排序的内容

1 排序概述
2 简单排序方法
3 经典排序方法
4 特殊排序方法

1

排序Sorting

每个记录内都有一个关键码域,这个域的值正是比较器所用到的.
- 线性顺序: 比较.

给定一组记录$r_1, r_2, \cdots, r_n$, 其关键码分别为$k_1, k_2, \cdots, k_n$, 排序问题就是要将这些记录排成顺序为$r_{s1}, r_{s2}, \cdots, r_{sn}$的一个序列,满足条件$k_{s1} \le k_{s2} \le \cdots \le k_{sn}$.

2

排序方法分类

- 按记录的存放位置分类有
  - 内排序：待排记录放在内存
  - 外排序：待排记录放在外存（数量大）
- 性能分类
  - 简单但相对较慢的排序算法，$O(n^2)$
  - 先进的排序算法，$O(n \log n)$
  - 一些特殊的方法，$O(n)$
- 待排序的记录序列通常有下列三种存贮方法
  - 顺序表
  - 静态链表：在排序过程，只需修改指针，不需要移动记录
  - 待排记录本身放在连续单元中，同时另建一索引表用于存放各记录存贮位置；排序时不移动记录本身，而移动索引表中的记录“地址”，在排序结束后再按地址调整记录的存贮位置

3

排序方法分类—排序策略

- 排序理论
- 交换类排序：冒泡排序、快速排序
- 选择类排序：选择排序、堆排序
- 插入类排序：插入排序、希尔排序
- 归并类排序：归并排序
- 分配式排序：分配排序、基数排序
- 并行排序
- 混合排序
- 其他排序：图拓扑排序

4

排序算法性能分析

- 分析排序算法时,需测量的代价:
  - 关键码比较的次数
  - 记录交换的次数

5

排序的稳定性

- 在待排记录序列中，任何两个关键字相同的记录，用某种排序方法排序后相对位置不变，则称这种排序方法是稳定的，否则称为不稳定的

例 设 49,38,65,97,76,13,27,49 是待排序列

排序后:13,27,38,49,49,65,76,97 —— 稳定

排序后:13,27,38,49,49,65,76,97——不稳定

6

约定

- 本章中排序算法的输入都是存储在数组中的一组记录
- 将按关键字递增的顺序排序
- 除非特别说明，本章中排序算法都适用于处理具有重复关键码的问题
- 为简洁起见，对待排记录只写出其关键字序列
- 关键字都采用了整数，或使用比较器
- 假设定义了一个swap函数，用于交换数组中两个位置的元素

7

排序的内容

1 **排序概述**
2 **简单排序方法**
3 **经典排序方法**
4 **特殊排序方法**

8

冒泡排序bubble sort

输入：

序列 $R_1, R_2, \cdots, R_n$ (凌乱无序)

冒泡排序基本思想：

比较并交换相邻元素对，直到所有元素都被放到正确的地方为止

9

冒泡排序过程

冒泡排序过程：

输入一个记录数组A，存放着n条记录

每次都从数组的底部（存放最后一个记录）开始，

将第n-1个位置的值与第n-2个位置的值比较，若为逆序，则交换两个位置的值，

然后，比较第n-2个位置和第n-3个位置的值，

依次处理，

直至第2个位置和第1个位置值的比较（交换）

重复这个过程n-1次，整个数组中的元素将按关键码非递减有序排列。

10

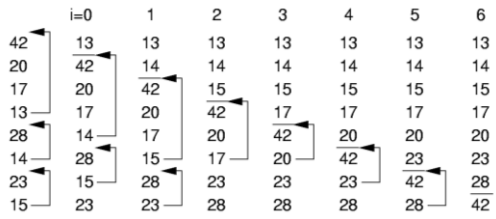
冒泡排序算法伪代码描述

```
template <class Elem, class Comp>
void bubsort(Elem A[], int n) {
  for (int i=0; i<n-1; i++)
    for (int j=n-1; j>0; j--)
      if (Comp::lt(A[j], A[j-1]))
        swap(A, j, j-1);
}
```

```
template <class Elem, class Comp>
void bubsort(Elem A[], int n) {
  for (int i=0; i<n-1; i++)
    for (int j=n-1; j>i; j--)
      if (Comp::lt(A[j], A[j-1]))
        swap(A, j, j-1);
}
```

11

冒泡排序示例



12

冒泡排序算法分析

不考虑数组中结点的组合情况，内层for循环比较的次数总会是i，因此时间代价为$\Theta(n^2)$

冒泡排序的最佳、平均和最差情况的运行时间几乎是相同的

假定交换的概率是平均情况下比较次数的一半，代价为$\Theta(n^2)$

13

冒泡排序特点

- 没有什么特殊的价值

一种相对较慢的排序、没有插入排序易懂，而且没有较好的最佳情况执行时间

- 冒泡排序为后面将要讨论的一种更好的排序提供了基础

14

插入排序基本思想

输入：

序列 $R_1, R_2, \cdots, R_n$ (凌乱无序)

插入排序基本思想：

逐个处理待排序的记录。每个新记录与前面已排序的子序列进行比较，将它插入到子序列中正确的位置。

15

直接插入排序过程

直接插入排序过程：

输入一个记录数组A，存放着n条记录

先将数组中第1个记录看成是一个有序的子序列，

然后从第2个记录开始，依次逐个进行处理（插入）

将第i个记录X，依次与前面的第i-1个、第i-2个，…，第1个记录进行比较，每次比较时，如果X的值小，则交换，直至遇到一个小于或等于X的关键码，或者记录X已经被交换到数组的第一个位置，本次插入才完成。

继续处理，直至最后一个记录插入完毕，整个数组中的元素将按关键码非递减有序排列。

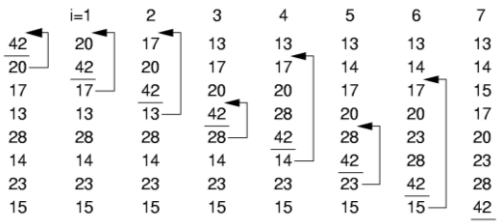
16

直接插入排序算法伪代码描述

```
template <class Elem, class Comp>
void inssort(Elem A[], int n) {
  for (int i=1; i<n; i++)
    for (int j=i; (j>0) &&
       (Comp::lt(A[j], A[j-1])); j--)
      swap(A, j, j-1);
}
```

17

直接插入排序示例



18

直接插入排序算法分析－比较次数

最佳情况

数组中的关键码按从小到大正序排列。

没有记录移动

总的比较次数为n-1次

最佳情况下插入排序的时间代价为$\Theta(n)$

19

直接插入排序算法分析－比较次数

最差情况

数组中的原始数据是逆序的。

每个记录都必须移动到数组的顶端

比较的次数为i

交换的次数为i

最差情况下总的比较次数为 $\sum_{i=2}^{n} i = (2+n)(n-1)/2 = \Theta(n^2)$

20

直接插入排序算法分析－比较次数

平均情况

当处理到第i个记录时，内层for循环的执行次数依赖于该记录“无序”的程度

逆置的数目（即数组中位于一个给定值之前，并比它大的值的数目）将决定比较及交换的次数

平均情况下，在数组的前i-1个记录中有一半关键码比第i个记录的关键码值大

平均的时间代价是最差情况的一半，为$\Sigma i = \Theta(n^2)$

21

直接插入排序算法分析－交换次数

平均情况

每执行一次内层for循环就要比较一次并交换一次

每一轮的最后一次比较找到应该插入的位置，此时没有发生交换

总排序次数是总比较次数减去n-1

最佳情况下为0，在最差及平均情况下为 $\Theta(n^2)$

22

直接插入排序算法的特点

特点

1) 算法简单
2) 时间复杂度为$\Theta(n^2)$，空间复杂度为$\Theta(1)$
3) 初始序列基本（正向）有序时，时间复杂度为$\Theta(n)$
4) 稳定排序

该方法适用于记录基本（正向）有序或n较少的情况

23

选择排序Selection Sort

输入：

序列 $R_1, R_2, \cdots, R_n$ (凌乱无序)

选择排序基本思想：

第i次时“选择”序列中第i小的记录，并把该记录放到序列的第i个位置上。

24

简单选择排序过程

简单选择排序过程：

输入一个记录数组A，存放着n条记录

对于n个记录的数组，共进行n−1趟排序。

每一趟在n−i+1个(i=1, 2, …, n−1)记录中通过n−i次关键字的比较选出关键码最小的记录和第i个记录进行交换

经过n-1趟，整个数组中的元素将按关键码非递减有序排列。

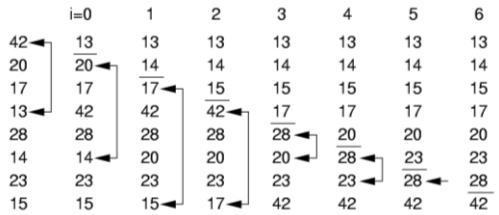
25

简单选择排序算法伪代码描述

```
template <class Elem, class Comp>
void selsort(Elem A[], int n) {
  for (int i=0; i<n-1; i++) {
    int lowindex = i; // Remember its index
    for (int j=n-1; j>i; j--) // Find least
      if (Comp::lt(A[j], A[lowindex]))
        lowindex = j; // Put it in place
    swap(A, i, lowindex);
  }
}
```

26

简单选择排序 举例



27

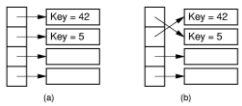
简单选择排序算法分析

比较的次数为$\Theta(n^2)$
但是交换的次数非常少
最佳情况 n-1次交换（重写）
最差情况 n-1次交换
平均情况 $\Theta(n)$

28

简单选择排序的特点

- 实质是延迟交换的冒泡排序
- 不稳定的排序方法
- 对于处理那些作一次交换花费代价（时间）较多的问题，选择排序是很有效的，适用于地址排序



29

简单排序算法的时间代价

	Insertion	Bubble	Selection
Comparisons:			
Best Case	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
Average Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Worst Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Swaps			
Best Case	0	0	$\Theta(n)$
Average Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$
Worst Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$

30

交换排序Exchange Sorting

迄今为止介绍的排序算法依靠交换（比较）相邻的元素.

交换排序的最小时间代价（平均来说）到底是多少呢?

- 每对元素平均需要 n(n-1)/2 交换.

31

知识点回顾

- 排序的概念
- 简单排序方法的思想/特点和实现
  - 冒泡排序
  - 直接插入排序
  - 简单选择排序

32

能力点小结

- **理解各种排序方法的原理和特点**
- **掌握各种排序方法的过程，能够使用**
- **能够实现各种排序方法**

黑色是基本能力
红色是提高能力

33

阅读教材

7.1
7.2

课后作业

任选10个整数的一个集合。

用本周学习的每种排序方法排序，给出每趟排序结果

34

排序的内容

1 **排序概述**
2 **简单排序方法**
3 **经典排序方法**
4 **特殊排序方法**

35

希尔排序Shell sort

动机:

插入排序算法简单，在n值较小时，效率比较高；在n值很大时，若序列按关键码基本有序，效率依然较高，其时间效率可提高到Θ(n)。

基本思想:

先将整个待排记录序列分割成若干个较小的子序列，对子序列分别进行插入排序，然后把有序子序列组合起来；待整个序列中的记录“基本有序”时，再对全体记录进行一次插入排序

*Shell, D. L. (1959). "A High-Speed Sorting Procedure" . Communications of the ACM. **2** (7): 30–32.*

36

Shell排序过程

希尔排序过程：

输入一个记录数组A，存放着n条记录，以及一个（递减）增量序列数组

按照递减的次序，对于每一个增量，

从数组的位置1开始，根据增量计算出子序列的最后一个值的位置，然后调用基于增量的插入排序函数；

从数组的位置2开始，根据增量计算出子序列的最后一个值的位置，然后调用基于增量的插入排序函数；

依次类推，计算出当前增量下的所有子序列，并排序

依法处理下一个增量，直至增量为1，执行一次标准的简单插入排序

37

Shell排序过程（续）

基于增量的插入排序过程：

输入一个记录数组A，起始位置i，结束位置n，增量incr

先将数组中第i位置的记录看成是一个有序的子序列，

然后从第i+j*incr位置的记录开始，依次对逐个增量位置进行处理（插入）

将第i+j*incr位置的记录X，依次与前面的第i+(j-1)*incr位置、第i+(j-2)*incr位置，…，第i位置的记录进行比较，每次比较时，如果X的值小，则交换，直至遇到一个小于或等于X的关键码，或者记录X已经被交换到第i位置，本次插入才完成。

继续处理，直至最后一个记录插入完毕，整个子序列有序。

38

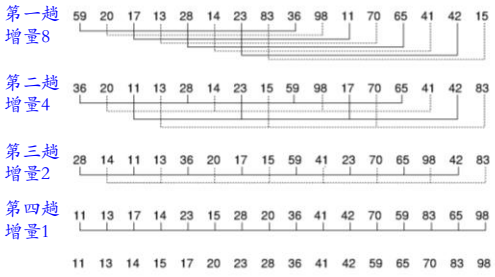
Shell排序算法伪代码描述

```
// Modified version of Insertion Sort
template <class Elem, class Comp>
void inssort2(Elem A[], int n, int incr) {
  for (int i=incr; i<n; i+=incr)
    for (int j=i;
         (j>=incr) &&
         (Comp::lt(A[j], A[j-incr])); j-=incr)
      swap(A, j, j-incr);
}

template <class Elem, class Comp>
void shellsort(Elem A[], int n) { // Shellsort
  for (int i=n/2; i>2; i/=2)  // For each incr
    for (int j=0; j<i; j++)   // Sort sublists
      inssort2<Elem,Comp>(&A[j], n-j, i);
  inssort2<Elem,Comp>(A, n, 1);
}
```

39

Shell排序算法举例



40

Shell排序的分析

- 性能分析

分析Shell排序是很困难的，因此我们必须不加证明地承认Shell排序的平均运行时间是$\Theta(n^{1.5})$(对于选择“增量每次除以3”递减)。选取其他增量序列可以减少这个上界。因此，Shell排序确实比插入排序或任何一种运行时间为$\Theta(n^2)$的排序算法要快

有人在大量实验的基础上推出，当n在某个特定范围内，所需比较和移动次数约为$n^{1.3}$

当$n \to \infty$时，可减少到$n(\log_2 n)^2$

- 增量序列可以有各种取法，但需注意：应使增量序列中的值没有除1之外的公因子，并且最后一个增量值必须等于1

41

Shell排序的特点

特点：

1）缩小增量排序。时间复杂度，取决于增量序列的选择，选择的好，效率优于插入排序

2）不稳定排序方法

3）适用于 n 较大情况（中等大小规模）

4） Shell排序说明有时可以利用一个算法的特殊性能

42

快速排序Quick sort

动机：

分治思想：划分交换排序

基本思想：

在待排序记录中选取一个记录R（称为轴值pivot），通过一趟排序将其余待排记录分割（划分）成独立的两部分，比R小的记录放在R之前，比R大的记录放在R之后，然后分别对R前后两部分记录继续进行同样的划分交换排序，直至待排序序列长度等于1，这时整个序列有序。

Hoare, C. A. R. (1961). "Algorithm 64: Quicksort". Comm. ACM. 4 (7): 321.

43

快速排序算法过程

快速排序过程：快速排序（划分过程）用递归实现。

若当前（未排序）序列的长度不大于1

返回当前序列

否则

在待排序记录中选取一个记录做为轴值，通过**划分算法**将其余待排记录划分成两部分，比R小的记录放在R之前，比R大的记录放在R之后；

分别用快速排序对前后两个子序列进行排序（注意轴值已经在最终排序好的数组中的位置。无须继续处理）

44

快速排序算法过程（续）

选取轴值，划分序列的过程：

记录数组A，待排子序列左、右两端的下标i和j

选取待排序子序列中间位置的记录为轴值

交换轴值和位置j的值

依据在位置j的轴值，将数组i-1到j之间的待排序记录划分为两个部分（i到k-1之间的记录比轴值小，k到j-1之间的记录比轴值大）

从数组i-1到j之间的待排序序列两端向中间移动下标，必要时交换记录，直到两端的下标相遇为止（相遇的位置记为k）

交换轴值和位置k的值

45

快速排序算法伪代码描述

```
template <class Elem, class Comp>
void qsort(Elem A[], int i, int j) {
  if (j <= i) return; // List too small
  int pivotindex = findpivot(A, i, j);
  swap(A, pivotindex, j);  // Put pivot at end
  // k will be first position on right side
  int k =
     partition<Elem,Comp>(A, i-1, j, A[j]);
  swap(A, k, j);          // Put pivot in place
  qsort<Elem,Comp>(A, i, k-1);
  qsort<Elem,Comp>(A, k+1, j);
}

template <class Elem>
int findpivot(Elem A[], int i, int j)
  { return (i+j)/2; }
```

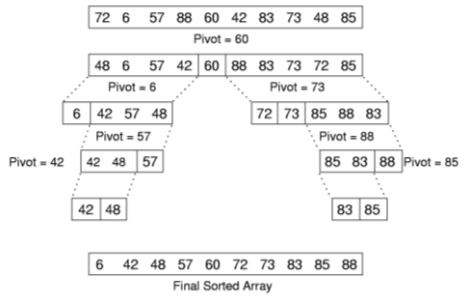
46

快速排序划分函数伪代码描述

```
template <typename E, typename Comp>
inline int partition(E A[], int l, int r, E&
  pivot) {
  do {//Move the bounds inward until they meet
    // Move l right and
    while (Comp::prior(A[++l], pivot));
    while ((l < r) &&
         Comp::prior(pivot, A[--r])); // r left
    swap(A, l, r); //Swap out-of-place values
  } while (l < r); // Stop when they cross
  return l;//Return first position in right part
}
```

47

快速排序算法举例



48

快速排序算法性能分析

时间开销：

找轴值：常数时间

划分：Θ(s)，s是数列的长度

最差情况: 糟糕的分割. $\Theta(n^2)$

最佳情况: 每次将数列分割为等长的两部分 Θ(nlogn)

平均情况:

$$\mathbf{T}(n) = cn + \frac{1}{n}\sum_{k=0}^{n-1}[\mathbf{T}(k) + \mathbf{T}(n-1-k)], \quad \mathbf{T}(0) = \mathbf{T}(1) = c.$$

Θ(nlogn)

空间开销：

辅助栈空间，O(logn)

49

快速排序的特点

- 冒泡排序的改进，划分交换排序
- 轴值的取值影响性能
- 时间复杂度为Θ（nlogn）
- 不稳定的排序方法
- 实际，快速排序是最好的内排序方法

快速排序的优化:

- 更好的轴值
- 对于小的子数列采用更好的排序算法
- 用栈来模拟递归调用

50

归并排序Merge sort

动机：

分治思想

基本思想：

将两个或多个有序表归并成一个有序表

John von Neumann (1945)

51

2路归并排序基本思想

2 路归并排序

1） 设有n个待排记录，初始时将它们分为n个长度为 1的有序子表；

2） 两两归并相邻有序子表，得到若干个长度2为的有序子表；

3） 重复2）直至得到一个长度为n的有序表

52

2路归并排序算法过程

归并排序过程：归并排序（划分过程）用递归实现。

若当前（未排序）序列的长度不大于1

返回当前序列

否则

将当前未排序序列分割为大小相等的两个子序列

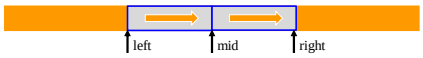
分别用归并排序对两个子序列进行排序

将返回的两个有序子序列合并成一个有序序列

```
List mergesort(List inlist) {
  if (inlist.length() <= 1)return inlist;
  List l1 = half of the items from inlist;
  List l2 = other half of items from inlist;
  return merge(mergesort(l1),
               mergesort(l2));
}
```

53

2路归并排序算法过程（续）



合并两个有序子序列的过程：

记录数组A，起始位置left，结束位置right，中间点mid

首先将两个子序列复制到辅助数组中，

首先对辅助数组中两个子序列的第一条记录进行比较，并把较小的记录作为合并数组中的第一个记录，复制到原数组的第一个位置上

继续使用这种方法，不断比较两个序列中未被处理的记录，并把结果较小的记录依次放到合并数组中，直到两个序列的全部记录处理完毕

注意要检查两个子序列中的一个被处理完，另一个未处理完的情况。只需依次复制未处理完的记录即可。

54

2路归并排序的伪代码描述

```
template <class Elem, class Comp>
void mergesort(Elem A[], Elem temp[],
               int left, int right) {
  int mid = (left+right)/2;
  if (left == right) return;
  mergesort<Elem,Comp>(A, temp, left, mid);
  mergesort<Elem,Comp>(A, temp, mid+1, right);
  for (int i=left; i<=right; i++) // Copy
    temp[i] = A[i];
  int i1 = left; int i2 = mid + 1;
  for (int curr=left; curr<=right; curr++) {
    if (i1 == mid+1)      // Left exhausted
      A[curr] = temp[i2++];
    else if (i2 > right)  // Right exhausted
      A[curr] = temp[i1++];
    else if (Comp::lt(temp[i1], temp[i2]))
      A[curr] = temp[i1++];
    else A[curr] = temp[i2++];
```

55

归并排序性能分析

在最佳、平均、最差情况下，时间复杂度为 $\Theta(n \log n)$

归并排序需要两倍的空间代价.

56

堆排序heap sort

动机：

选择排序：树形选择排序，最值堆

基本思想：

首先将数组转化为一个满足堆定义的序列。

然后将堆顶的最值取出，再将剩下的数排成堆，再取堆顶数值，…，如此下去，直到堆为空，就可得到一个有序序列

Williams, J. W. J. (1964), "Algorithm 232 - Heapsort", Communications of the ACM, 7 (6): 347 – 348

Floyd, Robert W. (1964), "Algorithm 245 - Treesort 3", Communications of the ACM, 7 (12): 701

57

堆排序算法过程

堆排序过程:

建堆：将输入序列用数组存储，利用堆的构建函数将数组转化为一个满足堆定义的序列（如果是递增排序，则构建最大值堆，反之，构建最小值堆）

然后将堆顶的最大元素取出，再将剩下的数排成堆，再取堆顶数值，…，如此下去，直到堆为空。

每次应将堆顶的最大元素取出放到数组的最后。

假设n个元素存于数组中的0到n-1位置上。把堆顶元素取出时，应该将它置于数组的第n-1个位置。这时堆中元素的数目为n-1个。再按照堆的定义重新排列堆，取出最大值并放入数组的第n-2个位置。到最后结束时，就排出了一个由小到大排列的数组。

58

堆排序算法伪代码描述

```
template <class Elem, class Comp>
void heapsort(Elem A[], int n) { // Heapsort
  Elem mval;
  maxheap<Elem,Comp> H(A, n, n);
  for (int i=0; i<n; i++)     // Now sort
    H.removemax(mval);  // Put max at end
}
```

59

堆排序的性能分析

堆排序的时间代价:

- 建堆：$\Theta(n)$
- n次取堆的最大元素：$\Theta(\log n)$

堆排序的总时间代价：$\Theta(2\ n \log n)$

堆排序的空间代价:

常数辅助空间： $\Theta(1)$

60

堆排序的特点

- 基于堆数据结构，具有许多优点：

整棵树是平衡的，而且它的数组实现方式对空间的利用率也很高，可以利用有效的建堆函数一次性把所有值装入数组中。

堆排序的最佳、平均、最差执行时间均为 $\Theta(n \log n)$，空间开销也是常数

堆排序是不稳定的排序算法

更适合于外排序，处理那些数据集太大而不适合在内存中排序的情况

61

知识点回顾

- 经典排序方法的思想/特点和实现
  - 希尔排序
  - 快速排序
  - 归并排序
  - 堆排序

62

能力点小结

- **理解各种排序方法的原理和特点**
- **掌握各种排序方法的过程，能够使用**
- **能够实现各种排序方法**

黑色是基本能力
红色是提高能力

63

阅读教材

7.3
7.4
7.5
7.6

实验

准备实验9 排序算法实验比较

写一个实验分析报告

64

排序的内容

1 **排序概述**
2 **简单排序方法**
3 **经典排序方法**
4 **特殊排序方法**

65

分配排序distribution sort

动机：

分配：按关键码划分；不进行关键码比较

基本思想：

关键码用来确定一个记录在排序中的最终位置

66

分配排序的伪代码描述

```
for (i=0; i<n; i++)
  B[A[i]] = A[i];


template <class E, class getkey>
void binsort(E A[], int n) {
  List<E> B[MaxKeyValue];
  E item;
  for (i=0; i<n; i++)
     B[A[i]].append(getkey::(A[i]));
  for (i=0; i<MaxKeyValue; i++)
    for (B[i].setStart();
         B[i].getValue(item); B[i].next())
      output(item);
}
```

67

桶式排序（bucket sort）

分治法：

- 将序列中的元素分配到一组（数量有限的）桶中
- 每个桶再分别排序（可以使用其它排序方法或递归使用桶式排序）
- 最后，依序遍历每个桶，将所有元素有序放回序列

68

分配排序的性能分析

时间代价：

- 分配排序的时间代价初看起来是$\Theta(n)$
- 实际上时间代价为$\Theta(n + \text{MaxKeyValue})$

如果MaxKeyValue很大，那么这个算法的时间代价可能会为$\Theta(n^2)$ 或更差

空间代价：

$O(n + \text{MaxKeyValue})$

69

分配排序的特点

- 不进行关键码的比较；适用于排序记录的关键码取值有限或分布有特点的情况
- 可用于分布式排序算法
- 适用于外排序

70

基数排序Radix sort

动机：

分配排序：按关键码划分；不进行关键码比较；而是一种多关键字的排序

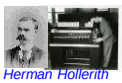
基本思想：

将关键码看成有若干个关键字复合而成。

然后对每个关键字进行分配（计数）排序

依次重复

最终得到一个有序序列


Herman Hollerith


Harold H. Seward. "Information sorting in the application of electronic digital computers to business operations," Tech. rep., MIT Digital Computer Laboratory, Report R-232, Cambridge, Mass, 1954

71

基数排序的算法过程

将所有待排序数值（正整数）按照基数 r 统一为同样的数位长度，数位较短的数前面补零。然后，从最低位开始，依次进行每趟（计数分配）排序

定义一个长度为r的辅助数组cnt。记录每个盒子里有多少个元素。初始值为0。

定义一个和原数组A一样大小的数组B。

依次处理每个元素，根据元素的值计算其盒子编号，统计出每个盒子需要存放的记录数。（cnt[j]存储了数位j（第j个盒子）在这一趟排序时分配的记录数）

利用cnt的值，计算该盒子在数组B中的（最后一个）下标位置

从后向前，依次把数组A中的元素，依据该元素在cnt中记录的下标，把元素值存入（分配）数组B的（盒子中）相应位置

将数组B的值依次复制到数组A，进行下一趟排序。

72

基数排序的伪代码描述

```
template <class Elem, class Comp>
void radix(Elem A[], Elem B[],
           int n, int k, int r, int cnt[]) {
  // cnt[i] stores # of records in bin[i]
  int j;
  for (int i=0, rtok=1; i<k; i++, rtok*=r) {
    for (j=0; j<r; j++) cnt[j] = 0;
    // Count # of records for each bin
    for(j=0; j<n; j++) cnt[(A[j]/rtok)%r]++;
    // cnt[j] will be last slot of bin j.
    for (j=1; j<r; j++)
      cnt[j] = cnt[j-1] + cnt[j];
    for (j=n-1; j>=0; j--)\
      B[--cnt[(A[j]/rtok)%r]] = A[j];
    for (j=0; j<n; j++) A[j] = B[j];
  }}
```

73

基数排序的性能分析

时间开销:

$\Theta(nk + rk)$ 其中n是记录数，k是趟数，和r是基数

n, k, 和 r 之间有关系吗?

如果关键码很小，那么基数排序的时间代价为 $\Theta(n)$.

如果有 n 个不同的关键码, 那么关键码的长度最小要大于$\log n$.

- 因而, 在一般情况下基数排序的时间代价为$\Omega(n \log n)$

空间开销:

需要辅助数组，$O(n)$

74

基数排序的特点

- 记录数目大于关键码长度，基数排序效率高
- 基数排序对一些数据类型比较难于实现
- 基数排序中对关键码是计算而非比较，在计算机中，比较操作比计算操作要快
- 稳定排序

75

知识点回顾

- 特殊排序方法的思想/特点和实现
  - 分配排序
  - 基数排序

76

能力点小结

- **理解各种排序方法的原理和特点**
- **掌握各种排序方法的过程，能够使用**
- **能够实现各种排序方法**

黑色是基本能力
红色是提高能力

77

77

阅读教材

7.7

78

各种排序算法的实验比较

Sort	10	100	1K	10K	100K	1M	Up	Down
Insertion	.00023	.007	0.66	64.98	7381.0	674420	0.04	129.05
Bubble	.00035	.020	2.25	277.94	27691.0	2820680	70.64	108.69
Selection	.00039	.012	0.69	72.47	7356.0	780000	69.76	69.58
Shell	.00034	.008	0.14	1.99	30.2	554	0.44	0.79
Shell/O	.00034	.008	0.12	1.91	29.0	530	0.36	0.64
Merge	.00050	.010	0.12	1.61	19.3	219	0.83	0.79
Merge/O	.00024	.007	0.10	1.31	17.2	197	0.47	0.66
Quick	.00048	.008	0.11	1.37	15.7	162	0.37	0.40
Quick/O	.00031	.006	0.09	1.14	13.6	143	0.32	0.36
Heap	.00050	.011	0.16	2.08	26.7	391	1.57	1.56
Heap/O	.00033	.007	0.11	1.61	20.8	334	1.01	1.04
Radix/4	.00838	.081	0.79	7.99	79.9	808	7.97	7.97
Radix/8	.00799	.044	0.40	3.99	40.0	404	4.00	3.99

3.4GHz英特尔奔腾4微处理器上运行Linux操作系统。
Shell排序、快速排序、归并排序和堆排序都给出了普通及优化算法。
基数排序给出了每轮处理4位和8位时的不同算法

79

排序问题的下限

排序算法的时间代价（在平均、最差情况）为$O(n \log n)$, 因为已经知道了有这个上限的算法.

想知道对所有可能的排序算法的时间代价的下限.

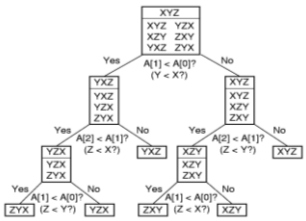
排序算法的I/O 时间下限为$\Omega(n)$.

排序问题（算法）的下限?

80

排序问题的下限

在最差情况下，任何一种基于比较的排序算法都需要$\Omega(n \log n)$时间代价



判定树（decision tree）

- n个数有$n!$ 个排列.
- 每个排序可以看成一个判定，排列是输入.
- 判定树的每个叶子对应一个排列.

在判定树中哪个结点对应最差情况呢?

- 有n个结点的树最少有$\Omega(\log n)$层, 所以有n!个叶结点的树的最小层数为 $\Omega(\log n!) = \Omega(n \log n)$.

81

排序问题下限研究的意义

没有任何一种基于关键码比较的排序算法可以把最差执行时间降低到Ω(*nlogn*)以下

广泛应用的排序算法已令人满意，没有必要死死追寻比nlogn还快的算法（基于关键码比较思路）

可以作为证明算法问题下限的模板

可以用排序问题的结果，利用规约方法可以推导其他问题的上限、下限

82

各种排序算法性能比较

	最佳	平均	最差	空间	稳定性	排序方法
直接插入排序	n	n^2	n^2	1	稳定	插入
冒泡排序	n^2	n^2	n^2	1	稳定	交换
简单选择排序	n^2	n^2	n^2	1	不稳定	选择
希尔排序	$nlogn$	$nlog^2n$	$nlog^2n$	1	不稳定	插入
快速排序	$nlogn$	$nlogn$	n^2	$logn$~n	不稳定	交换
归并排序	$nlogn$	$nlogn$	$nlogn$	n	稳定	归并
堆排序	$nlogn$	$nlogn$	$nlogn$	1	不稳定	选择
桶式排序	$n+r$	$n+r$	$n+r$	$n+r$	稳定	非比较
基数排序	$nk+rk$	$nk+rk$	$nk+rk$	$n+2^k$	稳定	非比较

83

关于排序方法的几个结论

1. 简单排序以直接插入排序最简单,当序列“基本有序”或n较小时,它是最佳排序方法,通常用它与“先进的排序方法”结合使用.如果记录空间大，宜用移动次数较少的简单选择排序.
2. 平均时间性能快速排序最佳,但最坏情况下的时间性能不如堆排序和归并排序.
3. 基数排序最适合n很大而关键字较小的序列
4. 从稳定性看,归并排序,基数排序,插入排序和冒泡排序是稳定的；而选择排序,快速排序,堆排序和希尔排序是不稳定的.
5. 简单排序，堆排序和快速排序对内存占用小；而归并排序和各种分配排序需要辅助空间.

84

能力点小结

- 理解各种排序方法的原理和特点
- 掌握各种排序方法的过程，能够使用
- 能够实现各种排序方法

黑色是基本能力
红色是提高能力

85

85