



北京大学

# 第八章 内排序

---

宋国杰

[gjsong@pku.edu.cn](mailto:gjsong@pku.edu.cn)

# 引言

## ➤ 排序技术应用广泛，是计算机科学中重要研究问题



The screenshot shows a Google search interface with the query '快速排序' (Quick Sort). The search results are displayed below the search bar, showing the number of results found (2,560,000) and the time taken (0.13 seconds). The first result is from Baidu Baike, titled '快速排序算法\_百度百科' (Quick Sort Algorithm - Baidu Baike). The second result is from Wikipedia, titled '快速排序- 维基百科，自由的百科全书' (Quick Sort - Wikipedia, the free encyclopedia). The third result is from MoreWindows, titled '白话经典算法系列之六快速排序快速搞定- MoreWindows - 博客园' (White Talk Classic Algorithm Series of Six Quick Sort Quick搞定 - MoreWindows - Blogcn). The fourth result is from CSDN, titled '快速排序算法- 结构之法算法之道- 博客频道- CSDN.NET' (Quick Sort Algorithm - Structure of the Law of Algorithm - Blog Channel - CSDN.NET).

Google 快速排序

网页 图片 地图 视频 更多 ▾ 搜索工具

找到约 2,560,000 条结果（用时 0.13 秒）

[快速排序算法\\_百度百科](#)  
[baike.baidu.com/view/19016.htm](http://baike.baidu.com/view/19016.htm) ▾

快速排序（Quicksort）是对冒泡排序的一种改进。由C. A. R. Hoare在1962年提出。它的基本思想是：通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的 ...

[快速排序- 维基百科，自由的百科全书](#)  
[zh.wikipedia.org/zh-cn/快速排序](http://zh.wikipedia.org/zh-cn/快速排序) ▾ 转为简体网页

快速排序是由東尼·霍爾所發展的一種排序算法。在平均狀況下，排序 $n$ 個項目要 $O(n \log n)$ 次比較。在最壞狀況下則需要 $O(n^2)$ 次比較，但這種狀況並不常見。事實上， ...

[演算法 - 競爭的排序演算法 - 正規的分析 - 選擇的關連性](#)

[白话经典算法系列之六快速排序快速搞定- MoreWindows - 博客园](#)  
[www.cnblogs.com/morewindows/archive/2011/08/13/2137415.html](http://www.cnblogs.com/morewindows/archive/2011/08/13/2137415.html) ▾

2011年8月13日 - 快速排序由于排序效率在同为 $O(N \log N)$ 的几种排序方法中效率较高，因此经常被采用，再加上快速排序思想----分治法也确实实用，因此很多软件 ...

[快速排序算法- 结构之法算法之道- 博客频道- CSDN.NET](#)  
[blog.csdn.net/v\\_JULY\\_v/article/details/6116297](http://blog.csdn.net/v_JULY_v/article/details/6116297) ▾

2011年1月4日 - 快速排序算法 作者Julv 二零一一年一月四日-----

# 分类

## ➤ 排序的分类

- ➡ **内部排序**：待排序记录数少放在内存，排序过程在内存进行，称为内部排序(internal sorting)
- ➡ **外部排序**：待排序记录数大，内存无法容纳所有记录，排序过程中还需要访问外存，称为外部排序 (external sorting)

# 8.1 基本概念

- 记录(Record): 进行排序的基本单位
- 关键码(Key): 唯一确定记录的一个或多个域
- 排序码(Sort Key): 作为排序运算依据的一个或多个域
- 序列(Sequence): 线性表, 由记录组成的集合
- 排序(Sorting) — 将序列中的记录按照排序码特定的顺序排列起来, 即排序码域的值具有不减(或不增)的顺序

➤ 给定一个序列  $R = \{r_1, r_2, \dots, r_n\}$ , 其排序码分别为  $k = \{k_1, k_2, \dots, k_n\}$ , 排序的目的就是将  $R$  中的记录按照特定的顺序重新排列, 形成一个新的有序序列  $R' = \{r'_1, r'_2, \dots, r'_n\}$

➡ 相应排序码为  $k' = \{k'_1, k'_2, \dots, k'_n\}$

➡ 其中  $k'_1 \leq k'_2 \leq \dots \leq k'_n$  或  $k'_1 \geq k'_2 \geq \dots \geq k'_n$ , 前者称**不减序**, 后者称**不增序**

### ➤ 稳定算法与不稳定算法

➡ 若记录序列中的任意两个记录  $R_x$ 、 $R_y$  的关键字  $K_x = K_y$ , 如果在排序之前和排序之后, 它们的相对位置保持不变, 则这种排序方法是稳定的, 否则是不稳定的。

# 排序算法的衡量标准

➤ 时间代价：记录的比较和交换次数

➡ 最小时间代价

➡ 最大时间代价

➡ 平均时间代价

➤ 空间代价：所需附加空间的大小

# 排序算法的分类

## ➤ 简单排序: $O(n^2)$

- ➡ 插入排序(Insert sort)
- ➡ 直接选择排序(Selection sort)
- ➡ 冒泡排序(Bubble sort)

## ➤ Shell排序(Shell sort)

## ➤ 分治排序

- ➡ 快速排序(Quicksort)
- ➡ 归并排序(Mergesort)

## ➤ 堆排序(Heapsort)

## ➤ 分配排序 (Binsort)

## 8.2 三种 $O(n^2)$ 的简单排序

- **插入排序(Insert Sort)**
- **冒泡排序(Bubble Sort)**
- **选择排序 (Selection Sort)**
  
- **优点：简单，易于实现**
- **缺点：复杂性高，不适合对大规模记录文件进行排序**



# 1、插入排序

## ➤ 基本思想

- ➡ 依次将每个待排序的记录插入到一个有序子文件的合适位置

➤ 例如：有待排序文件为：45, 34, 78, 12, 34, 32, 29, 64

{已排序集合}

{待排序集合}

- ➡ 依次将待排序集合中元素插入到已排序集合中的合适位置，直到最后一个记录。

45	34	78	12	34'	32	29	64
----	----	----	----	-----	----	----	----

# 排序算法

```
void StraightInsertSorter(Record Array[], int n){
```

```
    for (int i=1; i<n; i++) {                //依次插入第i个记录
```

```
        for (int j=i; j>0; j--){
```

```
            if (Array[j]< Array[j-1])
```

```
                swap(Array, j, j-1);
```

```
            else break;                        //此时i前面记录已排序
```

```
        }
```

```
    }
```

```
}
```

插入  
前面  
有序  
集合  
中恰  
当的  
位置

# 算法分析

➤ 算法是稳定的

➤ 空间代价： $\Theta(1)$ ，交换操作需要一个辅助空间

➤ 时间代价

➤ 最佳情况 (正序)：n-1次比较，0次交换， $\Theta(n)$ 复杂度

➤ 最差情况 (逆序)：比较和交换次数为

$$\sum_{i=1}^{n-1} i = n(n-1)/2 = \Theta(n^2)$$

➤ 平均情况： $\Theta(n^2)$

**实验表明：当记录数量n较小时，直接插入排序是一种高效的排序算法！**

# 优化的插入排序算法

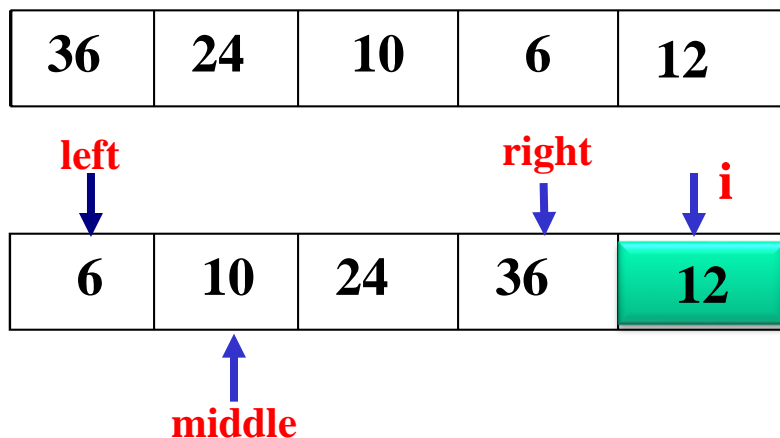
```
Void ImprovedInsertSorter (Record Array[], int n) {  
    Record TempRecord;           //临时变量  
    for (int i=1; i<n; i++) {     //依次插入第i个记录  
        TempRecord=Array[i];      //带插入记录存入临时变量  
        int j = i-1;  
        while ((j>=0) && (TempRecord < Array[j])){  
            Array[j+1] = Array[j]; j = j - 1;  
        }  
        Array[j+1] = TempRecord;  //临时变量值回填到j位置  
    }  
}
```

不用每次比较都要交换，但算法复杂性没有本质改变  
 $O(n^2)$ ，对大规模数据集性能会提高！

# 基于二分查找的插入排序

➤ **动机：**左侧待插入的集合已经有序，如何利用？

➡ **用二分法查找第 $i$ 个待插入记录的正确位置**



- 1) **移动条件：**  $< \text{middle}$ ,  $\text{right}$ 左移 ( $\text{middle}-1$ ) ;  
                   $\geq \text{middle}$ ,  $\text{left}$ 右移 ( $\text{middle}+1$ ) 。
- 2) **结束条件：**  $\text{left} > \text{right}$ ;
- 3) **插入位置：**  $\text{left}$

# 二分法插入排序算法

```
void BinaryInsertSorter (Record Array[], int n) {  
    Record TempRecord;  int left, right, middle;  
    for (int i=1;i<n;i++){  
        TempRecord = Array[i];  
        left = 0; right = i-1;  
        while(left <= right){  
            middle = (left+right)/2;  
            if (TempRecord < Array[middle])  
                right = middle-1;  
            else left = middle+1;  
        }  
        for(int j = i-1; j >= left; j --)  
            Array[j+1] = Array[j];  
        Array[left] = TempRecord;  
    }  
}
```

//依次插入第i个记录

//保存当前待插入记录

//记录已排好序序列的左右位置

//记录后移

//插入 left指针位置

# 算法分析

➤ 算法是稳定的

➤ 空间代价:  $\Theta(1)$

➤ 时间代价

- 比较次数降为 $n \log n$ 量级: 插入每个记录需 $\Theta(\log i)$ 次比较
- 移动次数仍为 $n^2$ 量级: 每插入一个记录最多移动 $i+1$ 次
- 因此, 最佳情况下总时间代价为 $\Theta(n \log n)$ , 最差和平均情况下仍为 $\Theta(n^2)$

## 2、冒泡排序

### ➤ 基本思想

➡ 不停地比较相邻的记录，如果不满足排序要求，就交换相邻记录，直到所有的记录都已经排好序

### ➡ 原理

— 若序列中有  $n$  个元素，通常进行  $n - 1$  趟。

- 第1趟，针对第 $r[0]$ 至 $r[n-1]$ 个元素进行。
- 第2趟，针对第  $r[1]$ 至 $r[n-1]$  个元素进行。……
- 第 $n-1$ 趟，针对第 $r[n-2]$ 至 $r[n-1]$ 个元素进行。

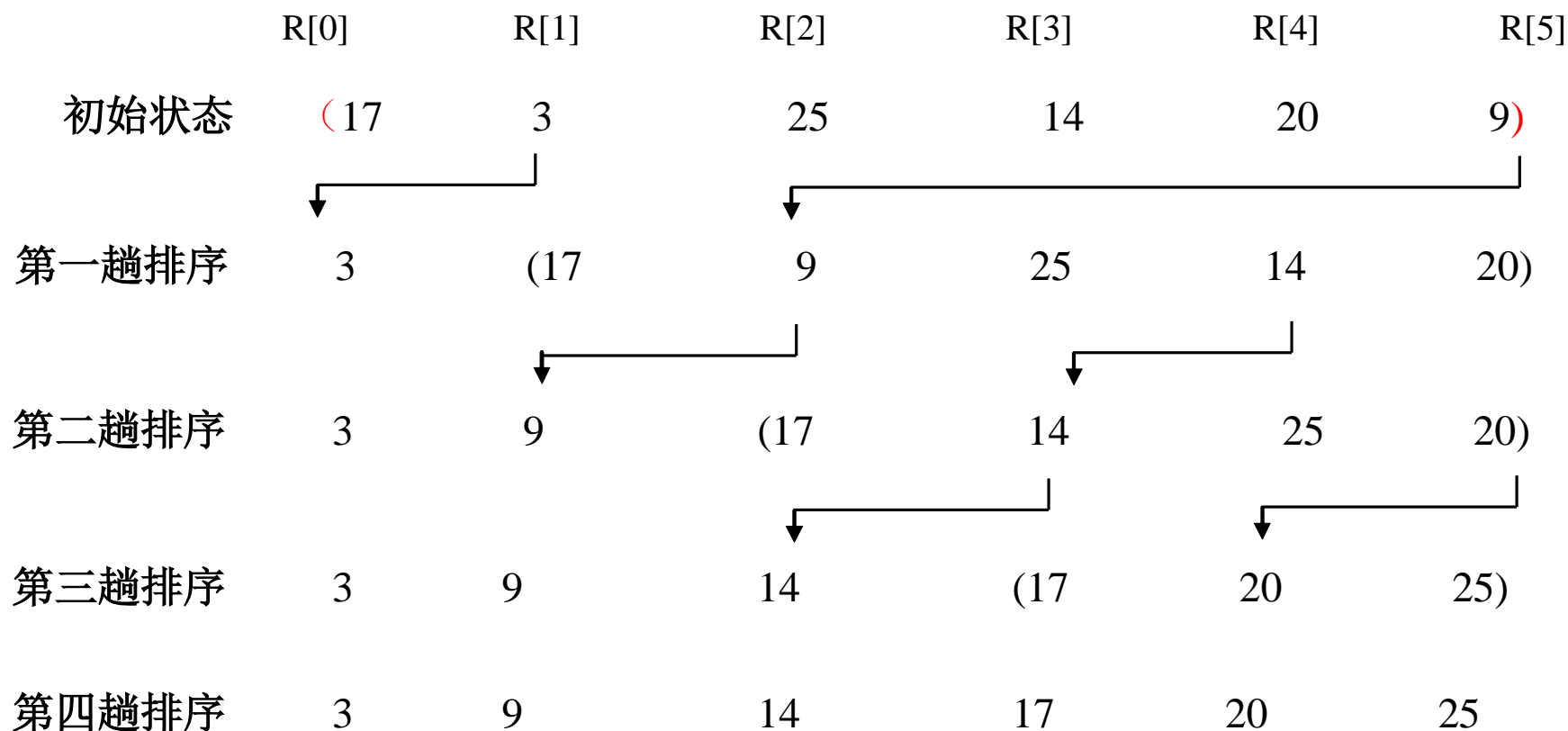
— 第 $i$ 趟进行的过程：

- 针对 $r[i-1]$ 至 $r[n-1]$  元素进行，比较两个相邻的元素。若相邻的元素的相对位置不正确，则进行交换；否则继续比较下面两个相邻的元素。



# 冒泡排序示例

➤ 由后至前，冒出最小



# 冒泡排序示例

➤ 由前至后，冒出最大

**例：**序列  $T=(21, 25, 49, 25^*, 16, 08)$  ， 冒泡排序过程

**初态：** 21, 25, 49, 25\*, 16, 08

**第1趟** 21, 25, 25\*, 16, 08, 49

**第2趟** 21, 25, 16, 08, 25\*, 49

**第3趟** 21, 16, 08, 25, 25\*, 49

**第4趟** 16, 08, 21, 25, 25\*, 49

**第5趟** 08, 16, 21, 25, 25\*, 49

# 冒泡排序算法

```
void BubbleSorter (Record Array[], int n){  
    for (int i=1; i<n; i++)                //向前冒泡  
        //依次比较相邻记录，如果发现逆置，就交换  
        for (int j=n-1; j>=i; j--)  
            if (Array[j] < Array[j-1])  
                swap(Array, j, j-1);  
}
```

# 算法分析

➤ 算法是稳定的

➤ 空间代价： $\Theta(1)$ 的临时空间

➤ 时间代价

➡ 比较次数

$$\sum_{i=1}^{n-1} (n-i) = n(n-1)/2 = \Theta(n^2)$$

➡ 交换次数最多为 $\Theta(n^2)$ ，最少为0，平均为 $\Theta(n^2)$ 。

➡ 最大，最小，平均时间代价均为 $\Theta(n^2)$ 。

# 优化的冒泡排序

## ➤ 优化思路

- ➡ 检查每次冒泡过程中是否发生过交换，如果没有，则表明整个数组已经排好序了，排序结束。
- ➡ 结束条件：不再元素交换

## ➤ 时间代价：

- ➡ 最小时间代价为 $\Theta(n)$ ：最佳情况下只运行第一轮循环
- ➡ 平均情况下时间代价仍为 $\Theta(n^2)$

# 优化的冒泡排序程序

```
void ImprovedBubbleSorter (Record Array[], int n) {  
    bool NoSwap;                // 是否发生交换的标志  
    for (int i=1; i<n; i++) {  
        NoSwap = true;  
        for (int j=n-1; j>=i; j--)  
            if (Array[j] < Array[j-1]) {  
                swap(Array, j, j-1);  
                NoSwap = false;  
            }  
        if (NoSwap) return;  
    }  
}
```

# 3、直接选择排序

## ➤ 基本思想

- ➡ 每一趟在后面 $n-i$ 个待排记录中选取**最小记录**和**第 $i$ 个记录**互换

## ➤ 具体过程

- ➡ 首先，在 $n$ 个记录中选择最小者与 $r[0]$ 互换；
- ➡ 然后，从剩余的 $n-1$ 个记录中选择最小者与 $r[1]$ 互换；
- ➡ ...如此下去，直到全部有序为止。

## ➤ 优点：实现简单

## ➤ 缺点：每趟只能确定一个元素，表长为 $n$ 时需要 $n-1$ 趟

**例：关键字序列T= (21, 25, 49, 25\*, 16, 08) , 请给出简单选择排序的具体实现过程。**

**原始序列：** 21, 25, 49, 25\*, 16, 08

最小值 08 与 r[0]  
交换位置

**第1趟** 08, 25, 49, 25\*, 16, 21

**第2趟** 08, 16, 49, 25\*, 25, 21

不稳定操作发  
生

**第3趟** 08, 16, 21, 25\*, 25, 49

**第4趟** 08, 16, 21, 25\*, 25, 49

**第5趟** 08, 16, 21, 25\*, 25, 49

**时间效率：**  $O(n^2)$  ——虽移动次数较少，但比较次数多。

**空间效率：**  $O(1)$  ——仅用到1个中间变量。

**算法的稳定性：** 不稳定 ——因为排序时，25\*到了25的前面。



# 直接选择排序算法

```
void StraightSelectSorter (Record Array[], int n) {  
    // 依次选出第i小的记录，即剩余记录中最小的那个  
    for (int i=0; i<n-1; i++){  
        int Smallest = i;           // 首先假设记录i就是最小的  
        for (int j=i+1; j<n; j++)    // 开始向后扫描所有剩余记录  
            if ( Array[j] < Array[Smallest])  
                Smallest = j;        // 如发现更小记录，记录其位置  
        swap(Array, i, Smallest);    // 将第i小的记录放在第i个位置  
    }  
}
```

**与冒泡排序的关系:** (1) 冒泡排序从后面两两交换冒出最小的；而直接选择排序是直接找到最小的和第一个进行交换！ (2) 前者是稳定的，后者是不稳定的！

# 4、简单排序算法的时间代价对比

比较次数	直接插入排序	改进的插入排序	二分法插入排序	冒泡排序	改进的冒泡排序	选择排序
最佳情况	$\Theta(n)$	$\Theta(n)$	$\Theta(n \log n)$	$\Theta(n^2)$	$\Theta(n)$	$\Theta(n^2)$
平均情况	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
最差情况	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$

移动次数	直接插入排序	改进的插入排序	二分法插入排序	冒泡排序	改进的冒泡排序	选择排序
最佳情况	0	$\Theta(n)$	$\Theta(n)$	0	0	$\Theta(n)$
平均情况	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$
最差情况	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$

总代价	直接插入排序	改进的插入排序	二分法插入排序	冒泡排序	改进的冒泡排序	选择排序
最佳情况	$\Theta(n)$	$\Theta(n)$	$\Theta(n \log n)$	$\Theta(n^2)$	$\Theta(n)$	$\Theta(n^2)$
平均情况	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
最差情况	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$

## 8.3 Shell排序

- Shell排序的提出基于直接插入排序的2个性质
  - ➡ 在待排序序列较短情形下效率高
  - ➡ 在整体有序的情形下时间代价低
- Shell排序又称为 “缩小增量排序”
  - ➡ 1959年由D.L.Shell提出

# 如何利用这两个性质？

## 1. 在初始无序时，进行等间隔的小序列分割

- ➡ 先将待排序序列转化为若干小序列，在这些小序列内进行直接插入排序

## 2. 在整个序列趋向有序后，逐步扩大序列规模

- ➡ 逐渐扩大小序列的规模，而减少小序列个数，使得待排序序列逐渐处于更有序的状态

## 3. 最后，对整个序列进行一次完整的插入排序

# 具体实现方法

1) 选定一个间隔增量序列 ( $n > d_1 > d_2 > \dots > d_t = 1$ )

➤  $n$ : 文件长度,  $d_i$ : 间隔增量,  $t$ : 排序趟数

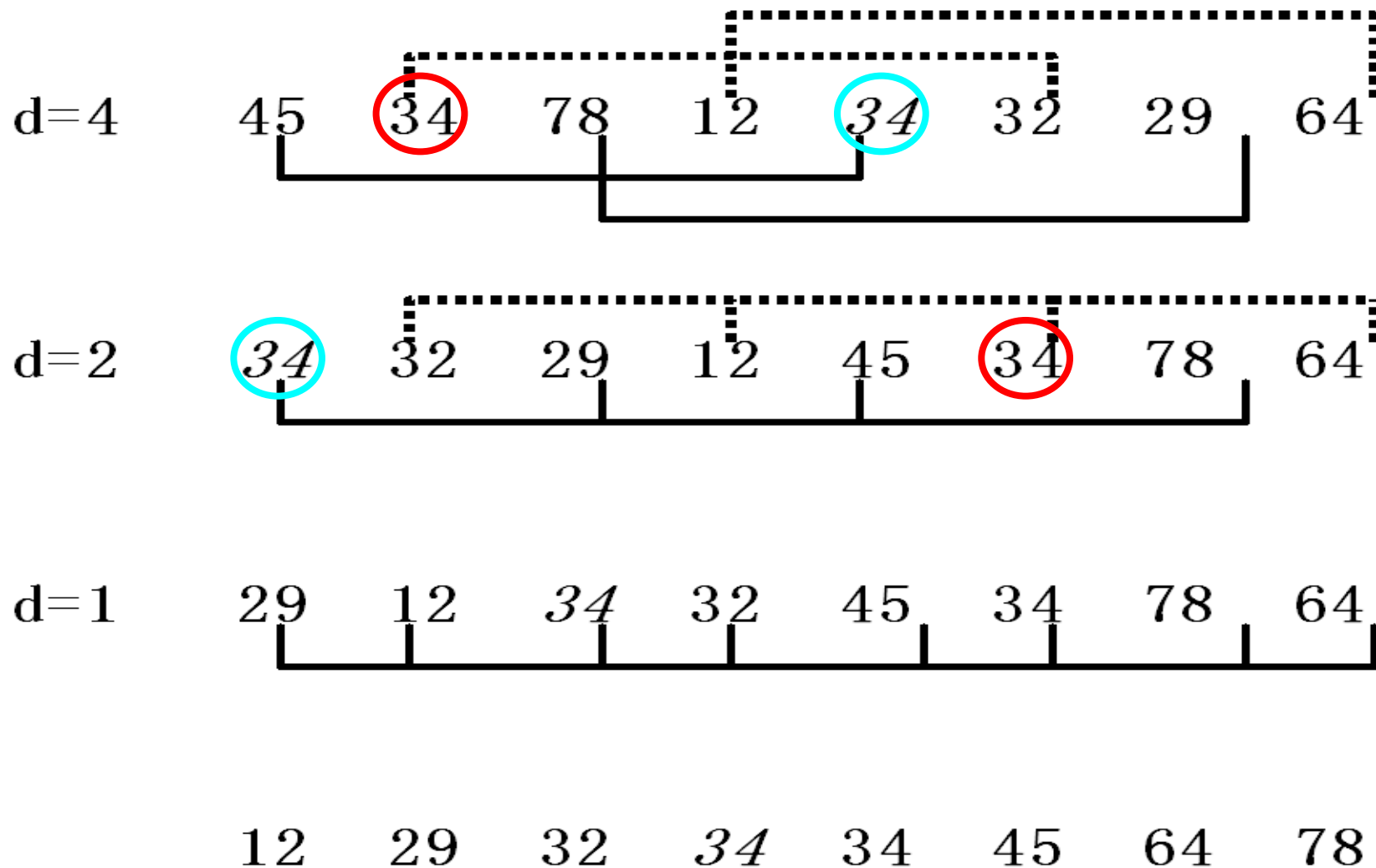
2) 将文件按  $d_1$  分组 (彼此相距  $d_1$  的记录划为一组), 在各组内采用直接插入法进行排序。

3) 分别按  $d_2, \dots, d_t$  重复上述分组和排序工作。

➤ Shell 最初提出的增量序列是

$$d_1 = \lfloor n / 2 \rfloor, \quad d_{i+1} = \lfloor d_i / 2 \rfloor$$

# Shell排序过程



# “delta=2” 的Shell排序

```
Void ShellSorter (Record Array[], int n){
```

```
    for (int delta=n/2; delta>0; delta/=2)
```

```
        //分别对delta个子序列排序
```

```
        for (int j=0; j<delta; j++)
```

```
            ModifiedInsertSort(&Array[j], n-j, delta);
```

```
    }
```



```
ModifiedInsertSort(Record Array[],int n, int delta) {//delta增量
```

```
// 对子序列中第i个记录排序
```

```
for (int i=delta; i<n; i+=delta)
```

```
    for (int j=i; j>=delta; j-=delta){
```

```
        if (Array[j] < Array[j-delta])
```

```
            swap(Array, j, j-delta);
```

```
        else break;
```

```
    }
```

```
}
```



# 分析

- 增量序列可有各种取法：任何正整数的递减序列 $d_1, d_2, \dots, d_t$ , 只要  $d_1 < n, d_t = 1$ , 原则上都可作为希尔排序的增量序列
- Hibbard增量序列
  - ➡  $\{2^k - 1, 2^{k-1} - 1, \dots, 7, 3, 1\}$ ,
  - ➡ Hibbard增量序列的Shell排序的效率可以达到 $\Theta(n^{3/2})$
- 选取其他增量序列还可以更进一步减少时间代价
- 希尔排序是一种不稳定的排序方法


## 8.4 基于分治法的排序

### ➤ “分而治之” 思想

- ➡ 将原始序列分为若干个子部分，然后分别进行排序
- ➡ 是解决问题的重要思想方法之一

### ➤ 两种算法

- ➡ 快速排序 (quick sorting)
- ➡ 归并排序 (merge sorting)



# 快速排序

C.A.R.Hoare (霍尔) 1962.

# 基于“分治”思想的快速排序



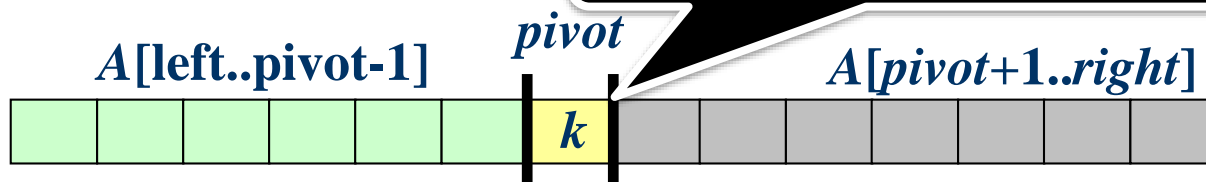
C.A.R.Hoare

## ➤ 分治思想



## ➤ 基于分治思想的快速排序

1. **轴值选择**: 从待排序列中选择轴值 $k$ (参考点)为划分基准;
2. **序列划分**: 划分为子序列 $L$ 和 $R$ ,  $L$ 中记录都 " $\leq$ "  $k$ ,  $R$ 中记录都 " $>$ "  $k$ ;



3. **递归排序**: 对子序列进行递归划分, 直到仅含1或0个元素

# 快速排序算法

```
void QuickSort(Record Array[], int left, int right) {  
    if (left < right){  
        int pivot = SelectPivot(left, right);           //选择轴值  
        pivot = Partition(Array, left, right);           //序列分割  
        QuickSort(Array, left, pivot-1);                 //对左子序列递归  
        QuickSort(Array, pivot + 1, right);               //对右子序列递归  
    }  
}
```

# 关键问题1：轴值选择

## ➤ 轴值选择的策略

➡ **目标：**使L和R长度尽可能相等

1、固定位置法

2、中值计算法

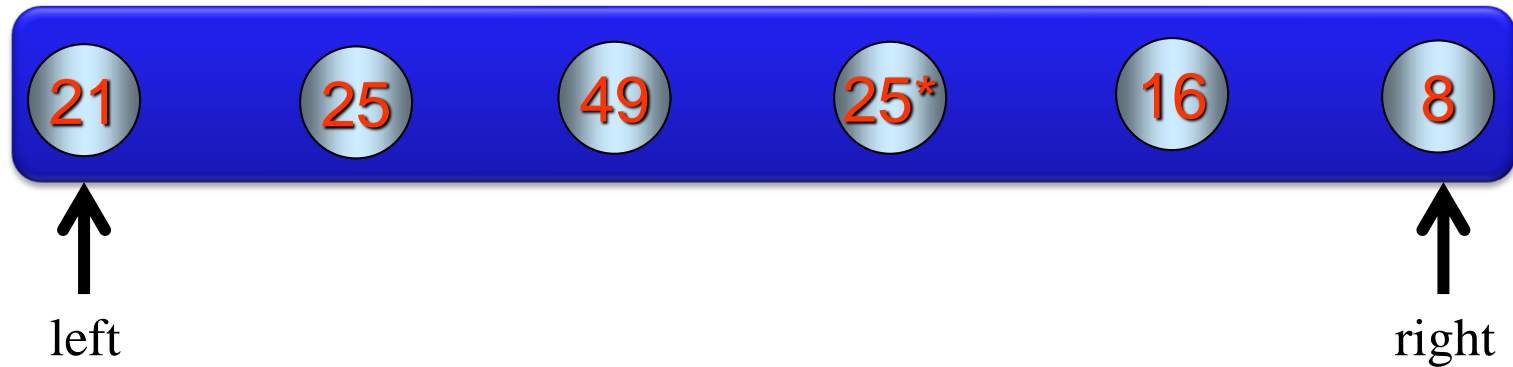
3、随机选择法

# 关键问题2：序列划分

## ➤ 序列划分的方法

➡ **选择首元素为轴值：** `pivotValue=Array[left];`

*pivotValue*



# 序列划分算法

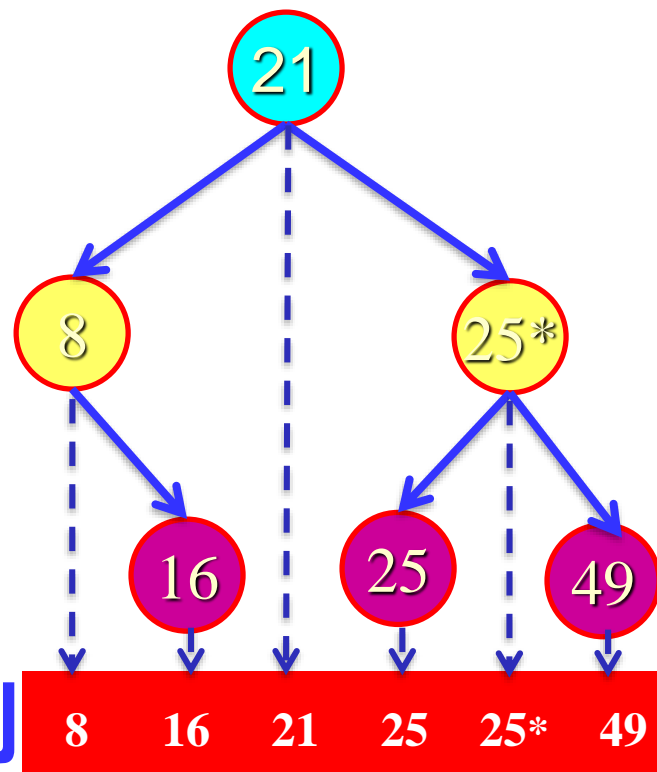
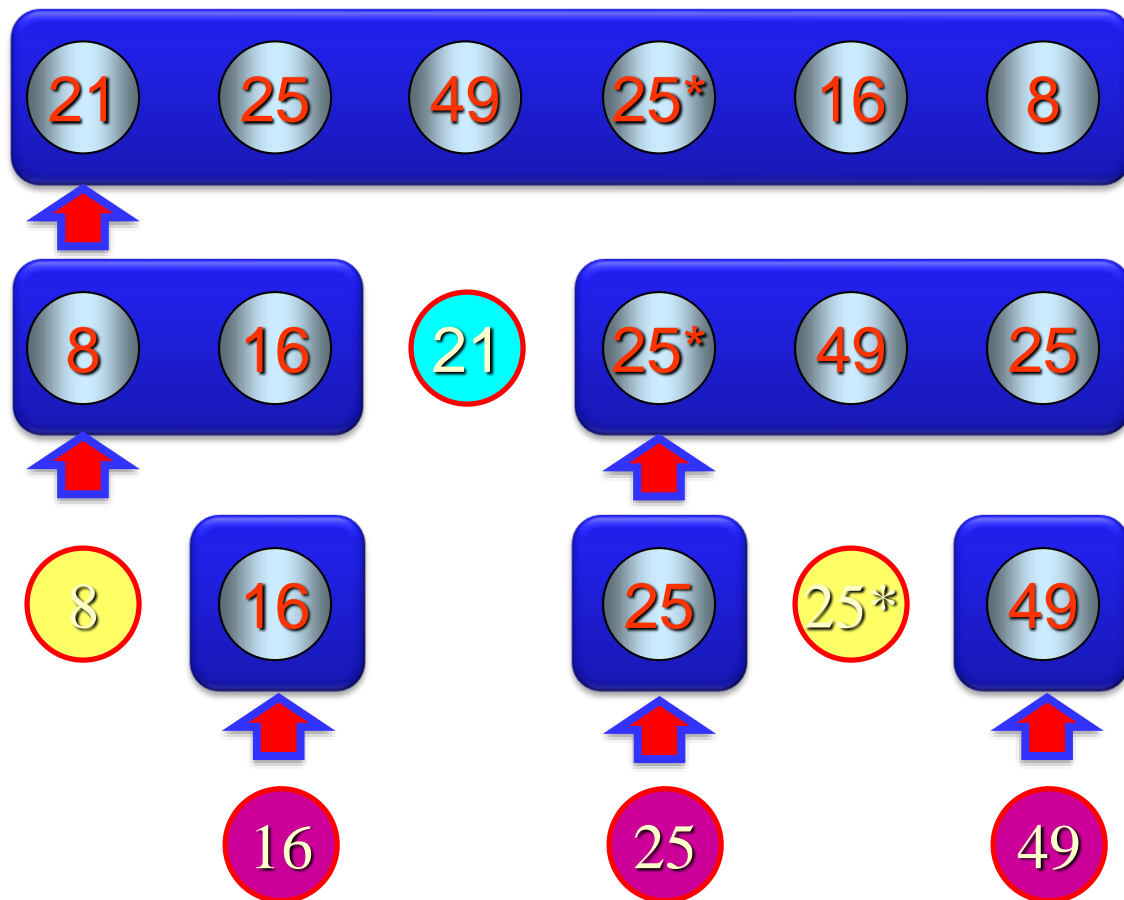
```
int Partition(Record Array[], int left,int right) {  
    int i = left, j = right;           //i为左指针, j为右指针  
    Record pivotValue = Array[left];   //将轴值放在临时变量中  
    while (i != j) {                   //划分过程  
        while ((Array[j] > pivotValue) && (i < j)) j--; //左移  
        if (i<j) {Array[i] = Array[j]; i++;}           //交换  
        while ((Array[i]<= pivotValue) && (i < j)) i++; //右移  
        if (i<j) { Array[j] = Array[i]; j--;}          //交换  
    }  
    Array[i] = pivotValue;              //轴值归位  
    return i;                           //返回轴值位置  
}
```



# 快速排序过程示例

递归划分过程

排序树



非稳定排序算法

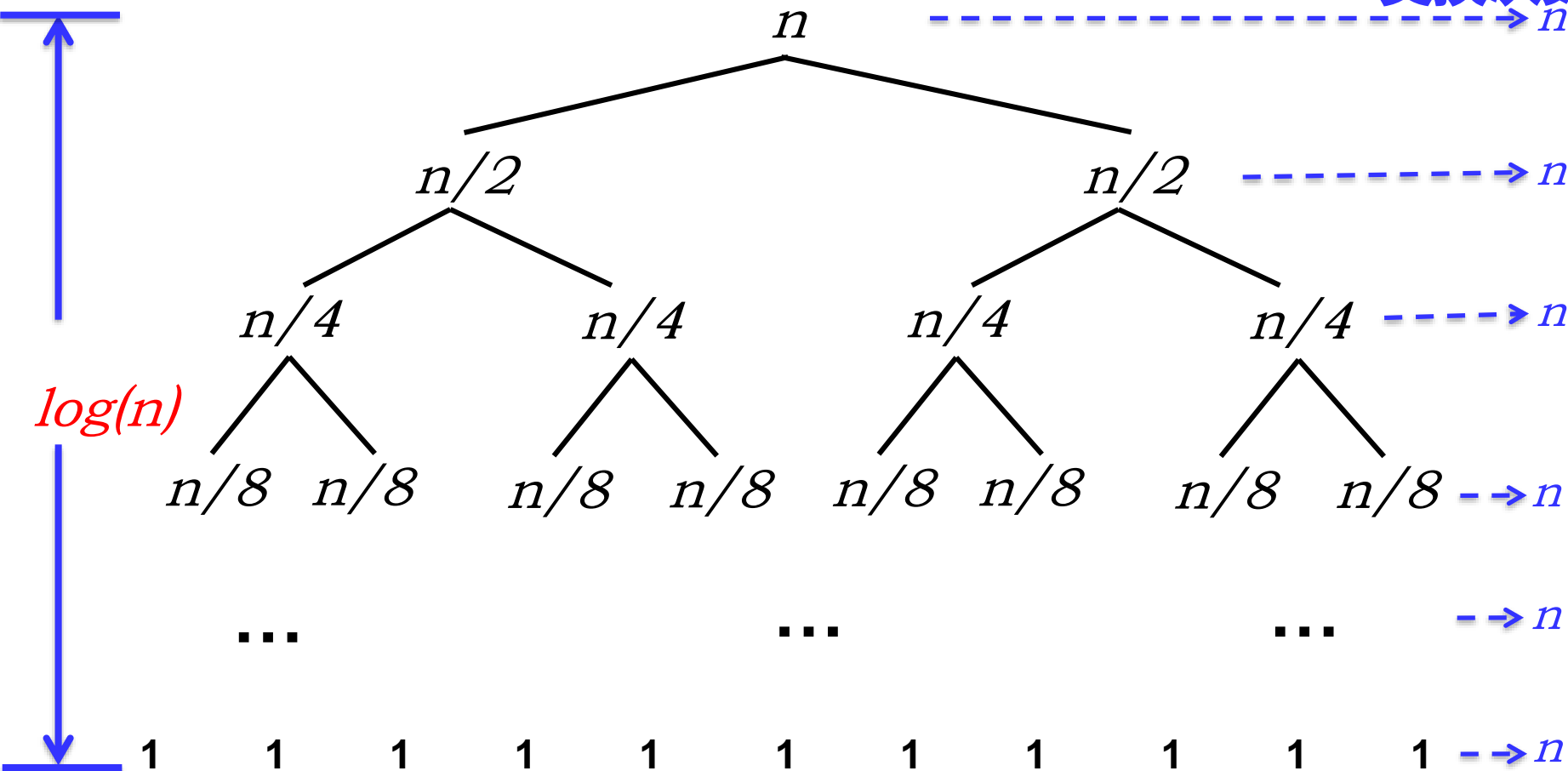
有序序列

# 算法分析(1)

➤ 最佳性能:  $T(n) = O(n \lg(n))$

每层比较

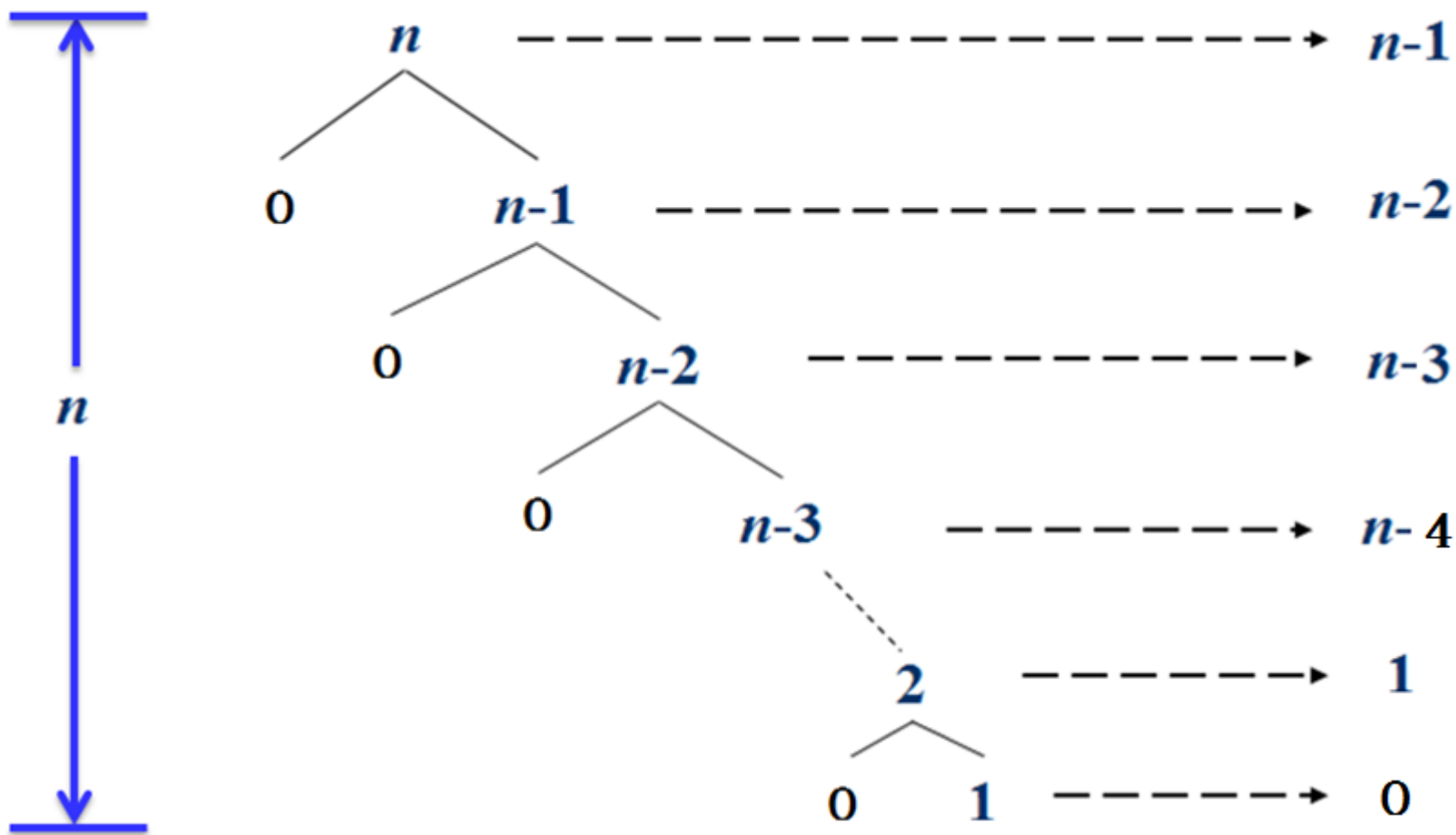
交换次数



# 算法分析(2)

➤ **最差性能:**  $T(n) = O(n^2)$

每层比较  
交换次数



# 算法分析(3)

➤ **平均性能:**  $T(n) = O(n \log n)$

$$T(n) = \frac{1}{n} \sum_{i=0}^{n-1} (T(i) + T(n-i-1)) + cn = \frac{2}{n} \sum_{i=0}^{n-1} T(i) + cn$$

$$nT(n) = 2 \sum_{i=0}^{n-1} T(i) + cn^2$$

$$nT(n) - (n-1)T(n-1) = 2T(n-1) + 2cn - c$$

$$nT(n) = (n+1)T(n-1) + 2cn \quad \text{公式两侧除以 } n(n+1)$$

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2c}{n+1}$$

# 算法分析(3)(续)

$$\frac{T(n)}{n+1} = \frac{T(n)}{n} + \frac{2c}{n+1}$$

$$\frac{T(n)}{n} = \frac{T(n-1)}{n-1} + \frac{2c}{n}$$

$$\frac{T(n-1)}{n-1} = \frac{T(n-2)}{n-2} + \frac{2c}{n-1}$$

...

$$\frac{T(2)}{2} = \frac{T(1)}{2} + \frac{2c}{3}$$

公式两侧分别相加求和

$$\frac{T(n)}{n+1} = \frac{T(1)}{2} + 2c \left( \frac{1}{n+1} + \frac{1}{n} + \dots + \frac{1}{3} \right)$$

$$\sum_{i=3}^{n+1} \frac{1}{i} = \log(n+1) + \gamma + \frac{3}{2}$$

$$\frac{T(n)}{n+1} \approx \frac{T(1)}{2} + 2c * \log(n+1)$$

调和级数

$$T(n) = O(n \log n)$$

In putting together this issue of *Computing in Science & Engineering*, we knew three things: it would be difficult to list just 10 algorithms; it would be fun to assemble the authors and read their papers; and, whatever we came up with in the end, it would be controversial. We tried to assemble the 10 algorithms with the greatest influence on the development and practice of science and engineering in the 20th century. Following is our list (here, the list is in chronological order; however, the articles appear in no particular order):

- Metropolis Algorithm for Monte Carlo
- Simplex Method for Linear Programming
- Krylov Subspace Iteration Methods
- The Decompositional Approach to Matrix Computations
- The Fortran Optimizing Compiler
- QR Algorithm for Computing Eigenvalues
- **Quicksort Algorithm for Sorting**
- Fast Fourier Transform
- Integer Relation Detection
- Fast Multipole Method

hand in developing the algorithm, and in other cases, the author is a leading authority.

## In this issue

Monte Carlo methods are powerful tools for evaluating the properties of complex, many-body systems, as well as nondeterministic processes. Isabel Beichl and Francis Sullivan describe the Metropolis Algorithm. We are often confronted with problems that have an enormous number of dimensions or a process that involves a path with many possible branch points, each of which is governed by some fundamental probability of occurrence. The solutions are not exact in a rigorous way, because we randomly sample the problem. However, it is possible to achieve nearly exact results using a relatively small number of samples compared to the problem's dimensions. Indeed, Monte Carlo methods are the only practical choice for evaluating problems of high dimensions.

John Nash describes the Simplex method for solving linear programming problems. (The use of the word *programming* here really refers to scheduling or

**快速排序是所有内排序方法中最好的方法！**

## 2、归并排序

### ➤ 基本思想

- ➡ 简单地将原始序列划分为两个子序列
- ➡ 分别对每个子序列递归划分，直至不可划分为止
- ➡ 最后将排好序的子序列合并为一个有序序列，即

### 归并过程

- ### ➤ 快速排序侧重于分割，而归并排序侧重于归并

# 归并思想





# 两路归并排序算法

```
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) {                                //如果序列中只有0或1个记录, 就不用排序
```

```
        middle = (left + right) / 2;    //从中间划分为两个子序列
```

```
        MergeSort(Array, TempArray, left, middle);    //对左边一半进行递归
```

```
        MergeSort(Array, TempArray, middle+1, right); //对右边一半进行递归
```

```
        Merge(Array, TempArray, left, right, middle); //进行归并
```

```
    }
```

```
}
```



```
template <class Record>
```

```
void Merge(Record Array[], Record TempArray[], int left, int right, int middle) {
```

```
    int i, j, index1, index2;
```

```
    for (j = left; j <= right; j++)
```

**// 将数组暂存入临时数组**

```
        TempArray[j] = Array[j];
```

```
    index1 = left;
```

**// 左边子序列的起始位置**

```
    index2 = middle+1;
```

**// 右边子序列的起始位置**

```
    i = left;
```

**// 从左开始归并**

```
    while (index1 <= middle && index2 <= right) {
```

```
        // 取较小者插入合并数组中
```

```
        if (TempArray[index1] <= TempArray[index2]) //为保稳定，相等时左边优先
```

```
            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，直接复制剩余的右序列
```

```
    Array[i++] = TempArray[index2++];
```

```
}
```

# 归并排序性能分析

➤ 容易看出，对  $n$  个记录进行归并排序的时间复杂度为  $O(n\log n)$ 。即：

- ➡ 每一趟归并的时间复杂度为  $O(n)$ ,
- ➡ 总共需进行  $\lceil \log n \rceil$  趟。
- ➡ 归并排序需要附加一倍的存储量  $O(n)$ 
  - 是辅助存储量最多的一种排序方法

➤ 是稳定排序算法

## 8.5 堆排序

➤ **直接选择排序**：直接从剩余记录中线性查找最大(小)记录

➤ **堆排序**

➡ 基于最大 (小) 堆来实现，效率更高

➤ **堆的分类**

$$\text{(最小堆)} \begin{cases} K_i \leq K_{2i+1} \\ K_i \leq K_{2i+2} \end{cases} \quad \text{或} \quad \text{(最大堆)} \begin{cases} K_i \geq K_{2i+1} \\ K_i \geq K_{2i+2} \end{cases} \quad (i=0, 1, \dots, \lfloor n/2 \rfloor - 1)$$

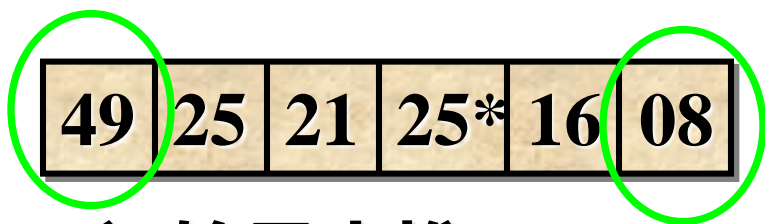
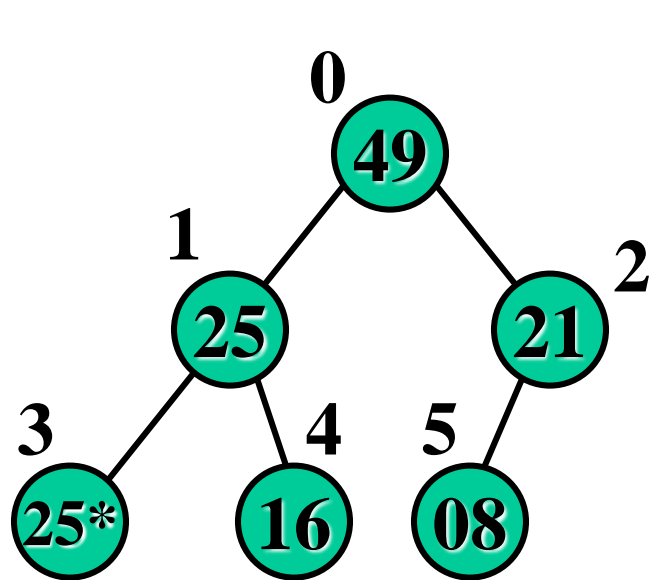
➤ **堆的完全二叉树表示**

➡ 堆可以用一棵完全二叉树表示，则  $K_{2i+1}$  是  $K_i$  的左孩子；  
 $K_{2i+2}$  是  $K_i$  的右孩子。

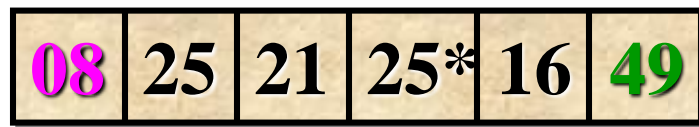
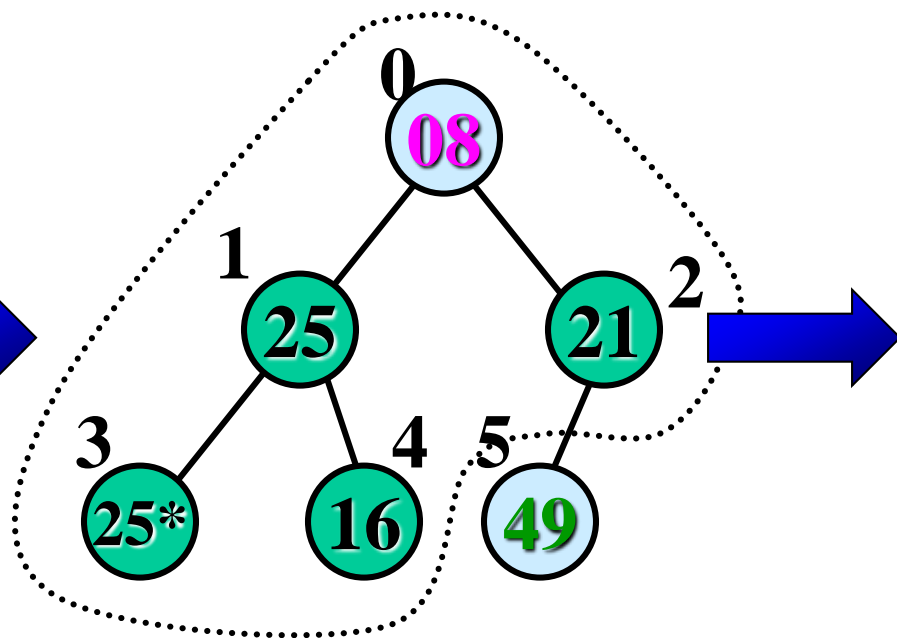
# 基本思想

1. 对所有记录建立最大堆 ( $O(n)$ )
2. 取出堆顶的最大记录移到数组末端, 放在下标 $n-1$ 的位置; 重新将剩下的 $n-1$ 个记录建堆 ( $O(\log n)$ ), 再取新堆顶最大的记录, 放到数组第 $n-2$ 位; ...; 不断重复这一操作, 直到堆为空。
3. 这时数组正好是按从小到大排序

# 例：对刚才建好的最大堆进行排序：

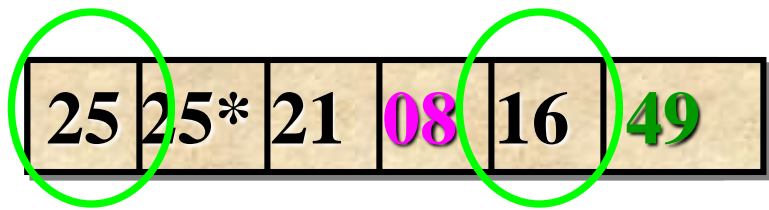
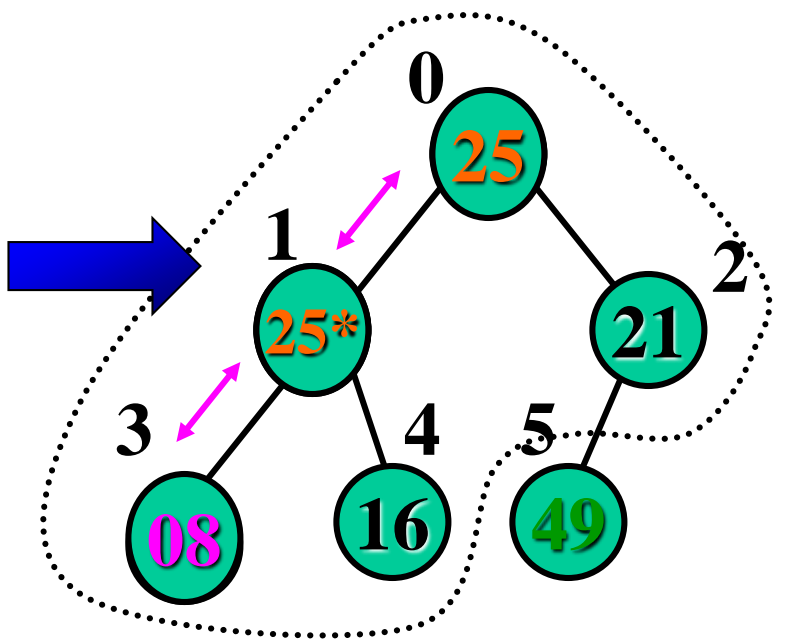


初始最大堆

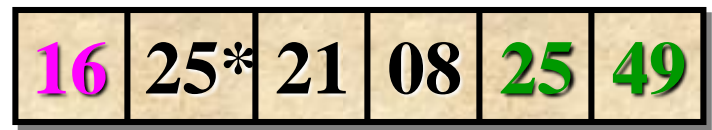
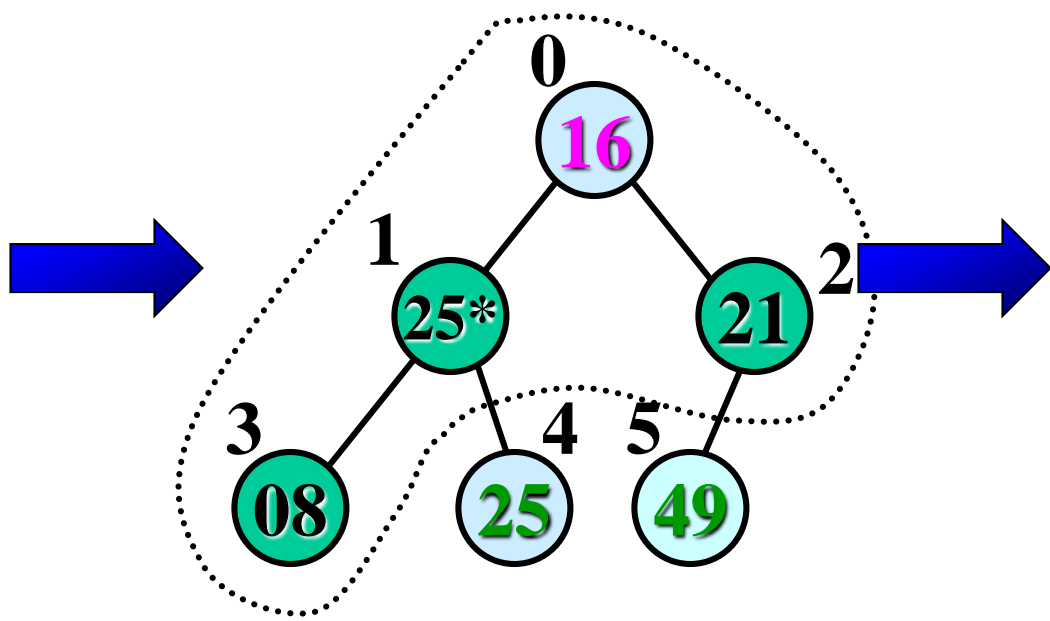


交换 0号与 5号记录

$r[0]$   $r[n-1]$

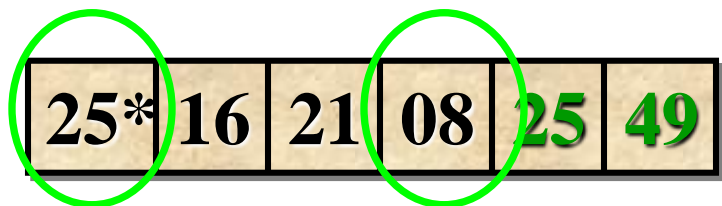
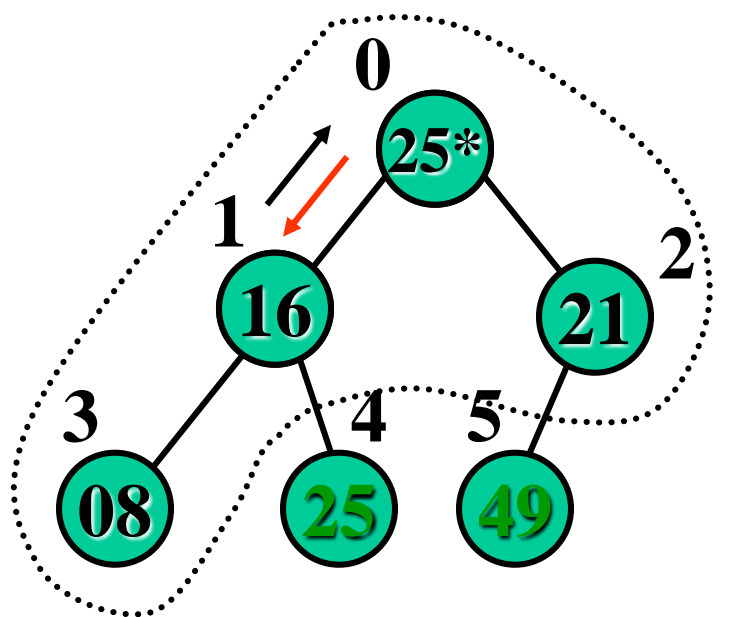


从 0 号到 4 号 重新  
调整为最大堆

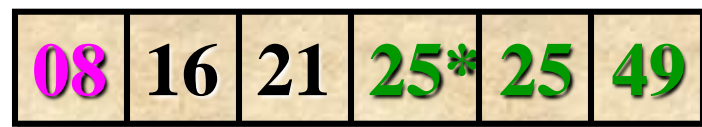
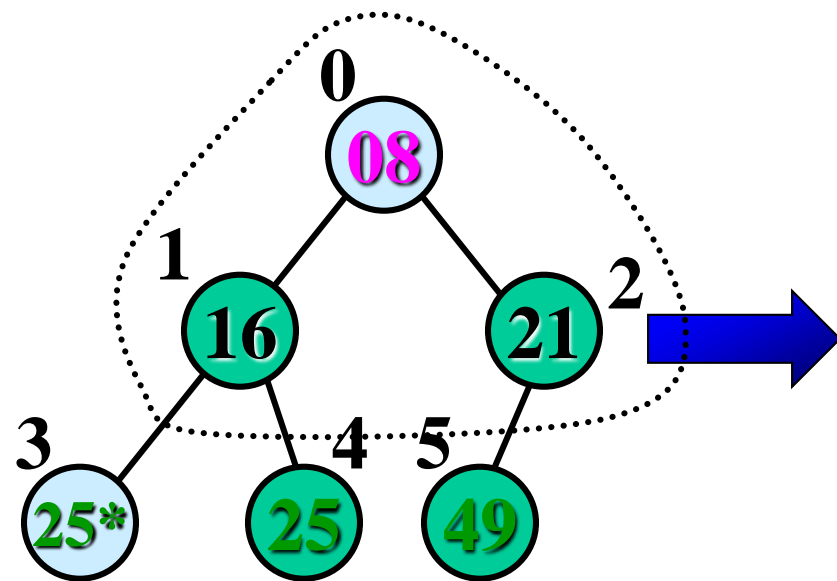


交换 0号与 4 号记录

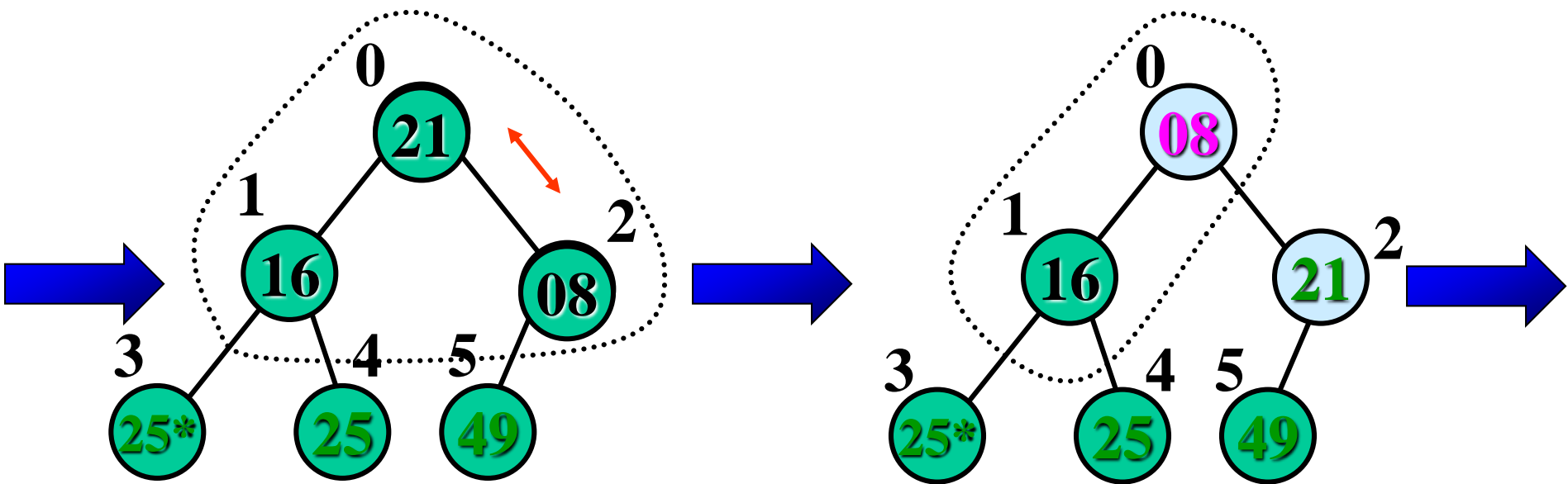




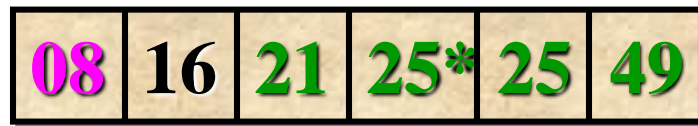
从 0 号到 3 号 重新  
调整为最大堆



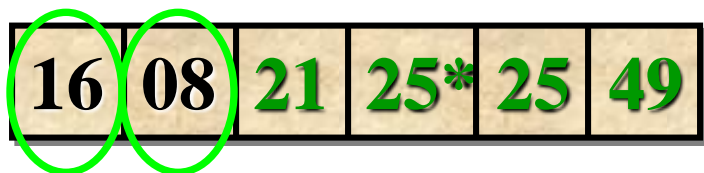
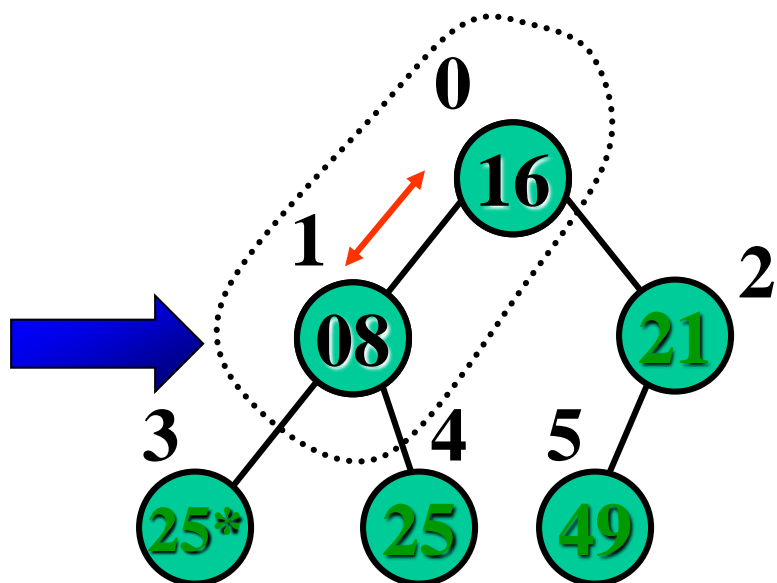
交换 0 号与 3 号记录



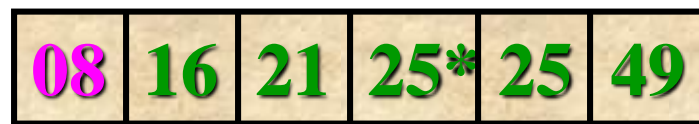
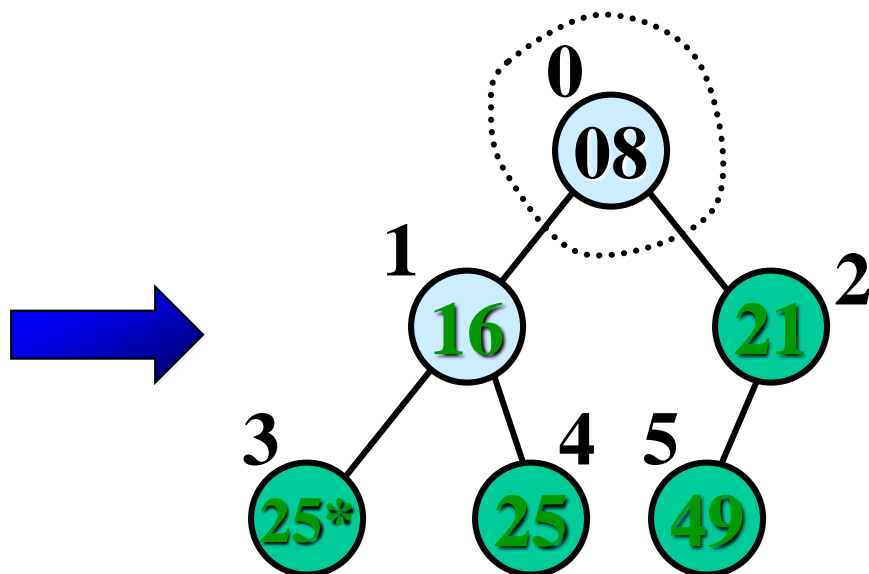
从 0 号到 2 号 重新  
调整为最大堆



交换 0 号与 2 号记录



从 0 号到 1 号 重新  
调整为最大堆



交换 0 号与 1 号记录  
排序完毕。

# 堆排序算法

```
template <class Record>
void sort(Record Array[], int n)
{
    int i;
    MaxHeap<Record> max_heap = MaxHeap<Record>(Array,n,n); // 建堆
    // 依次找出剩余记录中的最大记录，即堆顶
    for (i = 0; i < n; i++)
        max_heap.RemoveMax();
}
```

# 算法分析

- **非稳定性排序**
- **建堆：  $\Theta(n)$**
- **删除一次堆顶重新建堆：  $\Theta(\log n)$**
- **一次建堆,  $n$ 次删除堆顶, 总时间代价为  $\Theta(n \log n)$**
- **理论上, 堆排序最佳、最差、平均情况下的时间代价均为  $\Theta(n \log n)$**
- **辅助空间代价：  $\Theta(1)$**

# 8.6 分配排序

➤ 前述排序算法都是基于 “**比较 & 交换**” 完成

## ➤ 分配排序

➡ 不是通过 “比较” 和 “交换”

➡ 而是通过 “分配” 和 “收集”

## ➤ 前提假设

➡ 需事先知道待排序序列中记录的一些具体情况

– 如所有记录值都固定在某个区间 $[0, m)$

# 分配排序

---

**1. 桶排序**

**2. 基数排序**

**3. 索引排序**

# 1. 桶排序

## ➤ 桶排序 (Bucket Sorting)

- ➡ 序列中记录取值范围： $[0, m)$ ，标记为 $m$ 个桶
- ➡ 将相同值的记录**分配**到对应桶中
- ➡ 按桶号依次**收集**记录，组成有序序列

### 关键问题

如何将序列中的记录高效的分配到相应的桶中？



# 后继起始下标

待排数组: 7 3 8 9 6 1 8' 1' 2

Count数组:

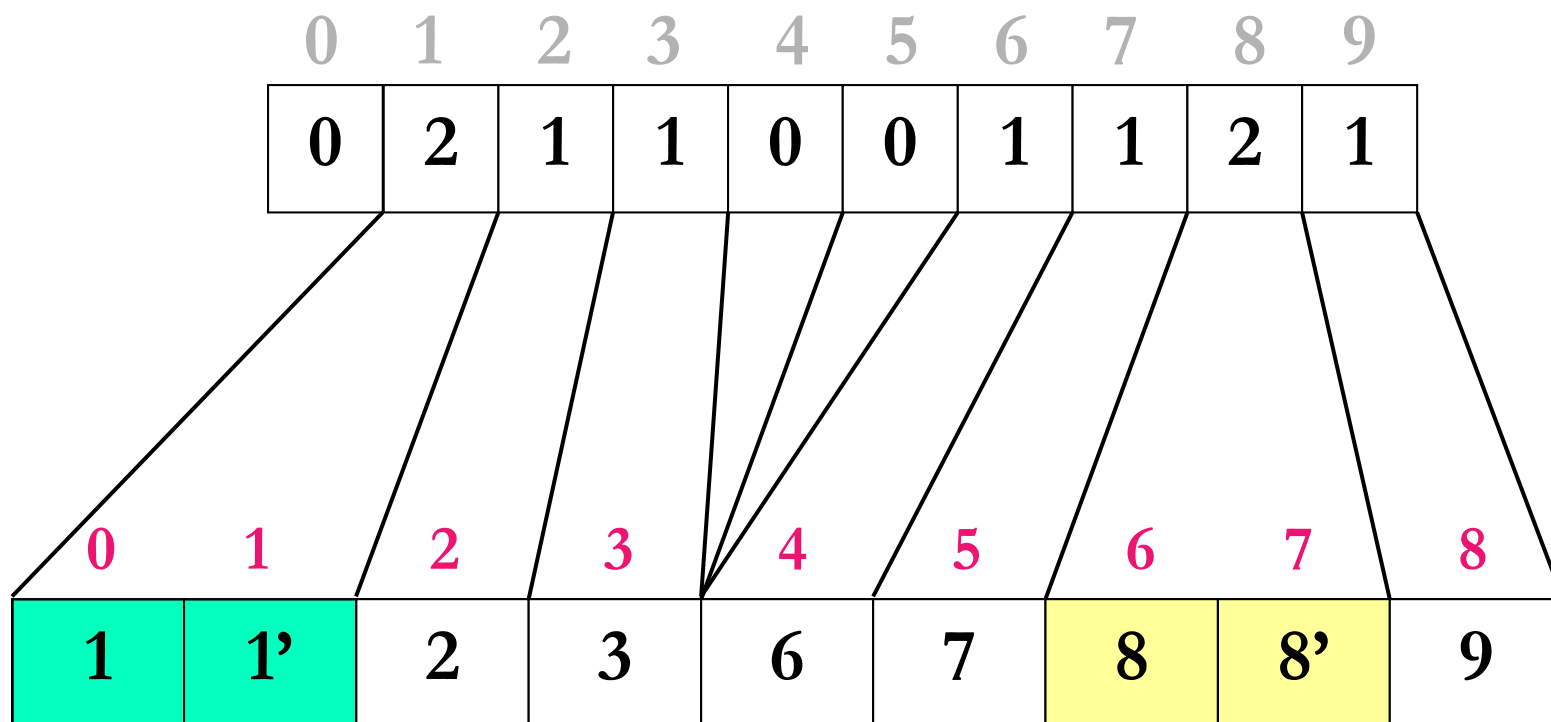
0	1	2	3	4	5	6	7	8	9
0	2	1	1	0	0	1	1	2	1

$$\text{count}[i] = \text{count}[i-1] + \text{count}[i]$$

后继起始下标:

0	2	3	4	4	4	5	6	8	9
0	1	2	3	4	5	6	7	8	9

# 桶计数与排序序列的映射关系



# 桶排序过程示例

收集自后向前，保持稳定！

待排数组: 7 3 8 9 6 1 8' 1' 2

第一趟count

0	1	2	3	4	5	6	7	8	9
0	2	1	1	0	0	1	1	2	1

后继起始下标:

0	0	2	4	4	4	5	6	7	9
---	---	---	---	---	---	---	---	---	---

0	1	2	3	4	5	6	7	8
1	1'	2	3	6	7	8	8'	9

# 桶式排序算法

```
template <class Record>
```

```
void BucketSort(Record Array[], int n, int m) {
```

```
    int *TempArray = new Record[n];          // 临时数组
```

```
    int *count = new int[m];                  // 小于或等于i的元素个数
```


```
    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 < m; i++)                    // 所有计数器初始都为0
```

```
        count[i] = 0;
```



```
for (i = 0; i < n; i++)
```

// 统计每个取值出现的次数

```
    count[Array[i]]++;
```

```
for (i = 1; i < m; i++)
```

// 后继起始地址计算

```
    count[i] = count[i-1] + count[i];
```

// 从尾部开始按顺序输出，保证排序的稳定性

```
for (i = n-1; i >= 0; i--)
```

```
    Array[--count[TempArray[i]]] = TempArray[i];
```

```
}
```

# 算法分析

## ➤ 时间代价

- ➡ 统计计数:  $\Theta(m+n)$
- ➡ 总时间代价:  $\Theta(m+n)$

## ➤ 空间代价

- ➡ 需要 $m$ 个计数器,  $n$ 个临时空间
- ➡ 总的空间代价:  $\Theta(m+n)$
- ➡ 适用于 $m$ 相对于 $n$ 很小的情况

## ➤ 稳定算法

# 对m的讨论

## ➤ m的取值决定了算法的复杂性

- ➡ 当m为 $\Theta(n)$ 数量级时，时间代价为 $\Theta(\Theta(n)+n)$ ，还是 $\Theta(n)$ 。
- ➡ 当m为 $\Theta(n \log n)$ 或 $\Theta(n^2)$ 时，时间代价变成 $\Theta(n \log n)$ 或 $\Theta(n^2)$ 。

## ➤ 因此，桶式排序只适合m很小的情况

**m非常大时如何处理？**

## 2、基数排序

### ➤ 排序码由多个部分组成

➡ 序列  $R = \{r_0, r_1, \dots, r_{n-1}\}$ , 每个排序码  $K$  由  $d$  位子排序码组成

–  $K = (k_{d-1}, k_{d-2}, \dots, k_1, k_0)$ ,

➡  $R$  有序就是对于任意两记录  $R_i, R_j$ , 满足

$$(k_{i,d-1}, k_{i,d-2}, \dots, k_{i,1}, k_{i,0}) \leq (k_{j,d-1}, k_{j,d-2}, \dots, k_{j,1}, k_{j,0})$$

– 其中  $k_{d-1}$  称为最高排序码,  $k_0$  称为最低排序码。

### ➤ $K_i$ 的取值范围称为基数, 记做 $r$



# 举例

## ➤ 0到9999之间整数排序

- ➡ 将4位数看作是由4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 $r=10$
- ➡ 可以按千、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（每位排序算法要保证稳定性!!）

## ➤ 扑克牌

- ➡ 若规定花色和面值的顺序关系为：
- ➡ 花色：♦ < ♣ < ♠ < ♥
- ➡ 面值：2 < 3 < 4 < 5 < 6 < 7 < 8 < 9 < 10 < J < Q < K < A
- ➡ 则可以先按花色排序，花色相同者再按面值排序；
- ➡ 也可以先按面值排序，面值相同者再按花色排序。

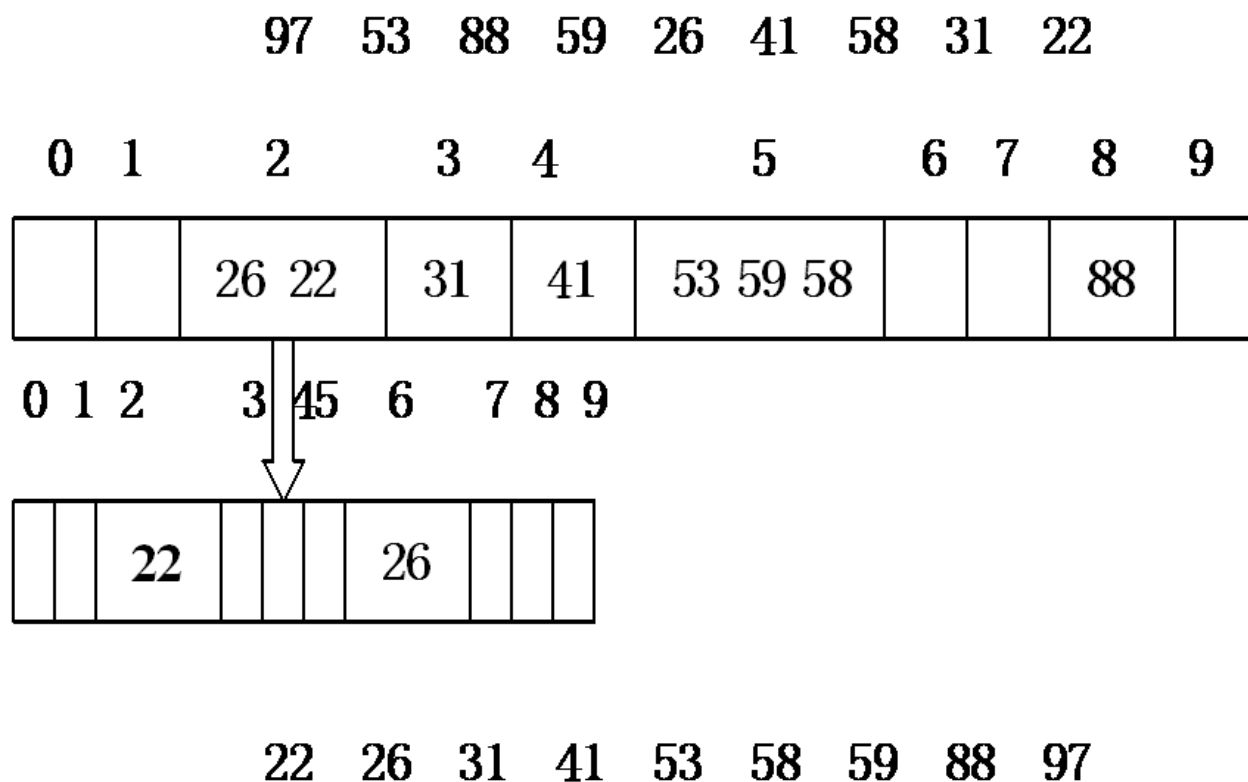
# 基数排序分类

---

- 高位优先法 (MSD, Most Significant Digit first)
- 低位优先法 (LSD, Least Significant Digit first)

# 高位优先法

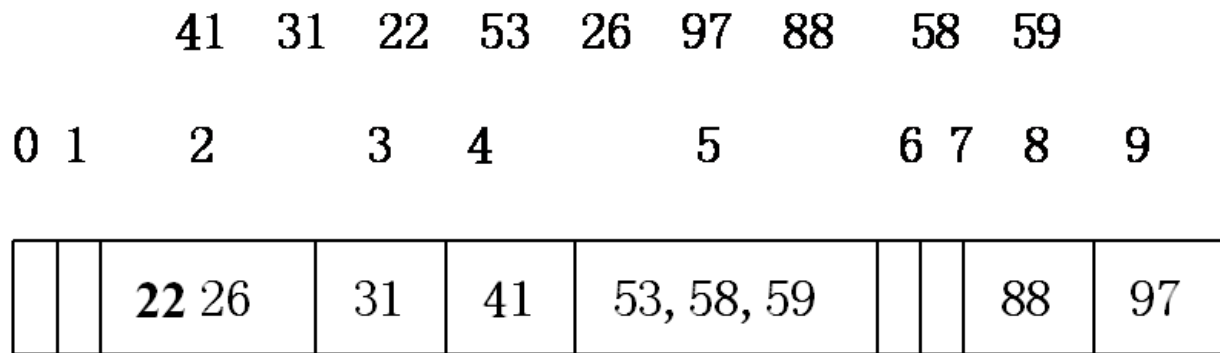
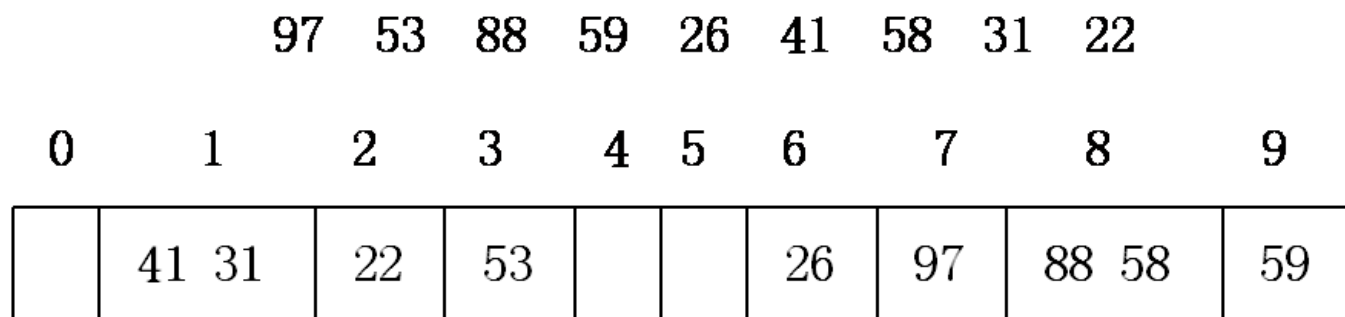
- 先按最高位 $k_{d-1}$ 进行桶式排序，将序列分成若干个桶中
- 再按次高位 $k_{d-2}$ 进行桶式排序，分成更小的桶
- 依次重复，直到对 $k_0$ 排序后，分成最小的桶，每个桶内含有相同的排序码 $(k_{d-1}, k_{d-2}, \dots, k_1, k_0)$
- 最后将所有的桶依次连接在一起，成为一个有序序列
- 这是一个分、分、...、分、收的过程
- 是一个递归分治问题



(a) 高位优先

# 低位优先法

- 从最低位 $k_0$ 开始对整个序列进行排序
- 对于排好序的整个序列再用次低位 $k_1$ 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
- 这是一个分、收；分、收；...；分、收的过程。
- 计算机常用



22 26 31 41 53 58 59 88 97

(b) 低位优先

# 基数排序的实现

---

A. 基于顺序存储

B. 基于静态链存储

**基于LSD方法进行算法实现**

# A. 基于顺序存储的基数排序

- 数组R长度:  $n$
- 基数:  $r$
- 排序码位数:  $d$



初始数组内容: 97 53 88 59 26 41 58 31 22

0 1 2 3 4 5 6 7 8 9

第一趟: count

0	2	1	1	0	0	1	1	2	1
---	---	---	---	---	---	---	---	---	---

0 1 2 3 4 5 6 7 8 9

按 count 分配桶:

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

收集:

41	31	22	53	26	97	88	58	59
----	----	----	----	----	----	----	----	----

0 1 2 3 4 5 6 7 8 9

第二趟: count

0	0	2	1	1	3	0	0	1	1
---	---	---	---	---	---	---	---	---	---

0 1 2 3 4 5 6 7 8 9

按 count 分配桶:

0	0	2	3	4	7	7	7	8	9
---	---	---	---	---	---	---	---	---	---

收集:

22	26	31	41	53	58	59	88	97
----	----	----	----	----	----	----	----	----

最终排序结果:

22 26 31 41 53 58 59 88 97

# 基于数组的基数排序算法

```
void RadixSorter<Record>::Sort(Record Array[], int n,int d, int r){
```

```
    //n为数组长度, d为排序码数, r为基数
```

```
    Record *TempArray =new Record[n];           //临时数组
```

```
    int* count= new int[r];                     //计数器
```


```
    int i,j,k;
```

```
    int Radix=1;
```

```
    for (i=1; i<=d; i++) {                     //取Array[j]的第i位排序码
```

```
        for (j=0; j<r; j++)                     //分别对第i个排序码分配
```

```
            count[j] = 0;                       // 初始计数器均为0
```



```
for (j=0; j<n; j++){ //统计每个桶中的记录数
    //取Array[j]的第i位排序码
    k=(Array[j] / Radix)%r;
    count[k]++;          //相应计数器加1
}
```

// 每个元素的后继起始下标地址

```
for (j=1; j<r; j++)
    count[j] = count[j-1] + count[j];
```

**//将所有桶中的记录依次收集到TempArray中**

**for (j=n-1; j>=0; j--) {**

**k=(Array[j] / Radix)%r; //取Array[j]的第i位排序码**

**TempArray[--count[k]] = Array[j];**

**}**

**for (j=0; j<n; j++)// 将临时数组中的内容复制到Array中**

**Array[j] = TempArray[j];**

**Radix\*=r;**

**}**

**}**

# 算法分析

## ➤ 空间代价:

➡ 临时数组,  $n$

➡  $r$  个计数器

➡  $\Theta(n+r)$

## ➤ 时间代价

➡ 桶式排序:  $\Theta(r+n)$

➡  $d$  次桶式排序

➡ 总的时间复杂性:  $\Theta(d \cdot (r+n))$

## B. 基于静态链的基数排序

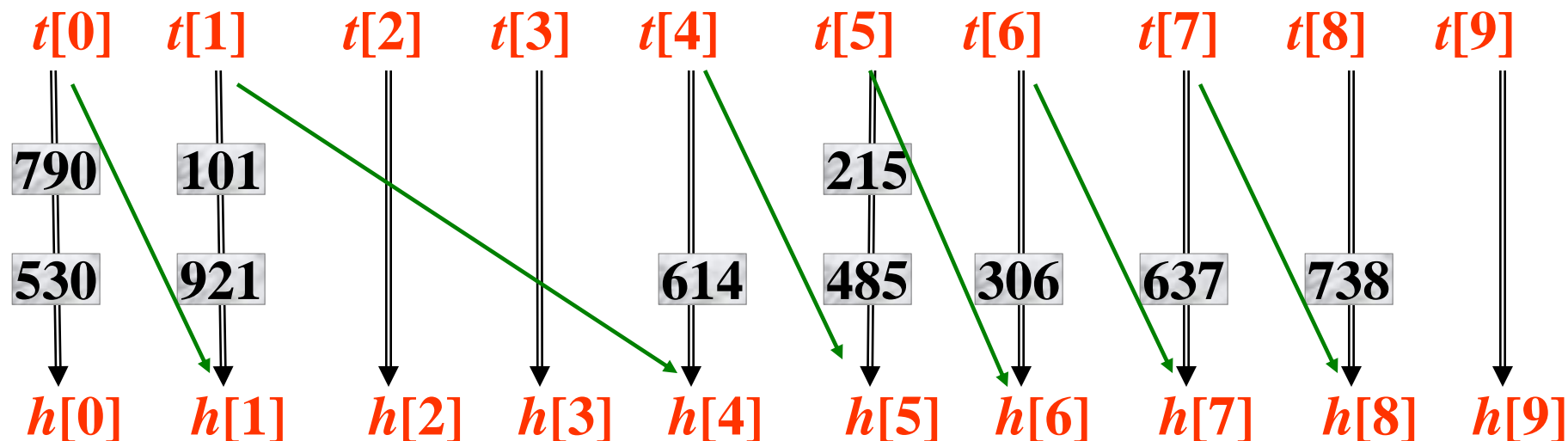
- 将分配出来的子序列存在 $r$ 个(静态链组织的)队列中
- 静态链式存储避免了在分配和收集时都需要移动所有记录，导致时间代价也高

# 举例

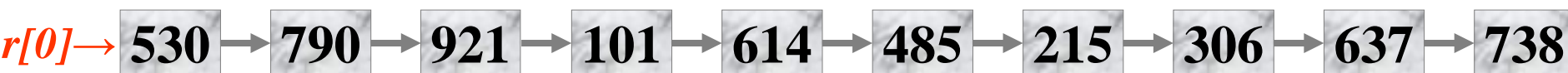
原始序列静态链表:



第一趟分配 (从最低位  $i = 3$  开始排序,  $h[ ]$  是队首指针,  $t[ ]$  为队尾指针)



第一趟收集 (让队尾指针  $t[i]$  链接到下一非空队首指针  $h[i+1]$  即可)

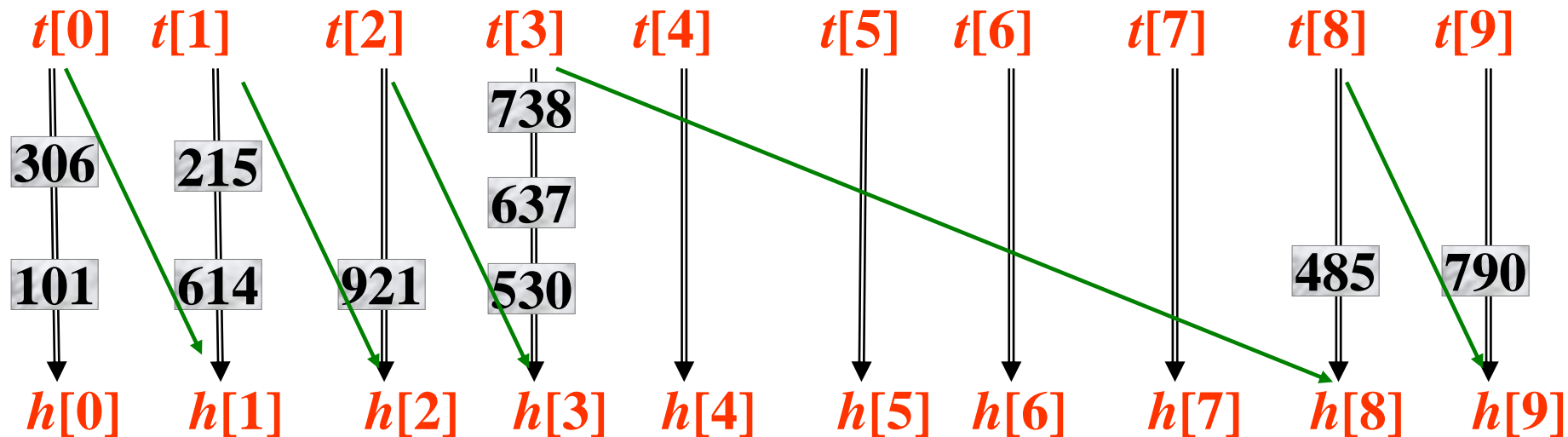




## 第一趟收集的结果:

$r[0] \rightarrow 530 \rightarrow 790 \rightarrow 921 \rightarrow 101 \rightarrow 614 \rightarrow 485 \rightarrow 215 \rightarrow 306 \rightarrow 637 \rightarrow 738$

## 第二趟分配 (按次低位 $i = 2$ )



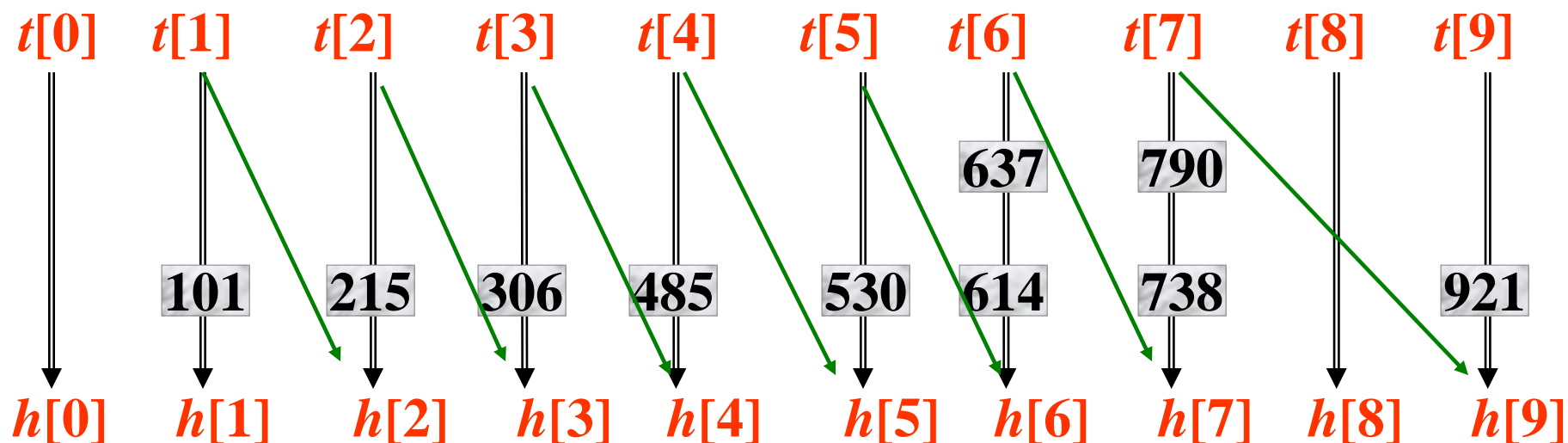
## 第二趟收集 (让队尾指针 $t[i]$ 链接到下一非空队首指针 $h[i+1]$ )

$r[0] \rightarrow 101 \rightarrow 306 \rightarrow 614 \rightarrow 215 \rightarrow 921 \rightarrow 530 \rightarrow 637 \rightarrow 738 \rightarrow 485 \rightarrow 790$

## 第二趟收集的结果:

$r[0] \rightarrow$  101  $\rightarrow$  306  $\rightarrow$  614  $\rightarrow$  215  $\rightarrow$  921  $\rightarrow$  530  $\rightarrow$  637  $\rightarrow$  738  $\rightarrow$  485  $\rightarrow$  790

## 第三趟分配 (按最高位 $i = 1$ )



## 第三趟收集 (让队尾指针 $t[i]$ 链接到下一非空队首指针 $h[i+1]$ )

$r[0] \rightarrow$  101  $\rightarrow$  215  $\rightarrow$  306  $\rightarrow$  485  $\rightarrow$  530  $\rightarrow$  614  $\rightarrow$  637  $\rightarrow$  738  $\rightarrow$  790  $\rightarrow$  921

排序结束!

# 静态队列定义

```
class Node{//结点类
```

```
public:
```

```
    int key; //结点的**关键码值
```

```
    int next;//后继结点的下标
```

```
};
```

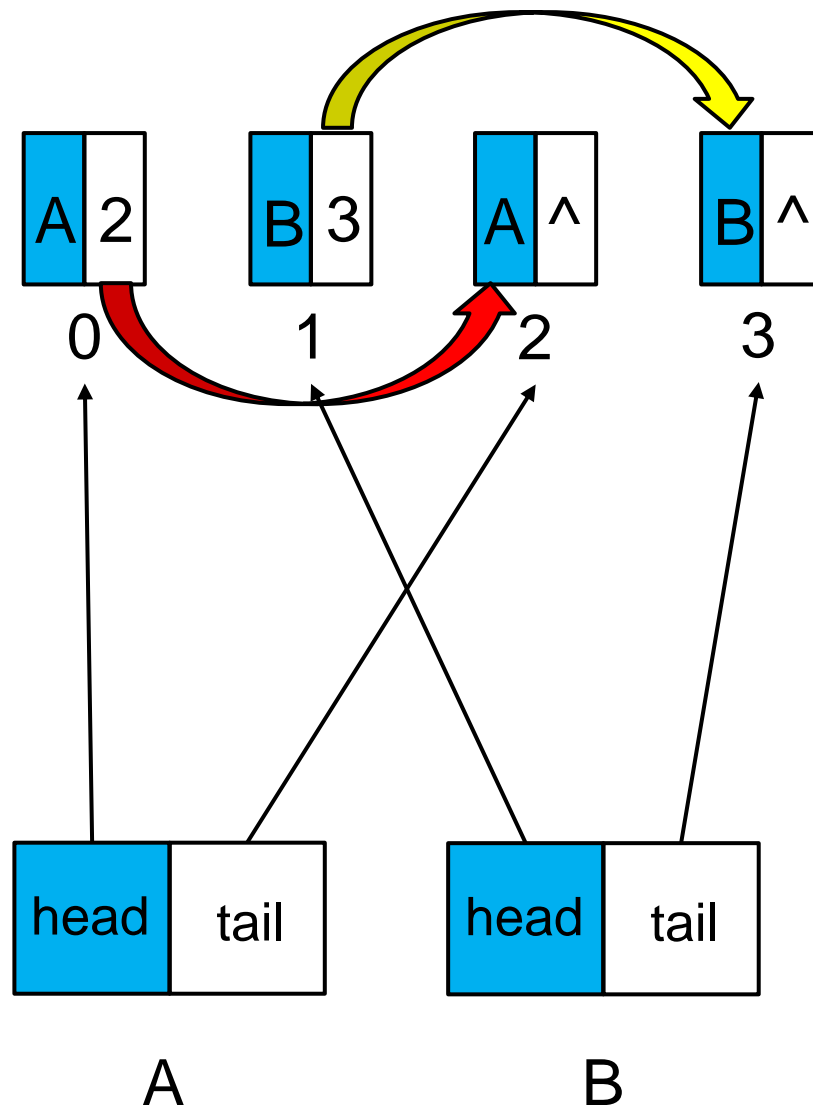
```
class StaticQueue{//静态队列类
```

```
public:
```

```
    int head; //头指针
```

```
    int tail;  //尾指针
```

```
};
```



# 基数排序算法

```
void RadixSort(Record *Array, int n, int d, int r) {
```

```
    int i, first = 0;           // first指向静态链中第一个记录
```

```
    //存放r个桶的静态队列
```

```
    StaticQueue *queue = new StaticQueue[r];
```

```
    //建链，初始为next域指向下一个记录
```

```
    for (i = 0; i < n-1; i++)
```

```
        Array[i].next = i + 1;
```

```
    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;**

**}**

# 分配过程

```
void Distribute(Record* Array,int first,int i,int r, StaticQueue*  
queue){
```

//first为静态链中的第一个记录，i为第i位排序码，r为基数

```
for (int j=0; j<r; j++)           //初始化r个队列
```

```
    queue[j].head=-1;
```

```
while (first != -1) {             //对整个静态链进行分配
```

```
    int k=Array[first].key;        //取第i位排序码数字
```

```
    for (int a=0;a<i;a++)
```

```
        k= k/r;
```

```
    k=k%r;
```



```
if (queue[k].head == -1)
```

```
    queue[k].head = first;
```

```
else    //否则加到子序列的尾部
```

```
    Array[queue[k].tail].next = first;
```

```
queue[k].tail = first; //first为子序列的尾部
```

```
first = Array[first].next;    //继续分配下一个记录
```

```
}
```

```
}
```

# 收集过程

```
Void Collect(Record* Array, int& first, int i, int r, StaticQueue*  
queue) {  
    int last, k=0;  
  
    // 找到第一个非空队列  
    while (queue[k].head == -1) k++;  
  
    first = queue[k].head;  
  
    last = queue[k].tail;  
  
    while (k<r-1)    { // 继续收集下一个非空队列  
        k++;  
    }
```





**//找到下一个非空队列**

**while (k<r-1 && queue[k].head== -1)     k++;**

**if (queue[k].head!= -1) {**//将非空序列连接起来****

**Array[last].next = queue[k].head;**

**last = queue[k].tail; **//尾部记录****

**}**

**}**

**Array[last].next = -1;     **//收集完毕****

**}**

# 算法分析

## ➤ 空间代价

- ➡  $n$ 个记录指针空间
- ➡  $r$ 个子序列的头尾指针
- ➡  $O(n + r)$

## ➤ 时间代价

- ➡ 不需移动记录，只需修改next指针
- ➡  $O(d \cdot (n+r))$

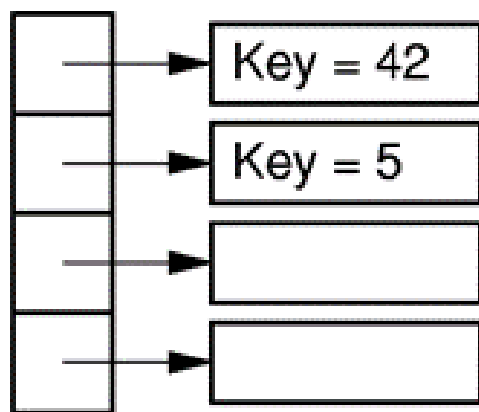
## ➤ 时间代价 $O(d \cdot n)$ ，线性复杂性吗？

- ➡ 实际上还是 $O(n \log n)$
- ➡ 没有重复编码的情况，需要 $n$ 个不同的编码来处理他们
- ➡ 也就是说， $d \geq \log_r^n$  即 $O(n \log n)$

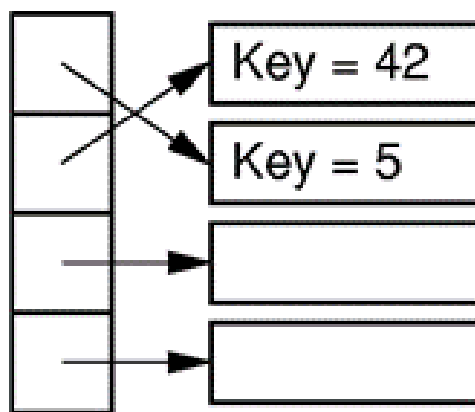


# 3、索引排序

- 记录自身规模较大时，减少记录移动次数能有效降低排序时间
- “索引排序”，或称“地址排序”
  - ➡ 可让数组中的每一个单元存储指向对应记录的地址，需要移动记录时，只移动地址（或索引），而不移动记录本身



Array



SortedArray

# 索引形式：两个哲学问题

➤ Index1: “我到哪里去?” 对Array而言

➡ IndexArray1[i]存放的是Array[i]中元素在排序后序列中的位置，即：  $\text{SortedArray}[\text{Index1}[i]] == \text{Array}[i]$

➤ Index2: “我从哪里来?” 对SortedArray而言

➡ IndexArray2[i]存放SortedArray[i]中元素在排序前序列中的位置，即：  $\text{SortedArray}[i] == \text{Array}[\text{Index2}[i]]$

原数组下标i	0	1	2	3	4	5	6	7
记录数组Array[i]的排序码	29	25	34	64	34'	12	32	45
索引1下标Index1[i]	2	1	4	7	5	0	3	6
索引2下标Index2[i]	5	1	0	6	2	4	7	3
结果数组SortedArray[i]排序码	12	25	29	32	34	34'	45	64

# 索引排序

- **面临的问题**：如何根据这两类索引，完成记录排序？
- 两种索引排序类似，**只讨论Index2**
- **两个关键问题**
  - I. **索引排序：Index Sort**
  - II. **记录调整：Adjust Record**

# 索引排序示例

		0	1	2	3	4	5	6	7
temp	Array	2	2	3	6	3	1	3	4
		9	5	4	4	4	2	2	5
0	Index2	5	1	0	6	2	4	7	3
1	SortedArray	1	2	2	3	3	3	4	6
		2	5	9	2	4	4	5	4

**Index数据和SortedArray可以共享!**

# 排序过程分析

- 前述索引排序过程中包含3个环 (Cycle)
  - ➡ 29-12-34-34: length 4
  - ➡ 25: length 1
  - ➡ 64-32-45: length 3
- 每个环中拷贝操作的数目为  $(length+1)$

# 索引排序算法

```
template<class Record>
```

```
void IndexSort(Record Array[], int IndexArray[], int n) {
```

```
    int i,j;
```

```
    for (i = 0; i < n; i++) IndexArray[i] = i;    // 初始化索引下标
```

```
    // 基于插入排序的索引排序过程
```

```
    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]])
```

```
                swap(IndexArray, j, j-1);
```

```
            else break;                            // 此时i前面记录已排序
```

```
    AdjustRecord(Array, IndexArray, n);
```

```
}
```



# 记录物理调整

```
template<class Record>
```

```
void AdjustRecord(Record Array[], int IndexArray[], int n) {
```

```
    Record TempRec;                // 临时空间
```

```
    int i, j;
```

```
    for (i = 0; i < n; i++) {
```

```
        j = i;                // j是循环链中的当前元素
```

```
        TempRec = Array[i];    // 暂存i下标中目前的记录
```

```
        while (IndexArray[j] != i) {    // 如下标不是i则顺链调整
```

```
            int k = IndexArray[j];    // k为链接j指向的下标
```

```
            Array[j] = Array[k];    // 把k下标中值复制j
```

```
    IndexArray[j] = j; // 因为是正确归位，索引j就是自身
    j = k;             // j换到循环链中的下一个，继续处理
}

Array[j]=TempRec;    // 第i大元素正确入位
IndexArray[j]=j;     // 因为是正确归位，索引j就是自身
}

}
```

在调整算法中， $i$ 下标是无回溯的for循环，while循环中的调整处理也都一次到位，每个元素最多参与一轮调整。因此，整个调整算法的时间代价为 $O(n)$ 。空间代价显然为 $O(1)$ 。

## 8.7 各种排序算法的理论和实验时间代价

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	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	$\Theta(1)$	稳定
二分法插入排序	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(1)$	稳定
冒泡排序	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(1)$	稳定
改进的冒泡排序	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	$\Theta(1)$	稳定
选择排序	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(1)$	不稳定
Shell排序(3)	$\Theta(n^{3/2})$	$\Theta(n^{3/2})$	$\Theta(n^{3/2})$	$\Theta(1)$	不稳定
快速排序	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(\log n)$	不稳定
归并排序	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n)$	稳定
堆排序	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(1)$	不稳定
桶式排序	$\Theta(n+m)$	$\Theta(n+m)$	$\Theta(n+m)$	$\Theta(n+m)$	稳定
基数排序	$\Theta(d \cdot (n+r))$	$\Theta(d \cdot (n+r))$	$\Theta(d \cdot (n+r))$	$\Theta(n+r)$	稳定



<u>数组规模</u>	<u>100</u>	<u>10K</u>	<u>1M</u>	<u>10K正序</u>	<u>10K逆序</u>
直接插入排序	0.000175	1.7266	_____	0.00031	3.44187
优化插入排序	0.000092	0.8847	_____	0.00031	1.76157
二分插入排序	0.000036	0.1434	_____	0.00453	0.28297
冒泡排序	0.000271	2.7844	_____	1.74641	3.47484
优化冒泡排序	0.000269	2.7848	_____	0.00032	3.44063
选择排序	0.000180	1.7199	_____	1.72516	1.72812
Shell排序(2)	0.000072	0.0166	4.578	0.00484	0.00781
Shell排序(3)	0.000055	0.0153	4.125	0.00121	0.00687

<u>数组规模</u>	<u>100</u>	<u>10K</u>	<u>1M</u>	<u>10K正序</u>	<u>10K逆序</u>
快速排序	0.000042	0.0068	1.001	0.00547	0.00547
优化快速排序	0.000031	0.0057	0.856	0.00408	0.00409
归并排序	0.000045	0.0070	1.105	0.00625	0.00546
优化归并排序	0.000033	0.0057	0.934	0.00531	0.00531
堆排序	0.000042	0.0075	1.562	0.00672	0.00688
基数排序/2	0.000294	0.0294	3.031	0.02937	0.02922
基数排序/4	0.000145	0.0147	1.515	0.01469	0.01469
基数排序/8	0.000095	0.0092	0.953	0.00922	0.00921

# 8.8 排序问题的界

## ➤ Lower Bound

➡ 解决排序问题能达到的最佳效率，即使尚未设计出算法

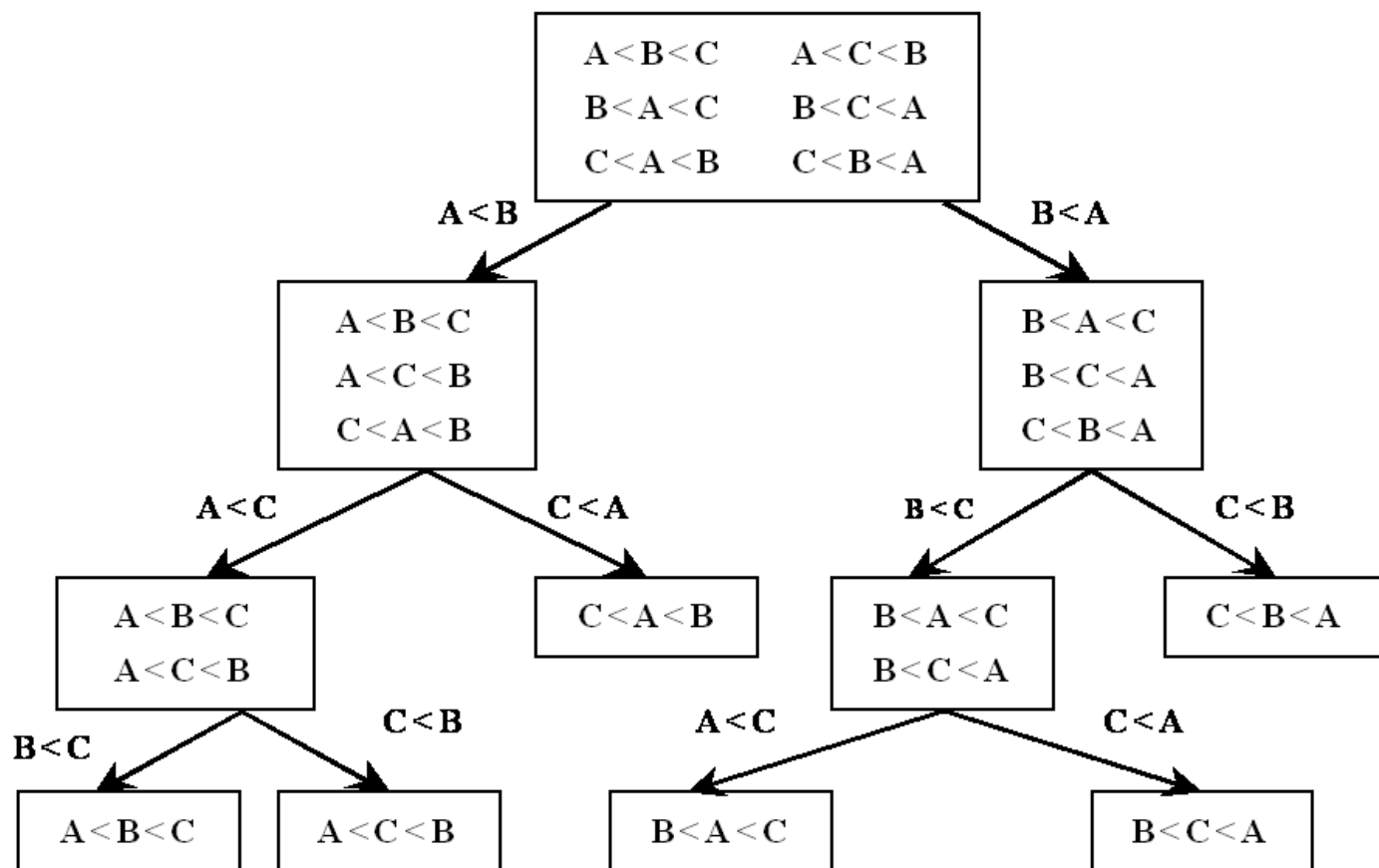
## ➤ Upper Bound

➡ 指已知最快算法所达到的最佳渐进效率

➤ 排序问题的下限应该在 $\Omega(n)$ 到 $O(n \log n)$  之间

# 判定树模拟基于比较的排序

- ▶ 判定树最大深度是排序算法在最差情况下需要的比较次数
- ▶ 最小深度就是最佳情况下的最小比较次数



# 基于比较的排序的下限

- 对 $n$ 个记录，共有 $n!$ 个叶结点
- 判定树的深度为 $\log(n!)$
- 最差情况下需要 $\log(n!)$ 次比较，即 $\Omega(n \cdot \log n)$
- 最差情况下任何基于比较的排序算法都需要 $\Omega(n \log n)$ 次比较





# 再见…

---

## 联系信息:

**电子邮件:** [gjsong@pku.edu.cn](mailto:gjsong@pku.edu.cn)

**电 话:** 62754785

**办公地点:** 理科2号楼2307室