

第八章 排序技术

本章的基本内容是：

8.1 排序的基本概念

8.2 插入排序

8.3 交换排序

8.4 选择排序

8.5 归并排序

1.1 排序的基本概念

排序 (Sorting)：给定一组记录的集合 $\{r_1, r_2, \dots, r_n\}$ ，其相应的关键码分别为 $\{k_1, k_2, \dots, k_n\}$ ，排序是将这些记录排列成顺序为 $\{r_{s1}, r_{s2}, \dots, r_{sn}\}$ 的一个序列，使得相应的关键码满足 $k_{s1} \leq k_{s2} \leq \dots \leq k_{sn}$ （称为升序）或 $k_{s1} \geq k_{s2} \geq \dots \geq k_{sn}$ （称为降序）。

正序：待排序序列中的记录已按关键码排好序。

逆序 (反序)：待排序序列中记录的排列顺序与排好序的顺序正好相反。

排序算法的稳定性

假定在待排序的记录集中，存在多个具有相同键值的记录，若经过排序，这些记录的**相对次序**仍然保持不变，即在原序列中， $k_i=k_j$ 且 r_i 在 r_j 之前，而在排序后的序列中， r_i 仍在 r_j 之前，则称这种排序算法是**稳定**的；否则称为**不稳定的**。

例如：

学号	姓名	高数	英语	思想品德
0001	王 军	85	88	88
0002	李 明	85	72	87
0003	汤晓影	78	72	86
...

单键排序与多键排序

单键排序： 根据一个关键码进行的排序；

多键排序： 根据多个关键码进行的排序。

学号	姓名	高数	英语	思想品德
0001	王 军	85	88	88
0002	李 明	85	72	87
0003	汤晓影	78	72	86
...

按**学号**排序——单键排序

按**成绩（高数+英语+思想品德）**排序——多键排序

多键排序

设关键码分别为 k_1, k_2, \dots, k_m ，多键排序有两种方法：

- (1) 依次对记录进行 m 次排序，第一次按 k_1 排序，第二次按 k_2 排序，依此类推。这种方法要求各趟排序所用的算法是稳定的。
- (2) 将关键码 k_1, k_2, \dots, k_m 分别视为字符串依次首尾连接在一起，形成一个新的字符串，然后，对记录序列按新形成的字符串排序。

多键排序



单键排序

排序算法的性能

- 1. 基本操作

- (1)比较：关键码之间的比较

- (2)移动：记录从一个位置移动到另一个位置

- 2. 辅助存储空间

辅助存储空间是指在数据规模一定的条件下，除了存放待排序记录占用的存储空间之外，执行算法所需要的其他存储空间。

- 3. 算法本身的复杂程度

排序算法的存储结构

从操作角度看，排序是线性结构的一种操作，待排序记录可以用顺序存储结构或链接存储结构存储。

假定1：采用顺序存储结构，关键码为整型，且记录只有关键码一个数据项。

```
int r[n+1];
```

//待排序记录存储在 $r[1] \sim r[n]$ ， $r[0]$ 留做他用

假定2：将待排序的记录序列排序为升序序列。

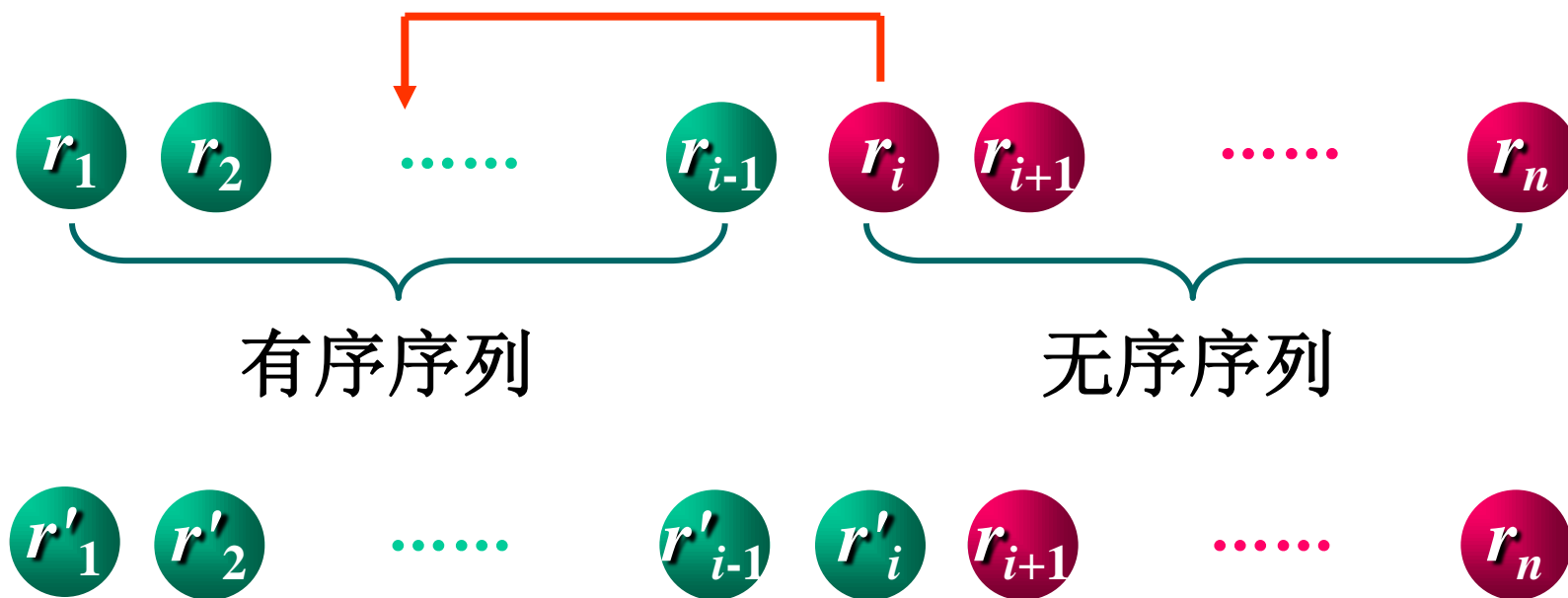
8.2 插入排序

插入排序(The Insertion Sort) 的主要操作是插入，其基本思想是：

每次将一个待排序的记录按其关键码的大小插入到一个已经排好序的有序序列中，直到全部记录排好序为止。

直接插入排序(The Direct Insertion Sort)

基本思想：在插入第 i ($i > 1$) 个记录时，前面的 $i-1$ 个记录已经排好序。



直接插入排序过程示例

r	0	1	2	3	4	5	6
		21	25	22	10	25*	18
$i = 2$		21	25	22	10	25*	18
$i = 3$	22	21	25	22	10	25*	18
$i = 4$	10	21	22	25	10	25*	18
$i = 5$	25	10	15	21	25	25*	18
$i = 6$	18	10	15	21	25	25*	18
		10	15	18	21	25	25*

① $r[0]$ 的作用?

① 暂存单元

② 监视哨

直接插入排序
算法是一种稳定的排序算法。

关键问题(1): 如何构造初始的有序序列?

解决方法:

将第1个记录看成是初始有序表, 然后从第2个记录起依次插入到这个有序表中, 直到将第 n 个记录插入。

算法描述:

```
for (i=2; i<=n; i++)  
{  
    插入第i个记录, 即第i趟直接插入排序;  
}
```

关键问题(2): 如何查找待插入记录的插入位置?

解决方法:

在 $i-1$ 个记录的有序区 $r[1] \sim r[i-1]$ 中插入记录 $r[i]$, 首先顺序查找 $r[i]$ 的正确插入位置, 然后将 $r[i]$ 插入到相应位置。

算法描述:

```
r[0]=r[i];  j=i-1;  
while (r[0]<r[j])  
{  
    r[j+1]=r[j];  
    j--;  
}
```

直接插入排序算法

```
void insertSort (int r[ ], int n)
{
    for (i=2; i<=n; i++)
    {
        r[0]=r[i]; j=i-1;
        while (r[0]<r[j])
        {
            r[j+1]=r[j];
            j=j-1;
        }
        r[j+1]=r[0];
    }
}
```

直接插入排序算法性能分析

最好情况下（正序）：

{ 比较次数： $n-1$
移动次数： $n-1$

时间复杂度为 $O(n)$ 。



直接插入排序算法性能分析

最好情况下（正序）：

比较次数： $n-1$
移动次数： $n-1$

时间复杂度为 $O(n)$ 。

最坏情况下（逆序或反序）：

比较次数： $\sum_{i=2}^n i = \frac{(n+2)(n-1)}{2}$
移动次数： $\sum_{i=2}^n (i+1) = \frac{(n+4)(n-1)}{2}$

时间复杂度为 $O(n^2)$ 。



直接插入排序算法性能分析

平均情况下（随机排列）：

$$\left\{ \begin{array}{l} \text{比较次数: } \sum_{i=2}^n \frac{i}{2} = \frac{(n+2)(n-1)}{4} \\ \text{移动次数: } \sum_{i=2}^n \frac{(i+1)}{2} = \frac{(n+4)(n-1)}{4} \end{array} \right.$$

时间复杂度为 $O(n^2)$ 。

直接插入排序算法性能分析

空间性能： 需要一个记录的辅助空间

稳定性： 直接插入排序算法是一种**稳定的**

结论：

- 直接插入排序算法简单、容易实现，**适用于待排序记录基本有序或待排序记录较少时。**
- 但是，当待排序的记录个数较多时，大量的比较和移动操作使直接插入排序算法的效率降低。

直接插入排序算法的改进

注意到，在插入第 i ($i > 1$) 个记录时，前面的 $i-1$ 个记录已经排好序，则在寻找插入位置时，
可以用折半查找来代替顺序查找，从而较少比较次数。

Do it by yourself

希尔排序(Shell Sort)

希尔排序的着眼点：

(1) 若待排序记录按关键码**基本有序**时，直接插入排序的效率可以大大提高；

(2) 由于直接插入排序算法简单，则在待排序记录数量 **n 较小**时效率也很高。

希尔排序

基本思想：

将整个待排序记录分割成若干个子序列，在子序列内分别进行直接插入排序，待整个序列中的记录基本有序时，对全体记录进行直接插入排序。

希尔排序

① 分割待排序记录的目的?

1. 减少待排序记录个数
2. 使整个序列向基本有序发展

基本有序：接近正序，例如{1, 2, 8, 4, 5, 6, 7, 3, 9}

局部有序：部分有序，例如{6, 7, 8, 9, 1, 2, 3, 4, 5}

局部有序不能提高直接插入排序算法的时间性能。

① 启示?

子序列的构成不能是简单地“逐段分割”，而是将相距某个“增量”的记录组成一个子序列。

希尔排序

① 需解决的关键问题：

- (1) 应如何分割待排序记录，才能保证整个序列逐步向基本有序发展？
- (2) 子序列内如何进行直接插入排序？

希尔插入排序过程示例

	1	2	3	4	5	6	7	8	9
初始序列	40	25	49	25*	16	21	08	30	13
d = 4	40	25	49	25*	16	21	08	30	13
	13	21	08	25*	16	25	49	30	40
d = 2	13	21	08	25*	16	25	49	30	40
	08	21	13	25*	16	25	40	30	49
d = 1	08	21	13	25*	16	25	40	30	49
	08	13	16	21	25*	25	30	40	49

关键问题(1): *应如何分割待排序记录?*

解决方法:

将相隔某个“增量”的记录组成一个子序列。

增量应如何取?

希尔最早提出的方法是 $d_1=n/2$, $d_{i+1}=d_i/2$ 。

算法描述:

```
for (d=n/2; d>=1; d=d/2)
{
    以d为增量, 进行组内直接插入排序;
}
```


关键问题(2): 子序列内如何进行直接插入排序?

解决方法:

- 在插入记录 $r[i]$ 时, 自 $r[i-d]$ 起往前跳跃式 (跳跃幅度为 d) 搜索待插入位置, 并且 $r[0]$ 只是暂存单元, 不是哨兵。当搜索位置 <0 , 表示插入位置已找到。
- 在搜索过程中, 记录后移也是跳跃 d 个位置。
- 在整个序列中, 前 d 个记录分别是 d 个子序列中的第一个记录, 所以从第 $d+1$ 个记录开始进行插入。

关键问题(2) 子序列内如何进行直接插入排序？

算法描述：

```
for (i=d+1; i<=n; i++) //将r[i]插入到所属的子序列中
{
    r[0]=r[i];           //暂存待插入记录
    j=i-d;               //j指向所属子序列的最后一个记录
    while (j>0 && r[0]<r[j])
    {
        r[j+d]=r[j];    //记录后移d个位置
        j=j-d;          //比较同一子序列的前一个记录
    }
    r[j+d]=r[0];
}
```

希尔排序算法的时间性能

希尔排序开始时增量较大，每个子序列中的记录个数较少，从而排序速度较快；当增量较小时，虽然每个子序列中记录个数较多，但整个序列已基本有序，排序速度也较快。

希尔排序算法的时间性能是所取增量的函数，而到目前为止尚未有人求得一种最好的增量序列。

研究表明，希尔排序的时间性能在 $O(n^2)$ 和 $O(n\log_2 n)$ 之间。当 n 在某个特定范围内，希尔排序所需的比较次数和记录的移动次数约为 $O(n^{1.3})$ 。

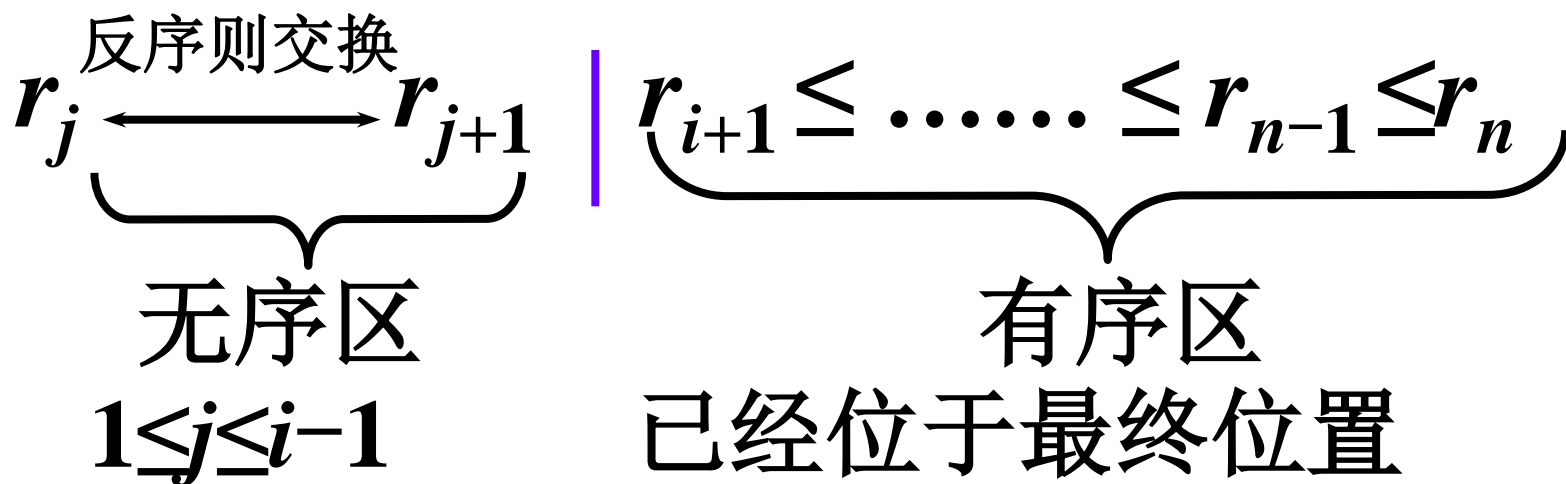
8.3 交换排序

交换排序的主要操作是**交换**，其主要思想是：在待排序列中选**两个**记录，将它们的关键码相比较，如果**反序**（即排列顺序与排序后的次序正好相反），则**交换**它们的存储位置。

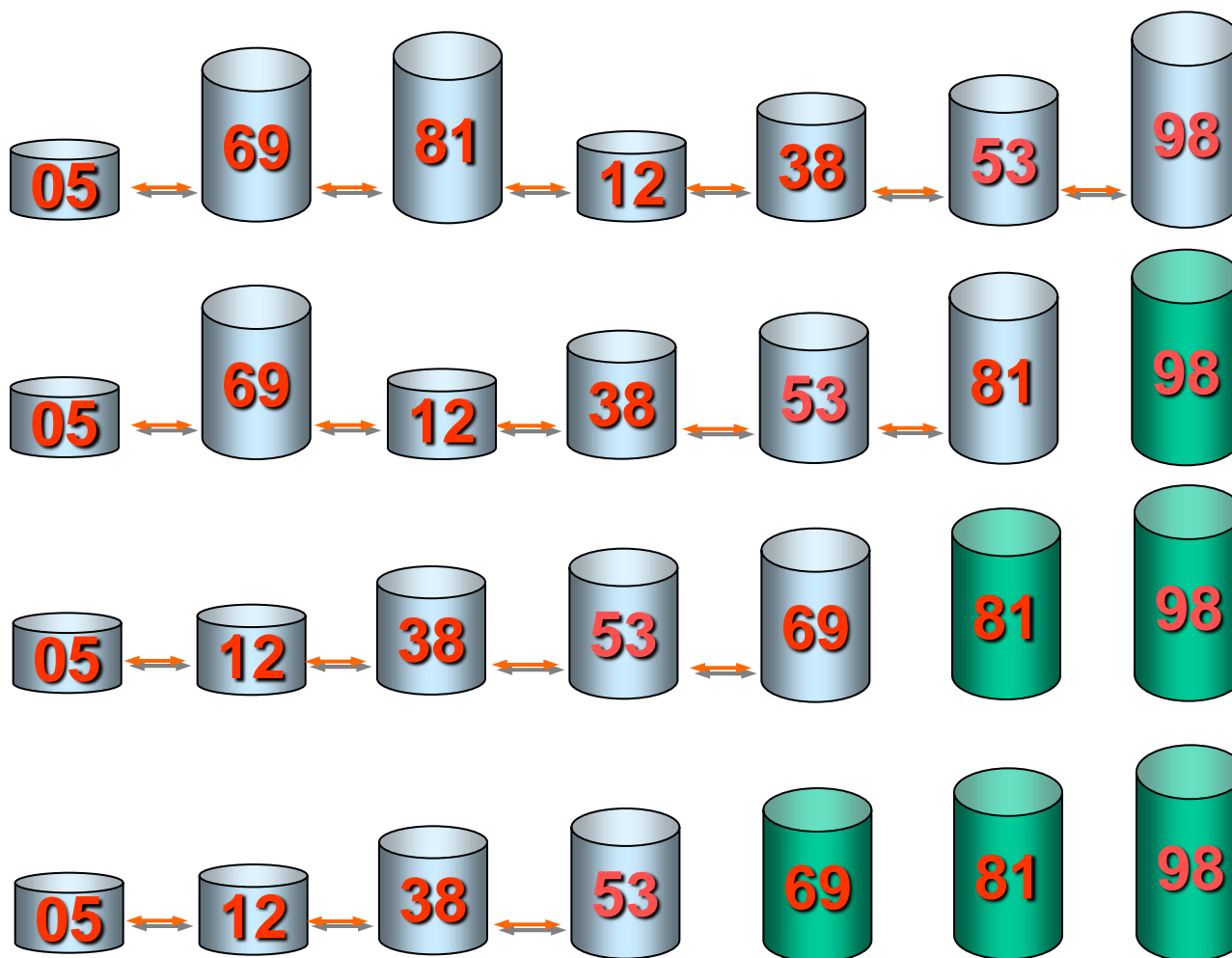


起泡排序(The Bubble Sort)

基本思想：两两比较**相邻**记录的关键码，如果反序则交换，直到没有反序的记录为止。



起泡排序过程示例



起泡排序

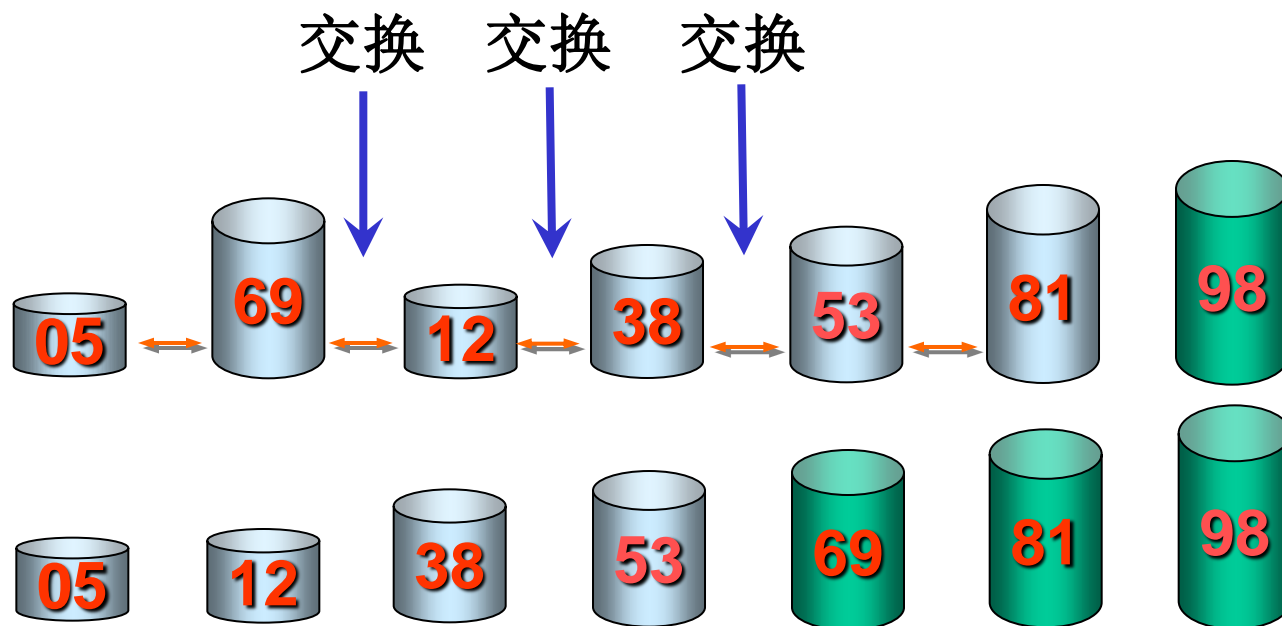
① 需解决的关键问题：

- (1) 在一趟起泡排序中，若有多个记录发生交换，应如何记载？
- (2) 如何确定起泡排序的范围，使得已经位于最终位置的记录不参与下一趟排序？
- (3) 如何判别起泡排序的结束？

关键问题(1): 如何记载一趟排序过程中的多次交换?

解决方法:

设变量exchange记载记录交换的位置, 则一趟排序后, exchange记载的一定是这一趟排序中记录的最后一次交换的位置, 且从此位置以后的所有记录均已经有序。



关键问题(1): 如何记载一趟排序过程中的多次交换?

解决方法:

设变量exchange记载记录交换的位置, 则一趟排序后, exchange记载的一定是这一趟排序中记录的最后一次交换的位置, 且从此位置以后的所有记录均已经有序。

算法描述:

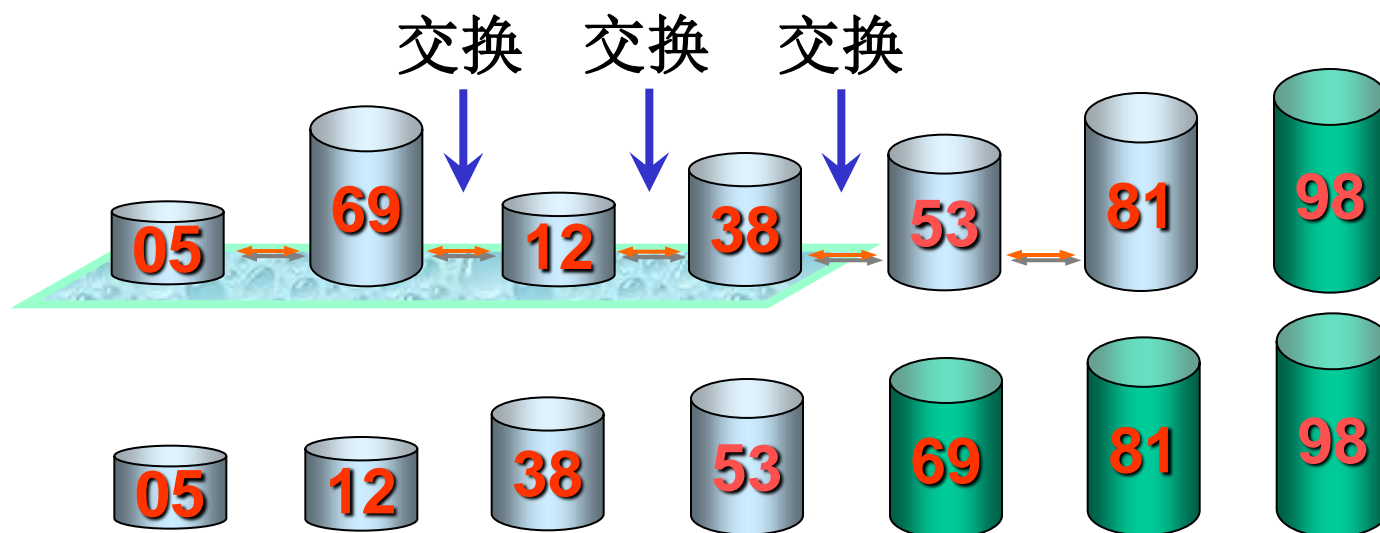
```
if (r[j]>r[j+1]) {  
    r[j]↔r[j+1];  
    exchange=j;  
}
```

关键问题(2): 如何确定起泡排序的范围?

解决方法:

设bound位置的记录是无序区的最后一个记录, 则每趟起泡排序的范围是 $r[1] \sim r[\text{bound}]$ 。

在一趟排序后, 从exchange位置之后的记录一定是有顺序的, 所以 $\text{bound} = \text{exchange}$ 。



关键问题(2): 如何确定起泡排序的范围?

解决方法:

设bound位置的记录是无序区的最后一个记录, 则每趟起泡排序的范围是 $r[1] \sim r[\text{bound}]$ 。

在一趟排序后, 从exchange位置之后的记录一定是有顺序的, 所以 $\text{bound}=\text{exchange}$ 。

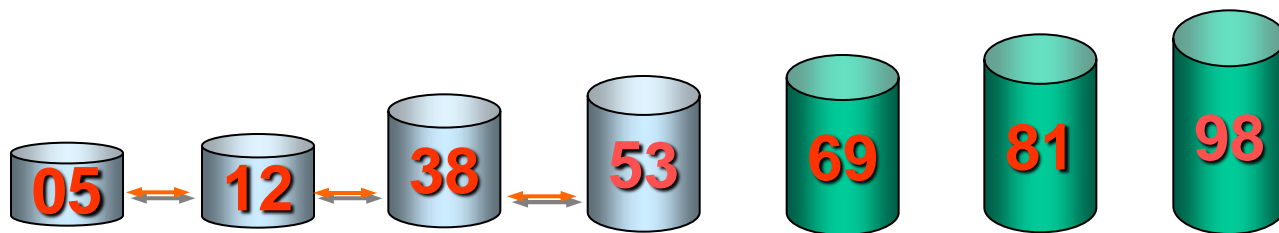
算法描述:

```
bound=exchange;  
for (j=1; j<bound; j++)  
    if (r[j]>r[j+1]) {  
        r[j]<=>r[j+1];  
        exchange=j;  
    }
```

关键问题(3): 如何判别起泡排序的结束?

解决方法:

在每一趟起泡排序之前, 令**exchange**的初值为**0**, 在以后的排序过程中, 只要有记录交换, **exchange**的值就会大于**0**。这样, 在一趟比较完毕, 就可以通过**exchange**的值是否为**0**来判别是否有记录交换, 从而判别整个起泡排序的结束。



关键问题(3): 如何判别起泡排序的结束?

解决方法:

在每一趟起泡排序之前, 令`exchange`的初值为0, 在以后的排序过程中, 只要有记录交换, `exchange`的值就会大于0。这样, 在一趟比较完毕, 就可以通过`exchange`的值是否为0来判别是否有记录交换, 从而判别整个起泡排序的结束。

算法描述:

```
while (exchange)
{
    执行一趟起泡排序;
}
```

起泡排序算法

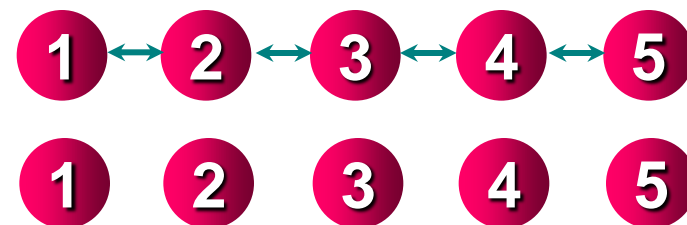
```
void BubbleSort (int r[ ], int n)
{
    exchange=n;
    while (exchange)
    {
        bound=exchange;
        exchange=0;
        for (j=1; j<bound; j++)
            if (r[j]>r[j+1]) {
                r[j]←→r[j+1];
                exchange=j;
            }
    }
}
```

起泡排序的时间性能分析

最好情况（正序）：

{ 比较次数： $n-1$
移动次数： 0

时间复杂度为 $O(n)$ 。



起泡排序的时间性能分析

最好情况（正序）：

{ 比较次数： $n-1$
移动次数： 0

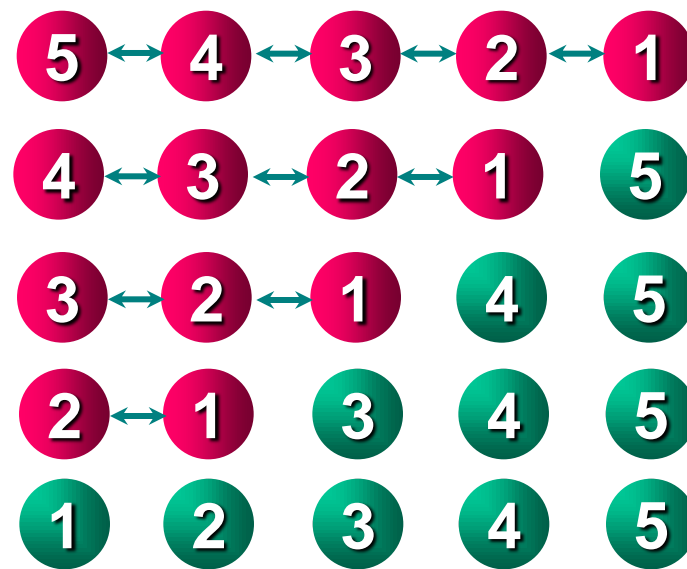
时间复杂度为 $O(n)$ ；

最坏情况（反序）：

{ 比较次数： $\sum_{i=1}^{n-1} (n-i) = \frac{n(n-1)}{2}$
移动次数： $\sum_{i=1}^{n-1} 3(n-i) = \frac{3n(n-1)}{2}$

时间复杂度为 $O(n^2)$ 。

平均情况：时间复杂度为 $O(n^2)$ 。



快速排序(QuickSort)

改进的着眼点：在起泡排序中，记录的比较和移动是在**相邻**单元中进行的，记录每次交换只能上移或下移**一个**单元，因而总的比较次数和移动次数较多。

减少总的比较次数和移动次数



增大记录的比较和移动距离



较大记录从前面直接移动到后面
较小记录从后面直接移动到前面

快速排序的基本思想

首先选一个轴值（即比较的基准），通过一趟排序将待排序记录分割成独立的两部分，前一部分记录的关键码均小于或等于轴值，后一部分记录的关键码均大于或等于轴值，然后分别对这两部分重复上述方法，直到整个序列有序。

① 需解决的关键问题？

- (1) 如何选择轴值？
- (2) 如何实现分割（称一次划分）？
- (3) 如何处理分割得到的两个待排序子序列？
- (4) 如何判别快速排序的结束？

关键问题(1): 如何选择轴值?

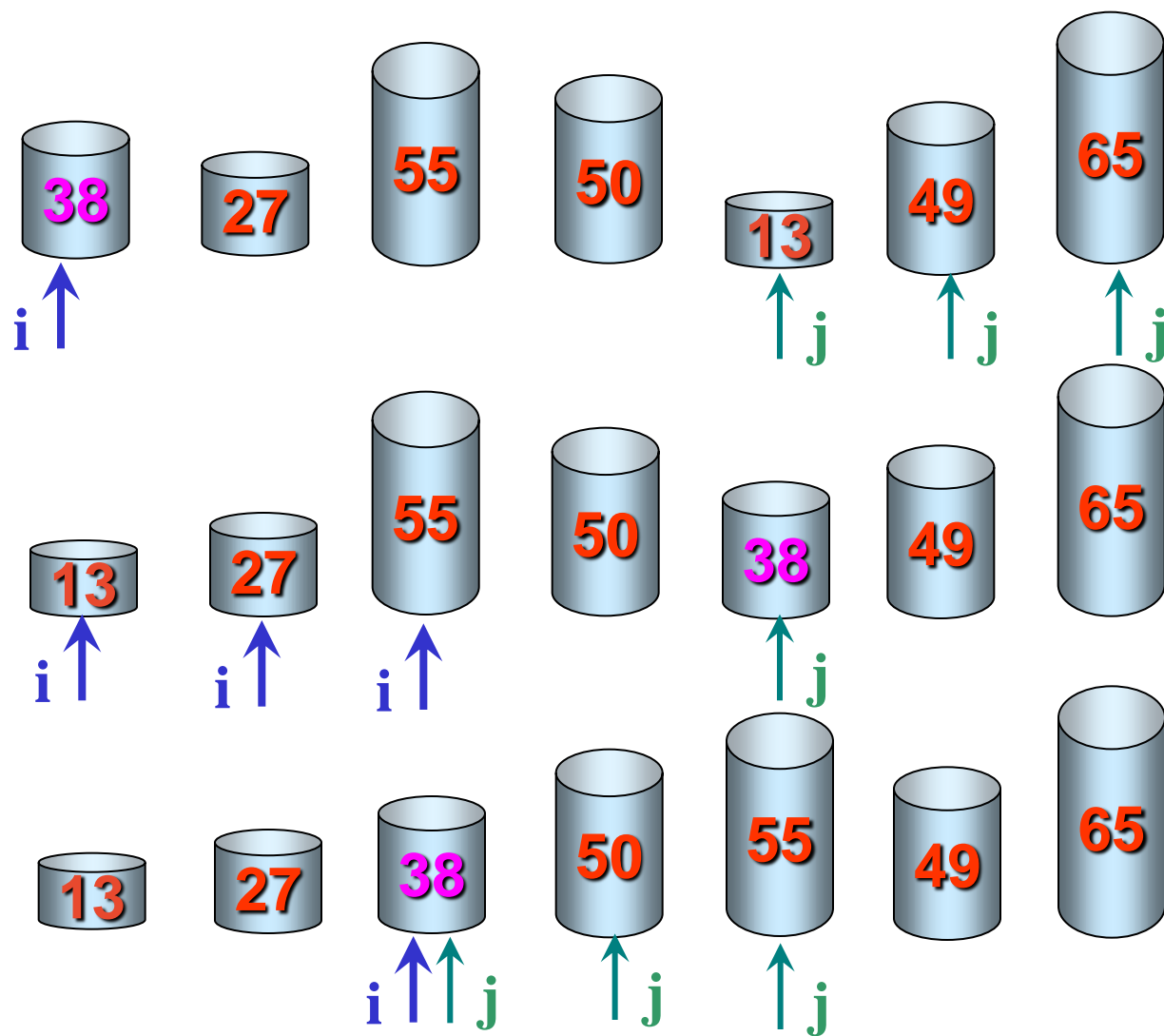
选择轴值的方法:

- 1.使用第一个记录的关键码
- 2.选取序列中间记录的关键码
- 3.比较序列中第一个记录、最后一个记录和中间记录的关键码, 取关键码居中的作为轴值并调换到第一个记录的位置
- 4.随机选取轴值

选取不同轴值的后果:

决定两个子序列的长度。当然子序列的长度最好相等。

关键问题(2): 如何实现一次划分?



关键问题(2): 如何实现一次划分?

解决方法:

设待划分的序列是 $r[s] \sim r[t]$, 设参数 i, j 分别指向子序列左、右两端的下标 s 和 t , 令 $r[s]$ 为轴值,

(1) j 从后向前扫描, 直到 $r[j] < r[i]$, 将 $r[j]$ 移动到 $r[i]$ 的位置, 使关键码小 (同轴值相比) 的记录移动到前面去;

(2) i 从前向后扫描, 直到 $r[i] > r[j]$, 将 $r[i]$ 移动到 $r[j]$ 的位置, 使关键码大 (同轴值比较) 的记录移动到后面去;

(3) 重复上述过程, 直到 $i=j$ 。

关键问题(2): 如何实现一次划分?

算法描述:

```

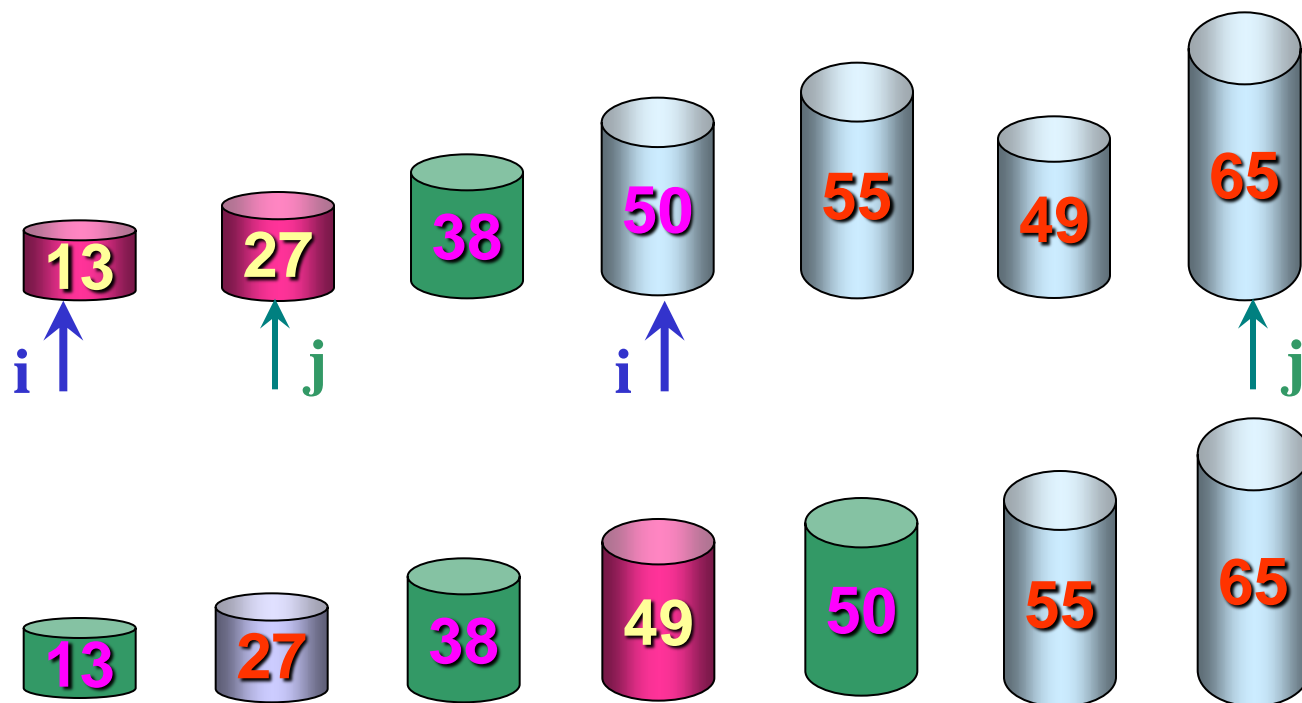
int Partition(int r[ ], int first, int end)
{
    i=first; j=end;    //初始化
    while (i<j)
    {
        while (i<j && r[i]<= r[j]) j--;    //右侧扫描
        if (i<j) {
            r[i]←→r[j]; i++; //将较小记录交换到前面
        }
        while (i<j && r[i]<= r[j]) i++;    //左侧扫描
        if (i<j) {
            r[j]←→r[i]; j--; //将较大记录交换到后面
        }
    }
    return i;    //i为轴值记录的最终位置
}

```

关键问题(3): 如何处理分割得到的两个待排序子序列?

解决方法:

对分割得到的两个子序列递归地执行快速排序。



关键问题(3): 如何处理分割得到的两个待排序子序列?

算法描述:

```
void QuickSort (int r[ ], int first, int end )  
{  
    pivotpos = Partition (r, first, end ); //一次划分  
    QuickSort (r, first, pivotpos-1);  
    //对前一个子序列进行快速排序  
    QuickSort (r, pivotpos+1, end );  
    //对后一个子序列进行快速排序  
}
```


关键问题(4): 如何判别快速排序的结束?

解决方法:

若待排序列中只有一个记录, 显然已有序, 否则进行一次划分后, 再分别对分割所得的两个子序列进行快速排序 (即递归处理)。

关键问题(4): 如何判别快速排序的结束?

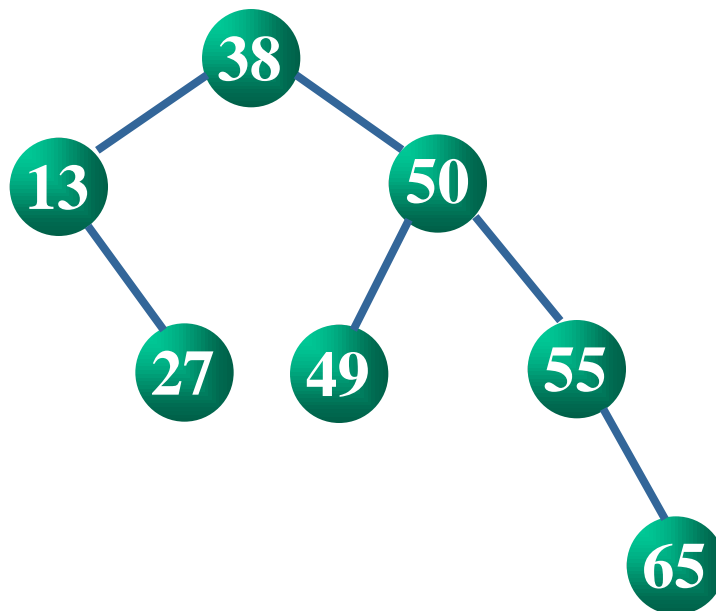
算法描述:

```
void QuickSort (int r[ ], int first, int end )  
{//在序列 first~end中递归地进行快速排序  
  if (first < end) {  
    pivotpos = Partition (r, first, end );  
    QuickSort (r, first, pivotpos-1);  
    QuickSort (r, pivotpos+1, end );  
  }  
}
```

快速排序的时间性能分析

快速排序的递归执行过程可以用**递归树**描述。

例：{38, 27, 55, 50, 13, 49, 65}的快速排序递归树如下：



快速排序的时间性能分析

最好情况：

每一次划分对一个记录定位后，该记录的左侧子表与右侧子表的长度相同，为 $O(n\log_2 n)$ 。

最坏情况：

每次划分只得到一个比上一次划分少一个记录的子序列（另一个子序列为空），为 $O(n^2)$ 。

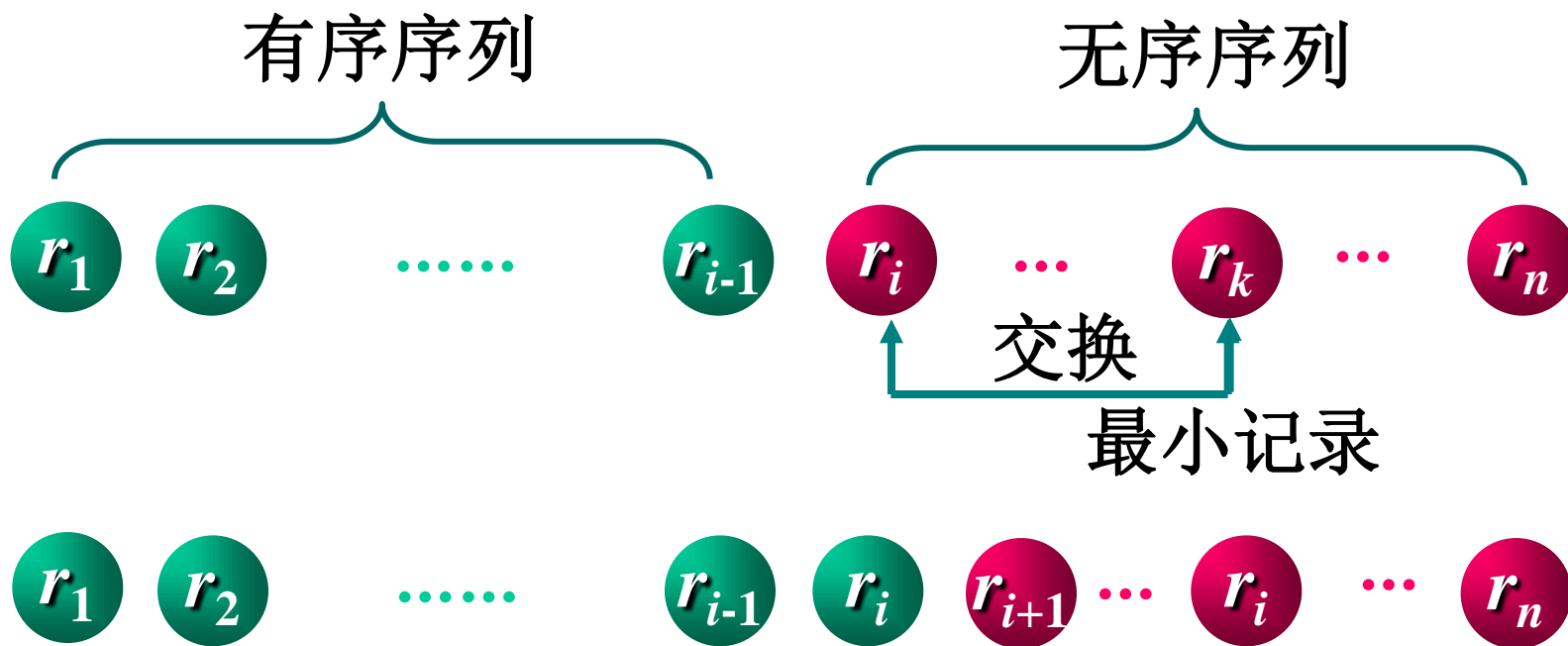
$$\sum_{i=1}^{n-1} (n-i) = \frac{1}{2} n(n-1) = O(n^2)$$

平均情况：近似为 $O(n\log_2 n)$ 。

8.4 选择排序

选择排序的主要操作是**选择**，其主要思想是：

每趟排序在当前待排序序列中选出关键码**最小**的记录，添加到有序序列中。



简单选择排序 (The Simple Selection Sort)

基本思想:

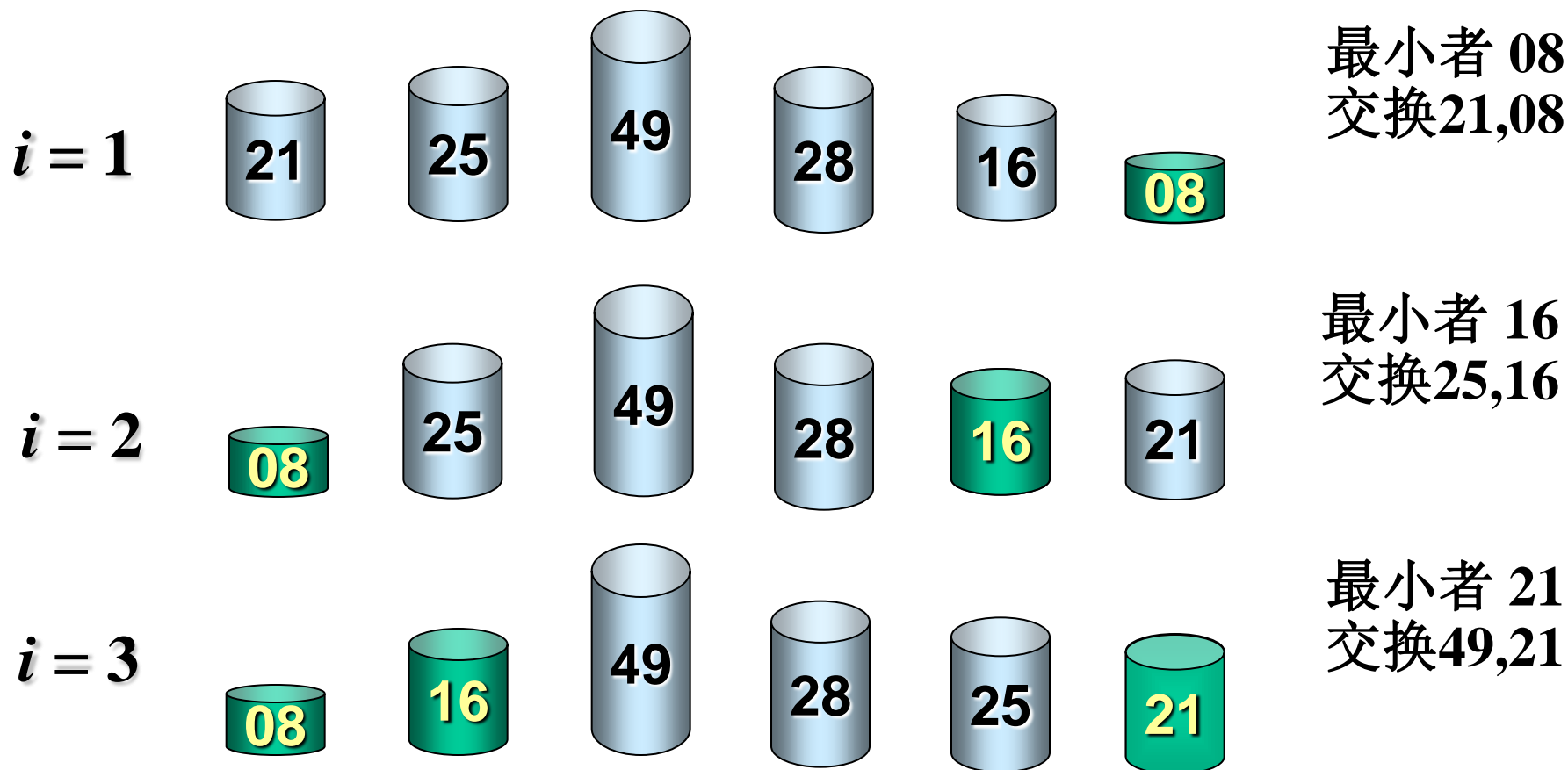
第 i 趟在 $n-i+1$ ($i=1,2,\dots,n-1$) 个记录中选取关键码最小的记录作为有序序列中的第 i 个记录。

① 需解决的关键问题?

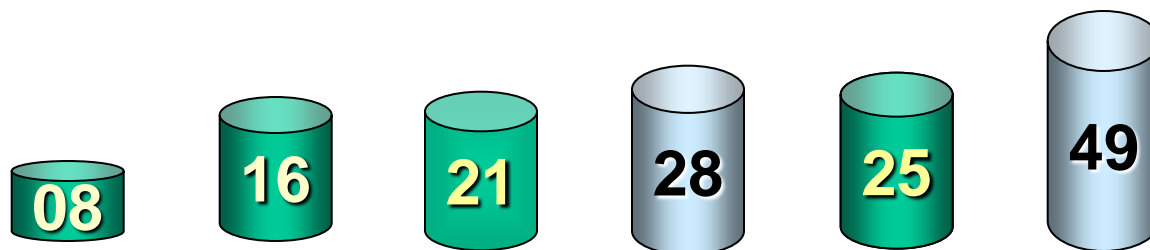
(1)如何在待排序序列中选出关键码最小的记录?

(2)如何确定待排序序列中关键码最小的记录在有序序列中的位置?

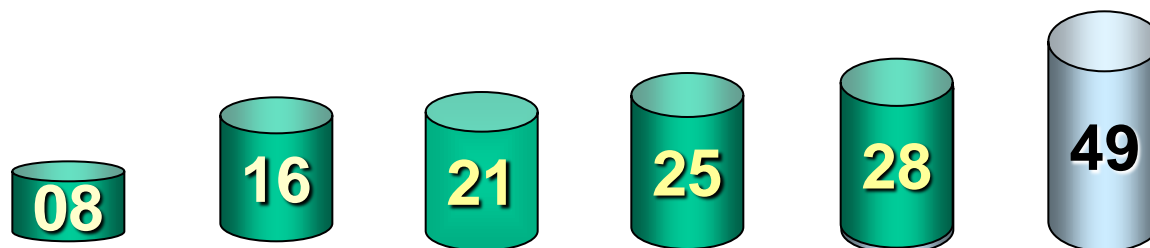
简单选择排序示例



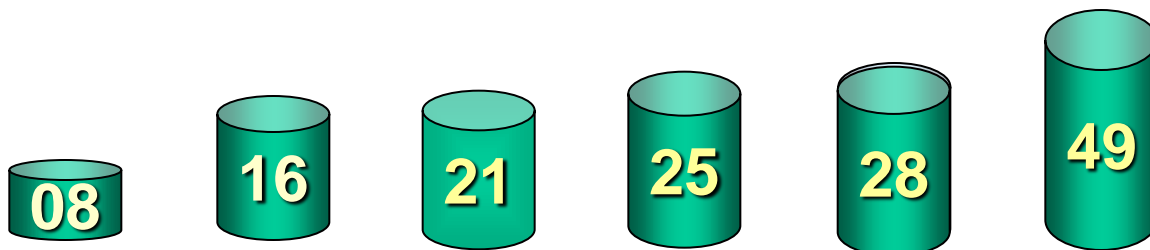
简单选择排序示例

 $i = 4$ 

最小者 25
交换25,28

 $i = 5$ 

最小者 28
不交换



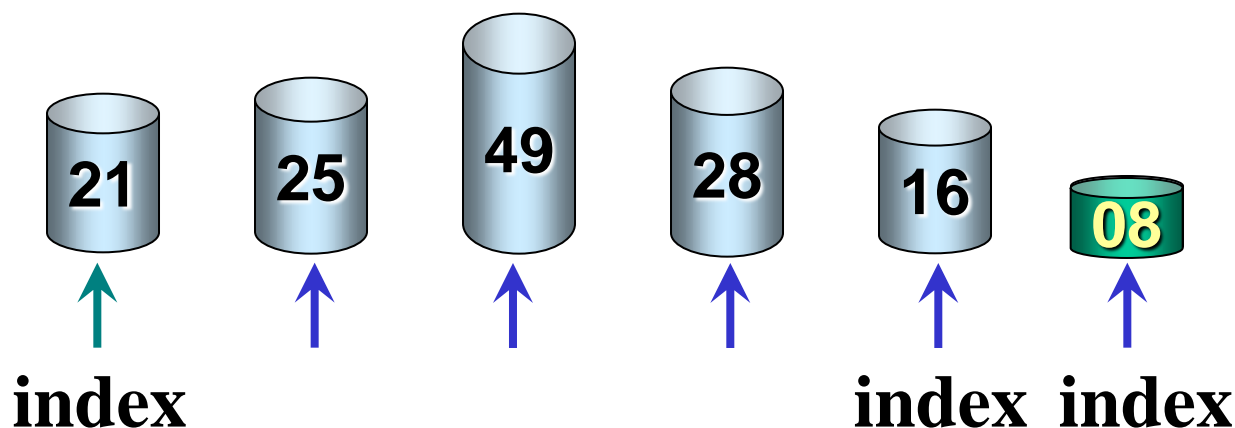
无序区只有
一个记录

跟初始状态有关系吗?

关键问题(1): 如何在无序区中选出关键码最小的记录?

解决方法:

设置一个整型变量**index**, 用于记录在一趟比较的过程中关键码最小的记录位置。



关键问题(1): 如何在无序区中选出关键码最小的记录?

解决方法:

设置一个整型变量`index`, 用于记录在一趟比较的过程中关键码最小的记录位置。

算法描述:

```
index=i;  
for (j=i+1; j<=n; j++)  
    if (r[j]<r[index]) index=j;
```

关键问题(2): 如何确定最小记录的最终位置?

解决方法:

第*i*趟简单选择排序的待排序区间是 $r[i] \sim r[n]$, 则 $r[i]$ 是无序区第一个记录, 所以, 将index所记载的关键码最小的记录与 $r[i]$ 交换。

算法描述:

if ($\text{index} \neq i$)

$r[i] \leftrightarrow r[\text{index}];$

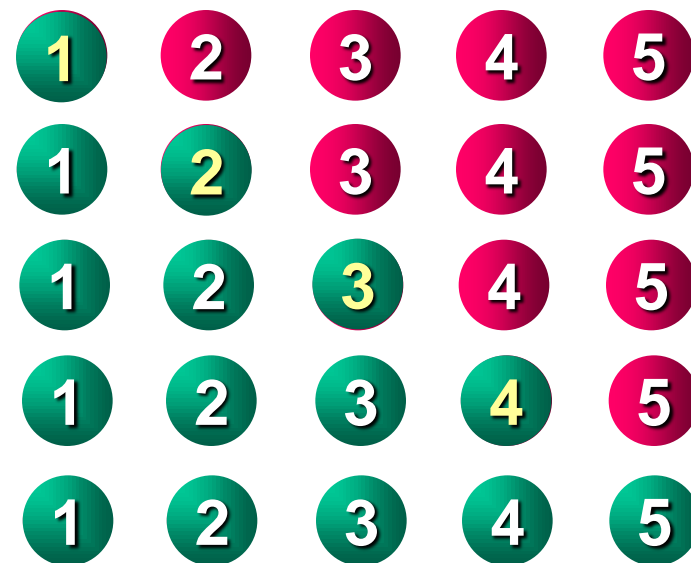
简单选择排序算法

```
void selectSort ( int  r[ ], int n)  
{  
    for ( i=1; i<n; i++)  
    {  
        index=i;  
        for (j=i+1; j<=n; j++)  
            if (r[j]<r[index]) index=j;  
        if (index!=i)  r[i]<==>r[index];  
    }  
}
```

简单选择排序算法的性能分析

移动次数：

最好情况（正序）：0次



简单选择排序算法的性能分析

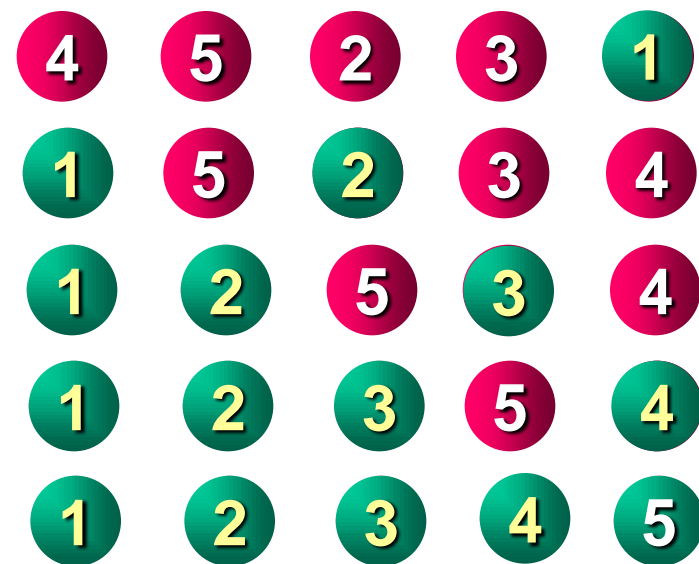
移动次数：

最好情况（正序）：0次

最坏情况： $3(n-1)$ 次

比较次数：

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



简单选择排序的时间复杂度为 $O(n^2)$ 。

空间性能：需一个辅助空间。

稳定性：简单选择排序是一种不稳定的排序算法。

堆排序(The Heap Sort)

改进的着眼点：如何减少关键码间的比较次数。

若能利用每趟比较后的结果，也就是在找出键值最小记录的同时，也找出键值较小的记录，则可减少后面的选择中所用的比较次数，从而提高整个排序过程的效率。

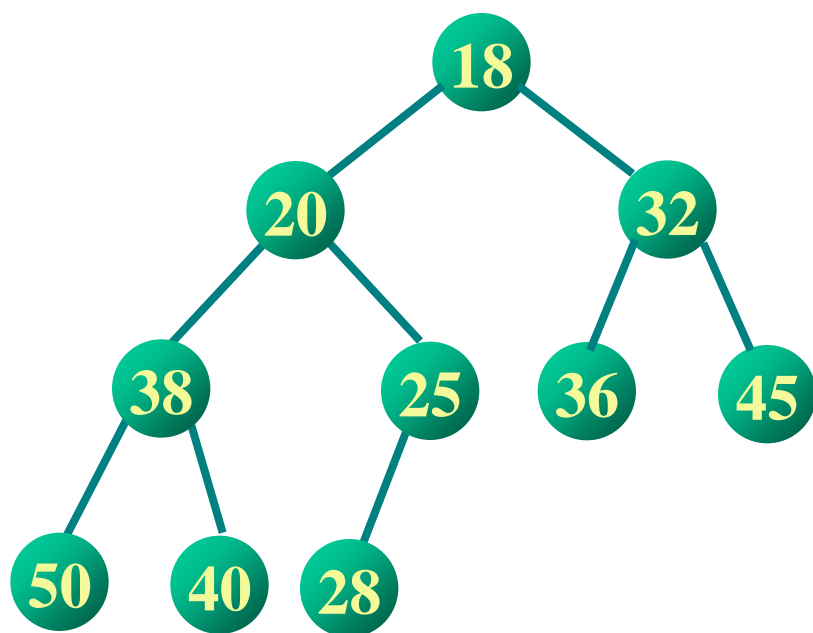
减少关键码间的比较次数




查找最小值的同时，找出较小值

堆的定义

堆是具有下列性质的完全二叉树：每个结点的值都小于或等于其左右孩子结点的值（称为**小根堆**），或每个结点的值都大于或等于其左右孩子结点的值（称为**大根堆**）。

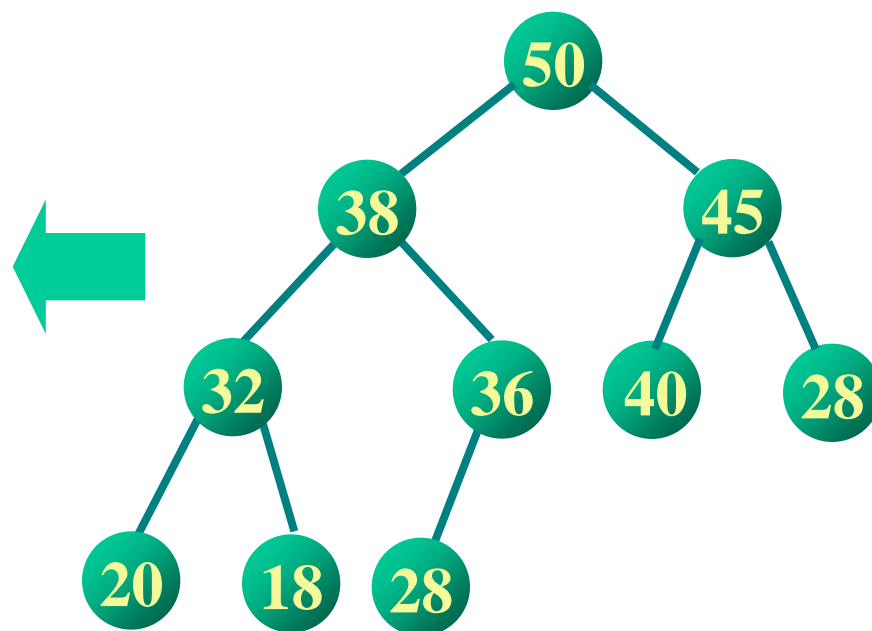


- 
1. 小根堆的根结点是所有结点的最小者
 2. 较小结点靠近根结点

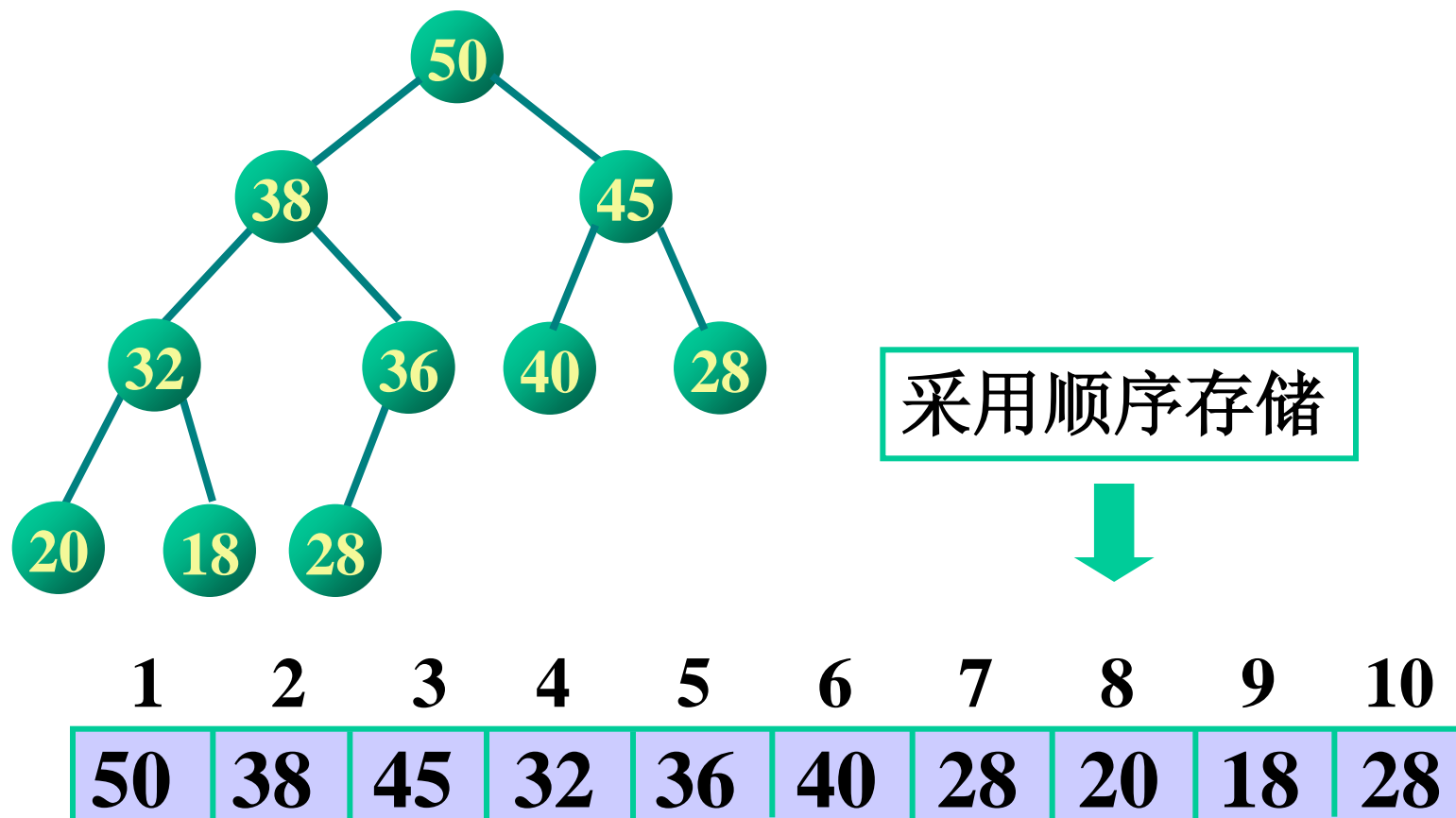
堆的定义

堆是具有下列性质的**完全二叉树**：每个结点的值都小于或等于其左右孩子结点的值（称为**小根堆**），或每个结点的值都大于或等于其左右孩子结点的值（称为**大根堆**）。

1. 大根堆的根结点是所有结点的最大者
2. 较大结点靠近根结点



堆和序列的关系



可见，将堆用顺序存储结构来存储，则堆对应一组序列，满足 $A_i \leq A_{2i}$, $A_i \leq A_{2i+1}$, 或 $A_i \geq A_{2i}$, $A_i \geq A_{2i+1}$

堆排序

基本思想：

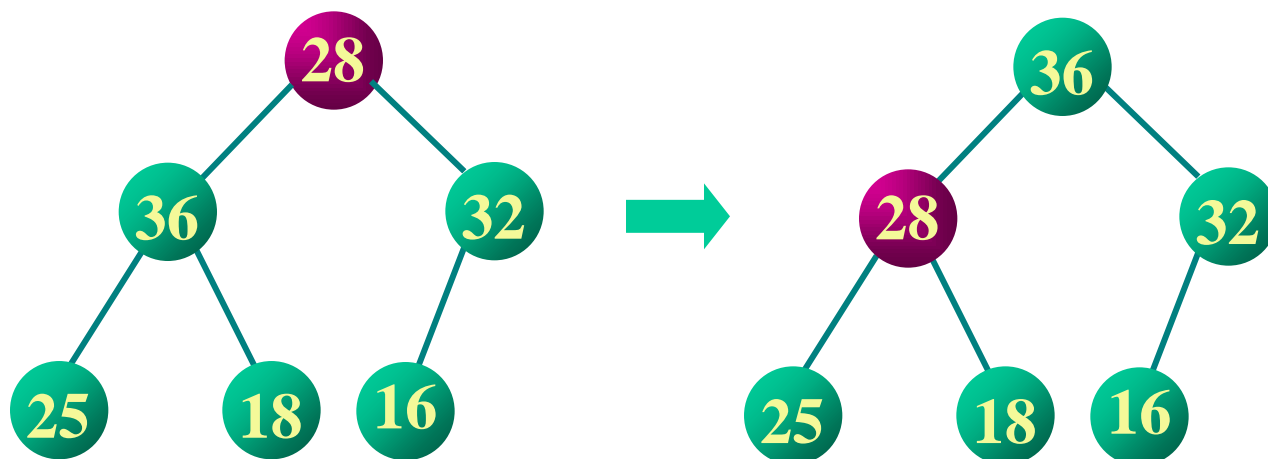
首先将待排序的记录序列**构造成一个堆**，此时，选出了堆中所有记录的最大者，然后将它从堆中移走，并将剩余的记录**再调整成堆**，这样又找出了次小的记录，以此类推，直到堆中只有一个记录。

① 需解决的关键问题？

- (1)如何由一个无序序列建成一个堆（即初始建堆）？
- (2)如何处理堆顶记录？
- (3)如何调整剩余记录，成为一个新堆（即重建堆）？

堆调整

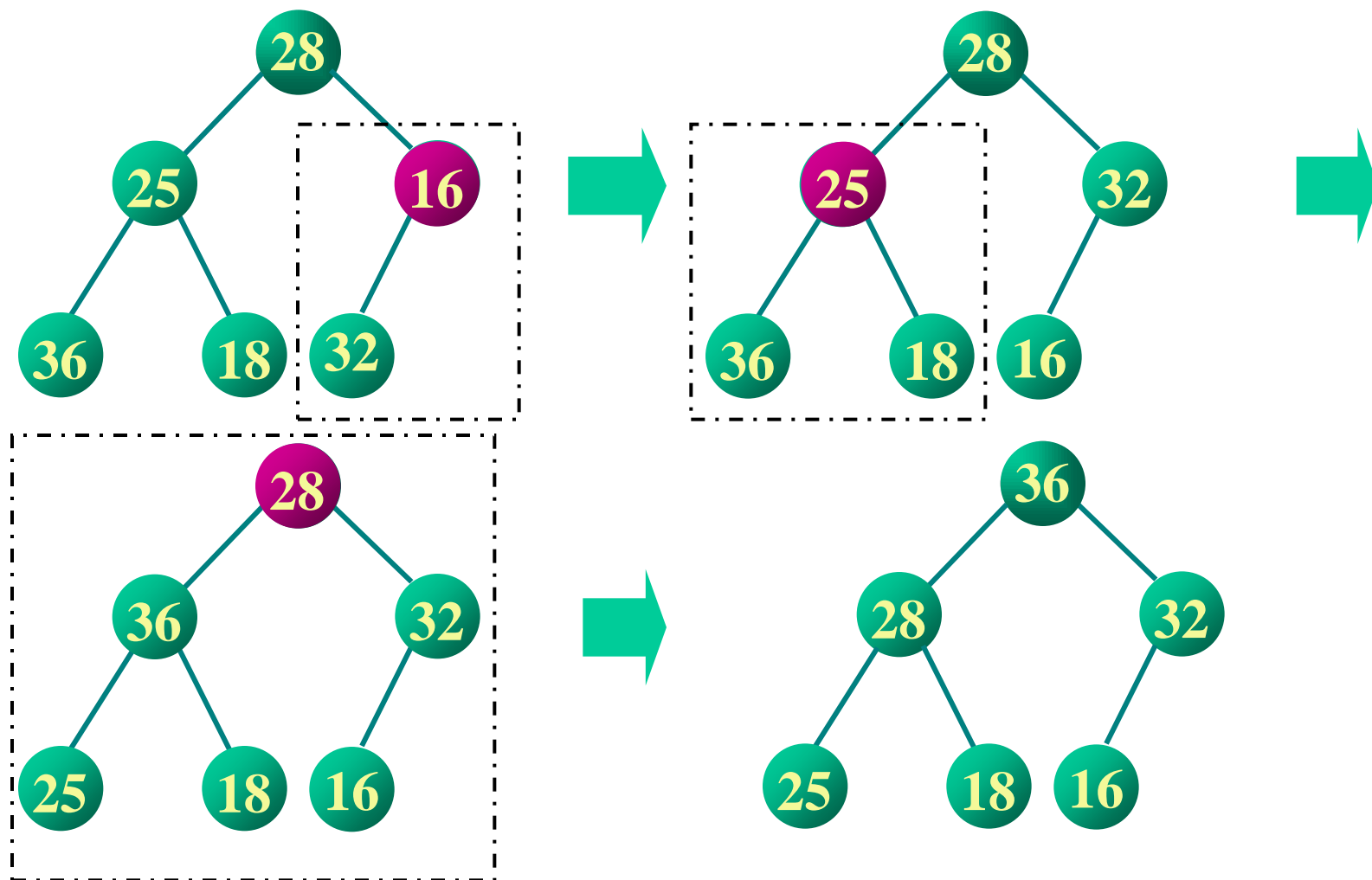
堆调整：在一棵完全二叉树中，若根结点的左右子树均是堆，如何调整根结点，使整个完全二叉树成为一个堆？



堆调整——算法描述:

```
void sift ( int r[ ], int k, int m )
{ //要筛选结点的编号为k, 堆中最后一个结点的编号为m
  i=k; j=2*i; //将i置为要筛选的结点, j为i的左孩子
  while (j<=m) //筛选还没有进行到叶子
  {
    if (j<m && r[j]<r[j+1]) j++; //左右孩子中取较大者
    if (r[i]>r[j]) break; //根结点已经大于左右孩子中的较大者
    else {
      r[i]←→r[j]; i=j; j=2*i;
    }
  }
}
```

关键问题(1): 如何由一个无序序列建成一个堆?



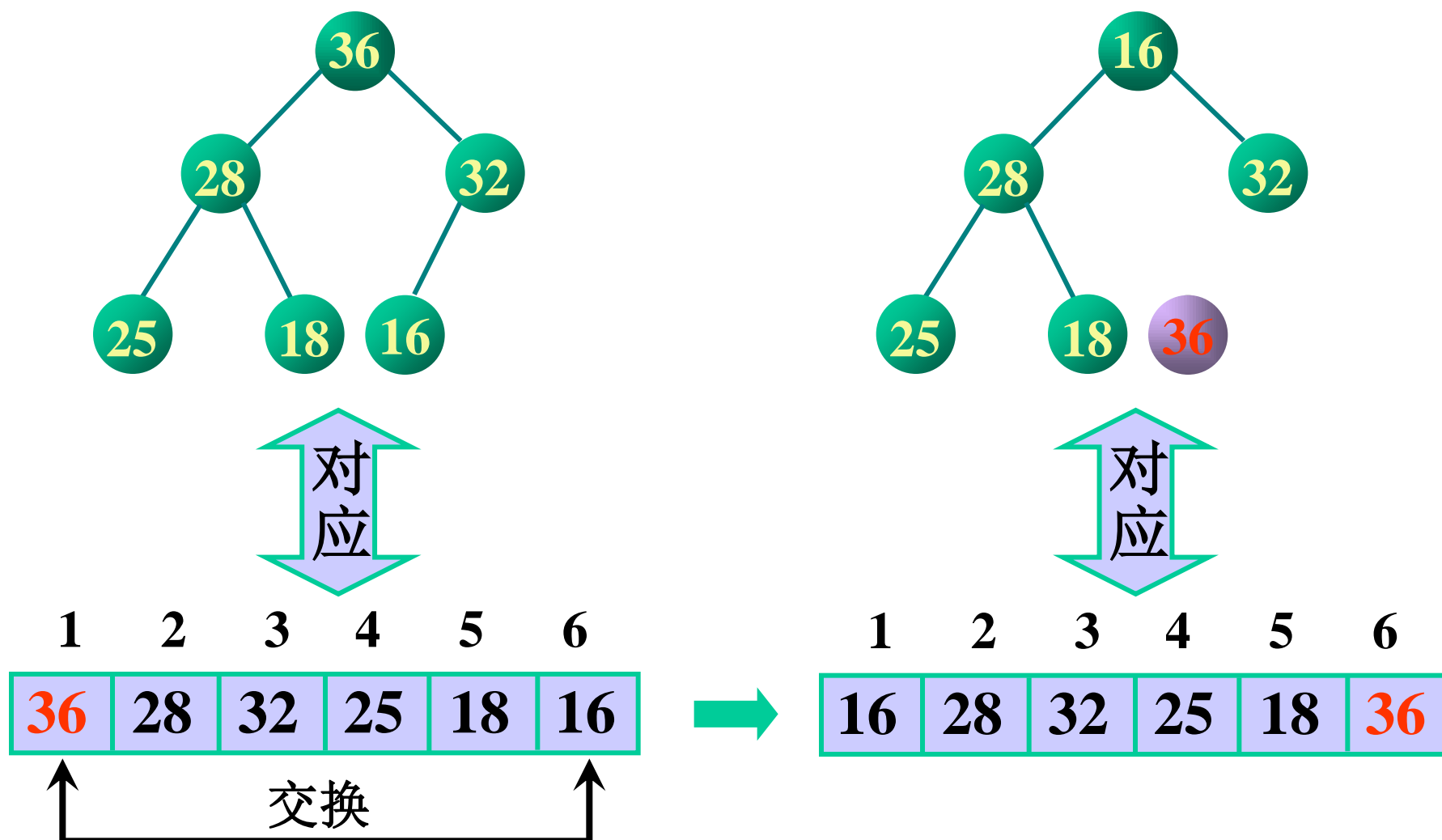
关键问题(1): 如何由一个无序序列建成一个堆?

算法描述:

```
for (i= $n/2$ ; i $\geq$ 1; i--)  
    sift(r, i, n);
```

最后一个结点（叶子）的序号是 n ，
则最后一个分支结点即为结点 n 的双亲，
其序号是 $n/2$ 。

关键问题(2): 如何处理堆顶记录?



关键问题(2): 如何处理堆顶记录?

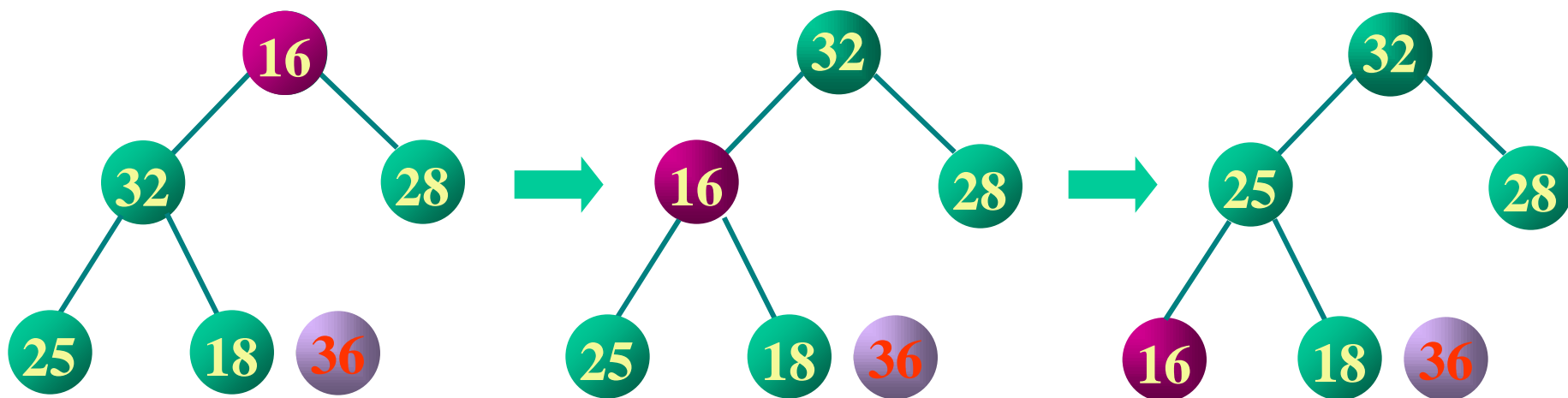
解决方法:

第 i 次处理堆顶是将堆顶记录 $r[1]$ 与序列中第 $n-i+1$ 个记录 $r[n-i+1]$ 交换。

算法描述:

$r[1] \leftrightarrow r[n-i+1];$

关键问题(3): 如何调整剩余记录, 成为一个新堆?



关键问题(3): 如何调整剩余记录, 成为一个新堆?

解决方法:

第 i 次调整剩余记录, 此时, 剩余记录有 $n-i$ 个, 调整根结点至第 $n-i$ 个记录。

算法描述:

sift(r, 1, n-i);

堆排序算法

```
void HeapSort ( int r[], int n)
{
    for (i=n/2; i>=1; i--)    //初建堆
        sift(r, i, n) ;
    for (i=1; i>n; i++ )
    {
        r[1]←→r[n-i+1];    //移走堆顶
        sift(r, 1, n-i);    //重建堆
    }
}
```

堆排序算法的性能分析

第1个for循环是**初始建堆**，需要 $O(n)$ 时间

第2个for循环是输出堆顶**重建堆**，第 i 次取堆顶记录**重建堆**需要 $O(\log_2 i)$ 时间，共需要取 $n-1$ 次堆顶记录，因此重建堆需要 $O(n\log_2 n)$ 时间

堆排序的**最好**、**最坏**和**平均**的时间代价为 $O(n\log_2 n)$

由于记录的比较和交换是跳跃进行的，因此，**堆排序是一种不稳定的排序方法**

堆排序**适用于**记录个数较多、原始记录任意排列的情况

8.5 归并排序

归并排序的主要操作是**归并**，其主要思想是：将若干有序序列逐步归并，最终得到一个有序序列。

归并：将两个或两个以上的有序序列合并成一个有序序列的过程。

二路归并排序

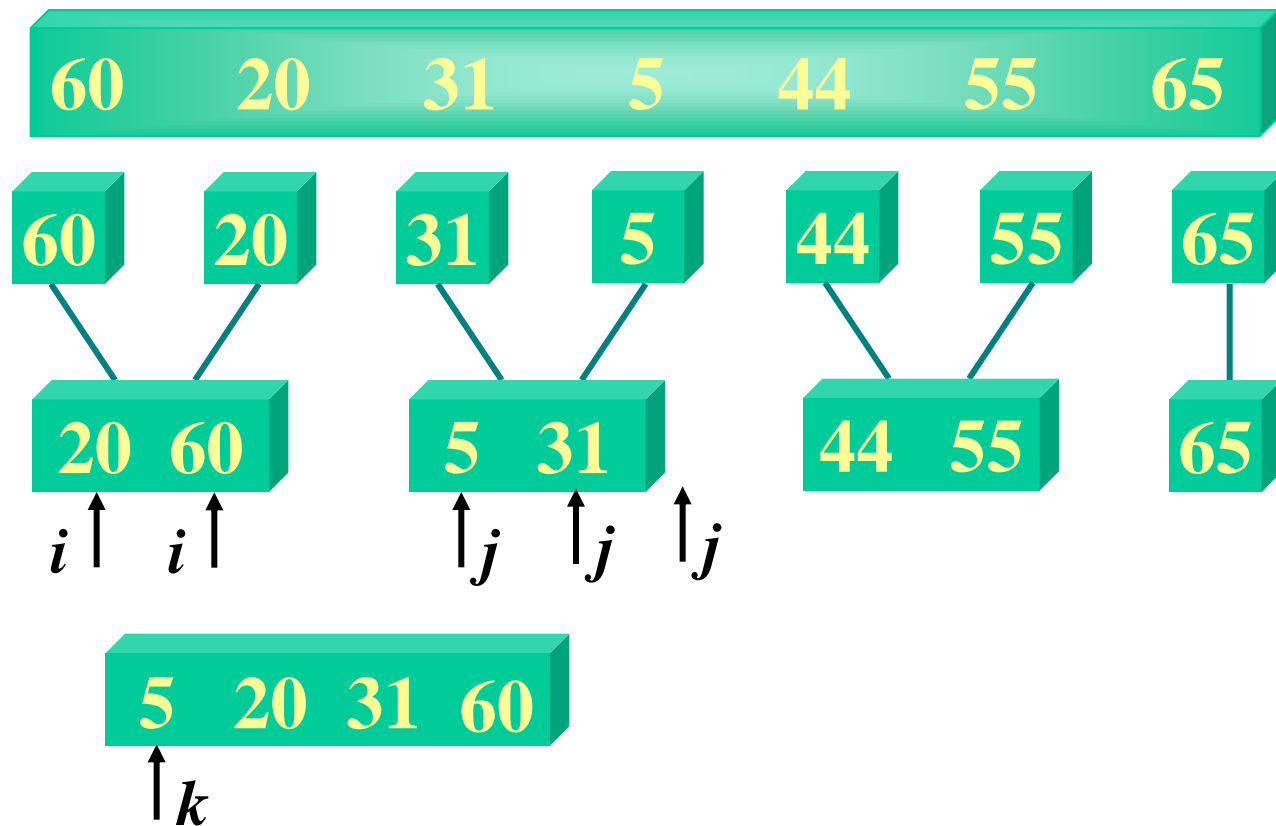
基本思想：

将一个具有 n 个待排序记录的序列看成是 n 个长度为1的有序序列，然后进行两两归并，得到 $n/2$ 个长度为2的有序序列，再进行两两归并，得到 $n/4$ 个长度为4的有序序列，……，直至得到一个长度为 n 的有序序列为止。

① 需解决的关键问题？

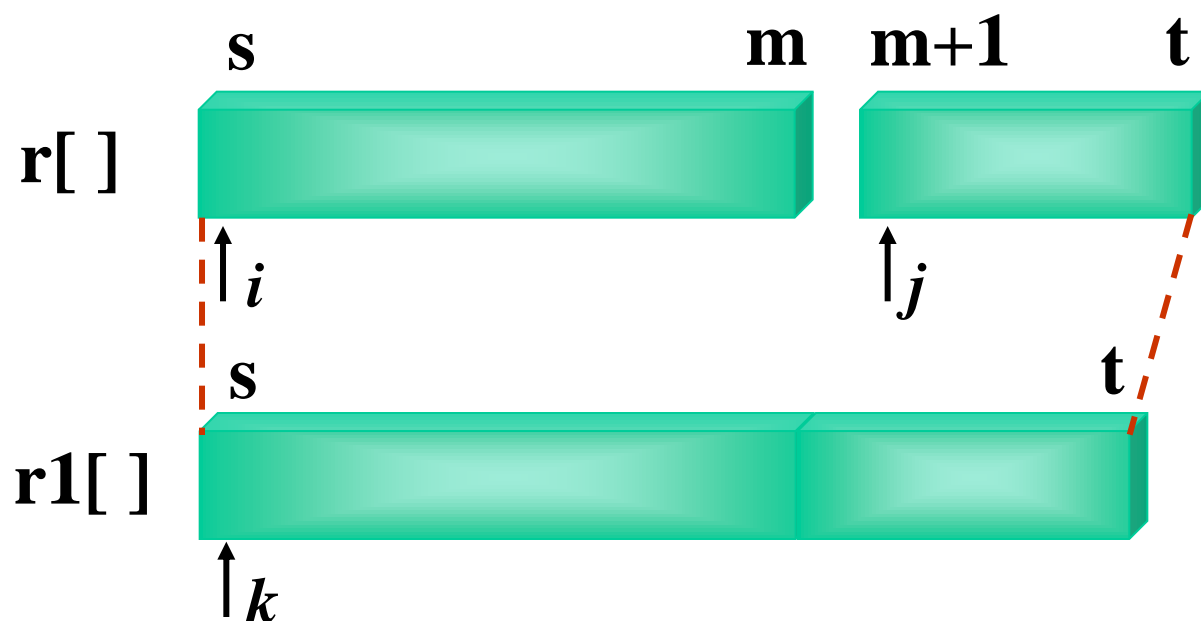
- (1) 如何将两个有序序列合成一个有序序列？
- (2) 怎样完成一趟归并？
- (3) 如何控制二路归并的结束？

关键问题(1): 如何将两个有序序列合成一个有序序列?



关键问题(1): 如何将两个有序序列合成一个有序序列?

设相邻的有序序列为 $r[s] \sim r[m]$ 和 $r[m+1] \sim r[t]$, 归并成一个有序序列 $r1[s] \sim r1[t]$

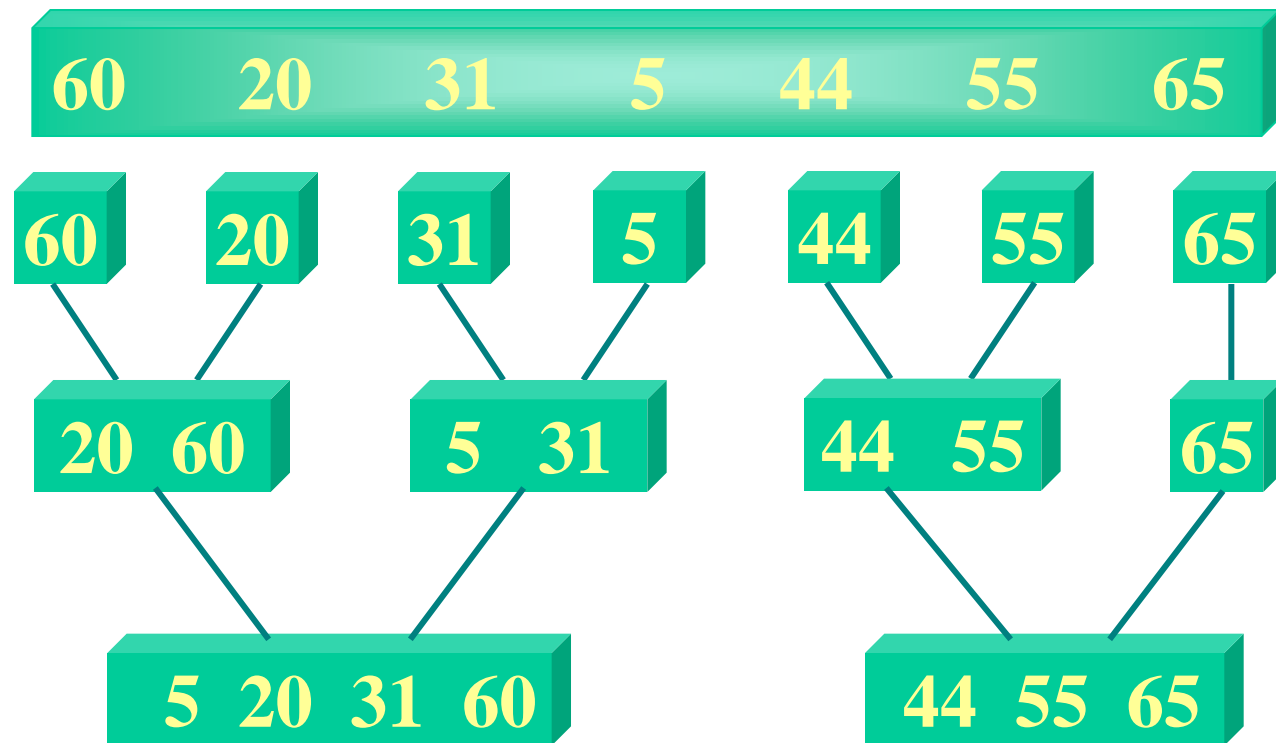


关键问题(1): 如何将两个有序序列合成一个有序序列?

算法描述:

```
void Merge (int r[ ], int r1[ ], int s, int m, int t )
{
    i=s; j=m+1; k=s;
    while (i<=m && j<=t)
    { //取r[i]和r[j]中的较小者放入r1[k]
        if (r[i]<=r[j]) r1[k++]=r[i++]; else r1[k++]=r[j++];
    }
    if (i<=m) while (i<=m) //收尾处理
        r1[k++]=r[i++]; //前一个子序列
    else while (j<=t)
        r1[k++]=r[j++]; //后一个子序列
}
```

关键问题(2): 怎样完成一趟归并?

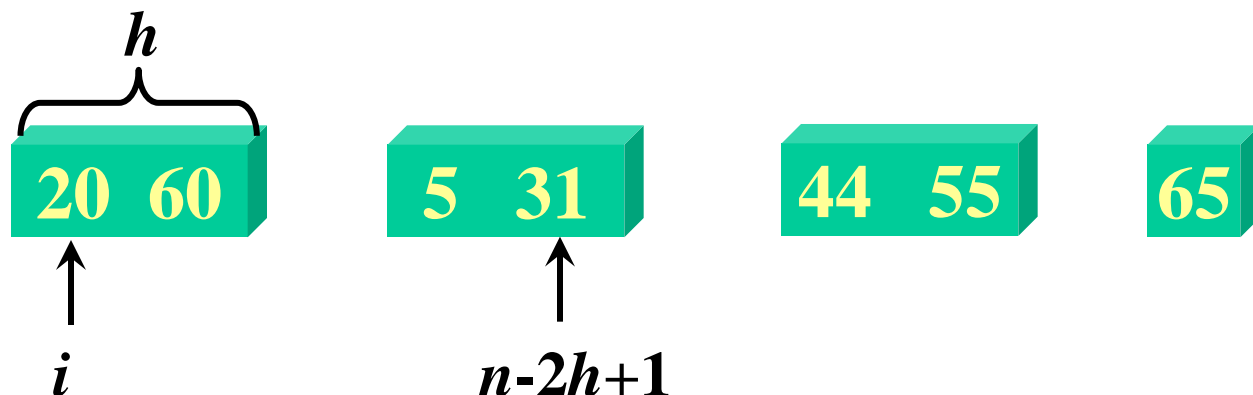


在一趟归并中，除最后一个有序序列外，其它有序序列中记录的个数相同，用长度 h 表示。

关键问题(2): 怎样完成一趟归并?

设参数 i 指向待归并序列的第一个记录，归并的步长是 $2h$ ，在归并过程中，有以下三种情况：

①若 $i \leq n-2h+1$ ，则相邻两个有序表的长度均为 h ，执行一次归并，完成后 i 加 $2h$ ，准备进行下一次归并；



$$i=1$$
$$n-2h+1=4$$

关键问题(2): 怎样完成一趟归并?

设参数 i 指向待归并序列的第一个记录, 归并的步长是 $2h$, 在归并过程中, 有以下三种情况:

①若 $i \leq n - 2h + 1$, 则相邻两个有序表的长度均为 h , 执行一次归并, 完成后 i 加 $2h$, 准备进行下一次归并;

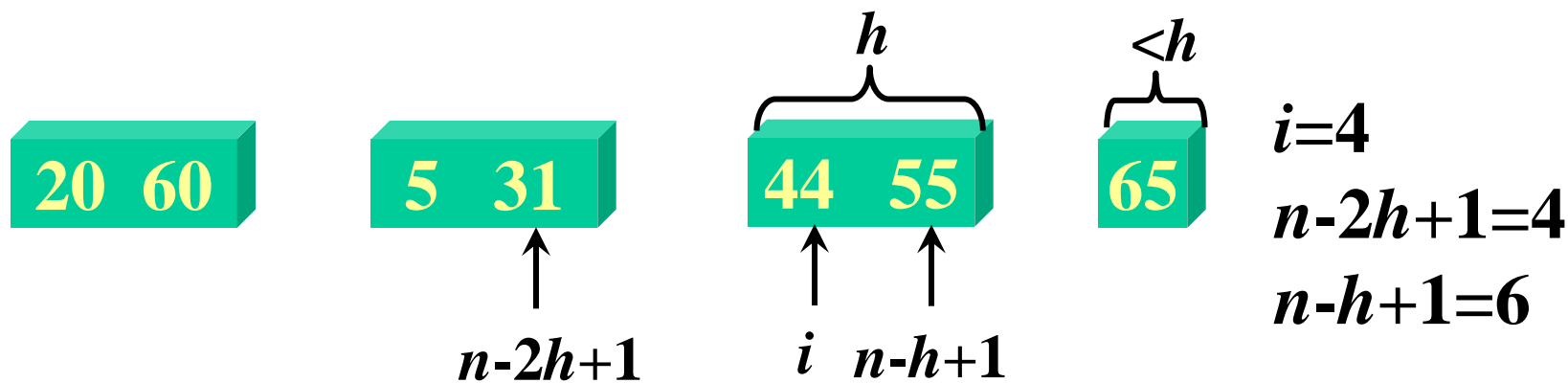
算法描述:

```
while (i ≤ n - 2h + 1)
{
    Merge (r, r1, i, i + h - 1, i + 2 * h - 1);
    i += 2 * h;
}
```

关键问题(2): 怎样完成一趟归并?

设参数 i 指向待归并序列的第一个记录，归并的步长是 $2h$ ，在归并过程中，有以下三种情况：

②若 $i < n-h+1$ ，则表示仍有两个相邻有序表，一个长度为 h ，另一个长度小于 h ，则执行两个有序表的归并，完成后退出一趟归并。



关键问题(2): 怎样完成一趟归并?

设参数 i 指向待归并序列的第一个记录，归并的步长是 $2h$ ，在归并过程中，有以下三种情况：

②若 $i < n-h+1$ ，则表示仍有两个相邻有序表，一个长度为 h ，另一个长度小于 h ，则执行两个有序表的归并，完成后退出一趟归并。

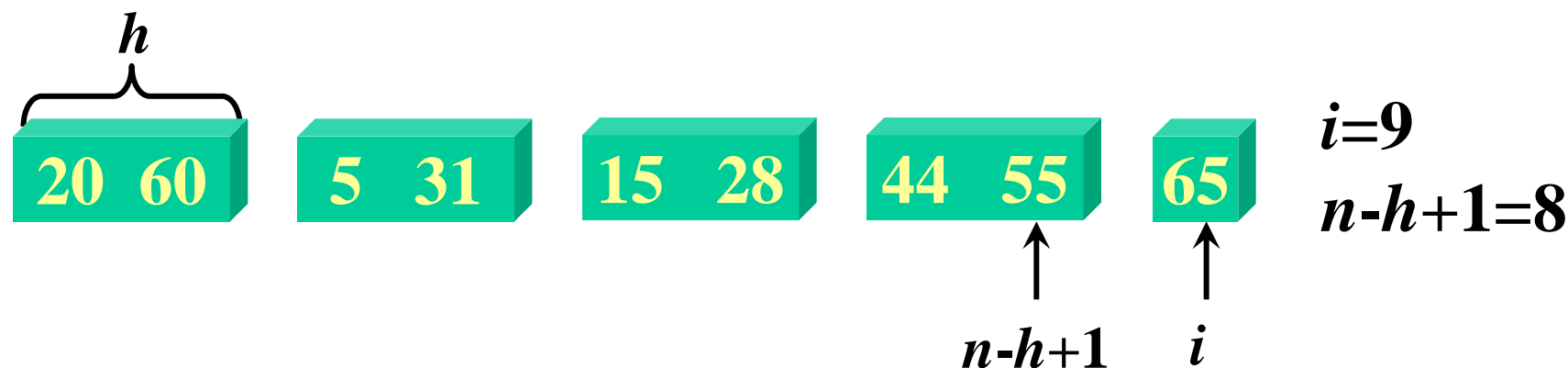
算法描述：

```
if ( $i < n-h+1$ ) Merge (r, r1, i, i+h-1, n);
```

关键问题(2): 怎样完成一趟归并?

设参数 i 指向待归并序列的第一个记录，归并的步长是 $2h$ ，在归并过程中，有以下三种情况：

③若 $i \geq n-h+1$ ，则表明只剩下一个有序表，直接将该有序表送到 $r1$ 的相应位置，完成后退出一趟归并。



关键问题(2): 怎样完成一趟归并?

设参数 i 指向待归并序列的第一个记录，归并的步长是 $2h$ ，在归并过程中，有以下三种情况：

③若 $i \geq n-h+1$ ，则表明只剩下一个有序表，直接将该有序表送到 $r1$ 的相应位置，完成后退出一趟归并。

算法描述：

```
if ( $i \geq n-h+1$ )  
    for ( $k=i$ ;  $k \leq n$ ;  $k++$ )  
         $r1[k]=r[k]$ ;
```

一趟归并排序算法

```
void MergePass (int r[ ], int r1[ ], int n, int h)
{
    i=1;
    while (i≤n-2h+1)           //情况1
    {
        Merge (r, r1, i, i+h-1, i+2*h-1);
        i+=2*h;
    }
    if (i<n-h+1) Merge (r, r1, i, i+h-1, n); //情况2
    else for (k=i; k≤n; k++) //情况3
        r1[k]=r[k];
}
```

关键问题(3): 如何控制二路归并的结束?

解决方法:

开始时, 有序序列的长度 $h=1$, 结束时, 有序序列的长度 $h=n$, 用有序序列的长度来控制排序的结束。

关键问题(3): 如何控制二路归并的结束?

算法描述:

```
void MergeSort (int r[ ], int r1[ ], int n )
{
    h=1;
    while (h<n)
    {
        MergePass (r, r1, n, h);
        h=2*h;
        MergePass (r1, r, n, h);
        h=2*h;
    }
}
```

二路归并排序算法的性能分析

时间性能:

一趟归并操作是将 $r[1] \sim r[n]$ 中相邻的长度为 h 的有序序列进行两两归并，并把结果存放到 $r1[1] \sim r1[n]$ 中，这需要 $O(n)$ 时间。整个归并排序需要进行 $\log_2 n$ 趟，因此，总的时间代价是 $O(n \log_2 n)$ 。这是归并排序算法的最好、最坏、平均的时间性能。

空间性能:

算法在执行时，需要占用与原始记录序列同样数量的存储空间，因此空间复杂度为 $O(n)$ 。

归并排序是稳定的。

小结：各种排序方法的比较

对排序算法应该从以下几个方面综合考虑：

- (1)时间复杂性
- (2)空间复杂性
- (3)稳定性
- (4)算法简单性
- (5)待排序记录个数 n 的大小
- (6)记录本身信息量的大小
- (7)关键码的分布情况

时间复杂度比较

排序方法	平均情况	最好情况	最坏情况
直接插入排序	$O(n^2)$	$O(n)$	$O(n^2)$
希尔排序	$O(n\log_2 n)$	$O(n^{1.3})$	$O(n^2)$
起泡排序	$O(n^2)$	$O(n)$	$O(n^2)$
快速排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n^2)$
简单选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$
堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$
归并排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$

空间复杂度比较

排序方法	辅助空间
直接插入排序	$O(1)$
希尔排序	$O(1)$
起泡排序	$O(1)$
快速排序	$O(\log_2 n) \sim O(n)$
简单选择排序	$O(1)$
堆排序	$O(1)$
归并排序	$O(n)$

稳定性比较

- (1) 直接插入排序、起泡排序和归并排序是稳定的。
- (2) 直接选择排序、希尔排序、快速排序和堆排序是不稳定的。

算法简单性比较

(1) 直接插入排序、冒泡排序和直接选择排序是一类简单的算法。

(2) 希尔排序、快速排序、堆排序和归并排序是一类改进后的算法，这些算法都很复杂。

待排序的记录个数比较

从待排序的记录个数 n 的大小看，

n 越小，采用简单排序方法越合适， n 越大，采用改进的排序方法越合适。

因为 n 越小， $O(n^2)$ 同 $O(n\log_2 n)$ 的差距越小，并且输入和调试简单算法比输入和调试改进算法要少用许多时间。

记录本身信息量比较

记录本身信息量越大，移动记录所花费的时间就越多，所以对**记录的移动次数**较多的算法不利。

排序方法	最好情况	最坏情况	平均情况
直接插入排序	$O(n)$	$O(n^2)$	$O(n^2)$
起泡排序	0	$O(n^2)$	$O(n^2)$
直接选择排序	0	$O(n)$	$O(n)$

关键码的分布情况比较

当待排序记录按关键码有序时，

直接插入排序和冒泡排序能达到 $O(n)$ 的时间复杂度；

对于快速排序而言，这是最坏的情况，此时的时间性能蜕化为 $O(n^2)$ ；

直接选择排序、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。

作业

习题8:

$4(2)(6)$, $5(6)(8)$

分配排序和基数排序

- 不需要进行比较
- 需要事先知道记录序列的一些具体情况

基数排序

- 假设长度为 n 的序列

$$\mathbf{R} = \{ \mathbf{r}_0, \mathbf{r}_1, \dots, \mathbf{r}_{n-1} \}$$

记录的排序码 \mathbf{K} 包含 d 个子排序码

$$\mathbf{K} = (\mathbf{k}_{d-1}, \mathbf{k}_{d-2}, \dots, \mathbf{k}_1, \mathbf{k}_0),$$

则称 \mathbf{R} 对排序码 $(\mathbf{k}_{d-1}, \mathbf{k}_{d-2}, \dots, \mathbf{k}_1, \mathbf{k}_0)$ 有序就是对于任意两个记录 $\mathbf{R}_i, \mathbf{R}_j$ ($0 \leq i < j \leq n-1$), 都满足

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

其中 \mathbf{k}_{d-1} 称为最高排序码, \mathbf{k}_0 称为最低排序码。

例子

- 例如：如果我们要对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，整个序列就排好序了。

基数排序分为两类

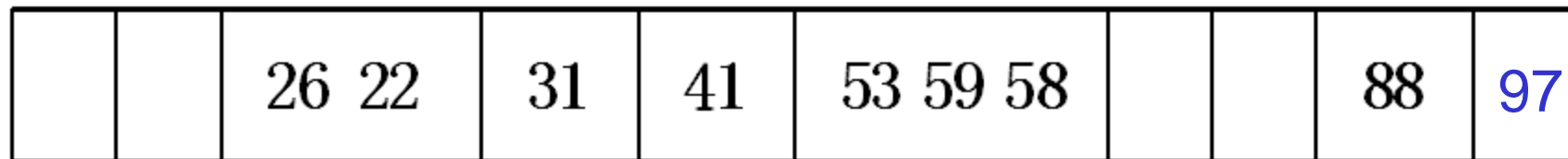
- 高位优先法（MSD, Most Significant Digit first）
- 低位优先法（LSD, Least Significant Digit first）

高位优先法 (MSD)

- 先对高位 k_{d-1} 进行桶式排序，将序列分成若干个桶中
- 然后对每个桶再按次高位 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。

97 53 88 59 26 41 58 31 22

0 1 2 3 4 5 6 7 8 9



0 1 2 3 4 5 6 7 8 9



22 26 31 41 53 58 59 88 97

(a) 高位优先

低位优先法 (LSD)

- 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。
- 这是一个分、收; 分、收; ...; 分、收的过程。
- 比较简单, 计算机常用

97 53 88 59 26 41 58 31 22

0 1 2 3 4 5 6 7 8 9

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

41 31 22 53 26 97 88 58 59

0 1 2 3 4 5 6 7 8 9

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

22 26 31 41 53 58 59 88 97

(b) 低位优先