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： 张喆

学 号： 1754060

指 导 教 师： 张颖

学院、 专业： 软件学院 软件工程

同济大学

Tongji University

目 录

[1 分析 1](#_Toc495668153)

[1.1 背景分析 1](#_Toc495668154)

[1.2 功能分析 1](#_Toc495668155)

[2 设计 2](#_Toc495668156)

[2.1 数据结构设计 2](#_Toc495668157)

[2.2 类结构设计 2](#_Toc495668158)

[2.3 成员与操作设计 2](#_Toc495668159)

[2.4 系统设计 3](#_Toc495668160)

[3 实现 4](#_Toc495668161)

[3.1 插入功能的实现 4](#_Toc495668162)

[3.1.1 插入功能流程图 4](#_Toc495668163)

[3.1.2 插入功能核心代码 5](#_Toc495668164)

[3.1.3 插入功能截屏示例 6](#_Toc495668165)

[3.2 删除功能的实现 7](#_Toc495668166)

[3.2.1 删除功能流程图 7](#_Toc495668167)

[3.2.2 删除功能核心代码 7](#_Toc495668168)

[3.2.3 删除功能截屏示例 8](#_Toc495668169)

[3.3 查找功能的实现 9](#_Toc495668170)

[3.3.1 查找功能流程图 9](#_Toc495668171)

[3.3.2 查找功能核心代码 9](#_Toc495668172)

[3.3.3 查找功能截图示例 10](#_Toc495668173)

[3.4 修改功能的实现 11](#_Toc495668174)

[3.4.1 修改功能流程图 11](#_Toc495668175)

[3.4.2 修改功能核心代码 11](#_Toc495668176)

[3.4.3 修改功能截屏示例 12](#_Toc495668177)

[3.5 统计功能的实现 13](#_Toc495668178)

[3.5.1 统计功能流程图 13](#_Toc495668179)

[3.5.2 统计功能核心代码 13](#_Toc495668180)

[3.5.3 统计功能截屏示例 14](#_Toc495668181)

[3.6 总体系统的实现 14](#_Toc495668182)

[3.6.1 总体系统流程图 14](#_Toc495668183)

[3.6.2 总体系统核心代码 14](#_Toc495668184)

[3.6.3 总体系统截屏示例 16](#_Toc495668185)

[4 测试 1](#_Toc495668186)7

[4.1 功能测试 1](#_Toc495668187)7

[4.1.1 插入功能测试 1](#_Toc495668188)7

[4.1.2 删除功能测试 18](#_Toc495668189)

[4.1.3 查找功能测试 19](#_Toc495668190)

[4.1.4 修改功能测试 19](#_Toc495668191)

[4.1.5 统计功能测试 19](#_Toc495668192)

[4.2 边界测试 2](#_Toc495668193)0

[4.2.1 初始化无输入数据 2](#_Toc495668194)0

[4.2.2 删除头结点 2](#_Toc495668195)0

[4.2.3 删除后链表为空 2](#_Toc495668196)1

[4.3 出错测试 2](#_Toc495668197)2

[4.3.1 考生人数错误 2](#_Toc495668198)2

[4.3.2 操作码错误 2](#_Toc495668199)2

[4.3.3 插入位置不存在 2](#_Toc495668200)2

[4.3.4 删除考号不存在 2](#_Toc495668201)3

[4.3.5 查找考号不存在 2](#_Toc495668202)3

[4.3.6 修改考号不存在 2](#_Toc495668203)3

1 分析

1.1 背景分析

排序在计算机数据处理中经常遇到，特别是在事务处理中，排序占了很大的比重。在日常的数据处理中，一般认为1/4的时间用在排序上，而对于安装程序，多大50%的时间花费在对表的排序上。

所谓排序，就是根据排序码递增或递减的顺序，吧数据元素一次排列起来，使一组任意排列的元素变成一组按其排序码线性有序的元素。

1.2 性能分析

排序算法的执行时间是衡量算法好坏的最重要的参数。排序的时间开销可用算法执行中**数据比较次数**与**数据移动次数**来衡量。

各算法运行时间代价的大略估算一般都按平均情况进行估算。对于那些受元素排序码序列初始排列及元素个数影响较大的，需要按最好情况和最欢情况进行估算。

1.3 定义

1. 排序的定义：对一个无序的序列进行排序的过程。

输入：n个数：a1,a2,a3,…,an。

输出：n个数的排列:a1,a2,a3,…,an，使得a1<=a2<=a3<=…<=an。

2. 排序的稳定性：相同值的节点相对位置是否会发生改变。

稳定：如果a原本在b前面，而a=b，排序之后a仍然在b的前面。

不稳定：如果a原本在b的前面，而a=b，排序之后a可能会出现在b的后面。

3. 排序的时间复杂度：一个算法执行所耗费的时间，一般在三种情况下考虑：最好情况、最坏情况、平均情况。

4. 空间复杂度：运行完一个程序所需内存的大小。

2 冒泡排序

## 2.1 基本思想

设待排序对象序列中 的对象个数为 n。最多作 n-1 趟，i = 1, 2, …, n-2 。在第 i 趟中顺次两两比较v[n-j-1].Key和 v[n-j].Key，j = n-1, n-2, …, i。 如果发生逆序，则交换v[n -j-1]和v[n-j]。

## 2.2 算法实现

### 2.2.1 冒泡排序核心代码

/\*冒泡排序\*/

void BubbleSort(vector<int> sequence)

{

Time\_Swap record("冒泡排序");

bool exchange = true;

record.Start();

//////////////////////////////////////////////////////////////////////////

for (int i = 0; i < sequence.size() - 1 && exchange; ++i)

{

exchange = false;

for (int j = 0; j < sequence.size() - i - 1; ++j)

{

if (sequence[j] > sequence[j + 1])

{

swap(sequence[j], sequence[j + 1]);

record.growExchange(3); //每次调用swap执行三次交换

exchange = true;

}

}

}

//////////////////////////////////////////////////////////////////////////

record.End();

record.showTime();

record.showExchange();

}

### 2.2.2 冒泡排序截屏示例

### 2.2.3 冒泡排序算法分析

1. 冒泡排序只需要一个值的空间用于交换节点，所以**空间复杂度**为O(1)。

2. **时间复杂度**

最坏情况：序列为逆序状态，则每一轮遍历都需要n次交换位置，所以时间复杂度为O(n^2)。   
最好情况：序列为正序状态，每一轮遍历不需要交换位置，所以时间复杂度为O(n)。   
平均情况：每一轮遍历需要n/2次交换位置，所以时间复杂度依然为O(n^2)。

3. **稳定性**

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，不会发生交换。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，也不会发生交换，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。

3 选择排序

## 3.1 基本思想

## 3.2 算法实现

### 3.2.1 选择排序核心代码

/\*选择排序\*/

void SelectSort(vector<int> sequence)

{

Time\_Swap record("选择排序");

int minindex;

record.Start();

//////////////////////////////////////////////////////////////////////////

for (int i = 0; i < sequence.size() - 1; ++i)

{

minindex = i;

record.growExchange(1);

for (int j = i + 1; j < sequence.size(); ++j)

{

if (sequence[j] < sequence[minindex])

{

minindex = j;

record.growExchange(1);

}

}

if (minindex != i)

{

swap(sequence[i], sequence[minindex]);

record.growExchange(3);

}

}

//////////////////////////////////////////////////////////////////////////

record.End();

record.showTime();

record.showExchange();

### 3.2.2 选择排序截屏示例

### 3.2.3 选择排序算法分析

1. 此过程需要一个额外的空间保存最小值用于交换，所以**空间复杂度**为O(1)。

2. **时间复杂度**

序列无论是正序还是逆序状态，每一轮的最小值需要比较到最后才能确定，所以最坏情况和最好情况下都需要 比较n次，再加上遍历整个序列的O(n)，总的复杂度为O(n^2)，平均情况的复杂度也是O(n^2)。

3. **稳定性**

直接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，如果当前元素比一个元素小，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。举个例子，序列5 8 5 2 9， 第一遍选择时第1个元素5会和2交换，那么原序列中2个5的相对前后顺序就被破坏了，所以选择排序不是一个稳定的排序算法。

4 直接插入排序

## 4.1 基本思想

直接插入排序的基本思想是：当插入第i (i ≥ 1)个对象时，前面的V[0], V[1], …, V[i-1] 已经排好序。这时，用v[i]的关键码与v[i-1] ,v[i-2] …的关键码顺序进行比较，找到插入位置即将v[i]插入，原来位置上的对象向后顺移。

## 4.2 算法实现

### 4.2.1 直接插入排序核心代码

/\*直接插入排序\*/

void InsertSort(vector<int> sequence)

{

Time\_Swap record("直接插入排序");

int i, j, temp;

record.Start();

//////////////////////////////////////////////////////////////////////////

for (i = 1; i < sequence.size(); ++i)

{

temp = sequence[i];

record.growExchange(1);

for (j = i - 1; j >= 0 && sequence[j] > temp; --j)

{

sequence[j + 1] = sequence[j];

record.growExchange(1);

}

sequence[j+1] = temp;

record.growExchange(1);

}

//////////////////////////////////////////////////////////////////////////

record.End();

record.showTime();

record.showExchange();

}

### 4.2.2 直接插入排序截屏示例

### 4.2.3 直接插入排序算法分析

1. 过程中需要一个额外的空间保存一个值用于交换节点，所以**空间复杂度**为O(1)。

2. **时间复杂度**

最坏情况：当待排序序列正好为逆序状态，首先遍历整个序列，之后一个个地将待插入元素放在已排序的序列最前面，之后的所有元素都需要向后移动一位，所以比较和移动的时间复杂度都是O(n)，再加上遍历整个序列的复杂度，总复杂度为O(n^2)。

最好情况：当待排序序列正好为正序状态，则遍历完整个序列，当插入元素时，只比较一次就够了，所以时间复杂度为O(n)。

平均情况：当被插入的元素放在已排序的序列中间位置时，为平均情况，比较和移动的时间复杂度为O(n/2)，所以总的时间复杂度依然为O(n^2)。

3. **稳定性**

插入排序是在一个已经有序的小序列的基础上，一次插入一个元素。当然，刚开始这个有序的小序列只有1个元素，就是第一个元素。比较是从有序序列的末尾开始，也就是想要插入的元素和已经有序的最大者开始比起，如果比它大则直接插入在其后面，否则一直往前找直到找到它该插入的位置。如果碰见一个和插入元素相等的，那么插入元素把想插入的元素放在相等元素的后面。所以，相等元素的前后顺序没有改变，从原无序序列出去的顺序就是排好序后的顺序，所以插入排序是稳定的。

5 希尔排序

## 5.1 基本思想

设待排序对象序列有 n 个对象，首先取一个整数 gap < n 作为间隔，将全部对象分为 gap 个子序列，所有距离为 gap 的对象放在同一个子序列中，在每一个子序列中分别施行直接插入排序。然后缩小间隔 gap ，例如取 gap = ⎡gap/2 ⎤ ，重复上述的子序列划分和排序工 作。直到最后取 gap == 1 ，将所有对象放在同一个序列中排序为止。

## 5.2 算法实现

### 5.2.1 希尔排序核心代码

/\*希尔排序\*/

void ShellInsert(vector<int> &sequence, int gap, Time\_Swap &record)

{

int i, j, k, temp;

for (i = gap; i < sequence.size(); ++i)

{

temp = sequence[i];

record.growExchange(1);

for (j = i; j >= gap && temp < sequence[j - gap]; j -= gap)

{

sequence[j] = sequence[j - gap];

record.growExchange(1);

}

sequence[j] = temp;

record.growExchange(1);

}

}

void ShellSort(vector<int> sequence)

{

Time\_Swap record("希尔排序");

record.Start();

//////////////////////////////////////////////////////////////////////////

for (int gap = sequence.size() / 2; gap >= 1; gap = (gap == 2 ? 1 : gap / 2.2))

{

ShellInsert(sequence, gap, record);

}

//////////////////////////////////////////////////////////////////////////

record.End();

record.showTime();

record.showExchange();

showSequence(sequence);

}

### 5.2.2 希尔排序截屏示例

### 5.2.3 希尔排序算法分析

1. 在这个过程中也只需要一个额外的空间保存一个值用于交换节点，所以**空间复杂度**为O(1)。

2. 对指定的待排序对象序列，可以准确的估算关键码的比较次数和对象的移动次数。但想要弄清关键码比较次数和对象移动次数与增量选择之间的依赖关系还无法做到。

3. 希尔排序是进行多次直接插入排序的算法，由于多次插入排序，虽然每一次插入排序是稳定的，不会改变相同元素的相对顺序，但在不同的插入排序过程中，相同的元素可能在各自的插入排序中移动，最后其稳定性就会被打乱，所以希尔排序是不稳定的。

6 快速排序

## 6.1 基本思想

任取待排序对象序列中的某个对象 (例如取第一个对象) 作为基准，按照该对象的关键码大小，将整个对象序列划分为左右两个子序列：

左侧子序列中所有对象的关键码都小于或等于基准对象的关键码

右侧子序列中所有对象的关键码都大于基准对象的关键码

基准对象则排在这两个子序列中间 (这也是该对象最终应安放的位置 ) 。

然后分别对这两个子序列重复施行上述方法，直到所有的对象都排在相应位置上为止。

## 6.2 算法实现

### 6.2.1 快速排序核心代码

/\*快速排序\*/

int Partition(vector<int> &sequence, int low, int high, Time\_Swap &record)

{

/\*选定基准为首元素\*/

int pivotpos = low;

int pivot = sequence[low];

record.growExchange(1);

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

{

if (sequence[i] < pivot && ++pivotpos != i)

//如果第一个条件满足则意味着这个元素应当被丢到基准前面

//换句话说, 基准要往后移一个位置, 腾出一个位置给这个更小的元素

//所以++pivotpos

{

swap(sequence[i], sequence[pivotpos]);

//现在的pivotpos位置是一个比基准大的元素, 反正它比基准大, 直接换到后面也无妨

//反正一会还要递归调快排 (个人认为这是快排的进化所在

record.growExchange(3);

}

}

/\*最后把基准从原来的low位置搬到应该到的位置pivotpos\*/

swap(sequence[low], sequence[pivotpos]);

record.growExchange(3);

/\*返回把原序列分为两部分的位置\*/

return pivotpos;

}

void QuickSort(vector<int> &sequence, int left, int right, Time\_Swap &record)

{

if (left < right)

{

/\*以pivotpos为中心, 把序列划分成两个部分\*/

int pivotpos = Partition(sequence, left, right, record);

/\*分别对左右部分递归快排\*/

QuickSort(sequence, left, pivotpos - 1, record);

QuickSort(sequence, pivotpos + 1, right, record);

}

}

void QuickSort(vector<int> sequence)

{

Time\_Swap record("快速排序");

int i, j, temp;

record.Start();

//////////////////////////////////////////////////////////////////////////

QuickSort(sequence, 0, sequence.size() - 1, record);

//////////////////////////////////////////////////////////////////////////

record.End();

record.showTime();

record.showExchange();

showSequence(sequence);

}

### 6.2.2 快速排序截屏示例

### 6.2.3 快速排序算法分析

1. 首先就地快速排序内部使用的空间是O(1)的，而真正消耗空间的是递归调用，因为每次递归就要保持一些数据，每一次都平分数组的情况下**空间复杂度**为O(logn) ，最差的情况下空间复杂度为O(n)。

2. **时间复杂度**   
最坏情况：每一次选取的基准元素都是最大或最小的，复杂度为O(n^2)   
最好情况：每一次选取的基准元素都能平分整个序列，由于快排涉及到递归调用，所以时间复杂度为O(nlog2n)。   
平均情况：平均情况下复杂度也是O(nlog2n)。

3. **稳定性**

快速排序有两个方向，左边的i下标一直往右走，当a[i]<=a[center\_index]，其中center\_index是中枢元素的数组下标，一般取为数组第0个元素。而右边的j下标一直往左走，当a[j]>a[center\_index]。如果i和j都走不动了，i<=j， 交换a[i]和a[j],重复上面的过程，直到i>j。 交换a[j]和a[center\_index]，完成一趟快速排序。在中枢元素和a[j]交换的时候，很有可能把前面的元素的稳定性打乱，比如序列为 5 3 3 4 3 8 9 10 11，现在中枢元素5和3(第5个元素，下标从1开始计)交换就会把元素3的稳定性打乱，所以快速排序是一个不稳定的排序算法，不稳定发生在中枢元素和a[j] 交换的时刻。

7 堆排序

## 7.1 基本思想

堆排序分为两个步骤：第一步，根据初始输入数据，利用堆的调整算法FilterDown( ) 形成初始堆，第二步，通过一系列的对象交换和重新调整堆进行排序。

## 7.2 算法实现

### 7.2.1 堆排序核心代码

/\*堆排序\*/

void FilterDown(vector<int> &heap, int i, const int EndOfHeap, Time\_Swap &record)

{

int current = i;

int child = 2 \* i + 1;

int temp = heap[i];

record.growExchange(1);

while (child <= EndOfHeap)

{

if(child+1<EndOfHeap &&

(heap[child]<heap[child+1]))

//如果右孩子大则切换到右孩子

{

child = child + 1;

}

if (temp >= heap[child]) { break; }

else

{

heap[current] = heap[child];

record.growExchange(1);

current = child; //切换到下一层

child = 2 \* child + 1;

}

}

heap[current] = temp;

record.growExchange(1);

}

void HeapSort(vector<int> sequence)

{

Time\_Swap record("堆排序");

int i, j;

record.Start();

//////////////////////////////////////////////////////////////////////////

/\*形成初始最大堆\*/

for (i = (sequence.size() - 2) / 2; i >= 0; --i)

//从最后一个非叶子结点开始逐步调成正最大堆

{

FilterDown(sequence, i, sequence.size() - 1, record);

}

for (i = sequence.size() - 1; i >= 1; --i)

{

swap(sequence[0], sequence[i]); //交换堆顶和堆底元素

record.growExchange(3);

FilterDown(sequence, 0, i - 1, record); //重建最大堆

}

//////////////////////////////////////////////////////////////////////////

record.End();

record.showTime();

record.showExchange();

}

### 7.2.2 堆排序截屏示例

### 7.2.3 堆排序算法分析

1. 堆排序需要两个个值的空间来存储临时变量，用于交换节点，一次用于存储子树最大节点用于交换子节点，一次用于存储堆顶的值用于交换最后的节点，所以**空间复杂度**为O(1)。

2. 采用堆的方式寻找最大值是降低时间复杂度的关键，假设有n个数据，需要n-1次建堆的过程，每次建堆的时间复杂度为log2n，但是无论序列的开始状态如何，都需要对堆进行遍历寻找最大值，所以在最好情况、最坏情况和平均情况下的**时间复杂度**都是O(nlog2n)。

3. 稳定性

堆排序是利用堆的特点，堆的结构是节点i的孩子为2\*i和2\*i+1节点，大顶堆要求父节点大于等于其2个子节点，小顶堆要求父节点小于等于其2个子节点。在一个长为n 的序列，堆排序的过程是从第n/2开始和其子节点共3个值选择最大(大顶堆)或者最小(小顶堆)，这3个元素之间的选择当然不会破坏稳定性。但当为n /2-1, n/2-2, …1这些个父节点选择元素时，就会破坏稳定性。有可能第n/2个父节点交换把后面一个元素交换过去了，而第n/2-1个父节点把后面一个相同的元素没有交换，那么这2个相同的元素之间的稳定性就被破坏了。所以，堆排序不是一个稳定的排序算法。

8 归并排序

## 8.1 基本思想

假设初始对象序列有 n 个对象，首先把它看成是 n 个长度为 1 的有序子序列 (归并项 )，先做两两归并，得到 ⎡n / 2 ⎤ 个长度为 2 的归并项 ( 如 果 n 为奇数为奇数，则最后一个有序子序列的长度为 1)；再做两两归并， …，如此重复，最后得到 一个长度为 n 的有序序列。

## 8.2 算法实现

### 8.2.1 归并排序核心代码

/\*归并排序\*/

void MergeList(vector<int> &sequence, int first, int mid, int last,

int tempList[], Time\_Swap &record)

{

int i = first, j = mid + 1;

int m = mid, n = last;

int k = 0;

//将两个指针分别指向两个序列的起始位置

while (i <= m && j <= n)

{

if (sequence[i] <= sequence[j])

{

tempList[k++] = sequence[i++];

record.growExchange(1);

}

else

{

tempList[k++] = sequence[j++];

record.growExchange(1);

}

}

//将第一条链的剩余部分链到后面

while (i <= m)

{

tempList[k++] = sequence[i++];

record.growExchange(1);

}

//将第二条链的剩余部分链到后面

while (j <= n)

{

tempList[k++] = sequence[j++];

record.growExchange(1);

}

for (i = 0; i < k; i++)

{

sequence[first + i] = tempList[i];

record.growExchange(1);

}

}

void Mergesort(vector<int> &sequence, int first, int last,

int tempList[], Time\_Swap &record)

{

if (first < last)

{

int mid = (first + last) / 2;

Mergesort(sequence, first, mid, tempList, record); //对左半部分递归调用归并排序

Mergesort(sequence, mid + 1, last, tempList, record); //对右半部分递归调用归并排序

MergeList(sequence, first, mid, last, tempList, record); //最后将两部分合并

}

}

void MergeSort(vector<int> sequence)

{

Time\_Swap record("归并排序");

int \*tempList = new int[sequence.size()];

record.Start();

//////////////////////////////////////////////////////////////////////////

Mergesort(sequence, 0, sequence.size() - 1, tempList, record);

//////////////////////////////////////////////////////////////////////////

record.End();

delete[]tempList;

record.showTime();

record.showExchange();

}

### 8.2.2 归并排序截屏示例

### 8.2.3 归并排序算法分析

1.  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:O(n)。

2. 时间复杂度

归并排序的时间主要花在了划分序列和合并序列上，由于是采用递归的方式进行合并，所以与快速排序类似，树的每层元元素的个数最多是n，也就代表着每层最多进行n次比较，而递归树最多只有log2n层，而且不管元素在什么情况下都要做这些步骤，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，都是O( nlogn )。

3. 稳定性

归并排序的原理是把序列递归地分成短序列，递归出口是短序列只有1个元素(认为直接有序)或者2个序列(1次比较和交换)，然后把各个有序的段序列合并成一个有序的长序列，不断合并直到原序列全部排好序。可以发现，在1个或2个元素时，1个元素不会交换，2个元素如果大小相等也不会交换，所以不会破坏稳定性。而在短的有序序列合并的过程中，稳定性也没有受到破坏，合并过程中可以保证如果两个当前元素相等时，把处在前面的序列的元素保存在结果序列的前面，这样就保证了稳定性。所以，归并排序是稳定的排序算法。

9 基数排序

## 9.1 基本思想

基数排序市调性的LSD排序方法，利用“分配”和“收集”两种运算对单关键码进行排序。在这种方法中，把但关键码看成是一个d元组，其中每个分量也可看作是一个关键码。

分量k有radix中取值，称radix为基数。针对d元组中的每一个分量，把对象序列中的所有对象，按k的取值，先“分配”到rd个队列中去然后再按个队列的顺序，依次把对象从队列中“收集”起来，这样所有对象按取值k排序完成。

如果对于所有对象的管家吗K0。。。Kn-1，依次对各位的分量，分别利用这种“分配”，“收集”的运算逐趟进行排序，在最后一趟“分配”，“收集”完成后，所有对象就按其管家吗的值从小到大排好序了。

## 9.2 算法实现

### 9.2.1 基数排序核心代码

/\*基数排序\*/

void distribute(vector<int> &sequence,vector<queue<int> > &alphabet,

int now, Time\_Swap &record)

//按照对应的位, 加入对应的队列中

{

int temp;

for (int i = 0; i < sequence.size(); ++i)

{

switch (now)

{

case 1:

//第一位(个位)

{

alphabet[sequence[i] % 10].push(sequence[i]);

record.growExchange(1);

break;

}

case 2:

//第二位(十位)

{

temp = sequence[i];

alphabet[(temp / 10) % 10].push(sequence[i]);

record.growExchange(1);

break;

}

case 3:

//第三位(百位)

{

temp = sequence[i];

alphabet[temp / 100].push(sequence[i]);

record.growExchange(1);

break;

}

default:cerr << "now error!" << endl; exit(-1);

break;

}

}

}

void collect(vector<int> &sequence, vector<queue<int> > &alphabet, Time\_Swap &record)

{

int cnt = 0;

for (int i = 0; i < RADIX; ++i)

//收集基数多个队列的信息

{

while (!alphabet[i].empty())

//当当前队列不空时, 反复将队列元素加入到序列中

{

sequence[cnt] = alphabet[i].front();

record.growExchange(1);

++cnt;

alphabet[i].pop();

}

}

}

void RadixSort(vector<int> sequence)

{

Time\_Swap record("基数排序");

vector<queue<int> > alphabet(RADIX);

record.Start();

//////////////////////////////////////////////////////////////////////////

for (int i = 1; i<= 3; ++i)

{

distribute(sequence,alphabet, i, record); //分发

collect(sequence, alphabet, record); //收集

}

//////////////////////////////////////////////////////////////////////////

record.End();

record.showTime();

record.showExchange();

}

### 9.2.2 基数排序截屏示例

### 9.2.3 基数排序算法分析

1. 若每个关键码有d位，需要重复执行d趟“分配”与“收集”。每趟对n个对象进行“分配”，对radix个队列进行“收集”。总时间复杂度为O(d(n+radix))。

2. 基数排序是稳定的排序算法