

# 第10章 内部排序

- 
- 概述
  - 插入排序
  - 交换排序
  - 选择排序
  - 归并排序
  - 小结

# 概 述

**排序**：将一组杂乱无章的记录按一定的规律顺次排列起来。

**关键字**(*key*)：通常数据记录有多个属性域，即多个数据成员组成，其中有一个属性域可用来区分记录，作为排序依据。该域即为关键字。

**主关键字**: 如果在待排序记录序列中各个记录的关键字互不相同, 这种关键字即主关键字。按照主关键字进行排序, 排序的结果是唯一的。

**次关键字**: 待排序记录序列中有些记录的关键字可能相同, 这种关键字称为次关键字。按照次关键字进行排序, 排序的结果可能不唯一。

**排序算法的稳定性**: 如果在记录序列中有两个记录 $R_{[i]}$ 和 $R_{[j]}$ , 它们的关键字  $key_{[i]} == key_{[j]}$ , 且在排序之前, 记录 $R_{[i]}$ 排在 $R_{[j]}$ 前面。如果在排序之后, 记录 $R_{[i]}$ 仍在记录 $R_{[j]}$ 的前面, 则称这个排序方法是稳定的, 否则称这个排序方法是不稳定的。

**内排序与外排序：** 内排序是指在排序期间数据记录全部存放在内存的排序；外排序是指在排序期间全部记录个数太多，不能同时存放在内存，必须根据排序过程的要求，不断在内、外存之间移动的排序。

**排序的时间开销：** 排序的时间开销是衡量算法好坏的最重要的标志。排序的时间开销可用算法执行中的记录比较次数与记录移动次数来衡量。各节给出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。对于那些受记录关键字序列初始排列及记录个数影响较大的，需要按最好情况和最坏情况进行估算。

**算法执行时所需的附加存储：** 评价算法好坏的另一标准。

- 待排序记录的存储方式:
  - 以一维数组作为存储结构，排序时必须实际移动记录；
  - 以链表(动态链表或静态链表)作为存储结构，排序时不用物理移动记录，只需修改指针；
  - 有的排序方法难于在链表上实现，且需要避免排序过程中的记录移动，可将待排序记录本身存储在一组地址连续的存储单元内，同时附设一个指示记录存储位置的地址向量，排序过程中不移动记录，而移动地址向量中这些记录的地址。

## 待排序记录的数据类型说明:

```
struct Record{  
    int key;  
    ...;  
};
```

```
Record r[n];
```

# 第十章 内部排序

- 概述
- 插入排序
- 交换排序
- 选择排序
- 归并排序
- 小结



# 插入排序 (Insert Sorting)

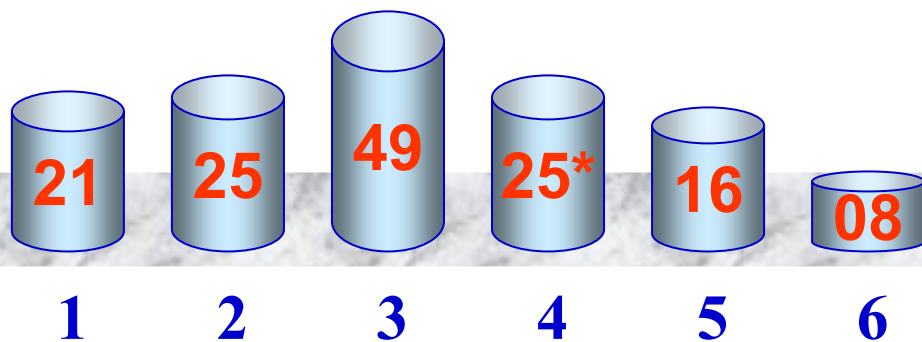
**基本方法：**每步将一个待排序的记录，按其关键字大小，插入到前面已经排好序的一组记录的适当位置上，直到记录全部插入为止。

## 直接插入排序 (Insert Sort)

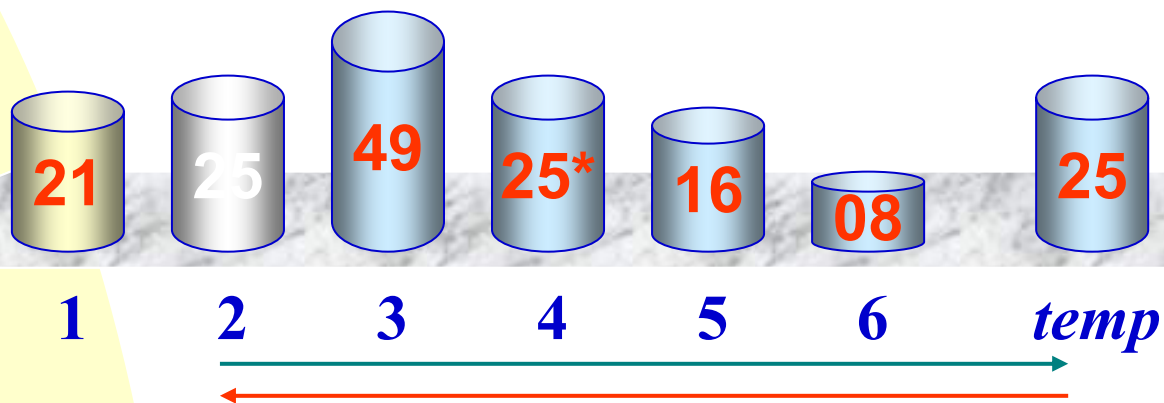
直接插入排序的基本思想是：当插入第 $i$  ( $i \geq 1$ ) 个记录时，前面的 $R[1], R[2], \dots, R[i-1]$ 已经排好序。这时，用 $R[i]$ 的关键字与 $R[i-1], R[i-2], \dots$ 的关键字顺序进行比较，找到插入位置即将 $R[i]$ 插入，原来位置上的记录向后顺移。



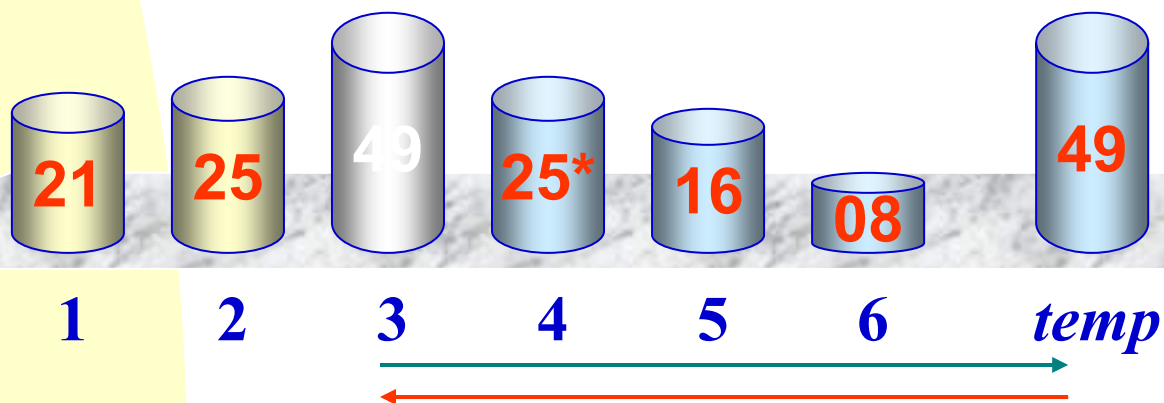
各趟排序结果



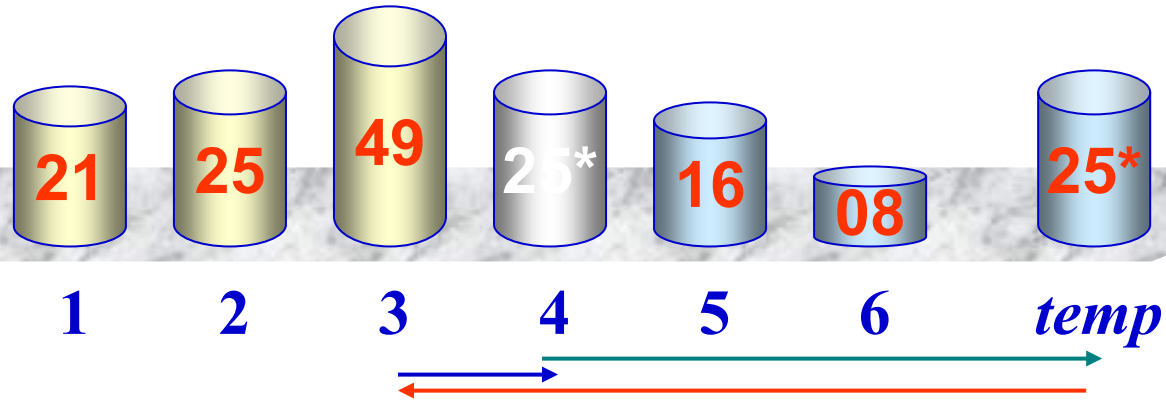
$i = 2$



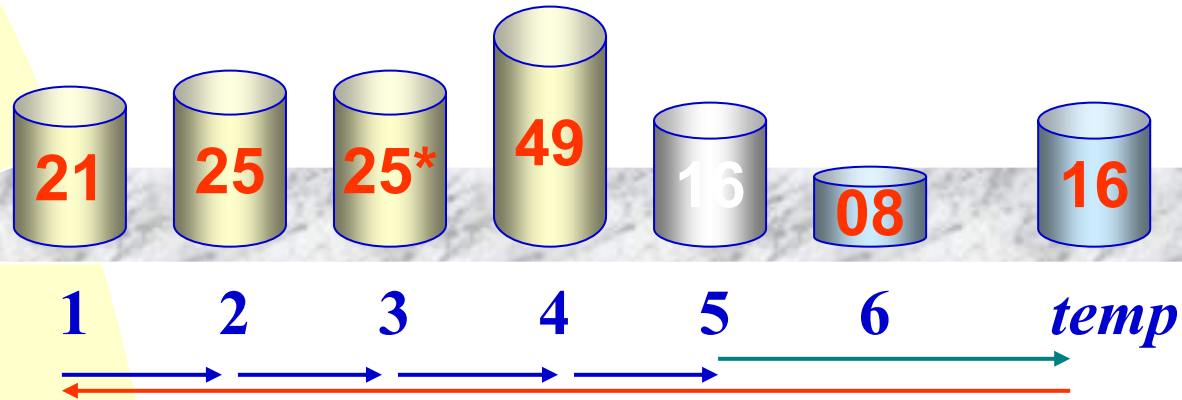
$i = 3$



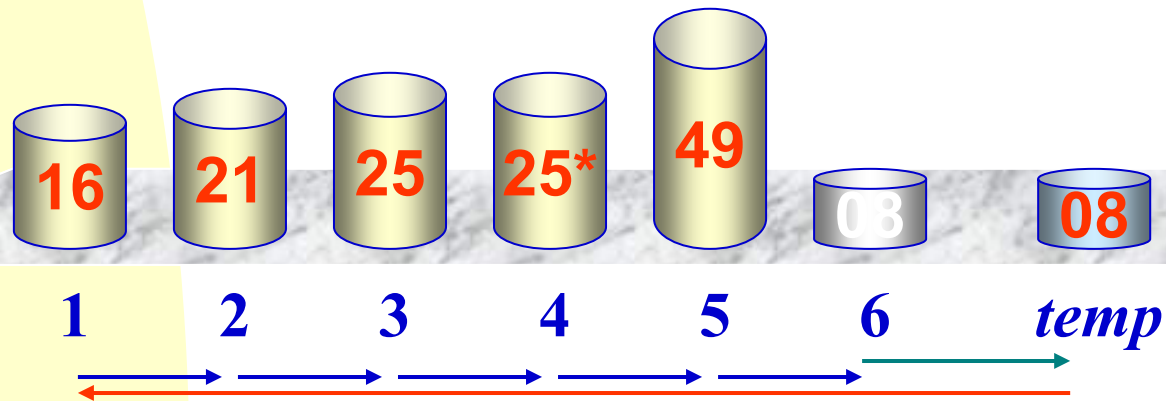
$i = 4$



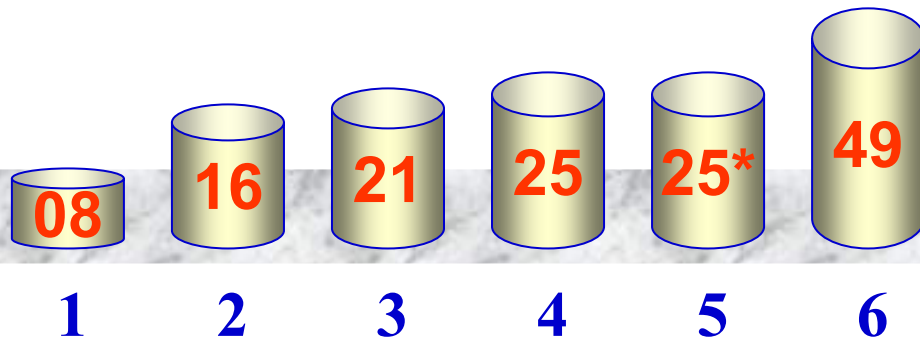
$i = 5$



$i = 6$

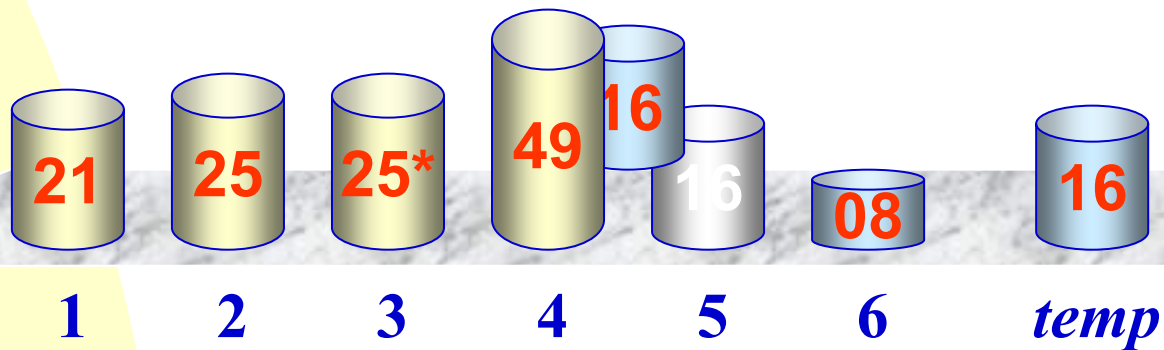


完成

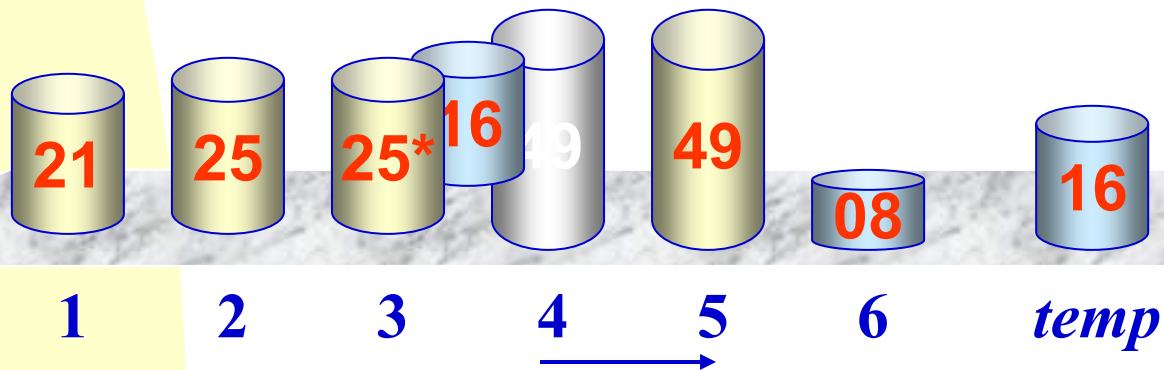


$i = 5$  时的排序过程

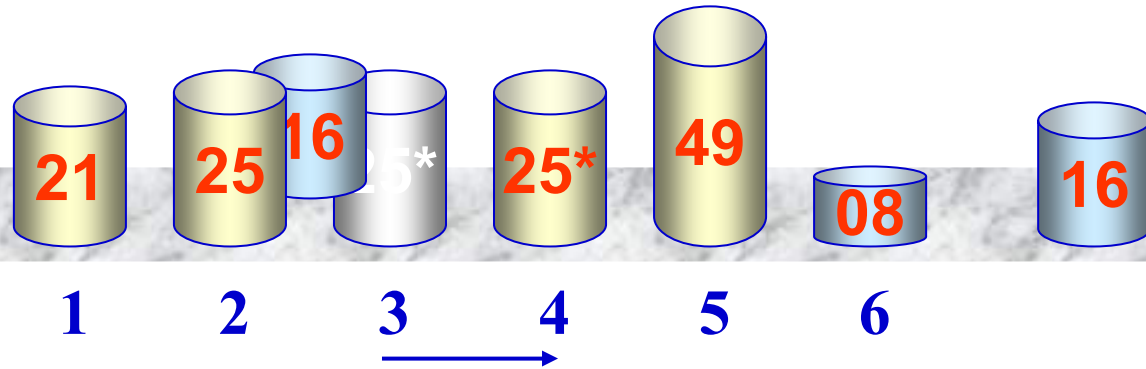
$i = 5$   
 $j = 4$



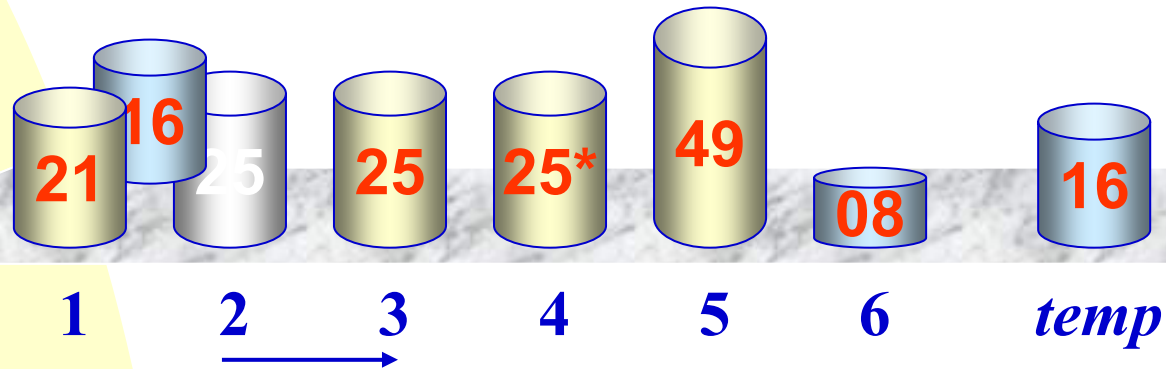
$i = 5$   
 $j = 3$



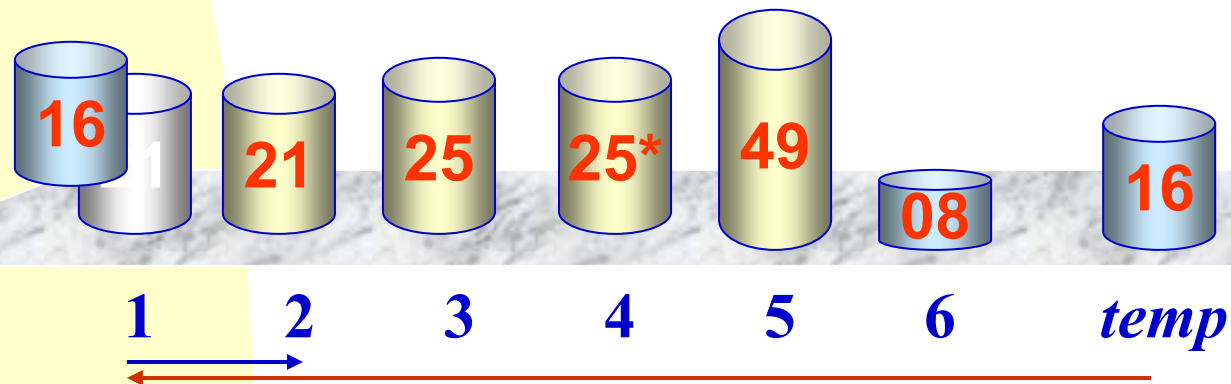
$i = 5$   
 $j = 2$



$i = 5$   
 $j = 1$



$i = 5$   
 $j = 0$



## 直接插入排序的算法：

```
void InsertSort( Record r[], int n )
{
    for( i=1; i<n; ++i )
    {
        temp=r[i];
        for(j=i-1; j>=0 && temp.key<r[j].key; --j )
            r[j+1]=r[j];    // 记录后移
        r[j+1]=temp;
    }
}
```



## 算法分析:

- 直接插入排序的算法简洁，容易实现。从时间来看，排序的基本操作为：比较两个记录的大小和移动记录。其中：
  - ◆ 最小比较次数： $C_{\min} = n-1 = O(n)$
  - ◆ 最大比较次数： $C_{\max} = (1+2+\dots+n-1) = n(n-1)/2 = O(n^2)$
  - ◆ 最小移动次数： $M_{\min} = 2(n-1) = O(n)$
  - ◆ 最大移动次数： $M_{\max} = (2+1 + 3+1 + \dots + n+1) = O(n^2)$

平均情况？

在平均情况下的关键字比较次数和记录移动次数约为  $n^2/4$ 。

因此，直接插入排序的时间复杂度为  $O(n^2)$ 。

- 关键字比较次数和记录移动次数与记录关键字的初始排列有关。
- 直接插入排序是一种稳定的排序方法。
- 从空间看，直接插入排序算法只需一个记录的辅助空间。

# 折半插入排序 (Binary Insertsort)

**基本思想:**

**如何提高每次插入操作的效率?**

$(R[1], R[2], \dots, R[i-1]) R[i]$

**前 $i-1$ 个记录已经排好序!**

**在插入  $R[i]$  时, 利用折半搜索法寻找  $R[i]$  的插入位置。**



## ■ 折半插入排序的算法:

```
void BInsertSort( Record r[], int n) {  
    for( i=1; i<n; ++i ){  
        temp=r[i]; // 将L.r[i]暂存到L.r[0]  
        low=0; high=i-1;  
        while( low <= high ){ // 在r[low..high]中折半查找有序插入的位置  
            mid = (low+high)/2; // 折半  
            if ( temp.key < r[mid].key )  
                high = mid-1; // 插入点在低半区  
            else  
                low = mid+1; // 插入点在高半区  
        }  
        for( j=i-1; j>=low; --j ) r[j+1]=r[j]; // 记录后移  
        r[low]=temp; // 插入  
    }  
}
```

## 算法分析：

折半插入排序所需要的关键字比较次数与待排序记录序列的初始排列的关系？

无关，关键字比较次数仅依赖于记录个数。

在插入第  $i$  个记录时，需要经过  $\lfloor \log_2 i \rfloor + 1$  次关键字比较，才能确定它应插入的位置。将  $n$  个记录用折半插入排序所进行的关键字比较次数为  $O(n \log_2 n)$ 。

当  $n$  较大时，总的关键字比较次数比直接插入排序的最坏情况要好得多，但比其最好情况要差。

折半插入排序的**记录移动次数**与直接插入排序相同，依赖于记录的初始排列。

折半插入排序的时间复杂度仍为 $O(n^2)$

折半插入排序是一个稳定的排序方法。

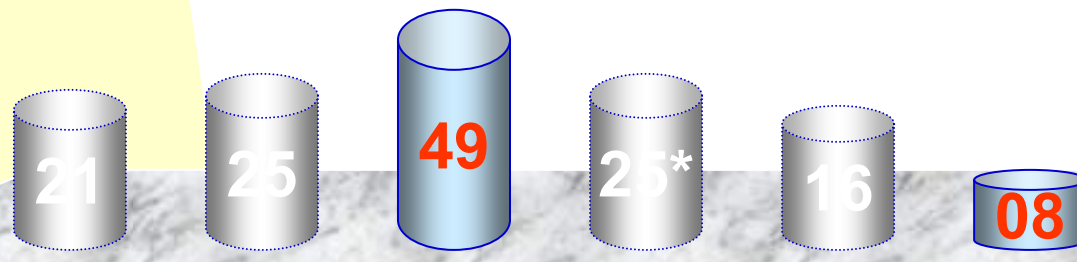
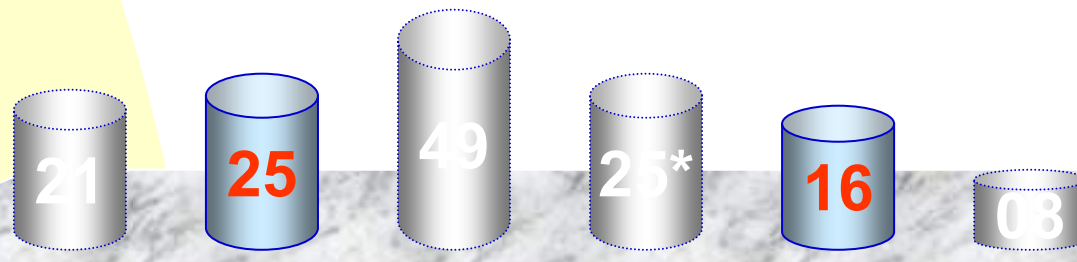
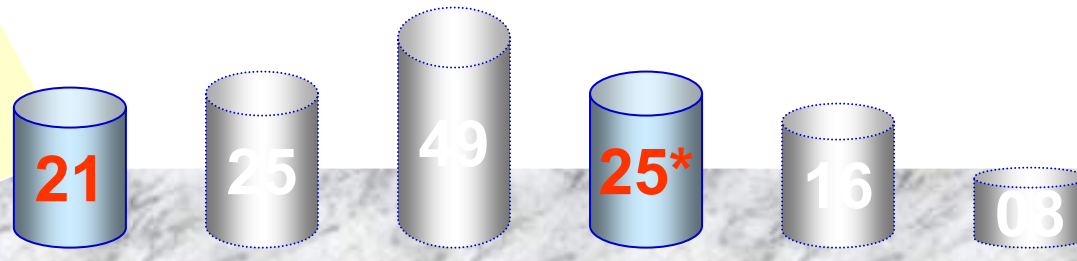
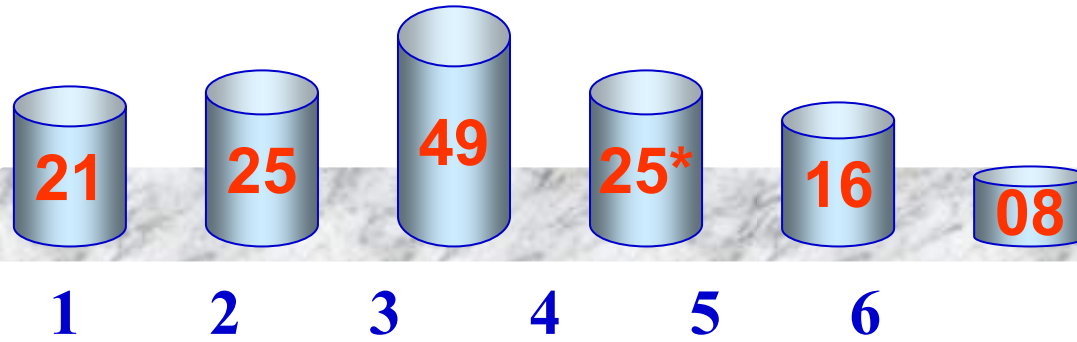
# 希尔排序 (Shell Sort)

- 直接插入排序在以下两种情况下效率较高：
  - 当待排序记录处于“基本有序”状态
  - 当待排序记录个数 $n$ 值较小时
- Shell 排序方法正是基于这两点考虑对直接插入排序加以改进而提出的一种插入排序算法。
- Shell 排序方法的基本思想？
  - 将待排序记录通过引入间隔(增量)分成若干个组（或称子序列），然后在每一组内进行直接插入排序；
  - 通过采用一种“逐渐缩小插入增量”的方法，逐步增长组的长度。

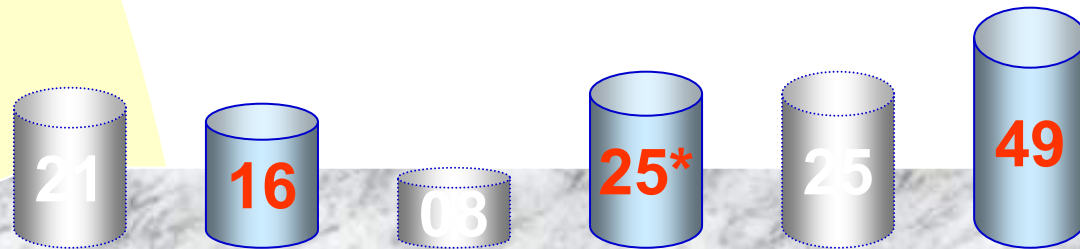
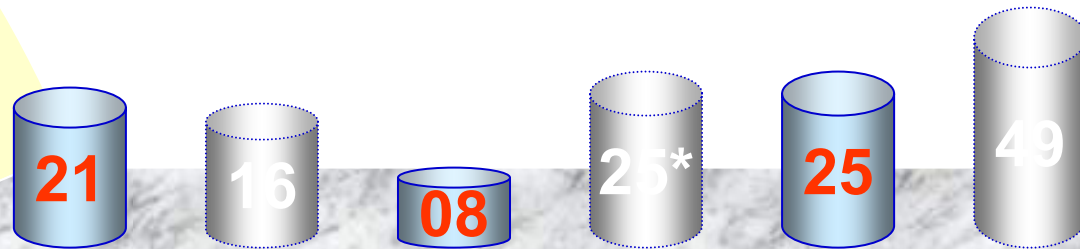
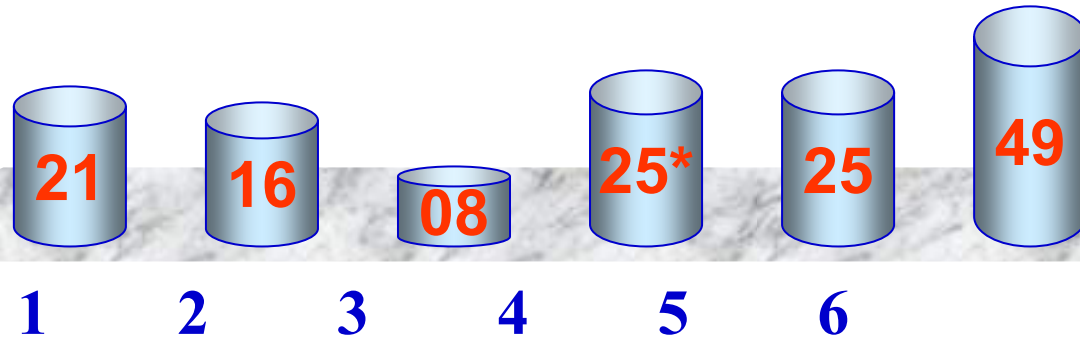
## Shell 排序的基本做法:

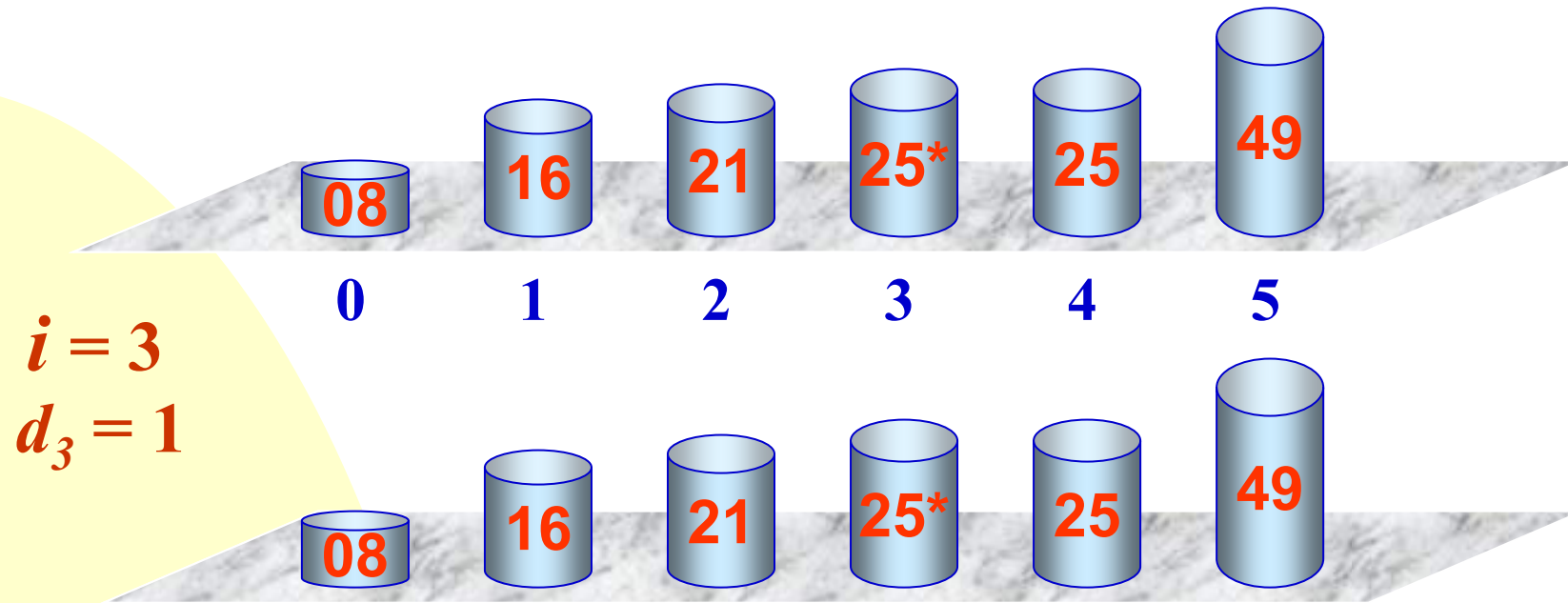
- 先确定一个小于 $n$  的整数 $d_1$ 作为第一个增量, 把记录序列分为 $d_1$ 个组, 所有距离为 $d_1$ 倍数的记录放在同一个组中, 在各组内进行直接插入排序;
- 然后, 取第二个增量 $d_2$  (  $d_2 < d_1$  ), 重复上述分组和排序, 直至所取的增量 $d_t=1$  (  $d_t < d_{t-1} < \dots < d_2 < d_1$  ), 即所有记录处在同一组中进行直接插入排序为止。

$i = 1$   
 $d_1 = 3$



$i = 2$   
 $d_2 = 2$





开始时 **增量**值较大，子序列中的**记录较少**，排序速度较快；

随着排序进展，增量值逐渐变小，子序列中记录个数逐渐变多，但由于前面工作的基础，大多数记录已**基本有序**，所以排序速度仍然很快。



## 希尔排序的算法:

```
void ShellSort(Record r[], int n, int dlta[], int t){  
    // 按增量序列dlta[0..t-1]作希尔排序  
    for(k=0; k<t; k++)  
        ShellInsert( r, n, dlta[k] ); // 一趟增量为dlta[k]的插入排序  
}  
  
void ShellInsert( Record r[], int n, int dk ){  
    for( i=dk; i<n; i++ ){ //需将r[i]插入有序增量子表  
        temp = r[i];  
        for( j=i-dk; j>=0 && temp.key<r[j].key; j-=dk)  
            r[j+dk] = r[j]; // 记录后移, 查找插入位置  
        r[j+dk] = temp;  
    }  
}
```

**增量的取法有多种。**

最初 shell 提出取  $d_1 = \lfloor n/2 \rfloor$ ,  $d_2 = \lfloor d_1/2 \rfloor$ , 直到  $d_t = 1$ 。

后来 knuth 提出取  $d_{i+1} = \lfloor d_i/3 \rfloor + 1$ 。

还有人提出都取奇数为好, 也有人提出各**增量互质**为好。

## **算法分析:**

对特定的待排序记录序列, 可以准确地估算关键字的比较次数和记录移动次数。

但想要弄清关键字比较次数和记录移动次数与**增量选择**之间的依赖关系, 并给出完整的数学分析, 还没有人能够做到。

Knuth利用大量的实验统计资料得出，当  $n$  很大时，关键字平均比较次数和记录平均移动次数大约在  $n^{1.25}$  到  $1.6n^{1.25}$  的范围内。

这是在利用直接插入排序作为子序列排序方法的情况下得到的。

# 第十章 内部排序

- 概述
- 插入排序
- 交换排序
- 选择排序
- 归并排序
- 小结

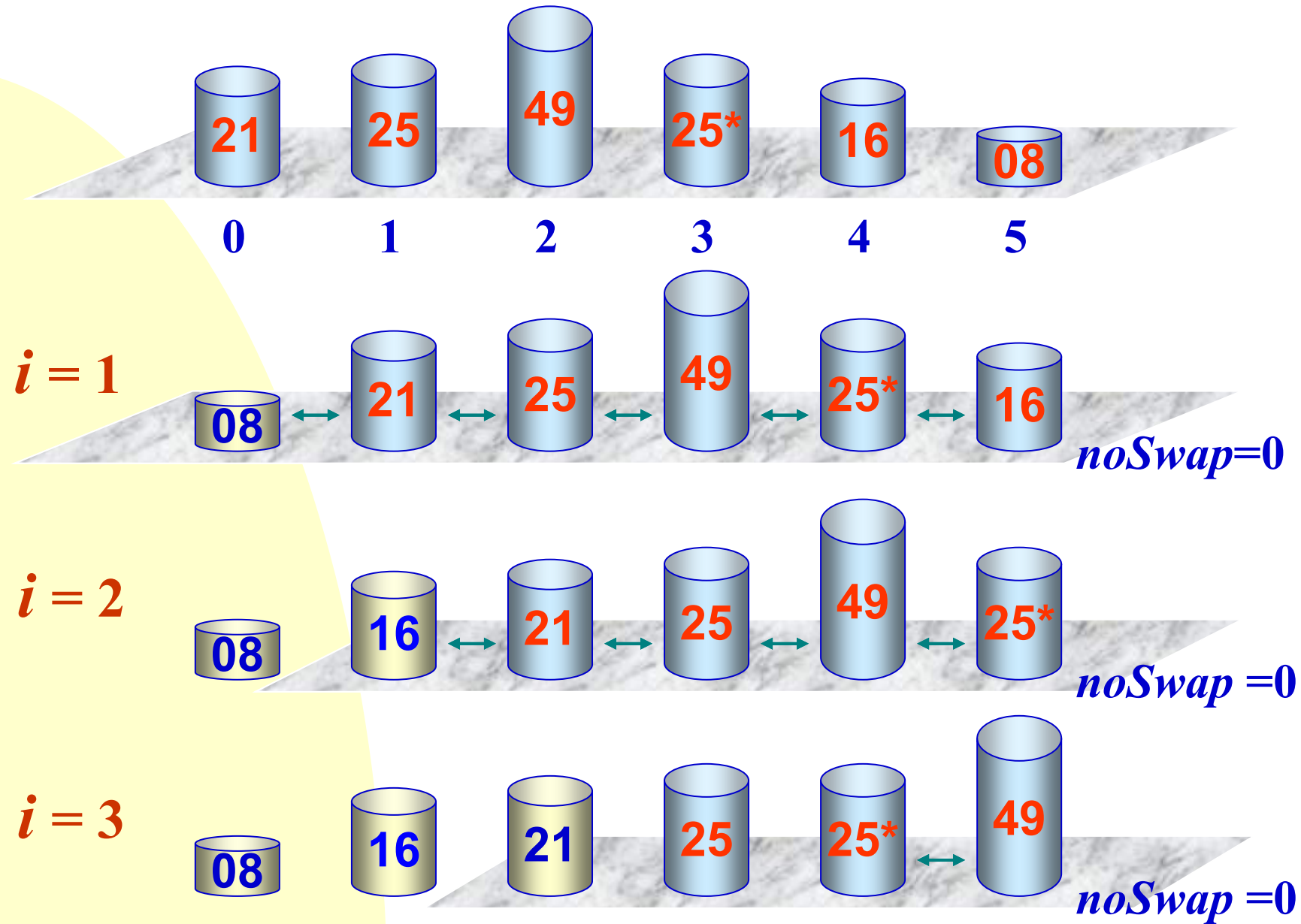


# 交换排序 ( Exchange Sort )

**基本思想：** 是两两比较待排序记录的关键字，如果发生逆序(即排列顺序与排序后的次序正好相反)，则交换之，直到所有记录都排好序为止。

## 起泡排序 (Bubble Sort)

**基本方法：** 比较相邻两个记录的关键字，若  $R[i].key > R[i+1].key$ ，则交换之，其中  $i$  从 0 到  $n-pass-1$  ( $pass$  的初值为 1) 称之为一趟起泡排序，其结果是使最大关键字的记录被交换到  $n-pass$  的位置上，如果某一趟起泡排序过程中没有进行一次记录交换，则排序过程结束。最坏情况下需  $n-1$  趟排序。



$i = 4$

08

0

16

1

21

2

25

3

25\*

4

49

5

*noSwap* = 1

# 起泡排序的算法:

```
void BubbleSort( Record r[], int n ){  
    // 从下往上扫描的起泡排序  
    for( i=0; i<n-1; i++ ){    // 做n-1趟排序  
        noSwap=true;        // 置未交换标志  
        for( j=n-1; j>i; j-- ) // 从下往上扫描  
            if( r[j].key < r[j-1].key ){  
                temp=r[j-1]; // 交换记录  
                r[j-1]=r[j];  
                r[j]=temp;  
                noSwap=false;  
            }  
        if( noSwap ) break; // 本趟扫描未发生交换, 则终止算法  
    }  
}
```

还能做什么进一步改进吗?



## 算法分析:

在记录的初始状态为**有序**时，此算法只执行一趟起泡，做  $n-1$  次关键字比较，不移动记录。这是最好的情形。

最坏的情形是算法执行了  $n-1$  趟起泡，第  $i$  趟 ( $1 \leq i < n$ ) 做了  $n-i$  次关键字比较，执行了  $n-i$  次记录交换。这样在最坏情形下总的关键字比较次数  $KCN$  和记录移动次数  $RMN$  为:

$$KCN = \sum_{i=1}^{n-1} (n - i) = \frac{1}{2} n (n - 1)$$

$$RMN = 3 \sum_{i=1}^{n-1} (n - i) = \frac{3}{2} n (n - 1)$$

起泡排序需要一个附加记录以实现记录值的对换。

起泡排序是一个**稳定的**排序方法。

# 快速排序 (Quick Sort)

## 基本思想:

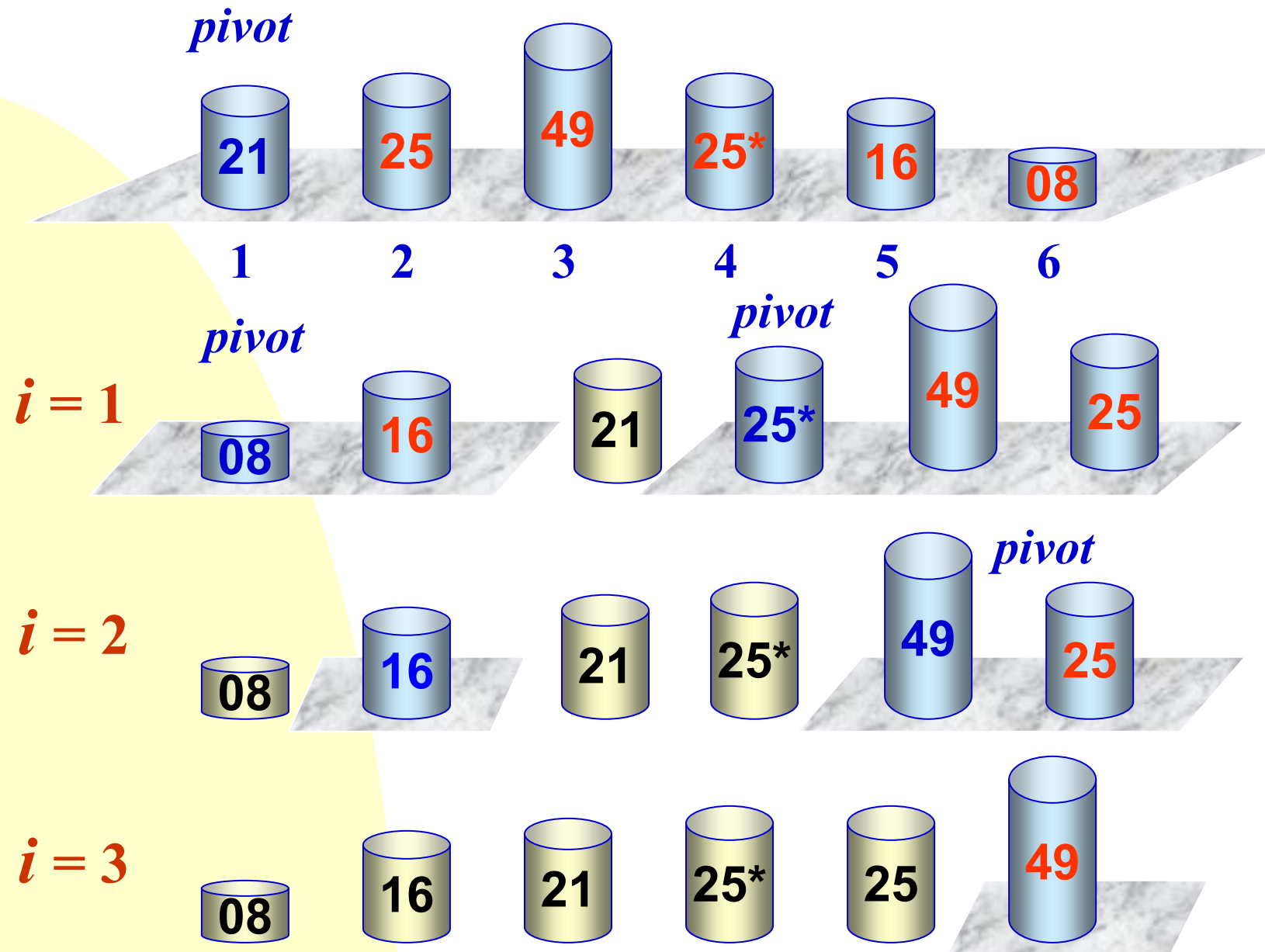
任取待排序记录序列中的某个记录（例如取第一个记录）作为**基准**，按照该记录的关键字大小，将整个记录序列划分为左右两个子序列：

左侧子序列中所有记录的关键字都小于（或等于）  
基准记录的关键字

右侧子序列中所有记录的关键字都大于（或等于）  
基准记录的关键字

基准记录则排在这两个子序列中间（这也是该记录最终应安放的位置）。

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



## 算法描述:

一趟划分算法:

```
int Partition( Record r[], int low, int high){  
    temp = r[low];    // 用子表的第一个记录作基准记录  
    while( low<high ){    // 从表的两端交替地向中间扫描  
        while(low<high && r[high].key>=temp.key) high--;  
        if(low<high) r[low++] = r[high];  
        while(low<high && r[low].key<= temp.key) low++;  
        if(low<high) r[high--] = r[low];  
    }  
    r[low] = temp;        //基准记录到位  
    return low;  
}
```

```
void QSort( Record r[], int low, int high)
{
    if( low < high ) // 长度大于1
    {
        pivotloc=Partition(r, low, high);
        QSort(r, low, pivotloc-1); // 对低子表递归排序
        QSort(r, pivotloc+1,high); // 对高子表递归排序
    }
}
```

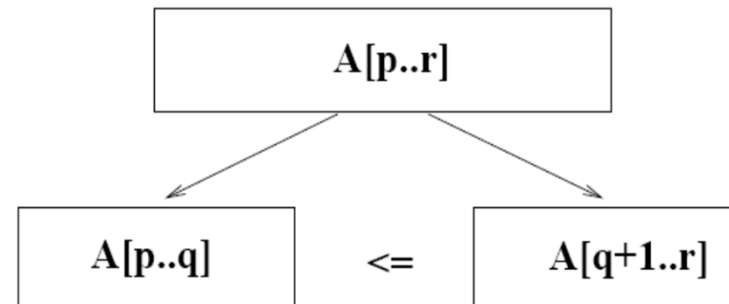
# 快速排序算法背后的思想?

分治法:

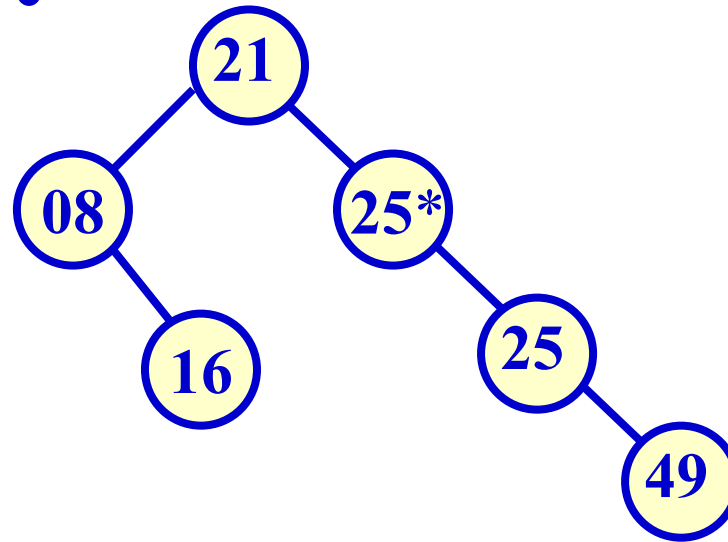
**Divide:**

**Conquer:**

**Combine:**



算法*quicksort*是一个递归的算法，其递归树如图所示。



✓ 算法*partition*利用序列第一个记录作为基准，将整个序列划分为左右两个子序列，将基准记录安放到位。

## 算法分析：

从快速排序算法的递归树可知，快速排序的趟数取决于递归树的深度。

最好的情况是？

如果每次划分对基准记录定位后，该记录的左侧子序列与右侧子序列的长度接近，则下一步将是对两个长度减半的子序列进行排序。

在 $n$ 个元素的序列中，对一个记录定位所需时间为 $O(n)$ 。若设 $T(n)$ 是对 $n$ 个元素的序列进行排序所需的时间，而且每次对一个记录正确定位后，正好把序列划分为长度相等的两个子序列，此时，总的计算时间为：



$$\begin{aligned} T(n) &\leq n + 2 T(n/2) \quad // \text{一趟划分比较次数为 } n-1 \text{ 次} \\ &\leq n + 2 ( n/2 + 2T(n/4) ) = 2n + 4T(n/4) \\ &\leq 2n + 4 ( n/4 + 2T(n/8) ) = 3n + 8T(n/8) \\ &\dots\dots\dots \\ &\leq n \log_2 n + nT(1) = O(n \log_2 n) \end{aligned}$$

可以证明，快速排序的平均计算时间也是  $O(n \log_2 n)$ 。

实验结果表明：就平均计算时间而言，快速排序是我们所讨论的所有内排序方法中最好的一个。

## 最坏的情况？

即待排序记录序列已经按其关键字从小到大排好序的情况下，其递归树成为单支树，每次划分只得到一个比上一次少一个记录的字序列。

这样，必须经过  $n-1$  趟才能把所有记录定位，而且第  $i$  趟需要经过  $n-i$  次关键字比较才能找到第  $i$  个记录的安放位置，总的关键字比较次数将达到：

$$\sum_{i=1}^{n-1} (n-i) = \frac{1}{2} n(n-1) \approx \frac{n^2}{2}$$



	0	1	2	3	4	5	<i>pivot</i>
初始	08	16	21	25	25*	49	08
$i = 1$	08	16	21	25	25*	49	16
$i = 2$	08	16	21	25	25*	49	21
$i = 3$	08	16	21	25	25*	49	25
$i = 4$	08	16	21	25	25*	49	25*
$i = 5$	08	16	21	25	25*	49	

用第一个记录作为基准记录  
快速排序退化的例子

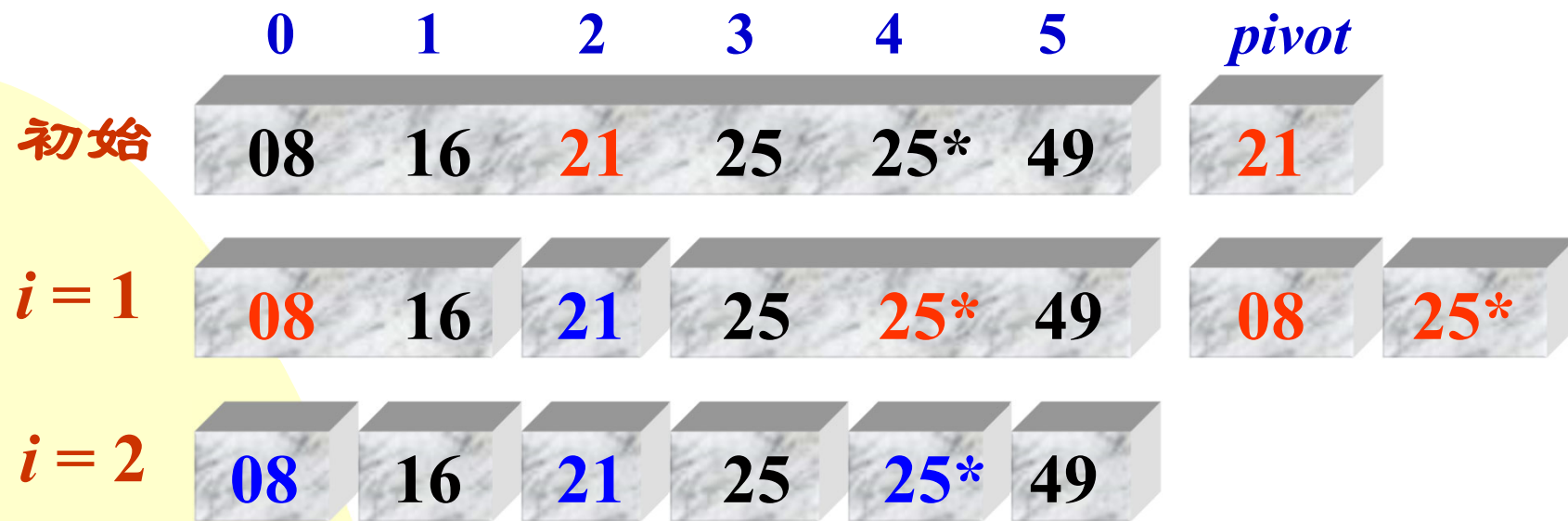
若能更合理地选择**基准记录**，使得每次划分所得的两个子序列中的记录个数尽可能地接近，可以加速排序速度；

但是由于记录的初始排列次序是随机的，这个要求很难办到！

有一种改进办法：

选择更好的**基准记录**；

取每个待排序记录序列的**第一个记录、最后一个记录**和**位置接近正中的3个记录**，取其关键字居中者作为**基准记录**。



用居中关键字记录作为基准记录

快速排序是一种不稳定的排序方法。

对于  $n$  较大的平均情况而言，快速排序是“快速”的，但是当  $n$  很小时，这种排序方法往往比其它简单排序方法还要慢。

# 第十章 内部排序

- 概述
- 插入排序
- 交换排序
- 选择排序
- 归并排序
- 小结



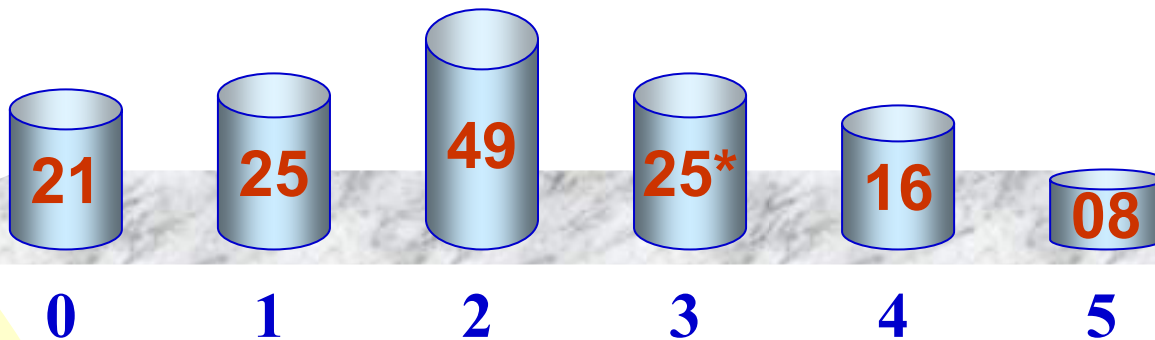
# 选择排序

**基本思想：**每一趟（例如第  $i$  趟， $i = 1, 2, \dots, n-1$ ）在后面  $n-i+1$  个待排序记录中选出关键字最小的记录，作为有序记录序列的第  $i$  个记录。待到第  $n-1$  趟做完，待排序记录只剩下1个，就不用再选了。

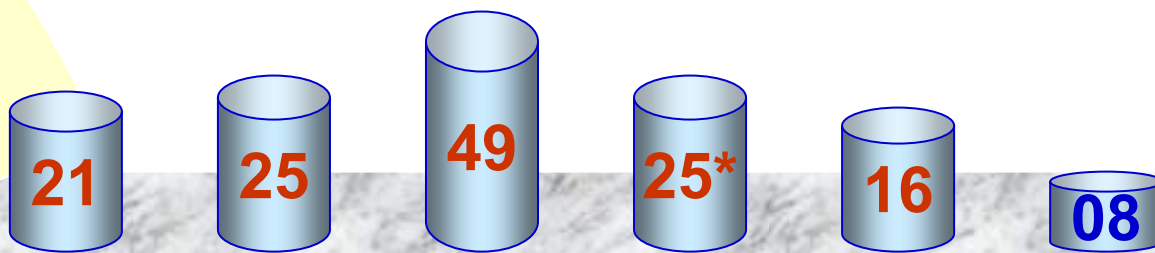
## 简单选择排序 (Select Sort)

首先在所有记录中选出关键字最小的记录，把它与第1个记录交换，然后在其余的记录中再选出关键字次最小的记录与第2个记录交换，以次类推.....，直到所有记录排序完成。

初始

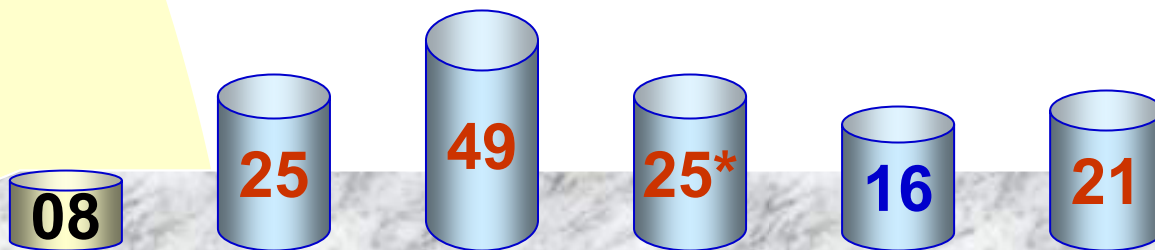


$i = 0$



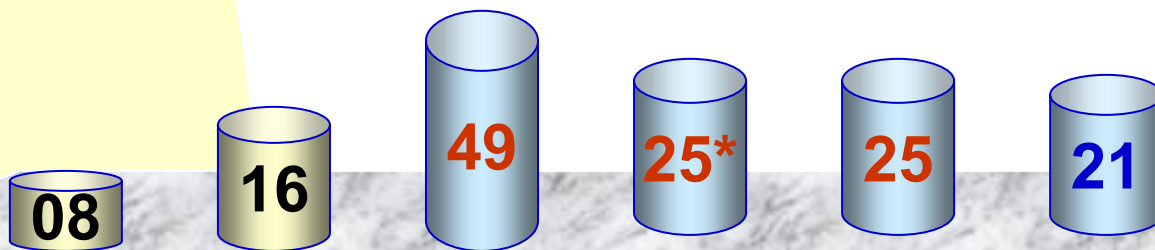
最小者 08  
交换21,08

$i = 1$



最小者 16  
交换25,16

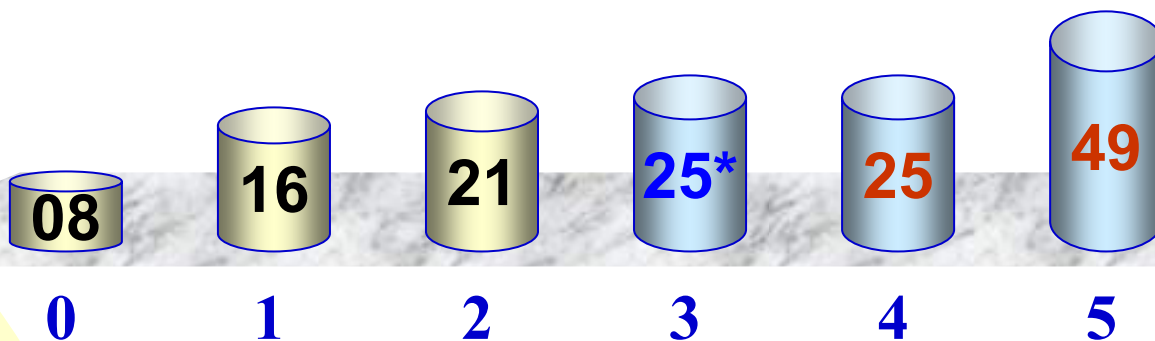
$i = 2$



最小者 21  
交换49,21

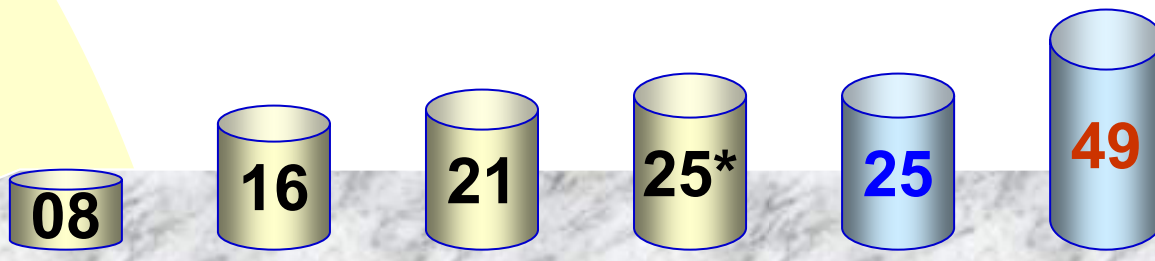


$i = 3$



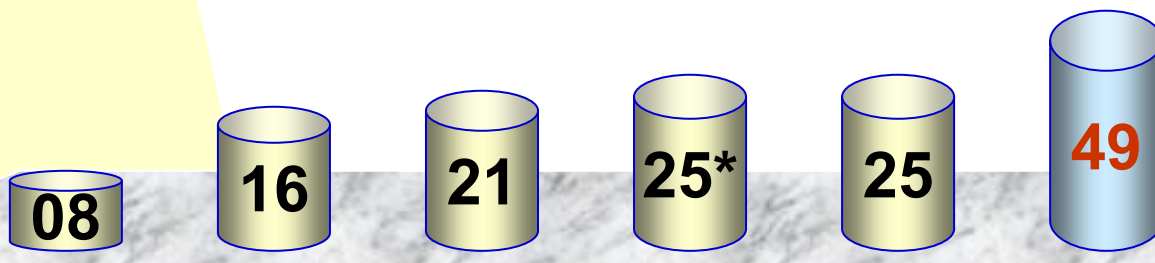
最小者 25\*  
无交换

$i = 4$



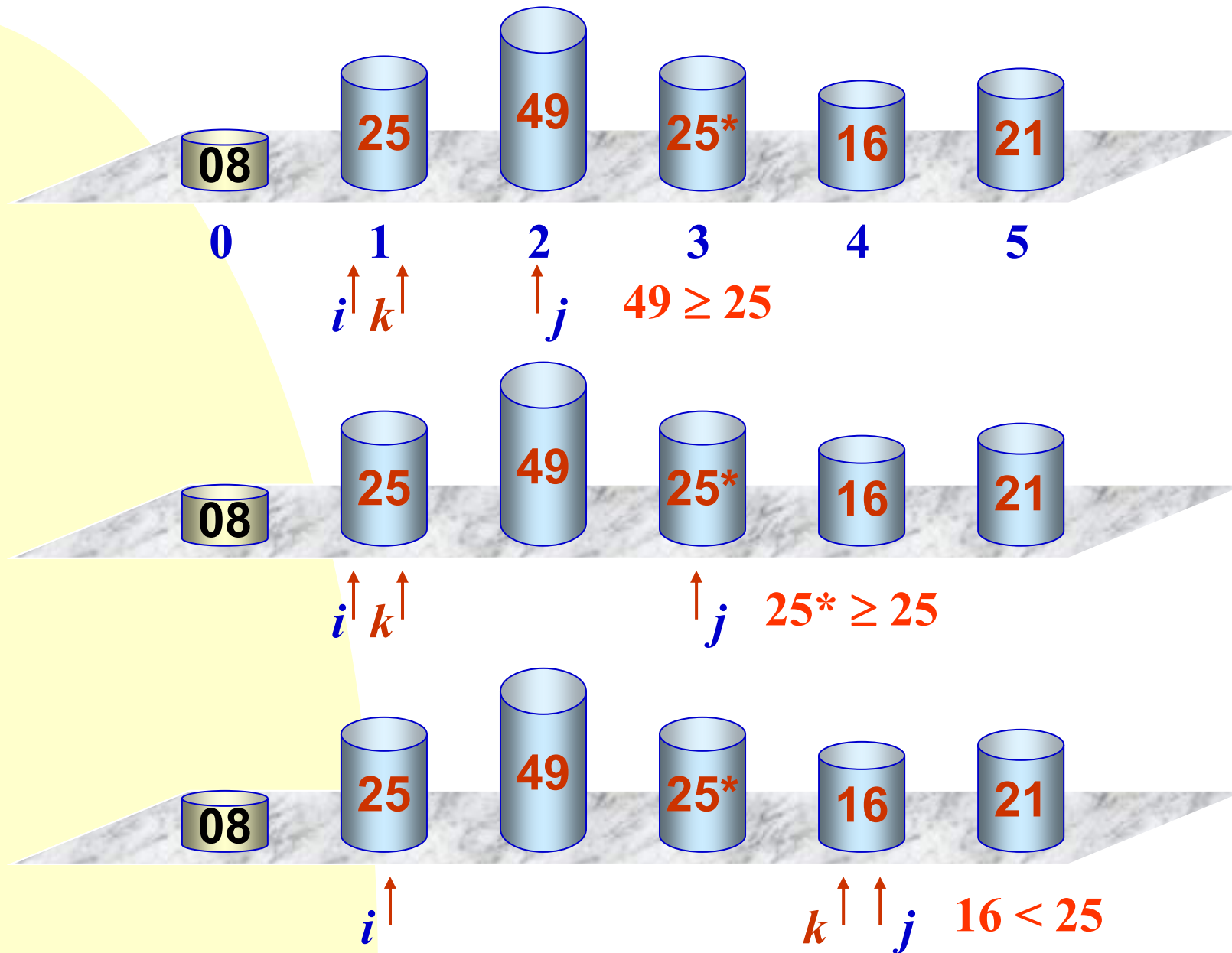
最小者 25  
无交换

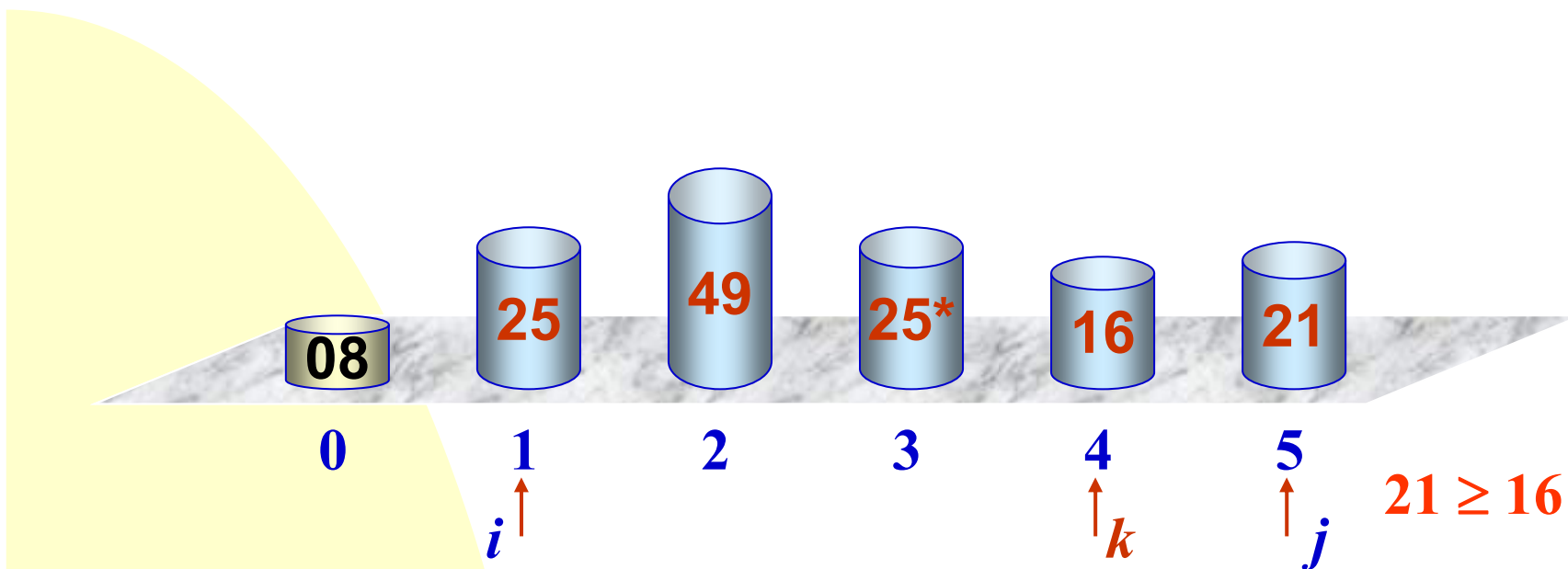
结果



各趟排序后的结果

## $i=1$ 时简单选择排序的过程





$k$  指示当前序列中最小者

## 简单选择排序的算法:

```
void SelectSort(Record r[], int n){  
    for( i=0; i<n-1; i++ ){    // 做n-1趟选择排序  
        k = i;  
        // 在当前无序区选关键字最小的记录  
        for( j=i+1; j<n; j++ )  
            if(r[j].key < r[k].key ) k = j ;  
        if( k != i ){  
            temp = L.r[i];  
            L.r[i] = L.r[k];  
            L.r[k] =temp;  
        }  
    }  
}
```

## 算法分析:

简单选择排序的**关键字比较次数** $KCN$ 与记录的初始排列无关。

第  $i$  趟选择具有最小关键字记录所需的比较次数总是  $n-i$  次, 此处假定整个待排序记录序列有  $n$  个记录。  
因此, 总的关键字比较次数为:

$$KCN = \sum_{i=1}^{n-1} (n-i) = \frac{n(n-1)}{2}$$

记录的移动次数与记录序列的初始排列有关。  
当这组记录的初始状态是按其关键字从小到大有序的时候，记录的移动次数 $RMN = 0$ ，达到最少。

最坏情况是每一趟都要进行交换，总的记录移动次数为 $RMN = 3(n-1)$ 。

简单选择排序总的时间复杂度是 $O(n^2)$

简单选择排序是一种不稳定的排序方法。

如何改进简单选择排序的效率？

# 堆排序 (Heap Sort)

堆的概念:

堆是一个关键字序列 $\{k_1, k_2, \dots, k_n\}$ , 它具有如下特性:

$$\begin{array}{lcl} k_i \leq k_{2i} & \text{或} & k_i \geq k_{2i} \\ k_i \leq k_{2i+1} & & k_i \geq k_{2i+1} \end{array} \quad (i = 1, 2, \dots, \lfloor n/2 \rfloor)$$

堆的特性在完全二叉树中解释为: 完全二叉树中任一结点的关键字都小于等于 (或大于等于) 它的左、右孩子的关键字。

如下列关键字序列均是堆:

$\{5, 23, 16, 68, 94, 72, 71, 73\}$  (小顶堆)

$\{96, 83, 27, 38, 11, 9\}$  (大顶堆)

由堆定义可知: 若关键字序列 $\{k_1, k_2, \dots, k_n\}$ 是堆, 则堆顶元素是 $n$ 个元素序列中的最小 (最大) 元素。

## 堆排序基本思想:

- 首先将初始待排序记录序列**建堆**，则堆顶元素必为含最大关键字或最小关键字的记录，输出该记录；
- 然后将剩余记录再**调整为堆**；
- 如此反复，直至排序结束。

- ① 如何将初始的无序序列构造成堆（建初始堆）？
- ② 若 $k_i$ 的左、右子树已经是堆，如何将以 $k_i$ 为根的完全二叉树也调整为堆？



■ 在堆排序主要需解决两个问题：

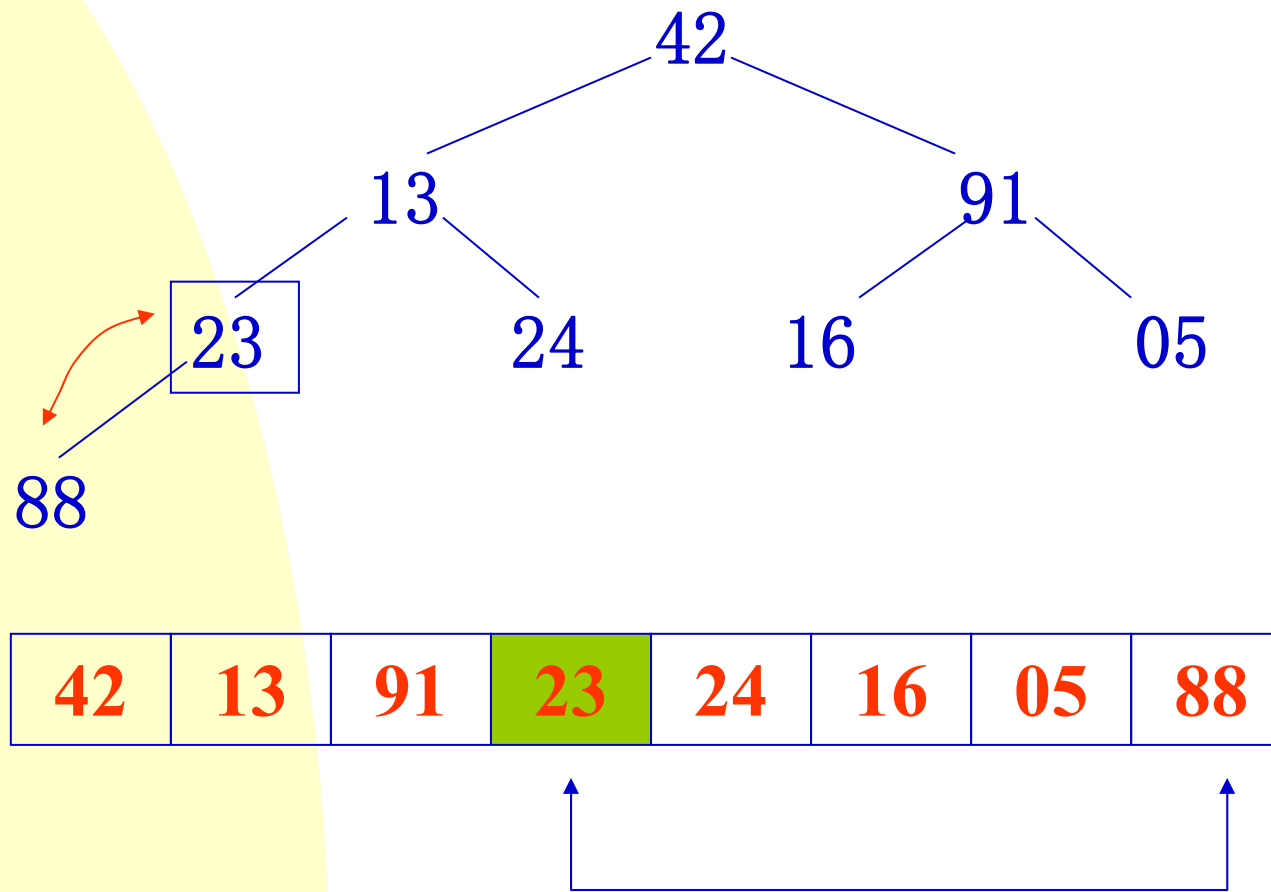
① 如何将初始的无序序列构造成堆（建初始堆）？

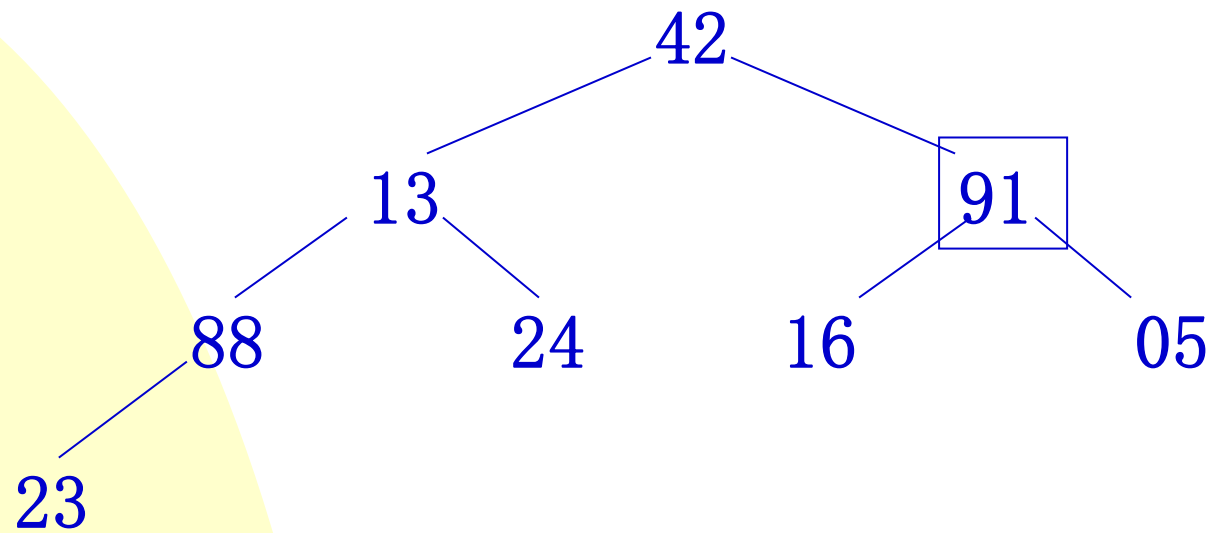
在完全二叉树中，所有序号 $i > \lfloor n/2 \rfloor$ 的结点都是叶子，因此，以这些结点为根的子树均已是堆，这样只需将以序号为 $\lfloor n/2 \rfloor, \lfloor n/2 \rfloor - 1, \lfloor n/2 \rfloor - 2, \dots, 1$ 的结点为根的子树都调整为堆即可。在按此次序调整每个结点时，其左、右子树均已是堆。

② 若 $k_i$ 的左、右子树已经是堆，如何将以 $k_i$ 为根的完全二叉树也调整为堆？

因 $k_i$ 的左、右子树已经是堆，所以必须在 $k_i$ 和它的左、右孩子中选出最大（或最小）的结点放到 $k_i$ 的位置上。不妨设 $k_{2i}$ 关键字最大，将 $k_i$ 与 $k_{2i}$ 交换位置，而交换之后有可能导致以 $k_{2i}$ 为根的子树不再为堆，于是可重复上述过程，将以 $k_{2i}$ 为根的子树调整为堆，……，如此逐层下去，最多可能一直调整到树叶，此方法称为“筛选法”。

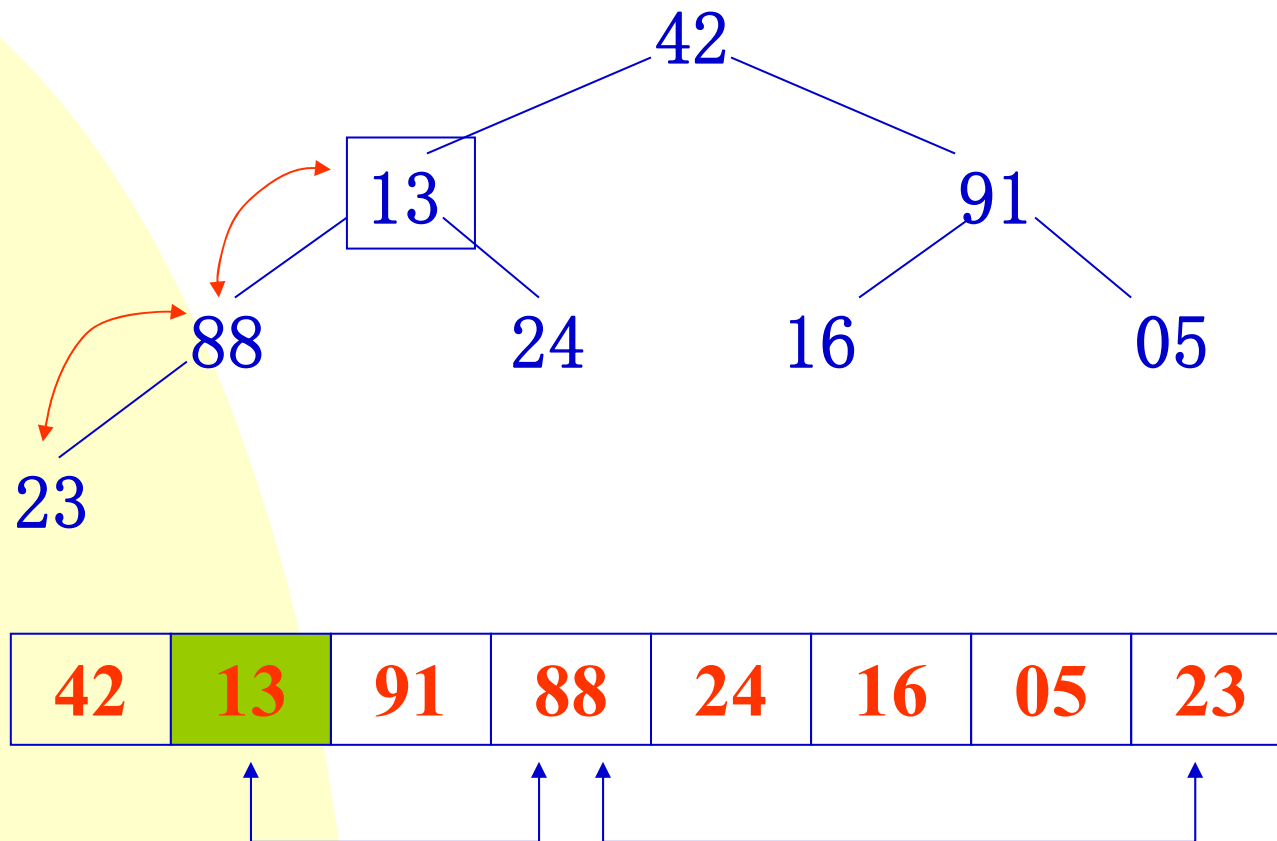
**例子：**关键字序列为 42, 13, 91, 23, 24, 16, 05, 88,  $n=8$ , 故从第四个结点开始调整

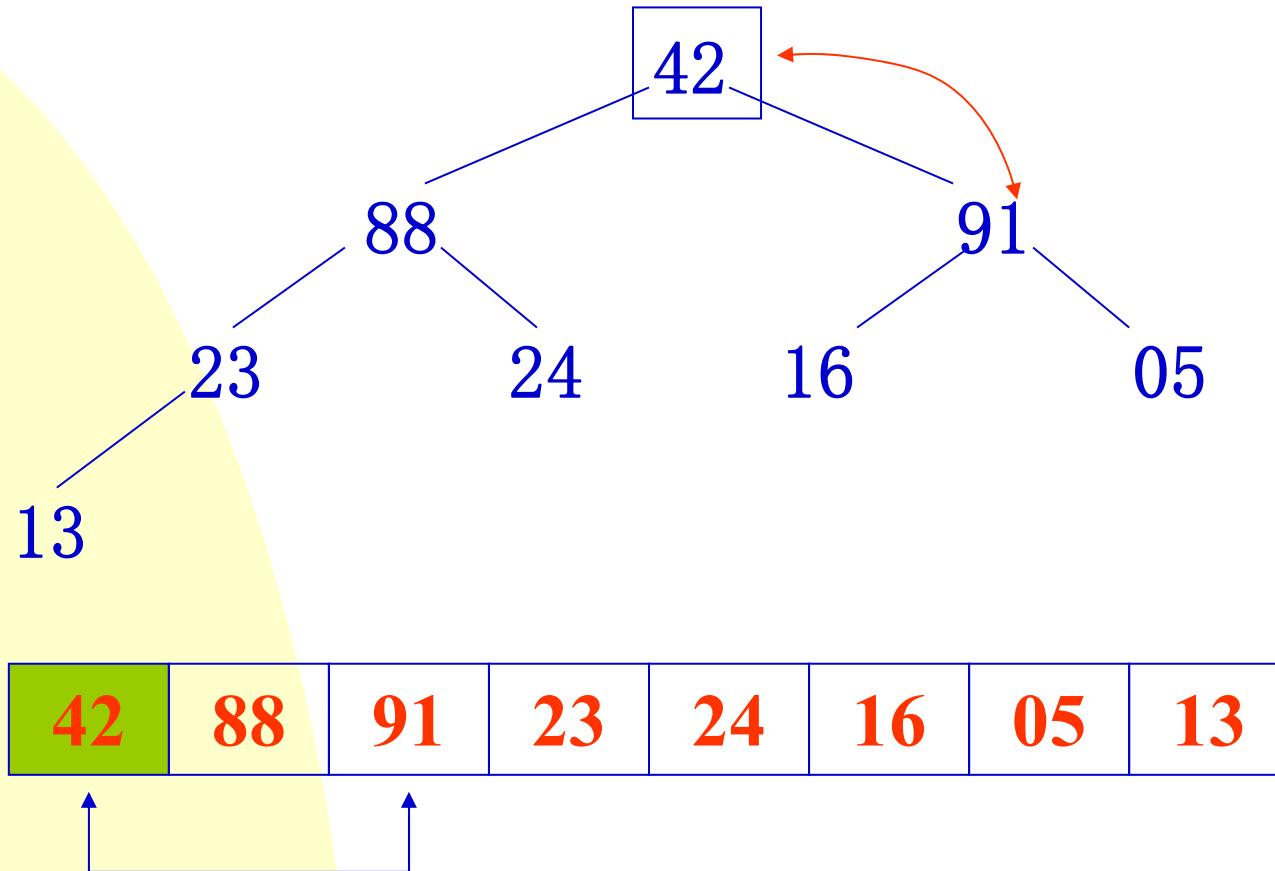


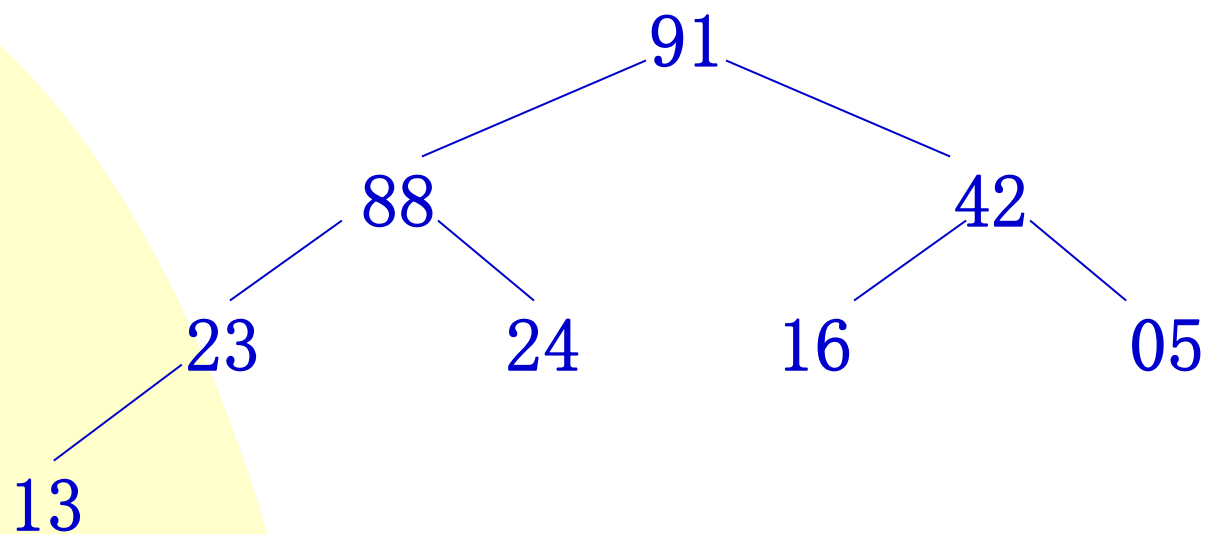


42	13	91	88	24	16	05	23
----	----	----	----	----	----	----	----

**不调整**







91	88	42	23	24	16	05	13
----	----	----	----	----	----	----	----

**建成的堆**

## 筛选算法 Sift:

// 已知 $r[k..m]$ 中记录的关键字除 $r[k].key$ 之外均满足堆的定义，本算法调整 $r[k]$  的关键字，使 $r[k..m]$ 成为一个大顶堆.

```
void Sift( Record r[], int k, int m )
{
    rc = r[k];
    for( j=2*k; j<=m; j*=2 ){ // 沿key较大的孩子结点向下筛选
        if( j<m && r[j].key<r[j+1].key ) j++;
        if( rc.key >= r[j].key ) break;    // rc应插入在位置k上
        r[k] = r[j];
        k = j;
    }
    r[k] = rc;
}
```

上述算法只是对一个指定的结点进行调整。  
有了这个算法，则将初始的无序区  $r[1]$  到  $r[n]$   
建成一个大顶堆，可用以下语句实现：

```
for ( i = n/2; i >= 1; i-- )
```

```
    Sift( r, i, n );
```

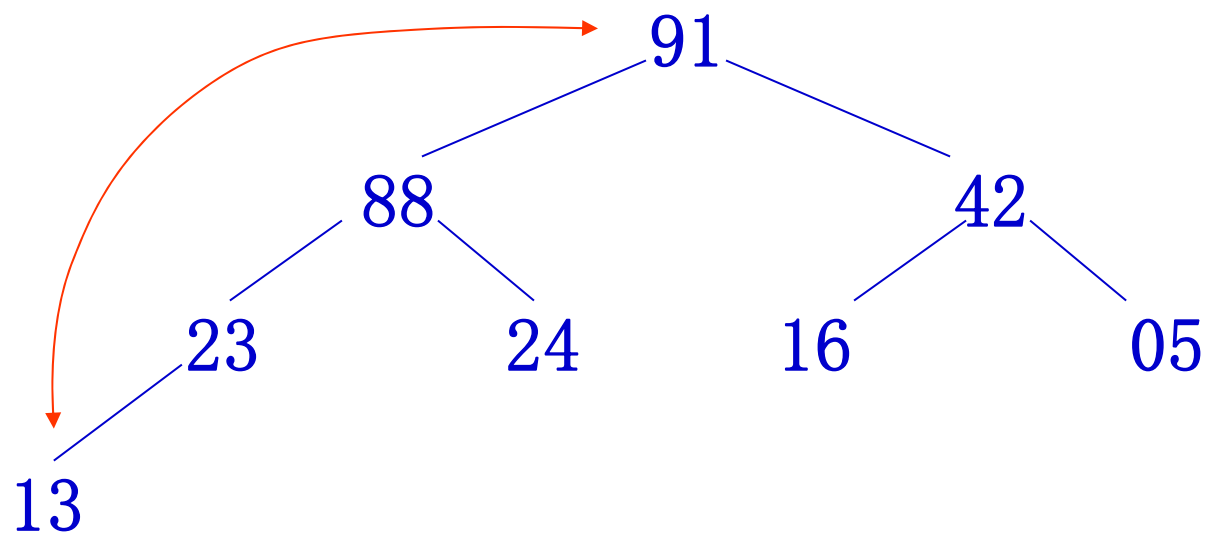


## 基于初始堆进行堆排序

大顶堆的第一个记录 $r[1]$ 具有最大的关键字，将 $r[1]$ 与 $r[n]$ 对调，把具有最大关键字的记录交换到最后；再对前面的 $n-1$ 个记录，使用堆的调整算法  $\text{Sift}(r, 1, n-1)$ ，重新建立大顶堆。结果具有次最大关键字的记录又上浮到堆顶，即 $r[1]$ 位置。

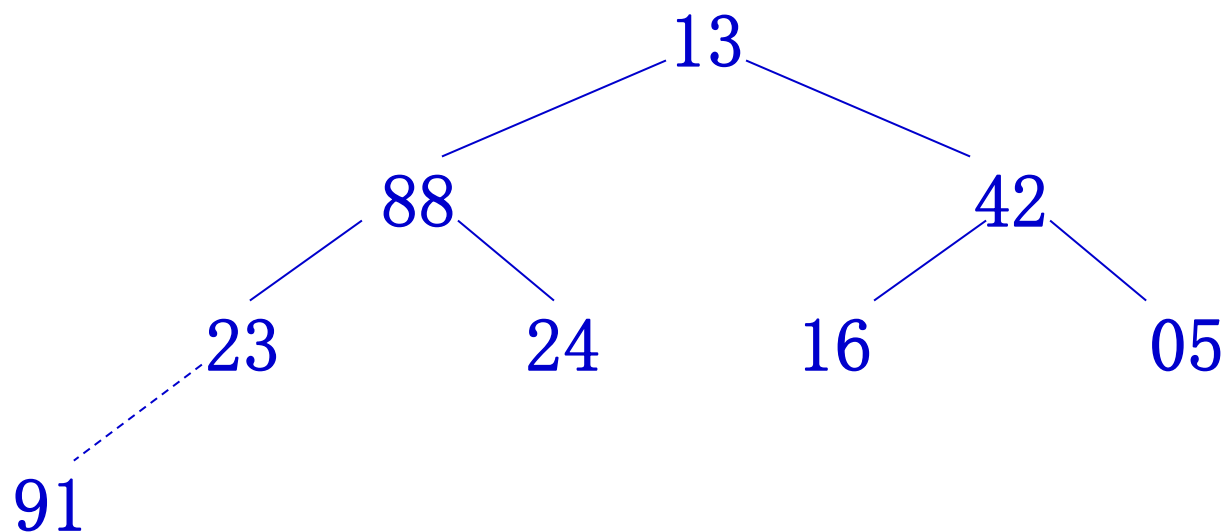
再对调 $r[1]$ 和 $r[n-1]$ ，调用  $\text{Sift}(r, 1, n-2)$ ，对前 $n-2$ 个记录重新调整...

如此反复执行，最后得到全部排序好的记录序列。这个算法即堆排序算法。



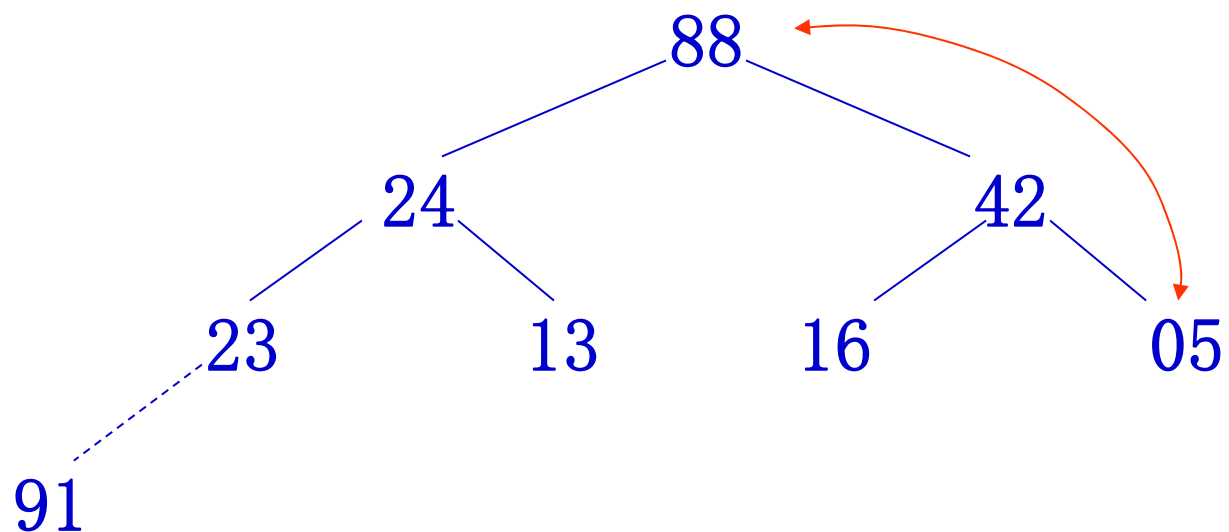
91	88	42	23	24	16	05	13
----	----	----	----	----	----	----	----

**(a) 初始堆r[1]到r[8]**



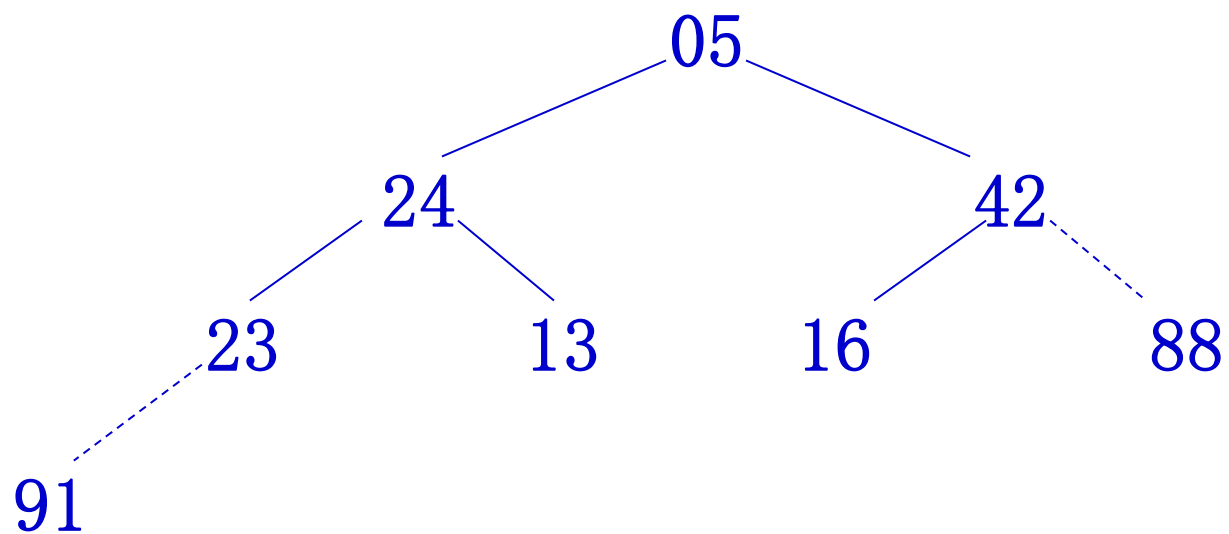
13	88	42	23	24	16	05	91
----	----	----	----	----	----	----	----

**(b) 第一趟排序之后**



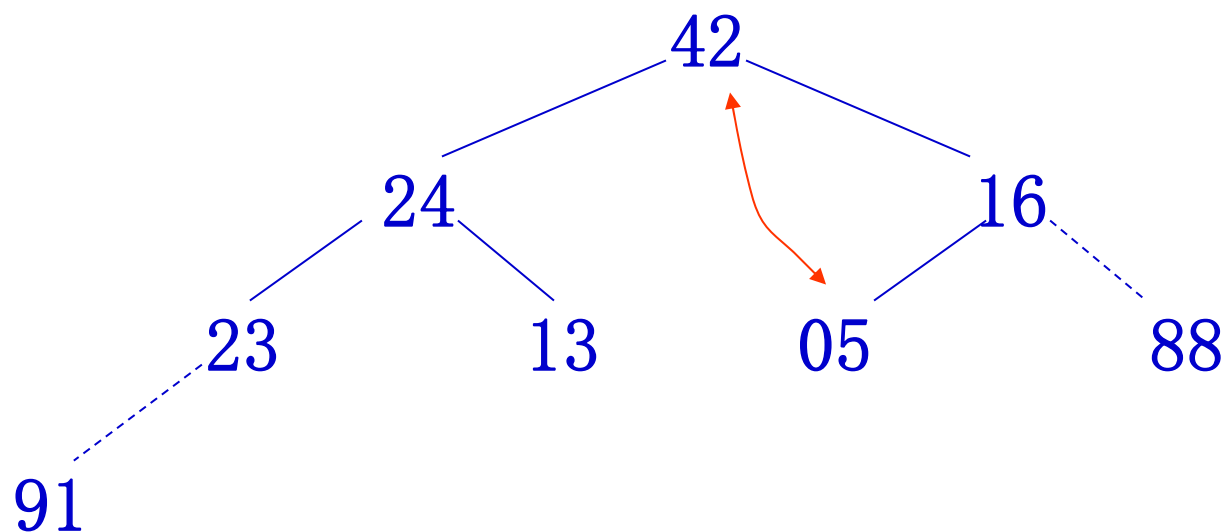
88	24	42	23	13	16	05	91
----	----	----	----	----	----	----	----

**(c) 重建的堆r[1]到r[7]**



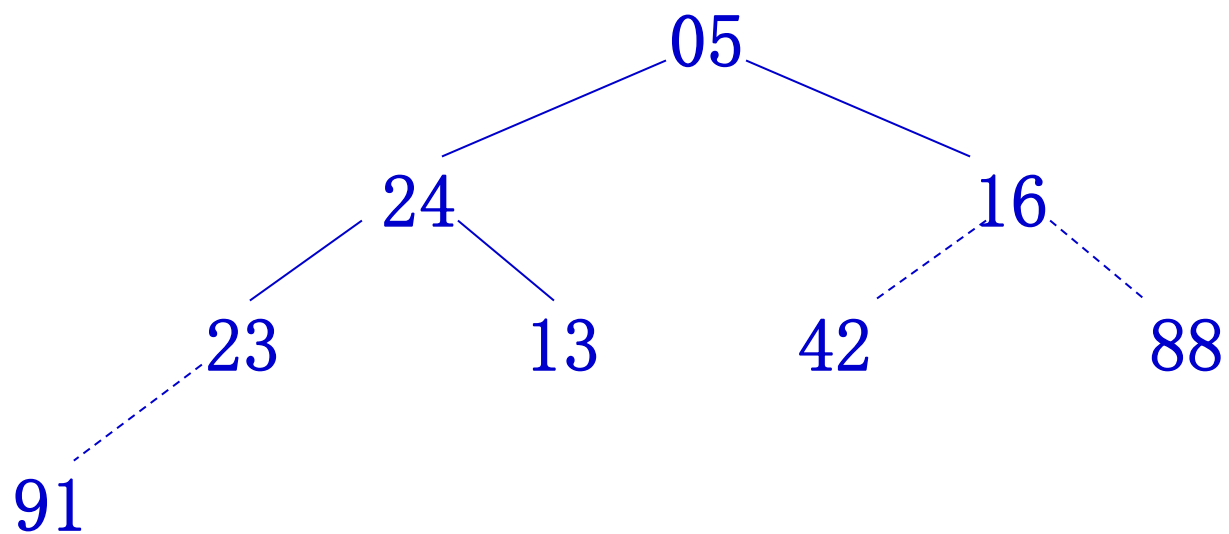
05	24	42	23	13	16	88	91
----	----	----	----	----	----	----	----

**(d) 第二趟排序之后**



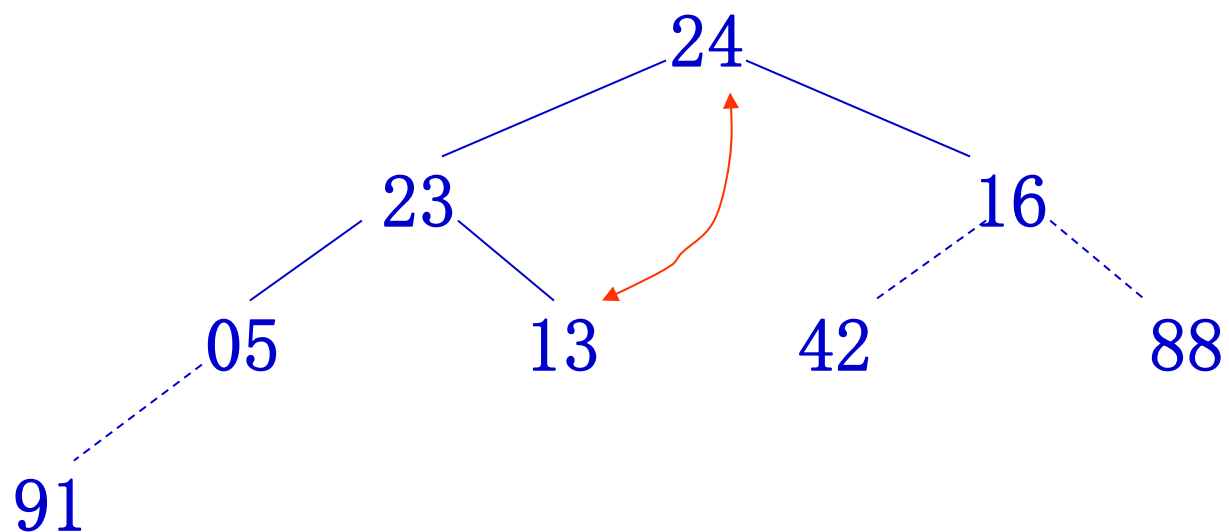
42	24	16	23	13	05	88	91
----	----	----	----	----	----	----	----

**(e) 重建的堆r[1]到r[6]**



05	24	16	23	13	42	88	91
----	----	----	----	----	----	----	----

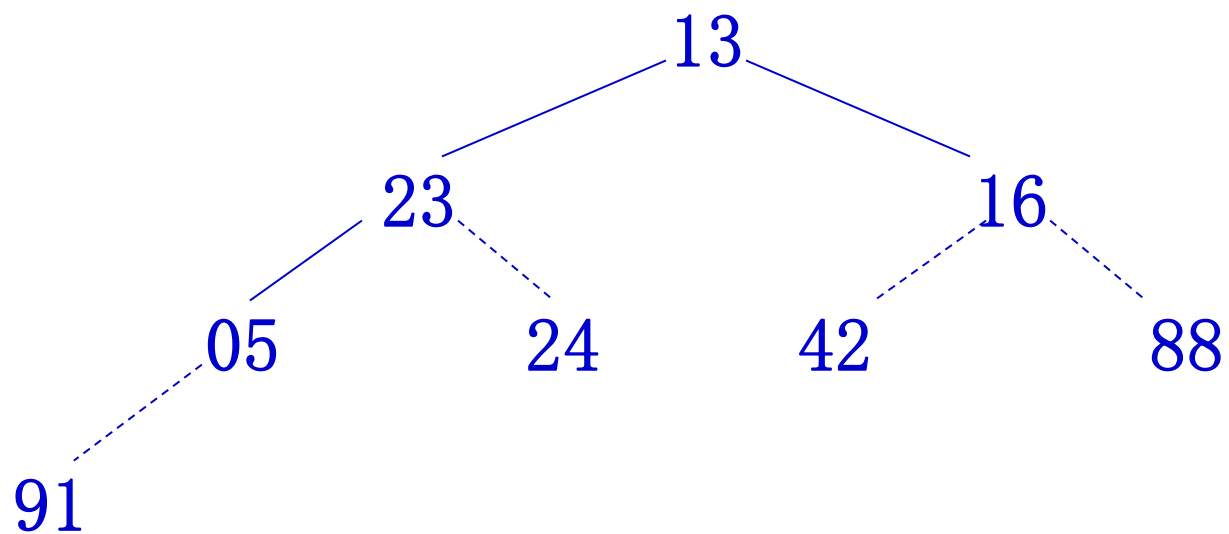
**(f) 第三趟排序之后**



24	23	16	05	13	42	88	91
----	----	----	----	----	----	----	----

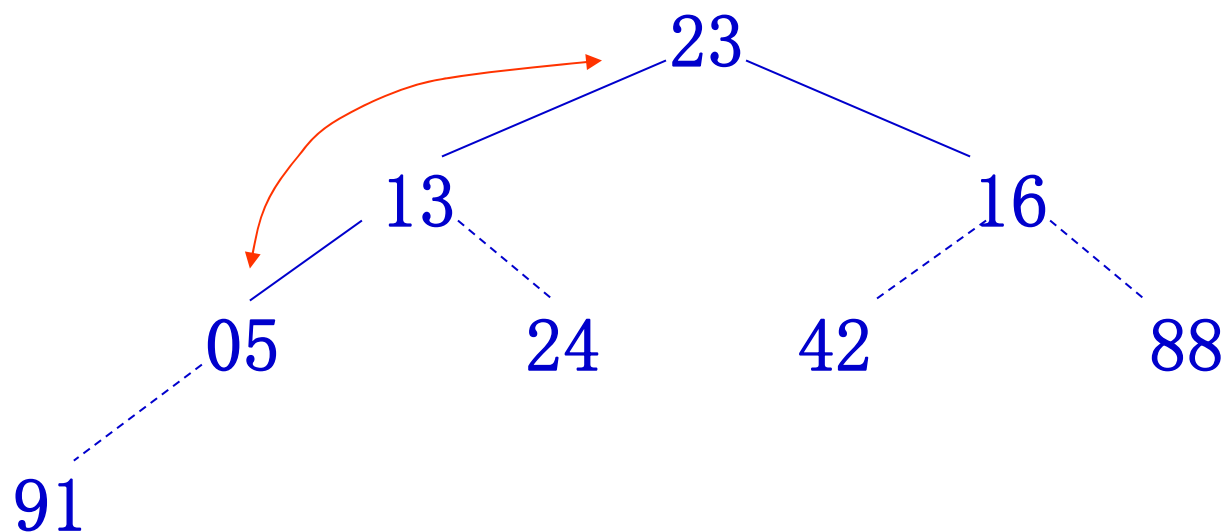
**(g) 重建的堆r[1]到r[5]**





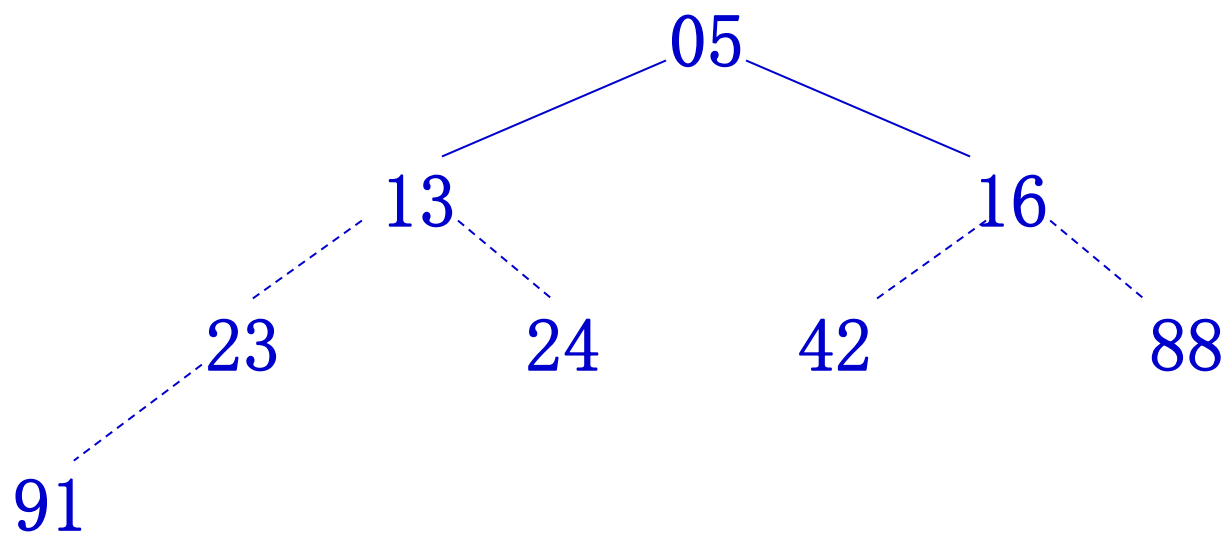
13	23	16	05	24	42	88	91
----	----	----	----	----	----	----	----

**(h) 第四趟排序之后**



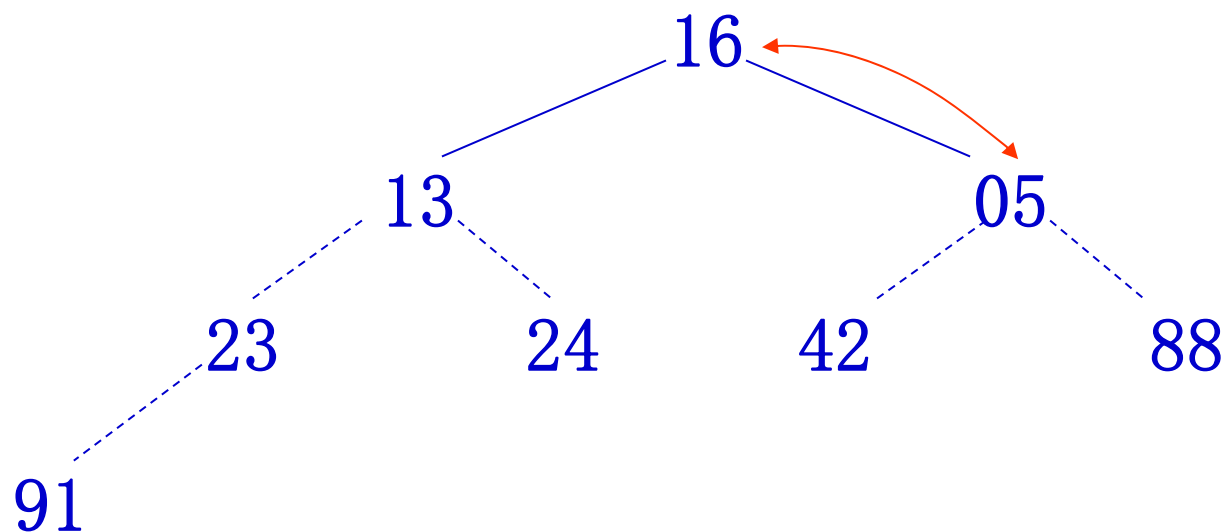
23	13	16	05	24	42	88	91
----	----	----	----	----	----	----	----

**(i) 重建的堆r[1]到r[4]**



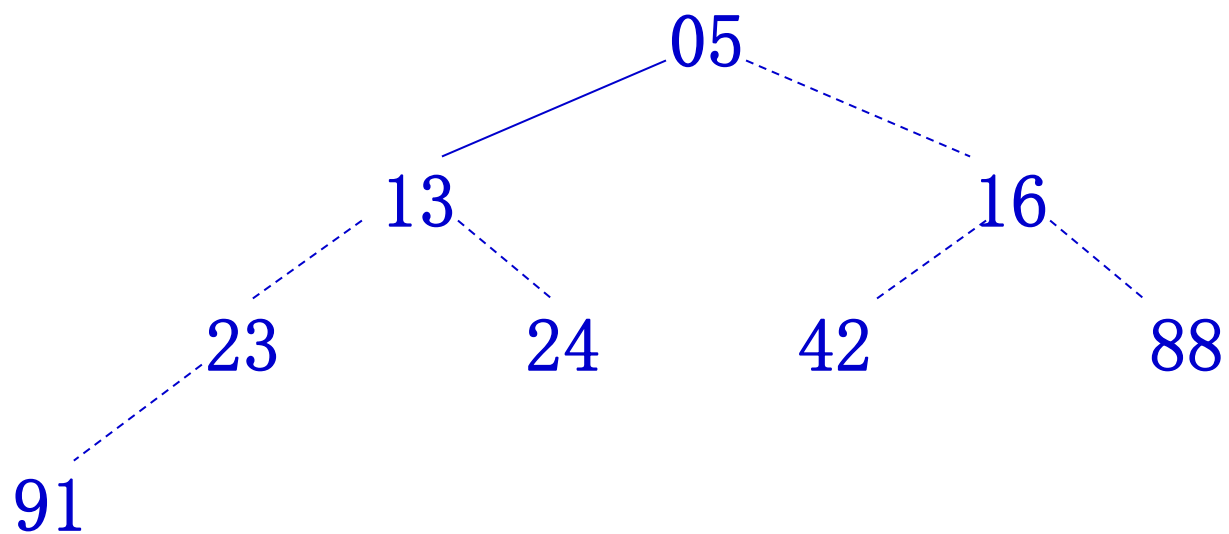
05	13	16	23	24	42	88	91
----	----	----	----	----	----	----	----

(j) 第五趟排序之后



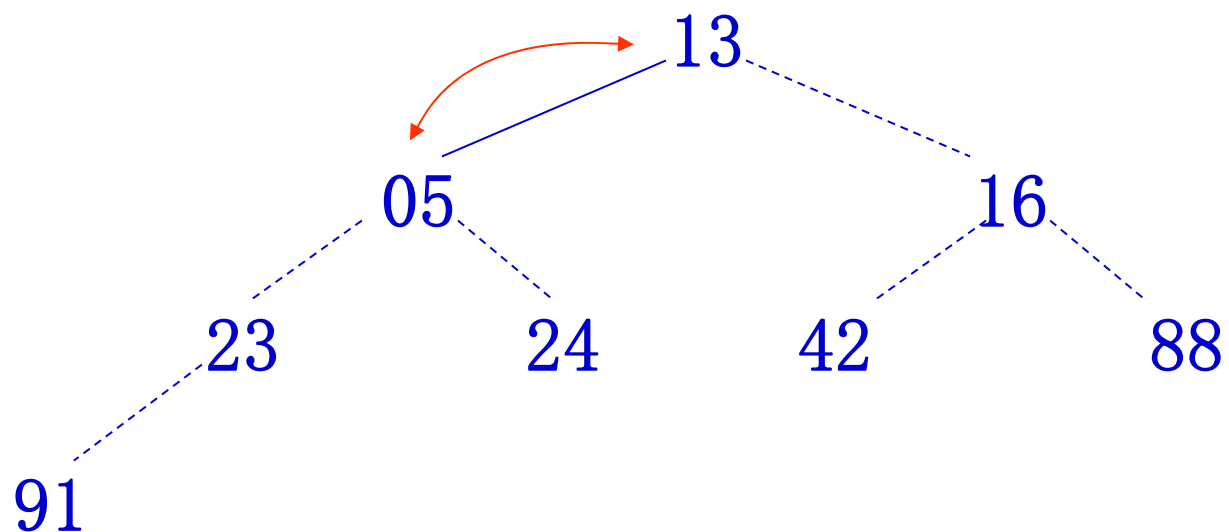
16	13	05	23	24	42	88	91
----	----	----	----	----	----	----	----

**(k) 重建的堆r[1]到r[3]**



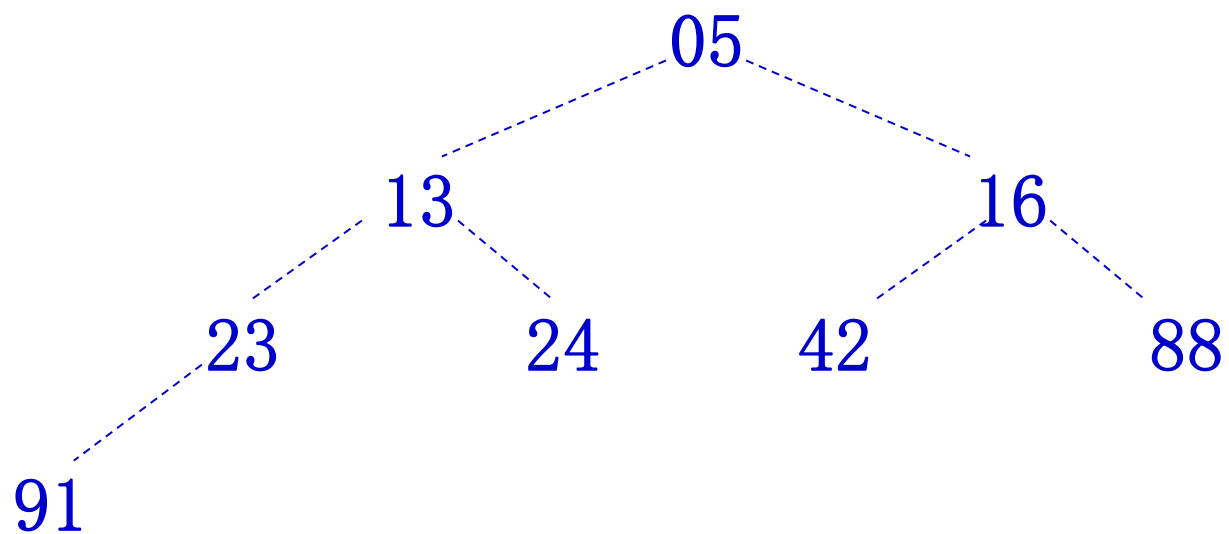
05	13	16	23	24	42	88	91
----	----	----	----	----	----	----	----

**(I) 第六趟排序之后**



13	05	16	23	24	42	88	91
----	----	----	----	----	----	----	----

**(m) 重建的堆r[1]到r[2]**



05	13	16	23	24	42	88	91
----	----	----	----	----	----	----	----

**(n) 第七趟排序之后**

## 堆排序的算法:

```
void HeapSort( Record r[], int n)
{
    for(i=n/2; i>=1; i--)    // 把r[1..n]建成大顶堆
        Sift( r, i, n);
    for( i=n; i>1; i-- ){
        temp = r[1];
        r[1] = r[i];
        r[i] = temp;
        Sift( r, 1, i-1 ); // 将r[1..i-1]重新调整为大顶堆
    }
}
```



## 算法分析:

堆排序的时间主要由**建立初始堆**和**不断重复调整堆**这两部分的时间开销构成。

若设堆中有  $n$  个结点，且对应的完全二叉树有  $k$  层，则  $2^{k-1} \leq n < 2^k$ 。在第  $i$  层上的结点数  $\leq 2^{i-1}$  ( $i = 1, 2, \dots, k$ )。在第一个建立初始堆的for循环中对每一个非叶结点调用了一次堆调整算法 *Sift()*，因此该循环所用的关键字比较次数为：

$$\sum_{i=k-1}^1 2^{i-1} \cdot 2(k-i) \leq 4n$$

其中， $i$  是层序号， $2^{i-1}$  是第  $i$  层的最大结点数， $(k-i)$  是第  $i$  层结点能够移动的最大距离。

堆排序过程中进行n-1次重建堆所进行的总关键字比较次数不超过下式：

$$2 * ( \lfloor \log_2(n-1) \rfloor + \lfloor \log_2(n-2) \rfloor + \dots + \lfloor \log_2 2 \rfloor ) < 2n \lfloor \log_2 n \rfloor \\ = O(n \log_2 n)$$

因此堆排序总的时间复杂度是  $O(n + n \log_2 n) = O(n \log_2 n)$ 。

堆排序在最坏情况下的时间复杂度也是  $O(n \log_2 n)$ ，相对于快速排序来说，这是堆排序的最大优点。

堆排序是一个不稳定的排序方法。

# 第十章 内部排序

- 概述
- 插入排序
- 交换排序
- 选择排序
- 归并排序
- 小结

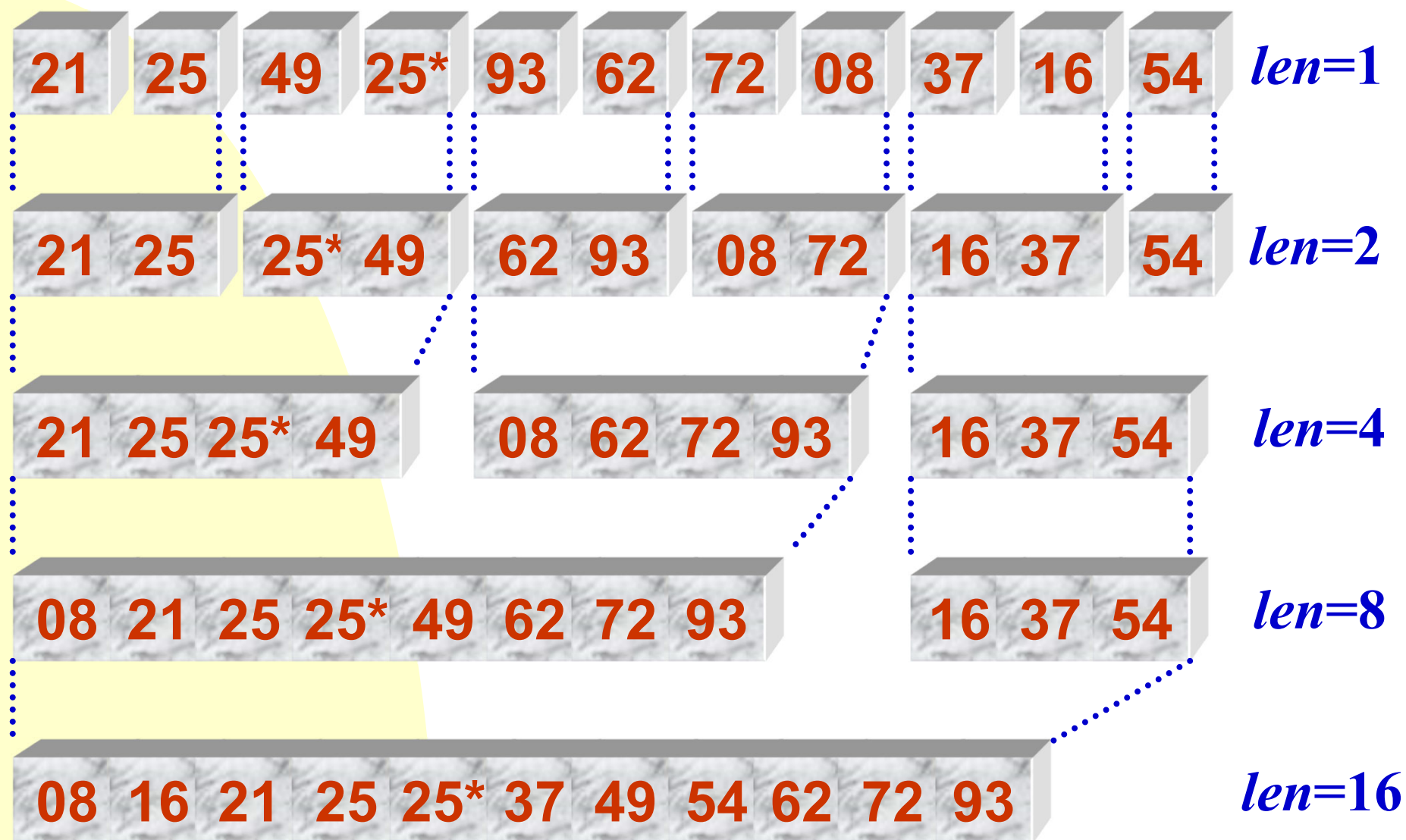


# 归并排序 (Merge Sort)

归并排序是利用“归并”技术来进行排序，所谓归并是指将两个有序的序列合并形成一个新的有序序列。

将待排序记录序列 $r_1, r_2, \dots, r_n$ 看成为 $n$ 个长度为1的有序子序列，把这些子序列两两进行归并，便得到 $\lceil n/2 \rceil$ 个长度为2的有序子序列，然后再两两归并，……，如此重复，直到最后得到一个长度为 $n$ 的有序记录序列为止，这种方法称为归并排序算法。

# 归并排序过程



## 归并算法:

// 将有序的SR[i..m]和SR[m+1..n]归并为有序的TR[i..n] 算法

```
void Merge( Record SR[], Record TR[], int i, int m, int n) {  
    int j, k, l;  
    for( j=m+1, k=i; i<=m && j<=n; ++k)  
        if ( SR[i].key <= SR[j].key )  
            TR[k] = SR[i++];  
        else  
            TR[k]=SR[j++];  
    if( i <= m )  
        for( l=0; l<=m-i; l++ )  
            TR[k+l]=SR[i+l]; // 将剩余的SR[i..m]复制到TR  
    if( j <= n )  
        for( l=0; l<=n-j; l++ )  
            TR[k+l]=SR[j+l]; // 将剩余的SR[j..n]复制到TR  
}
```

## 一趟归并算法:

```
void MergePass(Record R[], Record R1[], int length)
{
    i=0;
    while (i+2*length-1<n){
        Merge(R, R1, i, i+length-1, i+2*length-1);
        i=i+2*length;
    }

    if (i+length-1<n-1)
        Merge(R, R1, i, i+length-1, n-1);
    else
        for (j=i; j<n; j++) R1[j]=R[j];
}
```

## 归并排序算法:

```
void MergeSort(Record R[], int n) {  
    int length=1;  
    while ( length<n ) {  
        MergePass (R, R1, length) ;  
        length=2*length;  
        MergePass (R1, R, length) ;  
        length=2*length;  
    }  
}
```

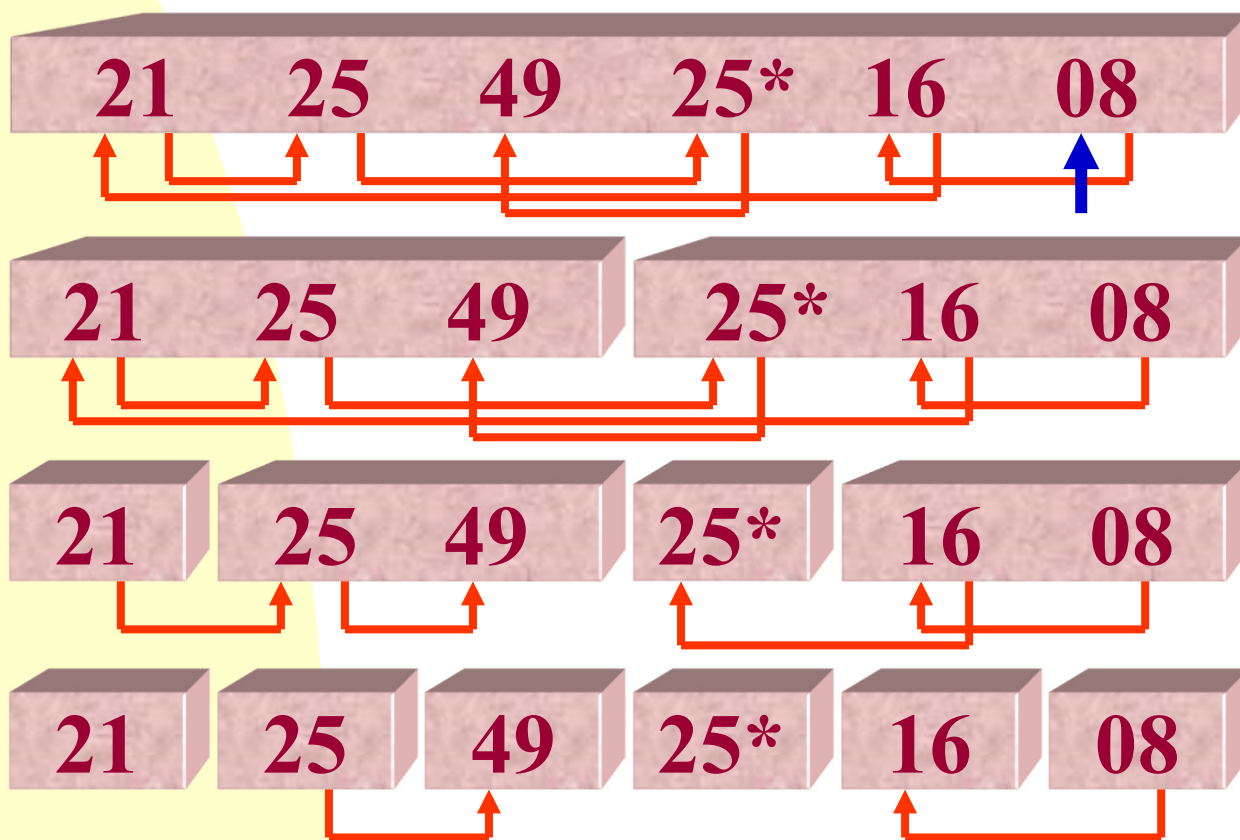
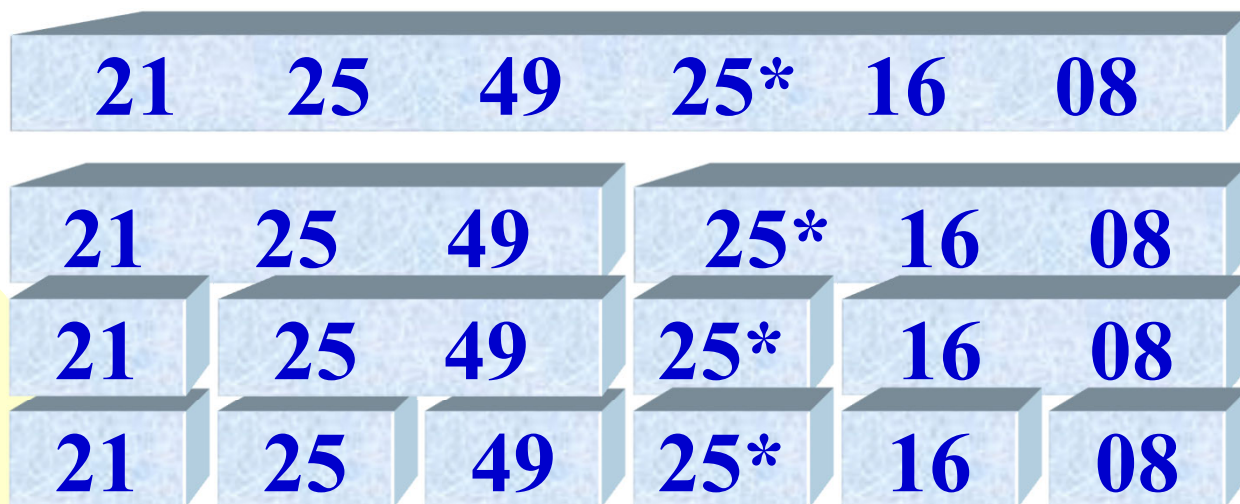


# 归并排序

归并排序算法本质上是个递归算法。

如何设计归并排序的递归过程？

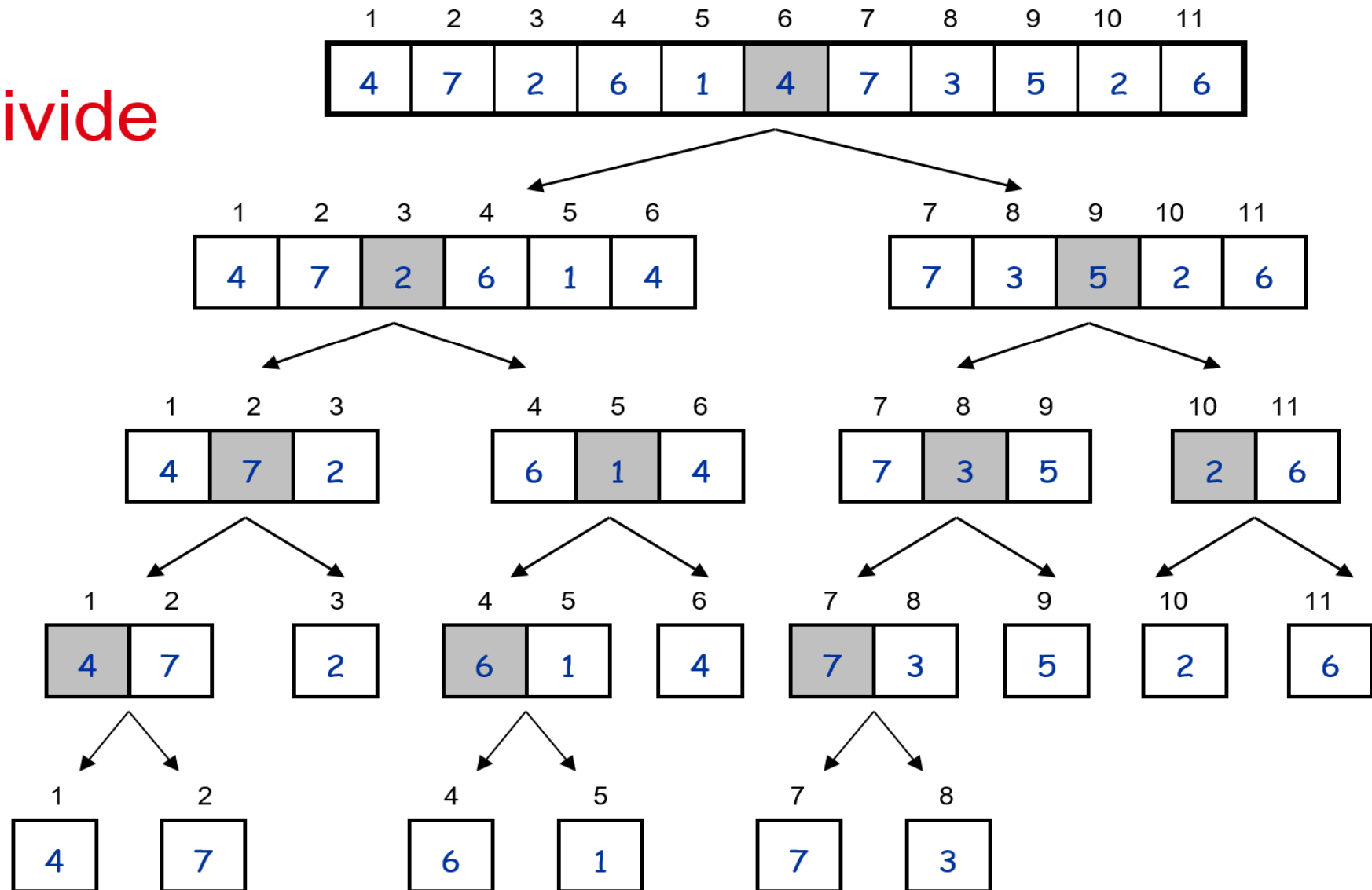
递归



回推

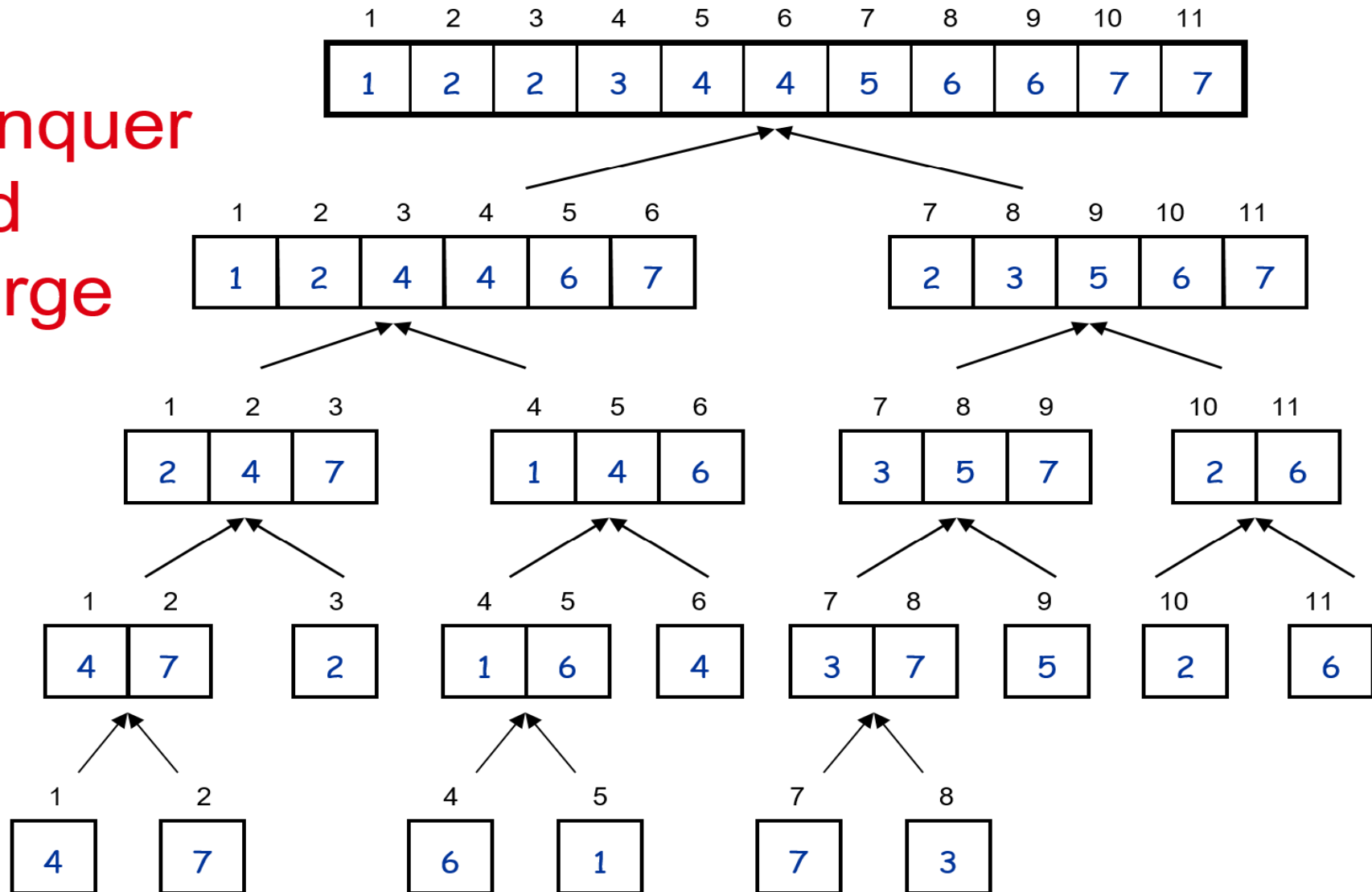
# Example

Divide



# Example

Conquer  
and  
Merge



## 递归的归并排序算法:

```
void MSort(Record r[], Record r1[], int s, int t)
{
    if( s == t ) r1[s]=r[s];
    else {
        m=(s+t)/2; // 将r[s..t]平分为r[s..m]和r[m+1..t]
        MSort( r, r2, s, m);
        MSort( r, r2, m+1, t);
        Merge( r2, r1, s, m, t);
    }
}
```

## 算法分析:

对于具有  $n$  个记录的文件排序, 必须做  $\lceil \log_2 n \rceil$  趟归并, 每趟归并所花时间为  $O(n)$ , 所以二路归并算法总的时间复杂度为:  $O(n \log_2 n)$ 。

归并排序方法是一种稳定的排序方法。

# 第十章 内部排序

- 概述
- 插入排序
- 交换排序
- 选择排序
- 归并排序
- 小结



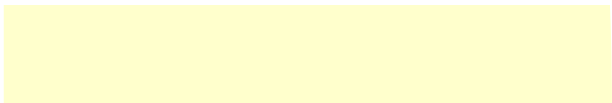
# 内部排序方法的比较和选择

选取排序方法时需要考虑的因素有：

- 待排序的记录数目
- 待排序记录序列的初始状态
- 记录本身信息量的大小
- 对排序稳定性的要求
- 语言工具的条件、辅助空间的大小



排序方法	最坏时间复杂度	平均时间复杂度	辅助空间	稳定性
直接插入排序	$O(n^2)$	$O(n^2)$	$O(1)$	稳定的
折半插入排序	$O(n^2)$	$O(n^2)$	$O(1)$	稳定的
Shell排序	$O(n^{1.3})$	$O(n^{1.3})$	$O(1)$	不稳定的
简单选择排序	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定的
堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	不稳定的
起泡排序	$O(n^2)$	$O(n^2)$	$O(1)$	稳定的
快速排序	$O(n^2)$	$O(n\log_2 n)$	$O(\log_2 n)$	不稳定的
归并排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n)$	稳定的
基数排序	$O(d(n+r))$	$O(d(n+r))$	$O(r+n)$	稳定的



## 各种排序法的比较:

- (1) 平均时间性能: 以快速排序法最佳, 但最坏情况下不如堆排序和归并排序;
- (2) 在 $n$ 较大时, 归并排序比堆排序快, 但所需辅助空间最多。
- (3) 简单排序以直接插入排序最简单, 当下列中记录“基本有序”或 $n$ 值较小时, 是最佳的排序方法。因此常和其他排序方法结合使用。
- (4) 当待排记录序列按关键字顺序有序时, 插入排序和起泡排序能达到 $O(n)$ 的时间复杂度;
- (5) 简单选择排序、堆排序和归并排序的时间性能不随记录序列中关键字的分布而改变。

**(5) 从稳定性来看，大部分时间复杂度为  $O(n^2)$  的简单排序法都是稳定的。然而，快速排序、堆排序和希尔排序等时间性能较好的排序都是不稳定的。一般来说，排序过程中的比较是在相邻的两个记录关键字之间进行的排序方法是稳定的。大多数情况下排序是按记录的主关键字进行的，则所有的排序方法是否稳定无关紧要。当排序是按记录的次关键字进行时，则应根据问题所需慎重选择。**

## 空间性能:

指的是排序过程中所需的辅助空间大小。

1. 所有的简单排序方法（包括：插入、起泡和选择排序）和堆排序的空间复杂度均为 $O(1)$ 。
2. 快速排序为 $O(\log n)$ ，为递归程序执行过程中栈所需的辅助空间。
3. 归并排序所需辅助空间最多，其空间复杂度为 $O(n)$ 。