



### 数据结构与算法(八)

张铭 主讲

采用教材:张铭,王腾蛟,赵海燕编写 高等教育出版社,2008.6 ("十一五"国家级规划教材)

http://www.jpk.pku.edu.cn/pkujpk/course/sjjg



### 大纲

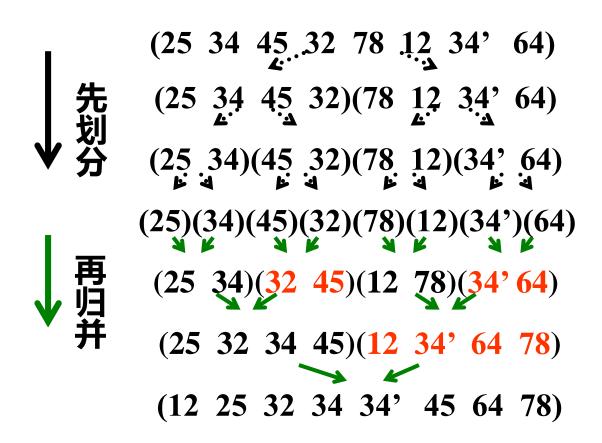
- · 8.1 排序问题的基本概念
- · 8.2 插入排序 (Shell 排序)
- · 8.3 选择排序(堆排序)
- · 8.4 交换排序
  - 8.4.1 冒泡排序
  - 8.4.2 快速排序
- · 8.5 归并排序
- · 8.6 分配排序和索引排序
- · 8.7 排序算法的时间代价
- · 内排序知识点总结





### 归并排序思想

- · 划分为两个子序列
- · 分别对每个子序列归并排序
- · 有序子序列合并





### 1 V V

#### 8.5 归并排序

### 两路归并排序

```
template <class Record>
void MergeSort(Record Array[], Record TempArray[], int
left, int right) {
  // Array为待排序数组, left, right两端
  int middle:
  if (left < right) { // 序列中只有0或1个记录,不用排序
     middle = (left + right) / 2; // 平分为两个子序列
     // 对左边一半进行递归
     MergeSort(Array, TempArray, left, middle);
     // 对右边一半进行递归
     MergeSort(Array, TempArray, middle+1, right);
     Merge(Array, TempArray, left, right, middle); // 归并
```





### 归并函数





```
while (index1 <= middle && index2 <= right) {
    // 取较小者插入合并数组中
    if (TempArray[index1] <= TempArray[index2])
        Array[i++] = TempArray[index1++];
    else Array[i++] = TempArray[index2++];
}
while (index1 <= middle) // 只剩左序列 , 可以直接复制
    Array[i++] = TempArray[index1++];
while (index2 <= right) // 与上个循环互斥 , 复制右序列
    Array[i++] = TempArray[index2++];
}
```





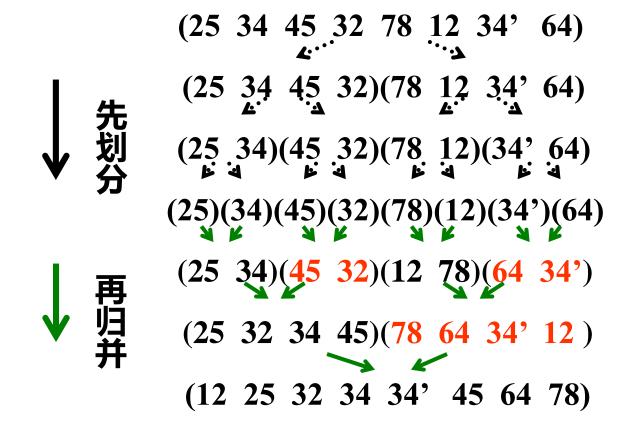
### 归并算法优化

- · 同优化的快速排序一样,对基本已排序序列直接插入排序
- · R.Sedgewick 优化:归并时从两端开始处理,向中间推进,简化边界判断



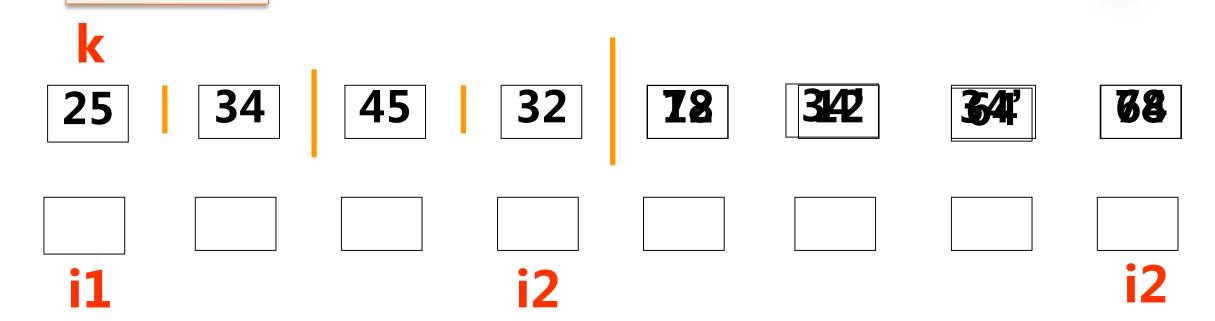


# R.Sedgewick优化归并思想









# R.Sedgewick优化归并





# 优化的归并排序(阈值28)

```
template <class Record>
void ModMergeSort(Record Array[], Record TempArray[], int left, int
right) { // Array为待排序数组, left, right两端
  int middle:
  if (right-left+1 > THRESHOLD) { // 长序列递归
    middle = (left + right) / 2; // 从中间划为两个子序列
     ModMergeSort(Array, TempArray, left, middle); // 左
     ModMergeSort(Array, TempArray, middle+1, right);// 右
     // 对相邻的有序序列进行归并
     ModMerge(Array, TempArray, left, right, middle); // 归并
  else InsertSort(&Array[left],right-left+1); // 小序列插入排序
```





### 优化的归并函数

```
template <class Record> void ModMerge(Record
Array[],Record TempArray[],int left,int right,int middle) {
                                  // 两个子序列的起始位置
  int index1,index2;
  int i,j,k;
  for (i = left; i <= middle; i++)
     TempArray[i] = Array[i]; // 复制左边的子序列
  for (j = 1; j <= right-middle; j++) // 颠倒复制右序列
     TempArray[right-j+1] = Array[j+middle];
  for (index1=left, index2=right, k=left; k<=right; k++)
     if (TempArray[index1] <= TempArray[index2])</pre>
        Array[k] = TempArray[index1++];
     else
        Array[k] = TempArray[index2--];
```





### 算法复杂度分析

- · 空间代价: Θ(n)
- ·划分时间、排序时间、归并时间

$$T(n) = 2T(n/2) + cn$$

- T(1) = 1
- ·归并排序总时间代价也为
  - $-\Theta(n\log n)$
- · 不依赖于原始数组的输入情况,最大、最小以及平均时间代价均为  $\Theta(n\log n)$



#### 8.5 归并排序

### 思考

- · 普通归并和 Sedgewick 算法都是稳定的吗?
- · 两个归并算法哪个更优?
  - 二者的比较次数和赋值次数
  - 归并时子序列下标是否需要边界判断



### 大纲

- · 8.1 排序问题的基本概念
- · 8.2 插入排序 (Shell 排序)
- · 8.3 选择排序(堆排序)
- · 8.4 交换排序
  - 8.4.1 冒泡排序
  - 8.4.2 快速排序
- · 8.5 归并排序
- · 8.6 分配排序和索引排序
- · 8.7 排序算法的时间代价
- · 内排序知识点总结



### 8.6 分配排序和基数排序

- 不需要进行纪录之间两两比较
- 需要事先知道记录序列的一些具体情况





#### 8.6.1 桶式排序

### 8.6.1 桶式排序

- ·事先知道序列中的记录都位于某个小区间段 [0, m) 内
- · 将具有相同值的记录都分配到同一个桶中 , 然后依次按照编号从桶中取出记录, 组 成一个有序序列



8.6.1 桶式排序



待排数组: 7 3 8 9 6 1 8 1 2

每个桶 count:



前若干桶的 累计count:

```
0 2 3 4 4 4 5 6 8 9
```







### 桶排序示意

待排数组: 7 3 8 9 6 1 8 1 2

每个桶count:



前若干桶的 累计count:



收集:





### 8.6.1 桶式排序



### 桶式排序算法

```
template <class Record> void BucketSort(Record Array[], int n, int max) {
  Record *TempArray = new Record[n]; // 临时数组
  int *count = new int[max]; // 桶容量计数器
  int i;
  for (i = 0; i < n; i++)
                            // 把序列复制到临时数组
    TempArray[i] = Array[i];
  for (i = 0; i < max; i++)
                        // 所有计数器初始都为0
    count[i] = 0;
                             // 统计每个取值出现的次数
  for (i = 0; i < n; i++)
    count[Array[i]]++;
  for (i = 1; i < max; i++) // 统计小于等于i的元素个数
    count[i] = count[i-1]+count [i]; // c [i]记录i+1的起址
  for (i = n-1; i >= 0; i--) // 尾部开始,保证稳定性
    Array[--count[TempArray[i]]] = TempArray[i];
```



#### 8.6.1 桶式排序

### 算法分析

- ·数组长度为 n, 所有记录区间 [0, m) 上
- · 时间代价:
  - 统计计数: Θ(n+m) , 输出有序序列时循环 n 次
  - 总的时间代价为  $\Theta(m+n)$
  - 适用于 m 相对于 n 很小的情况
- ・空间代价:
  - -m 个计数器,长度为n的临时数组, $\Theta(m+n)$
- ・稳定



#### 8.6.2 基数排序

### 8.6.2 基数排序

- · 桶式排序只适合 m 很小的情况
- · 基数排序: 当 m 很大时,可以将一个记录的值即排序码拆分为多个部分来进行比较





· 假设长度为 n 的序列

$$R = \{ r_0, r_1, ..., r_{n-1} \}$$
  
记录的排序码 K 包含 d 个子排序码  $K = (k_{d-1}, k_{d-2}, ..., k_1, k_0)$ 

· R 对排序码有序,即对于任意 两个记录  $R_i$  ,  $R_j$  (0  $\leq$  i< j  $\leq$  n-1),都满足 ( $k_{i,d-1}$  ,  $k_{i,d-2}$  , ... ,  $k_{i,1}$  ,  $k_{i,0}$  )  $\leq$  ( $k_{j,d-1}$  ,  $k_{j,d-2}$  , ... ,  $k_{j,1}$  ,  $k_{j,0}$  )



#### 8.6.2 基数排序

### 例子

例如:对0到9999之间的整数进行排序

- ·将四位数看作是由四个排序码决定,即千、百、十、个位,其中千位为最高排序码,个位为最低排序码。基数 r=10。
- ·可以按干、百、十、个位数字依次进行4次桶 式排序
- · 4趟分配排序后,整个序列就排好序了







- · 高位先排, 递归分治
  - 先按花色: **♣8 ♣1 ♦3 ♦7 ♥**J **♥9 ♠**3 **♠**9
  - 再按面值: **♣**1 **♣**8 **♦**3 **♦**7 **♥**9 **♥**J **♠**3 **▲**9
- · 低位先排,要求稳定排序!
  - 先面值: ♣1 ♠3 ♦3 ♦'7 ♣8 ♥9 ♠'9 ♥'J
  - 再花色: ♣1 ♣'8 ♦3 ♦'7 ♥9 ♥'J ♠3 ♠'9

#### 8.6.2 基数排序

# 高位优先法

- MSD, Most Significant Digit first
- 先处理高位  $k_{d-1}$  将序列分到若干桶中
- 然后再对每个桶处理次高位  $k_{d-2}$  , 分成更小的桶
- 依次重复,直到对  $k_0$  排序后,分成最小的桶, 每个桶内含有相同排序码( $k_{d-1}$  , ... ,  $k_1$  ,  $k_0$  )
- 最后将所有的桶中的数据依次连接在一起, 成为一个有序序列
- 这是一个分、分、...、分、收的过程





#### 8.6.2 基数排序

### 低位优先法

- · LSD , Least Significant Digit first
- · 从最低位  $k_0$  开始排序
- ·对于排好的序列再用次低位  $k_1$  排序;
- · 依次重复,直至对最高位  $k_{d-1}$  排好序后,整个序列成为有序的
- ·分、收;分、收;…;分、收的过程
  - 比较简单, 计算机常用



#### 8.6.2 基数排序

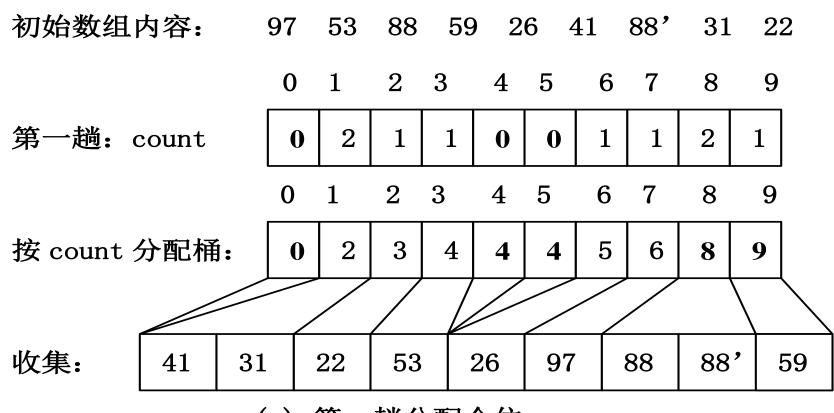
### 基数排序的实现

- 主要讨论 LSD
  - 基于顺序存储
  - 基于链式存储
- 原始输入数组 R 的长度为 n , 基数为 r , 排序 码个数为 d





# 基于顺序存储的基数排序

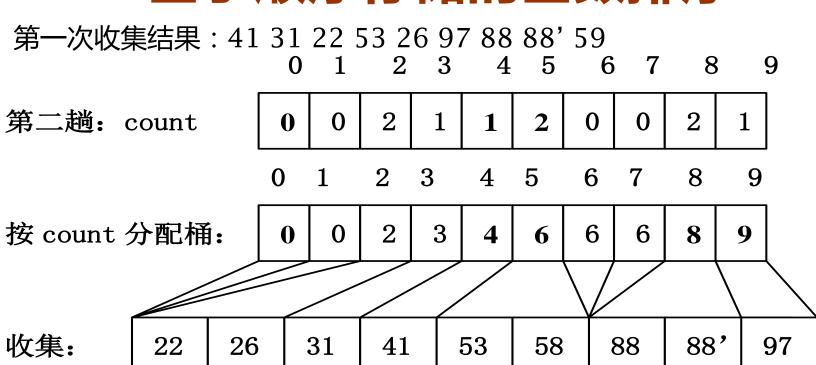








### 基于顺序存储的基数排序



最终排序结果: 22 26 31 41 53 59 88 88' 97

(b) 第二趟分配十位





### 基于数组的基数排序

```
template <class Record>
void RadixSort(Record Array[], int n, int d, int r) {
  Record *TempArray = new Record[n];
  int *count = new int[r]; int i, j, k;
  int Radix = 1; // 模进位,用于取Array[j]的第i位
  for (i = 1; i <= d; i++) { // 对第 i 个排序码分配
    for (i = 0; j < r; j++)
      count[j] = 0; // 初始计数器均为0
    for (j = 0; j < n; j++) { // 统计每桶记录数
      k = (Array[j] / Radix) % r; // 取第i位
      count[k]++; // 相应计数器加1
```

30





```
for (j = 1; j < r; j++) // 给桶划分下标界
  count[j] = count[j-1] + count[j];
for (j = n-1; j >= 0; j--) { // 从数组尾部收集
 k = (Array[j] / Radix ) % r; // 取第 i 位
  count[k]--; // 桶剩余量计数器减1
 TempArray[count[k]] = Array[i]; // 入桶
for (j = 0; j < n; j++) // 内容复制回 Array 中
 Array[j] = TempArray[j];
Radix *= r:
          // 修改模Radix
```





# 顺序基数排序代价分析

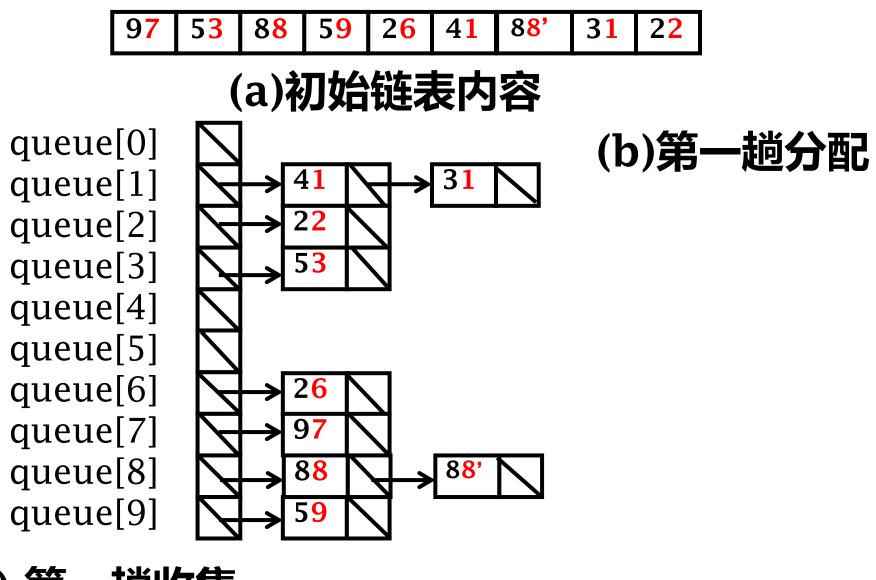
- ·空间代价:
  - 临时数组, n
  - r 个计数器
  - 总空间代价  $\Theta(n+r)$
- ·时间代价
  - 桶式排序:Θ(*n+r*)
  - d 次桶式排序
  - $-\Theta(d\cdot(n+r))$





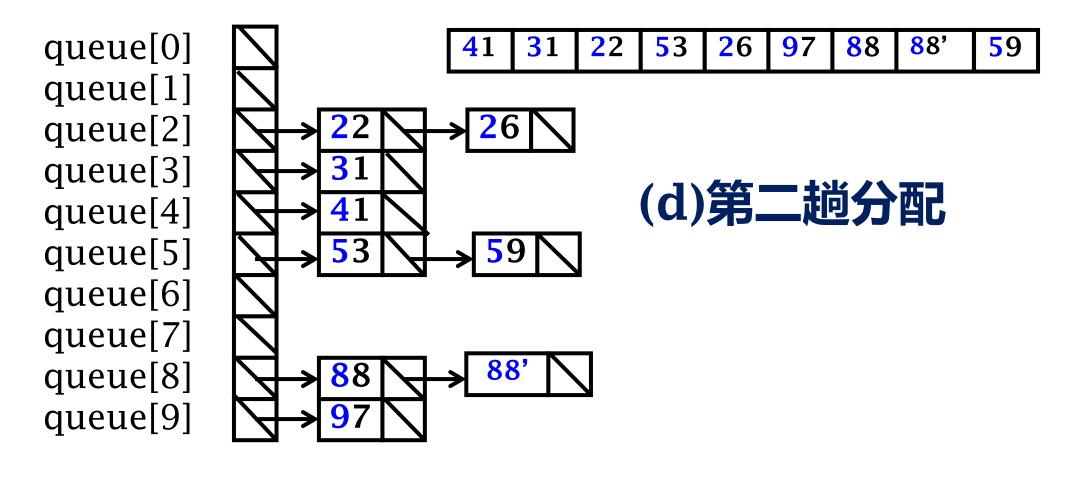
### 基于静态链的基数排序

- · 将分配出来的子序列存放在 r 个 (静 态链组织的) 队列中
- ·链式存储避免了空间浪费情况



### (c) 第一趟收集

41 31 22 53 26 97 88 88' 59



### (e) 第二趟收集结果 (最终结果)

 22
 26
 31
 41
 53
 59
 88
 88'
 97



### 静态队列定义

```
// 结点类
class Node {
public:
  int key;
                 // 结点的关键码值
                 // 下一个结点在数组中的下标
  int next;
};
class StaticQueue { // 静态队列类
public:
  int head;
  int tail;
};
```



### 基于静态链的基数排序

```
template <class Record>
void RadixSort(Record *Array, int n, int d, int r) {
  int i, first = 0; // first指向第一个记录
  StaticQueue *queue = new StaticQueue[r];
  for (i = 0; i < n-1; i++)
     Array[i].next = i + 1; // 初始化静态指针域
  Array[n-1].next = -1;  // 链尾next为空
  // 对第i个排序码进行分配和收集,一共d趟
  for (i = 0; i < d; i++) {
     Distribute(Array, first, i, r, queue);
     Collect(Array, first, r, queue);
  delete[] queue;
  AddrSort(Array, n, first); // 整理后,按下标有序
```



```
template <class Record>
void Distribute(Record *Array, int first, int i, int r,
StaticQueue *queue) {
  int j, k, a, curr = first;
  for (j = 0; j < r; j++) queue[j].head = -1;
  while (curr != -1) { // 对整个静态链进行分配
     k = Array[curr].key;
     for (a = 0; a < i; a++) // 取第i位排序码数字k
       k = k / r;
     k = k \% r:
     if (queue[k].head == -1) // 把数据分配到第k个桶中
        queue[k].head = curr;
     else Array[queue[k].tail].next = curr;
     queue[k].tail = curr;
     curr = Array[curr].next; // curr移动,继续分配
```



```
8 9 9
```

```
template <class Record>
void Collect(Record *Array, int& first, int r, StaticQueue
*queue) {
  int last, k=0; // 已收集到的最后一个记录
  while (queue[k].head == -1) k++; // 找到第一个非空队
  first = queue[k].head; last = queue[k].tail;
  while (k < r-1) { // 继续收集下一个非空队列
    k++;
    while (k < r-1 \&\& queue[k].head == -1)
       k++;
    if (queue[k].head != -1) { // 试探下一个队列
       Array[last].next = queue[k].head;
       last = queue[k].tail; // 最后一个为序列的尾部
  Array[last].next = -1; // 收集完毕
```





### 链式基数排序算法代价分析

- ·空间代价
  - n 个记录空间
  - r 个子序列的头尾指针
  - $-\Theta(n+r)$
- · 时间代价
  - 不需要移动记录本身 , 只需要修改记录的 next 指针
  - $-\Theta(d\cdot(n+r))$





### 基数排序效率

- ·时间代价为  $\Theta(d \cdot n)$  , 实际上还是  $\Theta(n \log n)$ 
  - 没有重复关键码的情况,需要 n 个不同的编码来表示它们
  - 也就是说 ,  $d >= \log_r n$  , 即在  $\Omega(\log n)$  中



### 8.6.1 桶式排序

# 思考:排序该排什么?

```
Template <class Elem >
void BucketSort(int n,int min,int max) {
  int i, num;
  int count[max-min];
  for (i=min; i<max; i++) // 计数器初始为0
     count[i]=0;
  for (i=0; i<n; i++) { // 开始统计计数
     cin >> num;
     count[num]++;
  for (i=min; i<max; i++) //按顺序输出
     while (count[i]>0) {
        cout << i;
        count[i]--;
```



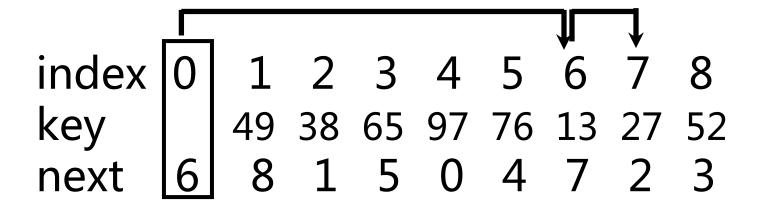


### 思考

- 1. 桶排事先知道序列中的记录都位于某个小区间段 [0, m)内。m 多大合适?超过这个范围怎么办?
- 2. 桶排中, count 数组的作用是什么?为什么桶 排要从后往前收集?



- 3. 顺序和链式基数排序的优劣?
- 4. 链式基数排序的结果整理?



有头结点的单链表的插入算法链式基数排序的结果



### 大纲

- · 8.1 排序问题的基本概念
- · 8.2 插入排序 (Shell 排序)
- · 8.3 选择排序(堆排序)
- · 8.4 交换排序
  - 8.4.1 冒泡排序
  - 8.4.2 快速排序
- · 8.5 归并排序
- · 8.6 分配排序和索引排序
- · 8.7 排序算法的时间代价
- · 内排序知识点总结



### 8.6 分配排序和基数排序

- 不需要进行纪录之间两两比较
- 需要事先知道记录序列的一些具体情况



#### 8.6.1 桶式排序

## 8.6.1 桶式排序

- ·事先知道序列中的记录都位于某个小区间段 [0, m) 内
- · 将具有相同值的记录都分配到同一个桶中 , 然后依次按照编号从桶中取出记录, 组 成一个有序序列



8.6.1 桶式排序



待排数组: 7 3 8 9 6 1 8 1 2

每个桶 count:



前若干桶的 累计count:

```
0 2 3 4 4 5 6 8 9
```



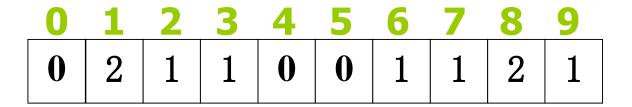




### 桶排序示意

待排数组: 7 3 8 9 6 1 8 1 2

每个桶count:



前若干桶的 累计count:



收集:





#### 8.6.1 桶式排序



### 桶式排序算法

```
template <class Record> void BucketSort(Record Array[], int n, int max) {
  Record *TempArray = new Record[n]; // 临时数组
  int *count = new int[max]; // 桶容量计数器
  int i;
  for (i = 0; i < n; i++)
                            // 把序列复制到临时数组
    TempArray[i] = Array[i];
  for (i = 0; i < max; i++)
                        // 所有计数器初始都为0
    count[i] = 0;
                             // 统计每个取值出现的次数
  for (i = 0; i < n; i++)
    count[Array[i]]++;
  for (i = 1; i < max; i++) // 统计小于等于i的元素个数
    count[i] = count[i-1]+count [i]; // c [i]记录i+1的起址
  for (i = n-1; i >= 0; i--) // 尾部开始,保证稳定性
    Array[--count[TempArray[i]]] = TempArray[i];
```



#### 8.6.1 桶式排序

### 算法分析

- ·数组长度为 n, 所有记录区间 [0, m) 上
- · 时间代价:
  - 统计计数: Θ(n+m) , 输出有序序列时循环 n 次
  - 总的时间代价为  $\Theta(m+n)$
  - 适用于 m 相对于 n 很小的情况
- ・空间代价:
  - -m 个计数器,长度为n的临时数组, $\Theta(m+n)$
- ・稳定



#### 8.6.2 基数排序

### 8.6.2 基数排序

- · 桶式排序只适合 m 很小的情况
- · 基数排序: 当 m 很大时,可以将一个记录的值即排序码拆分为多个部分来进行比较





· 假设长度为 n 的序列

$$R = \{ r_0, r_1, ..., r_{n-1} \}$$
  
记录的排序码 K 包含 d 个子排序码  
 $K = (k_{d-1}, k_{d-2}, ..., k_1, k_0)$ 

· R 对排序码有序,即对于任意 两个记录  $R_i$  ,  $R_j$  (0  $\leq$  i< j  $\leq$  n-1),都满足 ( $k_{i,d-1}$  ,  $k_{i,d-2}$  , ... ,  $k_{i,1}$  ,  $k_{i,0}$  )  $\leq$  ( $k_{j,d-1}$  ,  $k_{j,d-2}$  , ... ,  $k_{j,1}$  ,  $k_{j,0}$  )

#### 8.6.2 基数排序



例如:对0到9999之间的整数进行排序

- ·将四位数看作是由四个排序码决定,即千、百、十、个位,其中千位为最高排序码,个位为最低排序码。基数 r=10。
- ·可以按干、百、十、个位数字依次进行4次桶 式排序
- · 4趟分配排序后,整个序列就排好序了









- · 高位先排, 递归分治
  - 先按花色: **♣8 ♣1 ♦3 ♦7 ♥**J **♥9 ♠**3 **♠**9
  - 再按面值: **♣**1 **♣**8 **♦**3 **♦**7 **♥**9 **♥**J **♠**3 **♠**9
- · 低位先排,要求稳定排序!
  - 先面值: ♣1 ♠3 ♦3 ♦'7 ♣8 ♥9 ♠'9 ♥'J
  - 再花色: ♣1 ♣'8 ♦3 ♦'7 ♥9 ♥'J ♠3 ♠'9

#### 8.6.2 基数排序

### 高位优先法

- MSD, Most Significant Digit first
- 先处理高位  $k_{d-1}$  将序列分到若干桶中
- 然后再对每个桶处理次高位  $k_{d-2}$  , 分成更小的桶
- 依次重复,直到对  $k_0$  排序后,分成最小的桶, 每个桶内含有相同排序码( $k_{d-1}$  , ... ,  $k_1$  ,  $k_0$  )
- 最后将所有的桶中的数据依次连接在一起, 成为一个有序序列
- 这是一个分、分、...、分、收的过程



#### 8.6.2 基数排序

### 低位优先法

- · LSD , Least Significant Digit first
- · 从最低位  $k_0$  开始排序
- ·对于排好的序列再用次低位  $k_1$  排序;
- · 依次重复,直至对最高位  $k_{d-1}$  排好序后,整个序列成为有序的
- ·分、收;分、收;…;分、收的过程
  - 比较简单, 计算机常用



#### 8.6.2 基数排序

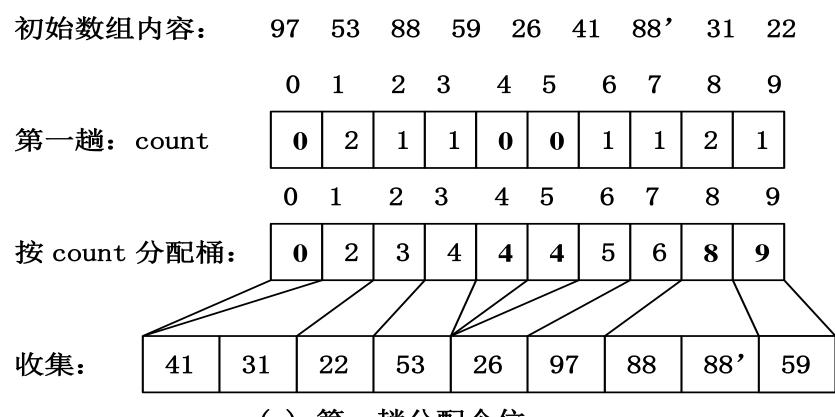
### 基数排序的实现

- 主要讨论 LSD
  - 基于顺序存储
  - 基于链式存储
- 原始输入数组 R 的长度为 n , 基数为 r , 排序 码个数为 d





### 基于顺序存储的基数排序



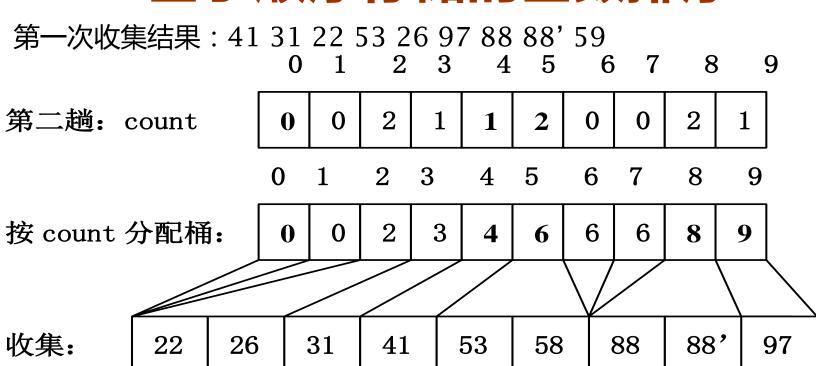
(a) 第一趟分配个位







### 基于顺序存储的基数排序



最终排序结果: 22 26 31 41 53 59 88 88' 97

(b) 第二趟分配十位





### 基于数组的基数排序

```
template <class Record>
void RadixSort(Record Array[], int n, int d, int r) {
  Record *TempArray = new Record[n];
  int *count = new int[r]; int i, j, k;
  int Radix = 1; // 模进位,用于取Array[j]的第i位
  for (i = 1; i <= d; i++) { // 对第 i 个排序码分配
    for (i = 0; j < r; j++)
      count[j] = 0; // 初始计数器均为0
    for (j = 0; j < n; j++) { // 统计每桶记录数
      k = (Array[j] / Radix) % r; // 取第i位
      count[k]++; // 相应计数器加1
```





```
for (j = 1; j < r; j++) // 给桶划分下标界
  count[j] = count[j-1] + count[j];
for (j = n-1; j >= 0; j--) { // 从数组尾部收集
 k = (Array[j] / Radix ) % r; // 取第 i 位
  count[k]--; // 桶剩余量计数器减1
 TempArray[count[k]] = Array[i]; // 入桶
for (j = 0; j < n; j++) // 内容复制回 Array 中
 Array[j] = TempArray[j];
Radix *= r:
          // 修改模Radix
```





## 顺序基数排序代价分析

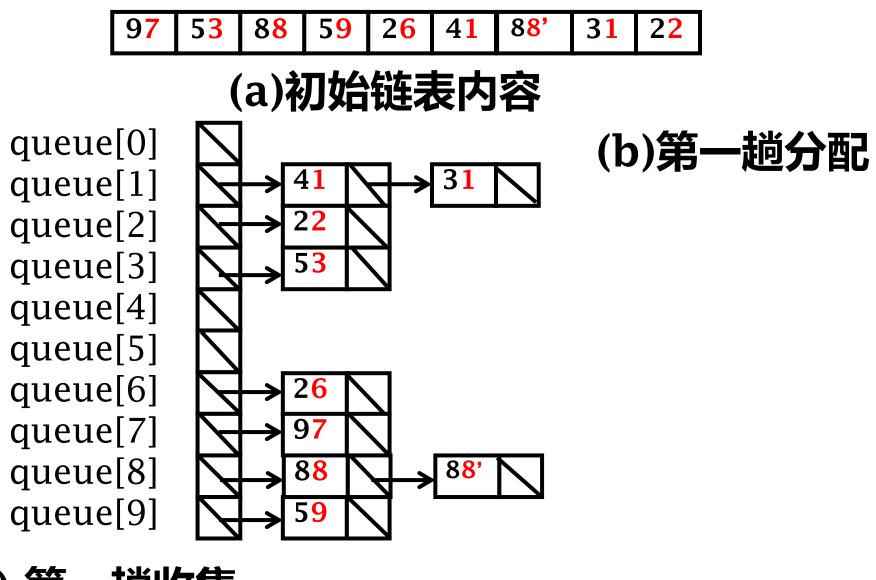
- ·空间代价:
  - 临时数组, n
  - r 个计数器
  - 总空间代价 Θ(n+r)
- ·时间代价
  - 桶式排序:Θ(*n+r*)
  - d 次桶式排序
  - $-\Theta(d\cdot(n+r))$





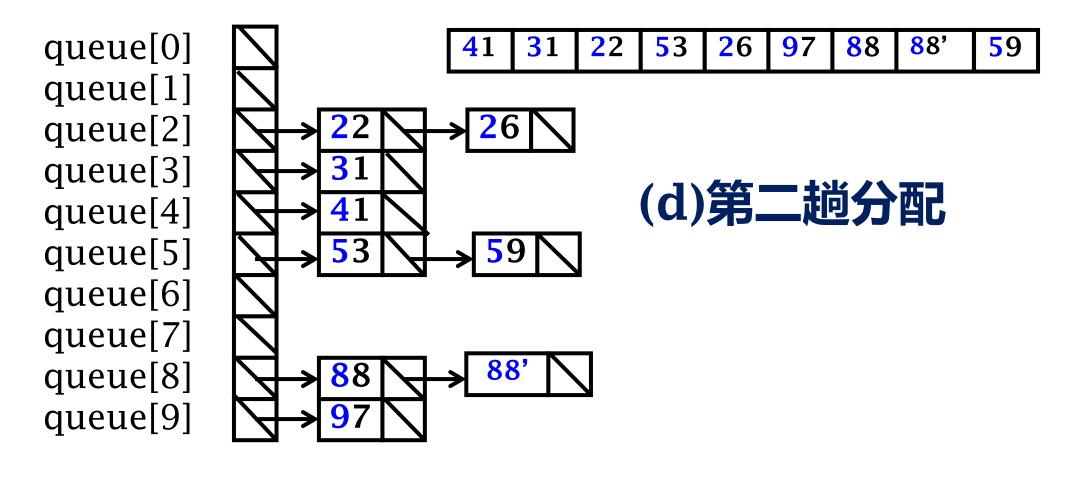
### 基于静态链的基数排序

- · 将分配出来的子序列存放在 r 个 (静 态链组织的) 队列中
- ·链式存储避免了空间浪费情况



### (c) 第一趟收集

41 31 22 53 26 97 88 88' 59



### (e) 第二趟收集结果 (最终结果)

 22
 26
 31
 41
 53
 59
 88
 88'
 97



### 静态队列定义

```
// 结点类
class Node {
public:
  int key;
                 // 结点的关键码值
                 // 下一个结点在数组中的下标
  int next;
};
class StaticQueue { // 静态队列类
public:
  int head;
  int tail;
};
```



#### 8.6.2 基数排序

### 基于静态链的基数排序

```
template <class Record>
void RadixSort(Record *Array, int n, int d, int r) {
  int i, first = 0; // first指向第一个记录
  StaticQueue *queue = new StaticQueue[r];
  for (i = 0; i < n-1; i++)
     Array[i].next = i + 1; // 初始化静态指针域
  Array[n-1].next = -1;  // 链尾next为空
  // 对第i个排序码进行分配和收集,一共d趟
  for (i = 0; i < d; i++) {
     Distribute(Array, first, i, r, queue);
     Collect(Array, first, r, queue);
  delete[] queue;
  AddrSort(Array, n, first); // 整理后,按下标有序
```



```
template <class Record>
void Distribute(Record *Array, int first, int i, int r,
StaticQueue *queue) {
  int j, k, a, curr = first;
  for (j = 0; j < r; j++) queue[j].head = -1;
  while (curr != -1) { // 对整个静态链进行分配
     k = Array[curr].key;
     for (a = 0; a < i; a++) // 取第i位排序码数字k
       k = k / r;
     k = k \% r:
     if (queue[k].head == -1) // 把数据分配到第k个桶中
        queue[k].head = curr;
     else Array[queue[k].tail].next = curr;
     queue[k].tail = curr;
     curr = Array[curr].next; // curr移动,继续分配
```





```
8 9 9
```

```
template <class Record>
void Collect(Record *Array, int& first, int r, StaticQueue
*queue) {
  int last, k=0; // 已收集到的最后一个记录
  while (queue[k].head == -1) k++; // 找到第一个非空队
  first = queue[k].head; last = queue[k].tail;
  while (k < r-1) { // 继续收集下一个非空队列
    k++;
    while (k < r-1 \&\& queue[k].head == -1)
       k++;
    if (queue[k].head != -1) { // 试探下一个队列
       Array[last].next = queue[k].head;
       last = queue[k].tail; // 最后一个为序列的尾部
  Array[last].next = -1; // 收集完毕
```

70





### 链式基数排序算法代价分析

- ·空间代价
  - n 个记录空间
  - r 个子序列的头尾指针
  - $-\Theta(n+r)$
- · 时间代价
  - 不需要移动记录本身 , 只需要修改记录的 next 指针
  - $-\Theta(d\cdot(n+r))$



#### 8.6.2 基数排序

## 基数排序效率

- ·时间代价为  $\Theta(d \cdot n)$  , 实际上还是  $\Theta(n \log n)$ 
  - 没有重复关键码的情况,需要 n 个不同的编码来表示它们
  - 也就是说 ,  $d >= \log_r n$  , 即在  $\Omega(\log n)$  中



### 8.6.1 桶式排序

# 思考:排序该排什么?

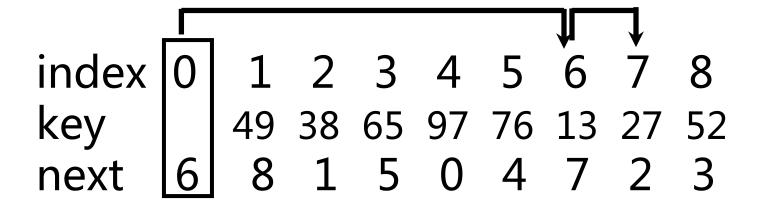
```
Template <class Elem >
void BucketSort(int n,int min,int max) {
  int i, num;
  int count[max-min];
  for (i=min; i<max; i++) // 计数器初始为0
     count[i]=0;
  for (i=0; i<n; i++) { // 开始统计计数
     cin >> num;
     count[num]++;
  for (i=min; i<max; i++) //按顺序输出
     while (count[i]>0) {
        cout << i;
        count[i]--;
```







- 思考
- 1. 顺序和链式基数排序的优劣?
- 2. 链式基数排序的结果整理?



有头结点的单链表的插入算法链式基数排序的结果

### 内排序



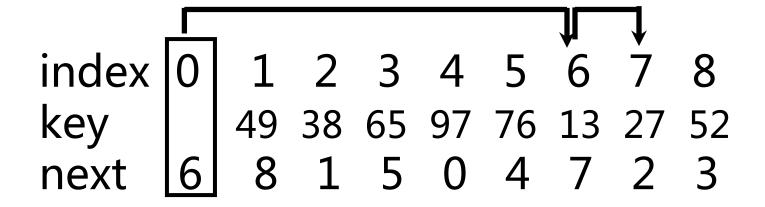
## 大纲

- · 8.1 排序问题的基本概念
- · 8.2 插入排序 (Shell 排序)
- · 8.3 选择排序(堆排序)
- · 8.4 交换排序
  - 8.4.1 冒泡排序
  - 8.4.2 快速排序
- 8.5 归并排序
- · 8.6 分配排序和索引排序
- · 8.7 排序算法的时间代价
- · 内排序知识点总结





# 基数排序的结果地址索引表示



有头结点的单链表的插入算法 链式基数排序的结果





# 线性时间整理静态链表

```
template <class Record>
void AddrSort(Record *Array, int n, int first) {
  int i, j;
  j = first;
                       // i待处理数据下标
  Record TempRec;
  for (i = 0; i < n-1; i++) {// 循环,每次处理第 i 个记录
     TempRec = Array[j]; // 暂存第 i 个的记录 Array[j]
     swap(Array[i], Array[j]);
    Array[i].next = j; // next 链要保留调换轨迹j
    j = TempRec.next; // j 移动到下一位
     while (j <= i) // j 比 i 小 , 则是轨迹 , 顺链找
       j = Array[j].next;
```





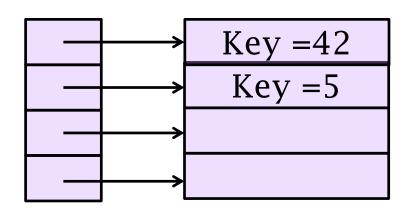
# 索引数组

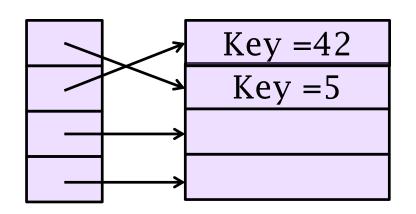
数据域很大,交换记录的代价比较高





# 索引数组





交换指针,减少交换记录的次数



# 索引结果:

- · 结果下标 IndexArray[i] 存放的是 Array[i] 中应该摆放的数据位置。
- · 整理后 Array[i] 对应原数组中 Array[IndexArray[i]]
- · 下标 0 1 2 3 4 5 6 7
- ·排序码 29 25 34 64 34' 12 32 45
- ・结果 5 1 0 6 2 4 7 3
  - 0 1 2 3 4 5 6 7
  - 12 25 29 32 34 34' 45 64



### 内排序



### 8.6.3 索引地址排序

# 索引排序的适用性

- ·一般的排序方法都可以
  - -那些赋值(或交换)都换成对 index 数组的赋值(或交换)
- · 举例:插入排序





# 插入排序的索引地址排序版本

```
template<class Record>
void AddrSort(Record Array[], int n) {
  //n为数组长度
  int *IndexArray = new int[n], TempIndex;
  int i,j,k;
  Record TempRec;
                       //只需一个临时空间
  for (i=0; i<n; i++)
    IndexArray[i] = i;
  for (i=1; i<n; i++) // 依次插入第i个记录
    for (j=i; j>0; j--) // 依次比较,发现逆置就交换
       if ( Array[IndexArray[j]] <Array[IndexArray[j-1]]))</pre>
         swap(IndexArray, j, j-1);
       else break; //此时i前面记录已排序
```







# 对索引数组的顺链整理

- 下标 0 1 2 3 4 5 6 7
- 排序码 29 25 34 64 34′12 32 45
- 索引 5 1 0 6 2 4 7 3

 0
 1
 2
 3
 4
 5
 6
 7

 • 结果
 12
 25
 29
 32
 34
 34'
 45
 64





### 插入排序的索引地址排序版本(续)

```
for(i=0;i<n;i++) { // 调整为按下标有序
    j = i;
    TempRec = Array[i];
    while (IndexArray[j] != i) {
      k=IndexArray[j];
      Array[j]=Array[k];
      IndexArray[j] = j;
      j = k;
    Array[j] =TempRec;
    IndexArray[j] = j;
```





# 第二种索引方法

- · 结果下标 IndexArray[i] 存放的是 Array[i] 中数据应该待的位置。
- ・排好序的 Array[IndexArray[i]] 对应原数组 中 Array[i]
- · 下标 0 1 2 3 4 5 6 7
- ·排序码 29 25 34 64 34' 12 32 45
- ・结果 2 1 4 7 5 0 3 6
  - 0 1 2 3 4 5 6 7
  - 12 25 29 32 34 34' 45 64







# 对第二种索引的顺链整理

- 下标 0 1 2 3 4 5 6 7
- 排序码 29 25 34 64 34' 12 32 45
- 索引 2 1 4 7 5 0 3 6

### 内排序



### 思考

- 1. 证明地址排序整理方案的时间代价为  $\theta(n)$
- 2. 修改快速排序,得到第一种索引结果
- 3. 采用 Rank 排序得到第二种索引的方法
- 4. 对静态链的基数排序结果进行简单变换得到第二种索引的方法

### 内排序



## 大纲

- · 8.1 排序问题的基本概念
- · 8.2 插入排序 (Shell 排序)
- · 8.3 选择排序(堆排序)
- 8.4 交换排序
  - 8.4.1 冒泡排序
  - 8.4.2 快速排序
- 8.5 归并排序
- · 8.6 分配排序和索引排序
- · 8.7 排序算法的时间代价





# 8.7 排序算法的时间代价

- ·简单排序算法的时间代价
- ·排序算法的理论和实验时间
- ・排序问题的下限



### 8.7 排序算法的时间代价

## 原因

- · 一个长度为 n 的序列平均有 n(n-1)/4 对逆置
- ·任何一种只对相邻记录进行比较的排序算法的平均时间代价都是  $\Theta(n^2)$



# 8.7.2 排序算法的理论和实验时间

算法	最大 时间	平均 时间	最小 时间	辅助 空间代价	稳定性
直接插入 排序	Θ(n <sup>2</sup> )	Θ(n <sup>2</sup> )	Θ(n)	Θ(1)	稳定
冒泡排序	Θ(n <sup>2</sup> )	Θ(n <sup>2</sup> )	Θ(n)	Θ(1)	稳定
选择排序	Θ(n <sup>2</sup> )	Θ(n <sup>2</sup> )	Θ(n <sup>2</sup> )	Θ(1)	不稳定

#### 第八章





### 8.7.2 排序算法的理论和实验时间

算法	最大时间	平均时间	最小时间	辅助空间	稳定性
Shell 排序(3)	$\Theta(n^{3/2})$	$\Theta(n^{3/2})$	$\Theta(n^{3/2})$	Θ(1)	不稳定
快速排序	$\Theta(n^2)$	Θ(nlog n)	Θ(nlog n)	Θ(log n)	不稳定
归并排序	Θ(nlog n)	Θ(nlog n)	Θ(nlog n)	Θ(n)	稳定
堆排序	Θ(nlog n)	Θ(nlog n)	Θ(nlog n)	Θ(1)	不稳定
桶式排序	Θ(n+m)	Θ(n+m)	Θ(n+m)	Θ(n+m)	稳定
基数排序	$\Theta(d \cdot (n+r))$	$\Theta(d \cdot (n+r))$	Θ(d·(n+r))	Θ(n+r)	稳定



# 小结

- · n 很小或基本有序时插入排序比较有效
- · Shell 排序选择增量以3的倍数递减
  - 需要保证最后一趟增量为1
- ·综合性能快速排序最佳



# 测试环境

- ・硬件环境
  - CPU: Intel P4 3G
  - 内存:1G
- ・软件环境
  - Windows XP
  - Visual C++ 6.0



# 随机生成待排序数组

```
//设置随机种子
inline void Randomize() {
  srand(1);
// 返回一个[0,n-1]之间的随机整数值
inline int Random(int n) {
  return rand() % (n);
//产生随机数组
ELEM *sortarray = new ELEM[1000000];
for(int i=0; i<1000000; i++)
  sortarray[i] = Random(32003);
```



# 时间测试

```
#include <time.h>
#define CLOCKS_PER_SEC 1000
clock_t tstart = 0; // 开始的时间
// 初始化计时器
void Settime() {
  tstart = clock();
// 上次 Settime() 之后经过的时间
double Gettime() {
  return (double)((double)clock() -
    (double)tstart) / (double)CLOCKS_PER_SEC:
```



# 排序的时间测试

```
Settime();
for (i=0; i<ARRAYSIZE; i+=listsize) {
   sort<int>(&array[i], listsize);
}
cout << "Sort with list size " << listsize
<< ", array size " << ARRAYSIZE << ", and
threshold " <<
THRESHOLD << ": " << Gettime() << "
seconds\n";</pre>
```







		-						
数组	10	100	1K	10K	100K	1M	10K	10K
规模							正序	逆序
直接插入 排序	0.00000047	0.000020	0.001782	0.175 2	17.917		0.00011	0.35094
选择排序	0.00000110	0.000041	0.002922	0.277 8	36.500		0.27781	0.29109
冒泡排序	0.0000160	0.000156	0.015620	1.561 7	207.69		0.00006	2.44840
Shell排序 (2)	0.00000156	0.000036	0.000640	0.010 9	0.1907	3.0579	0.00156	0.00312
Shell排序 (3)	0.00000078	0.000016	0.000281	0.003 8	0.0579	0.8204	0.00125	0.00687
堆排序	0.00000204	0.000027	0.000344	0.004	0.0532	0.6891	0.00406	0.00375
快速排序	0.0000169	0.000021	0.000266	0.003 0	0.0375	0.4782	0.00190	0.00199
优化快排 /16	0.00000172	0.000020	0.000265	0.002 0	0.0235	0.3610	0.00082	0.00088
优化快排 /28	0.00000062	0.000011	0.000141	0.001 8	0.0235	0.2594	0.00063	0.00063
山	0.00000219	0.000028	0.000375	0.004	0.0532	0.5969	0.00364	0.00360





数组规模	10	100	1K	10K	100K	1M	<b>10K</b> 正序	10K 逆序
优化归并/16	0.00000063	0.000014	0.000188	0.0030	0.0375	0.4157	0.00203	0.00265
优化归并/28	0.00000062	0.000013	0.000204	0.0027	0.0360	0.4156	0.00172	0.00265
顺序基数/8	0.00000610	0.000049	0.000469	0.0048	0.0481	0.4813	0.00484	0.00469
顺序基数/16	0.00000485	0.000034	0.000329	0.0032	0.0324	0.3266	0.00328	0.00313
链式基数/2	0.00002578	0.000233	0.002297	0.0234	0.2409	3.4844	0.02246	0.02281
链式基数/4	0.00000922	0.000075	0.000719	0.0075	0.0773	1.3750	0.00719	0.00719
链式基数/8	0.00000704	0.000048	0.000466	0.0049	0.0502	0.9953	0.00469	0.00469
链式基数/16	0.00000516	0.000030	0.000266	0.0028	0.0295	0.6570	0.00281	0.00281
链式基数/32	0.00000500	0.000027	0.000235	0.0028	0.0297	0.5406	0.00263	0.00266

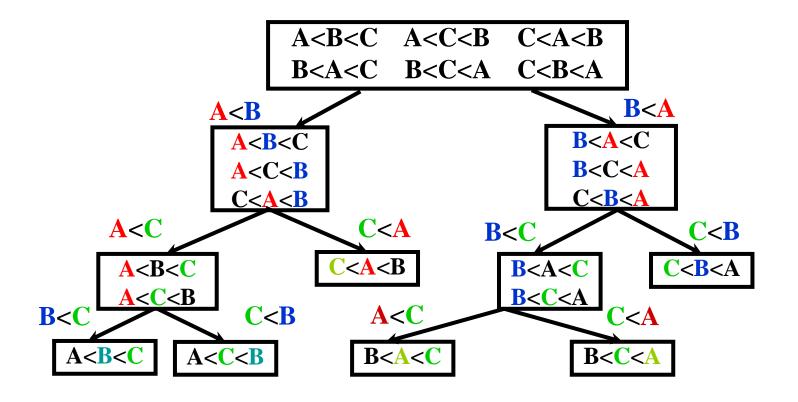


# 排序问题的下限

- ·排序问题的时间代价在  $\Omega(n)$  (单趟扫描) 到  $O(n \log n)$  (平均, 最差情况) 之间
- ·基于比较的排序算法的下限也为 $\Omega(n \cdot \log n)$
- ·用 判定树 (Decision Tree) 可以说明
  - 判定树中叶结点的最大深度就是排序算法在最差情况下需要的最大比较次数
  - 叶结点的最小深度就是最佳情况下的最小比较次数



### 用判定树模拟基于比较的排序





# 基于比较的排序的下限

- · 对 n 个记录, 共有 n! 个叶结点
- ·判定树的深度至少为 log(n!)
- ·在判定树深度最小的情况下最多需要  $\log(n!)$  次比较,即  $\Omega(n \cdot \log n)$
- ·在最差情况下任何基于比较的排序算法都至少需要  $\Omega(n\log n)$  次比较



# 排序问题的时间代价

- ·排序问题需要的运行时间也就是  $\Omega(n \cdot \log n)$
- · 所有排序算法都需要  $O(n \cdot \log n)$ 的运行时间, 因此可以推导出排序问题的时间代价为  $\Theta(n \cdot \log n)$



# 基数排序效率

- ·时间代价为  $\theta(d\cdot(n+r))(r \ll n)$ , 实际上还是  $\theta(n\log n)$ 
  - 没有重复关键码的情况,需要 n 个不同的编码来表示它们
  - $-即 d>=\log_r n$ ,在 $\Omega(\log n)$ 中



## 讨论:

- 1. 本章讨论的排序算法都是基于数组实现的,可否应用于动态链表?性能上是否有差异?
- 2. 试总结并证明各种排序算法的稳定性,若算法稳定,如何修改可以使之不稳定?若算法不稳定,可否通过修改使之稳定?
- 3. 试调研STL中的各种排序函数是如何组合各种排序算法的。





### 数据结构与算法

#### 谢谢聆听

国家精品课"数据结构与算法" http://www.jpk.pku.edu.cn/pkujpk/course/sjjg/

> 张铭,王腾蛟,赵海燕 高等教育出版社,2008. 6。"十一五"国家级规划教材