3.1 冒泡排序算法

### 核心代码

int pass = 1;//已排好序的元素个数（也即遍历趟数）

int exchange = 1;//一趟遍历中是否有元素交换的标识

long long int change\_time = 0;//统计交换次数

while (pass < list.CurrentSize && exchange) {

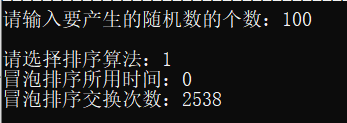
BubbleExchange(list, pass, exchange);

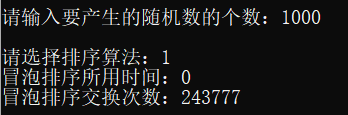
change\_time += exchange;//交换次数累加

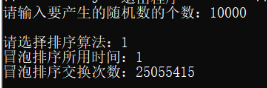
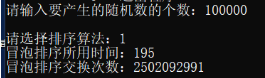
pass++;

}

### 性能测试截屏示例





### 冒泡排序算法分析

首先从存储开销来看，冒泡排序只在BubbleExchange中用到一个附加对象保存交换对象的值，因此存储开销不大。但从所用时间上来看，由上面的实验现象可见当数据数量较小时排序所用时间还算客观，但当随机数个数达到上万时，所用时间和交换次数就显著增长。因此冒泡排序的优点是存储开销较小，且具有稳定性，缺点是速度慢、用时长。

## 3.2 选择排序算法

### 选择排序算法核心代码

for (int i = 0; i < list.CurrentSize-1; i++)

change\_time+=SelectExchange(list, i);

int SelectExchange(datalist& list, const int i)

{//两两比较，将最小的元素交换到第i个位置

int k = i;

int change\_time = 0;

for (int j = i + 1; j < list.CurrentSize; j++) {

if (list.vector[j].getKey() < list.vector[k].getKey())

k = j;

if (k != i) {

list.swap(list.vector[i], list.vector[k]);

change\_time++;//交换次数加一

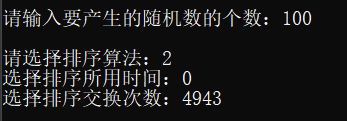
}

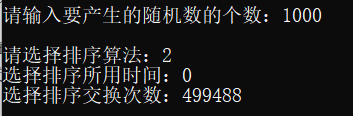
}

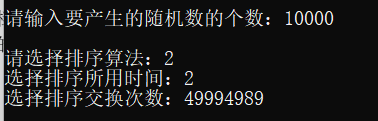
return change\_time;

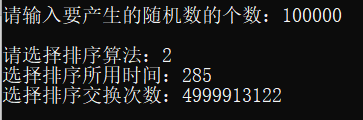
}

### 性能测试截屏示例









### 选择排序算法分析

选择排序算法就是不断的选择队列中最小的元素交换到前面尚未确定的第一个位置。同冒泡排序一样，选择排序的存储开销也只有一个附加对象保存交换对象的值。其计算时间和交换次数也与冒泡排序相当，在随机数个数较小时用时较短，但当个数达到上万用时和交换次数骤增，程序运行时也需要等很长时间才能计算出来。所以选择排序算法的优点是存储开销小，缺点是在交换过程中存在远距离交换，不具有稳定性，计算时间在样本容量大的时候很长。

## 3.3 直接插入排序

### 核心代码

for (int i = 1; i < list.CurrentSize; i++)

change\_time+=Insert(list, i);

int Insert(datalist& list, int i)

{//将第i个元素在list的第0-（i-1）个位置中找到符合顺序的插入位置

int change\_time=0;//统计插入一个元素的比较次数

element temp = list.vector[i];

int j = i;

//将j挪到首个不比vector[i]大的元素后,比它大的元素都往后挪一位

while (j > 0 && temp.getKey() < list.vector[j - 1].getKey()) {

list.vector[j] = list.vector[j - 1];

j--;

change\_time++;

}

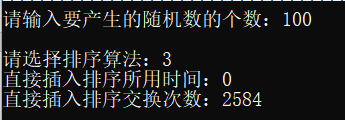
list.vector[j] = temp;//插入vector[i]到第j个位置

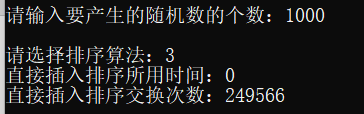
change\_time++;

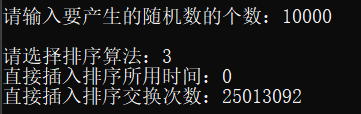
return change\_time;

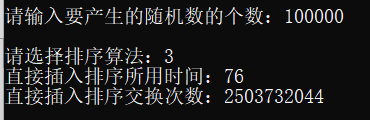
}

### 性能测试截屏示例









### 直接插入排序算法分析

直接插入排序算法的存储开销也只用到一个附加的临时变量保存要交换对象的值。其用时相比前两个算法较少，即使在随机数个数达到一万时计算时间仍然为0，在个数达到十万时计算时间也比前两个算法好很多，不过能看出其用时随着个数的增势也较大，不难设想当个数进一步增大时其用时也会很大。因此直接插入排序算法的优点是存储开销小，在一定数据量范围内用时短。缺点是当数据量大到一定值时，随着数据量增多其用时也会明显增多。

## 3.4 希尔排序

### 核心代码

int gap = list.CurrentSize / 2;//子序列间隔

while (gap) {//循环，直到gap为零

change\_time+=ShellInsert(list, gap);//对各子序列用直接插入排序法排序

gap = gap == 2 ? 1 : (gap / 2.2);//修改gap大小

}

int ShellInsert(datalist& list, const int gap)

{//对list中相隔gap的元素分到一段序列，用直接插入排序法对每段序列进行排序

int change\_time = 0;//统计一趟希尔排序交换次数

for (int i = gap; i < list.CurrentSize; i++) {

int j = i;

//将vector[i]在其gap序列中找到合适位置插入

element temp = list.vector[i];

while (j >= gap && list.vector[j].getKey() < list.vector[j - gap].getKey()) {

list.vector[j] = list.vector[j -= gap];

change\_time++;

}

list.vector[j] = temp;

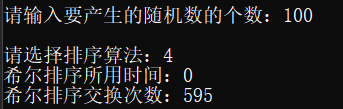
change\_time++;

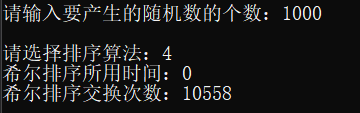
}

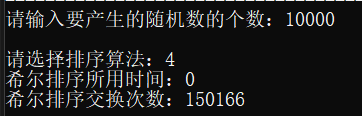
return change\_time;

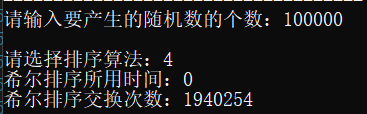
}

### 性能测试截屏示例









### 希尔排序算法分析

希尔排序的存储开销仅一个附加的临时element类变量用于保存要交换的对象。从上面的实验现象可见其计算时间很短，相比前三个算法优化不少，不仅在数值较小时所用时间和交换次数都很少，在数值达到一万甚至十万时其用时仍然没有显著变化，交换次数也不像前三个算法那样骤增，但是希尔排序中存在远距离交换，因此是不稳定的。由上分析可知，希尔排序的优点是存储开销小，计算时间短。缺点是不具有稳定性。

## 3.5 快速排序

### 核心代码

int QuickSort(datalist& list, int left, int right)

{//在待排序区间left~right中递归进行快速排序

static int change\_time;

if (left < right) {

int pivotpos= Partition(list, left, right,change\_time);//划分左右子区间,统计交换次数

QuickSort(list, left, pivotpos-1);//在左子区间递归进行快速排序

QuickSort(list, pivotpos + 1, right);//在右子区间递归进行快速排序

}

return change\_time;

}

int Partition(datalist& list, int low, int high,int& change\_time)

{//将比基准位置元素小的元素都交换到基准元素左边，pivotpos返回经划分后基准元素的位置

int pivotpos = low;//设置初始基准位置

element& pivot = list.vector[low];

for (int i = low + 1; i <= high; i++) {

int j = i;

if (list.vector[i].getKey() < pivot.getKey() && ++pivotpos != i) {

//每找到一个比pivot小的元素将pivotpos往后移一位，

//此时若pivotpos指向的元素不为刚找到的元素，则交换二者

list.swap(list.vector[i], list.vector[pivotpos]);

change\_time++;//交换次数+1

}

}

//将pivot与pivotpos指向的元素，即最后一个比pivot小的元素交换位置

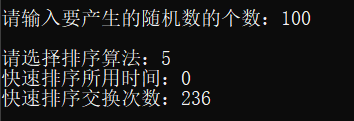
list.swap(pivot, list.vector[pivotpos]);

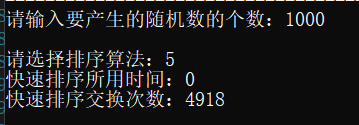
change\_time++;

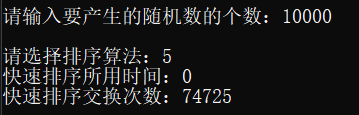
return pivotpos;

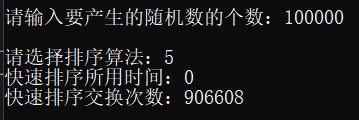
}

### 性能测试截屏示例









### 快速排序算法分析

由代码可知快速排序是递归的，需要有一个栈存放每层递归调用时的指针和参数，最大递归调用曾次数与递归树的深度一致，理想情况为log2(n+1)，因此，要求存储开销为O(log2n);最坏情况是待排序对象已经按关键码从小到大排好序，递归树成为单支树，则占用附加存储将达到O(n)。从上面的实验现象可见快速排序算法的所用时间和交换次数在当前随机数个数范围内是最短的，但是从理论分析来讲，最不理想情况是待排序序列已按从小到大有序排好， 递归树成为单支树，层数最大，此时其排序速度退化到简单排序的水平，比直接插入排序还慢。此外当产生随机数的个数大到一定值是将出现很多重复的数字，这时就会出现每次划分结束后基准位置只往后移动一个位置的情况，用时将大大增加。所以，快速排序算法的优点是在一定数据量范围内用时短、交换次数少，缺点是存储开销较大，并且存在远距离交换是不稳定的。

## 3.6 堆排序算法

### 核心代码

for (int i = (list.CurrentSize - 2) / 2; i >= 0; i--)//从最后一个非叶结点开始

change\_time+=FilterDown(list,i, list.CurrentSize - 1);//将表转化为初始堆

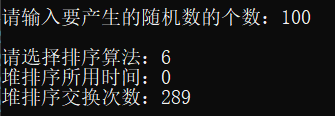
for (int i = list.CurrentSize - 1; i >= 1; i--) {

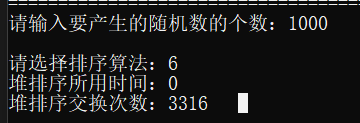
list.swap(list.vector[0], list.vector[i]);//将本趟的堆顶，即尚未排好序的元素中最大的元素交换到从后往前数第一个未确定的位置

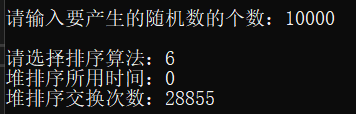
change\_time+=FilterDown(list, 0, i - 1);//重建最大堆

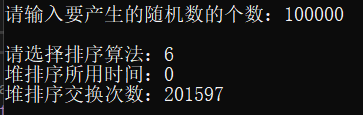
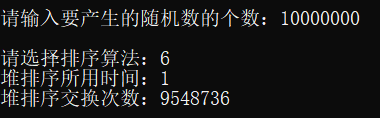
}

### 性能测试截屏示例







### 堆排序算法分析

该算法的附加存储主要是在第二个for循环中用来执行对象交换时所用的一个临时对象。因此该算法的空间复杂度为O(1)。其所用时间和交换次数从上面的实验现象可见在十万以内的数据量范围内非常可观，甚至比快速排序还要优越，经后续实验即使到了一千万，其用时也没有明显增加，到了一亿的时候比前三种算法在十万的时候用时还短。但是堆排序也存在远距离交换，是不稳定的。因此堆排序算法的优点是存储开销小，计算时间短。缺点是不稳定。

## 3.6 归并排序算法

### 核心代码

datalist templist(list.MaxSize);//创建一个辅助数据表

int len = 1;

while (len < list.CurrentSize) {

comp\_time+=MergePass(list, templist, len);

len \*= 2;

comp\_time+=MergePass(templist, list, len);

len \*= 2;

}

int MergePass(datalist& initlist, datalist& mergedlist, const int len)

{//进行一趟归并排序

int comp\_time=0;//统计比较次数

int i = 0;

while (i + 2 \* len <= initlist.CurrentSize - 1) {

comp\_time +=merge(initlist, mergedlist, i, i + len,i + 2 \* len);

i += 2 \* len;

}

if (i + len <= initlist.CurrentSize - 1)

//剩余序列长度大于len小于2len，拆成len长度+剩余的一段序列，将二者合并

comp\_time += merge(initlist, mergedlist, i, i + len,initlist.CurrentSize);

else for (int j = i; j <= initlist.CurrentSize - 1; j++)

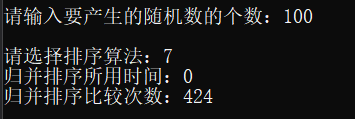
//剩余序列长度小于len直接照搬

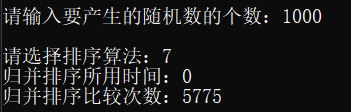
mergedlist.vector[j] = initlist.vector[j];

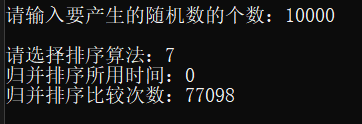
return comp\_time;

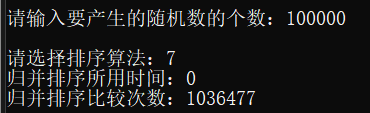
}

### 性能测试截屏示例









### 归并排序算法分析

该算法的附加存储需要另外一个与原待排序对象数组同样大小的辅助数组，存储开销较大。其所用时间和交换次数从上面的实验现象可见仍比前三个算法改进不少，在十万以内的数据量范围内用时和交换次数基本与快速排序算法相当，其比较次数比快速排序的交换次数略大，但是它不存在远距离交换，是一个稳定的排序方法。因此堆排序算法的优点是计算时间短，稳定。缺点是存储开销大。

## 3.6 基数排序算法

### 核心代码

int \*rear=new int[radix], \*front=new int[radix];//开辟n个队列，front和rear分别指向队头队尾

for (int i = 1; i < list.CurrentSize; i++)

list.vector[i].setLink(i + 1);

list.vector[list.CurrentSize-1].setLink(0); //静态链表初始化

int current = 1; //链表扫描指针

for (int i = d; i > 0; i--) { //做 d 趟分配.收集

for (int j = 0; j < radix; j++) front[j] = 0;

while (current != 0) { //逐个对象分配

int k = list.vector[current].getKey(i);

//取当前对象关键码的第 i 位

if (front[k] == 0) //原链表为空,对象链入

front[k] = current;

else //原链表非空,链尾链入

list.vector[rear[k]].setLink(current);

rear[k] = current; //修改链尾指针

current = list.vector[current].getLink();

change\_time++;

}

int j = 0; //从0号队列开始收集

while (front[j] == 0) j++; //空队列跳过

current = front[j];

list.vector[0].setLink(current);

int last = rear[j];

change\_time++;

for (int k = j + 1; k < radix; k++) //逐个队列链接

if (front[k]) {//队列不为空,收集

list.vector[last].setLink(front[k]);

last = rear[k];

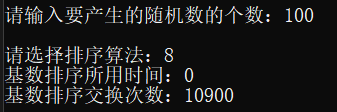
change\_time++;

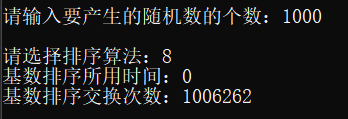
}

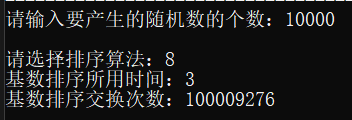
list.vector[last].setLink(0);

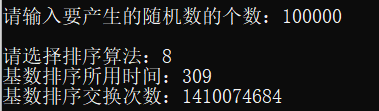
}

### 性能测试截屏示例









### 基数排序算法分析

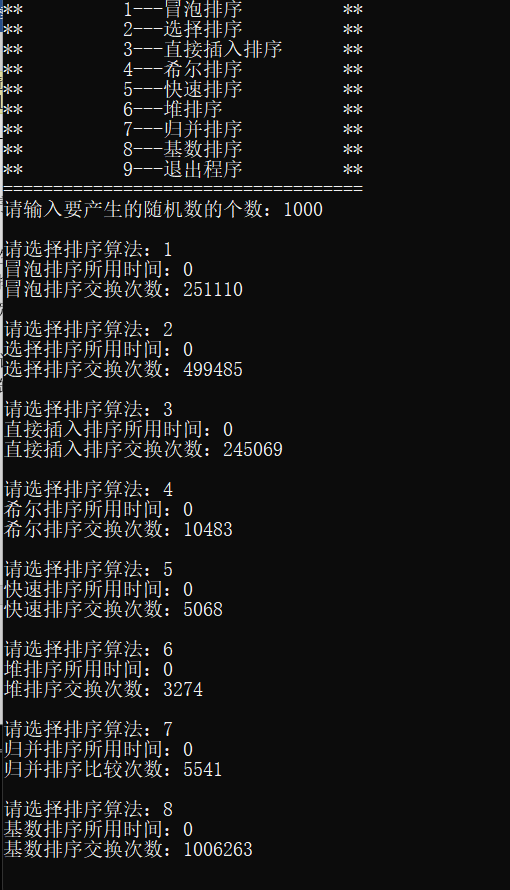
基数排序算法需要对每个元素增加一个链接指针，还要开辟基数个队列，每个队列需要头尾两个指针，因此该算法的附加存储是n+2radix个附加链接指针，存储开销较大。其所用时间和交换次数从上面的实验现象可见在当前确定基数的方案下都非常大是所有算法中用时最长的，但是它不存在远距离交换，是一个稳定的排序方法。因此堆排序算法的优点是稳定。缺点是存储开销大、在radix较大时用时很长。

# 4 测试

## 4.1 功能测试

**测试用例**：随机数个数1000

**实验结果**



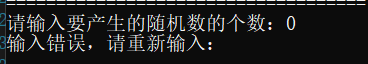
## 4.2 边界测试

### 4.2.1 初始化输入数据无效

**测试用例：**初始输入结点个数为0或1

**预期结果：**给出错误提示，程序运行正常不崩溃。

**实验结果：**



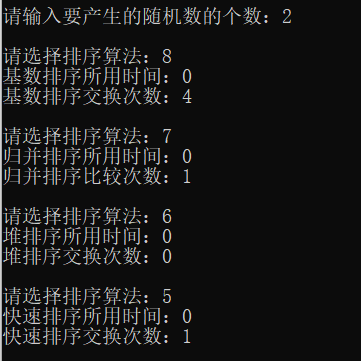


### 4.2.2 初始化输入数据最少

**测试用例：**初始输入结点个数为2

**预期结果：**程序运行正常。

**实验结果：**



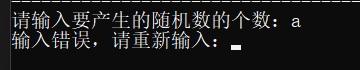
## 4.3 出错测试

### 4.3.1 随机数个数输入错误

**测试用例：**输入考生人数为a

**预期结果：**程序给出提示信息，程序正常运行不崩溃。

**实验结果：**

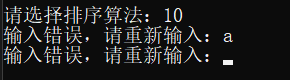
****

### 4.3.2 操作码错误

**测试用例：**输入操作码错误

**预期结果：**程序给出提示信息，程序正常运行不崩溃。

**实验结果：**

****