Sorting 2

Zibin Zheng（郑子彬）

School of Data and Computer Science， SYSU

http://www.inpluslab.com

课程主页：http://inpluslab.sysu.edu.cn/dsa2016/

排序方法

- Insertion Sort (直接插入、希尔排序)
- Exchange sort (冒泡排序、快速排序)
- Selection Sort (**简单选择排序**、堆排序) ★
- Merge Sort (归并排序)
- Radix Sort (基数排序)

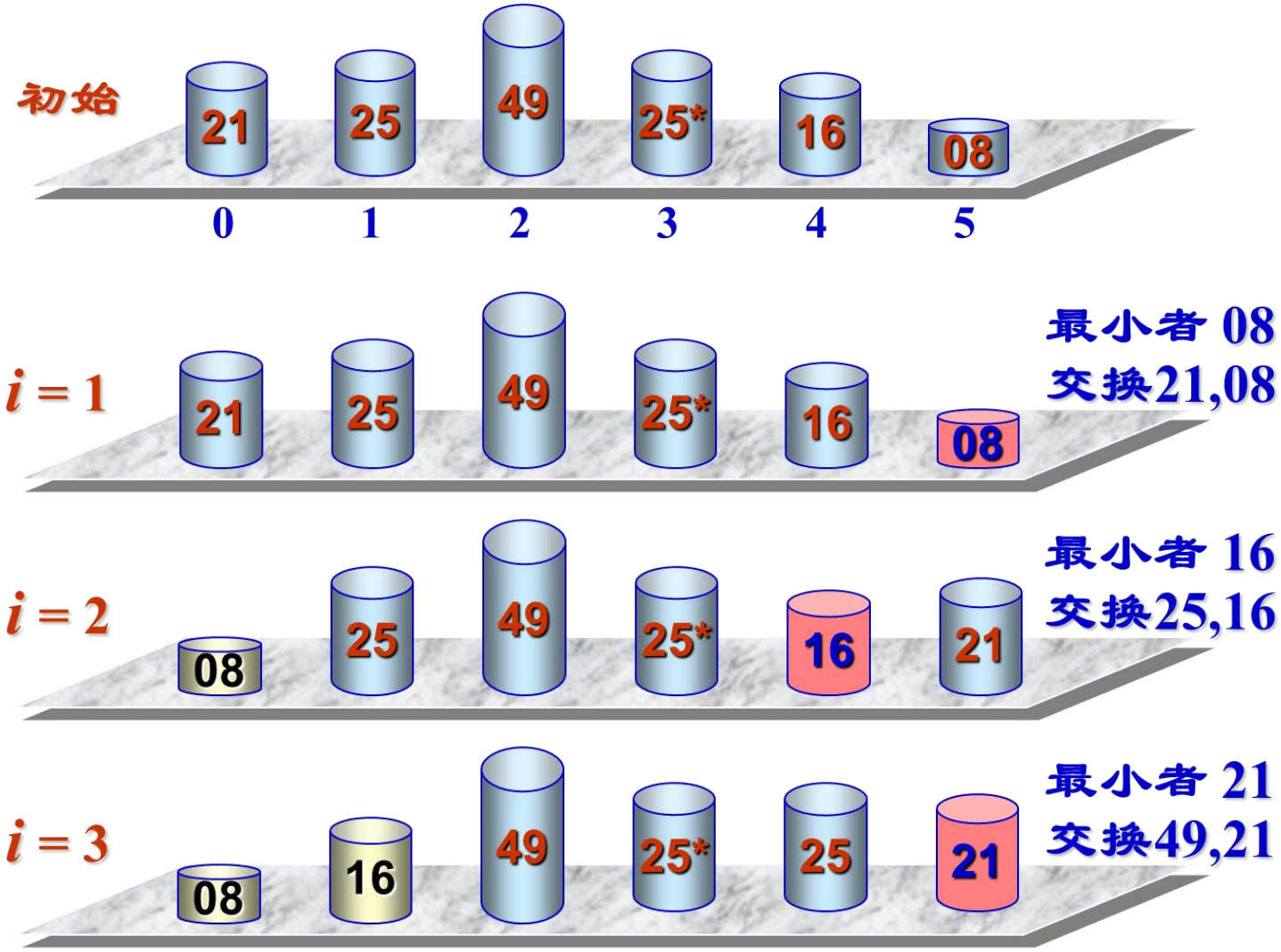
选择排序(Selection Sort)

- 选择排序的基本思想是：每一趟 (例如第 i 趟，$i = 1, \ldots, n\text{-}1$) 在后面的 $n\text{-}i\text{+}1$ 个待排序对象中选出关键字最小的对象, 作为有序对象序列的第 i 个对象。待到第 $n\text{-}1$ 趟作完，待排序对象只剩下1个，就不用再选了。

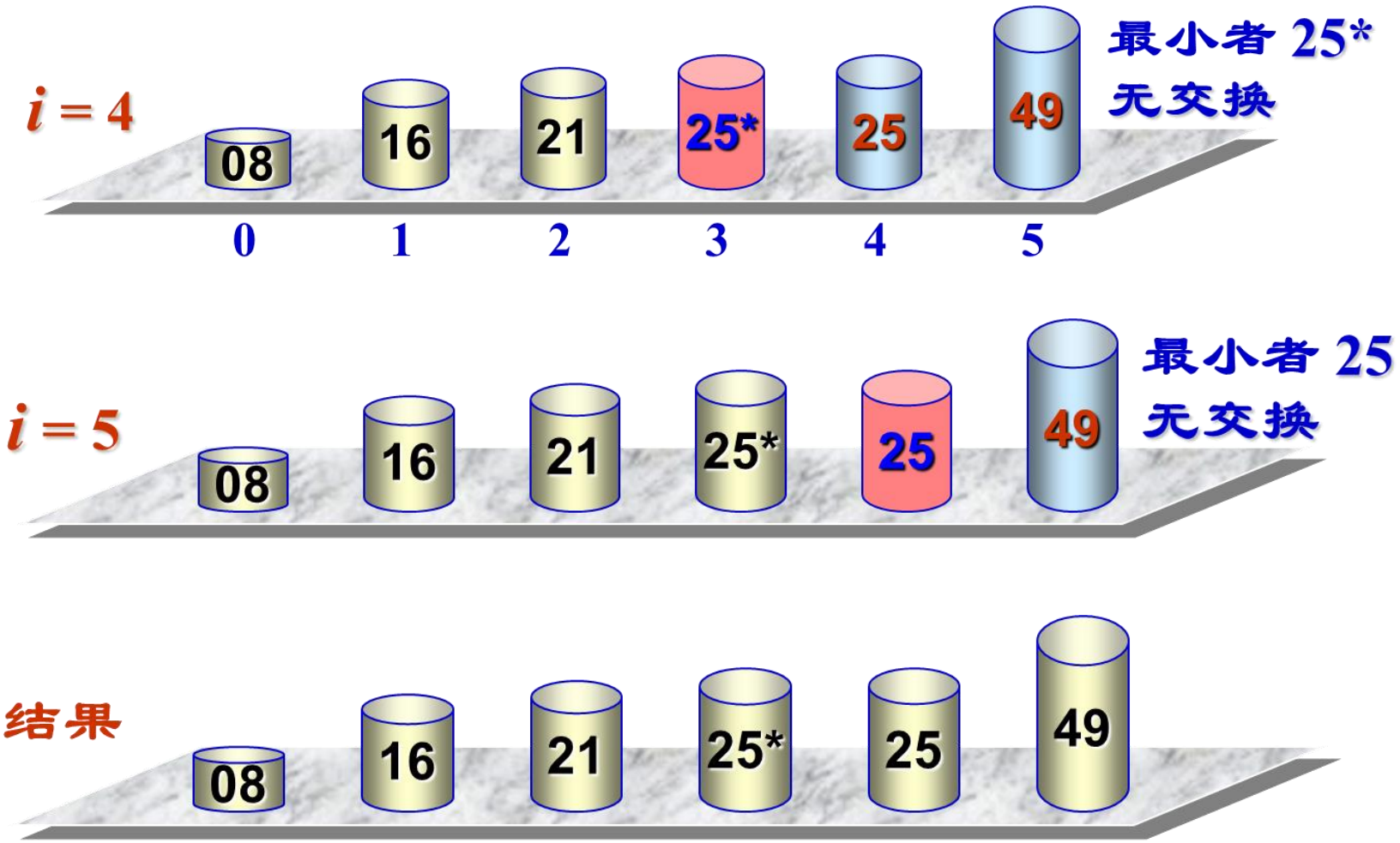
简单选择排序 (Simple Selection Sort)

- 基本步骤为：i从1开始，直到n-1，进行n-1趟排序，第 i 趟的排序过程为： 在一组对象r[i] ~ r[n] (i=1,2,…,n-1)中选择具有最小关键字的对象；并和第 i 个对象进行交换；

简单选择排序 (Simple Selection Sort)



简单选择排序 (Simple Selection Sort)



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] ~ 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（index!=i）
    r[i]←→r[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 (Simple Selection Sort)

- 直接选择排序的关键字比较次数KCN与对象的初始排列无关。第 i 趟选择具有最小关键字对象所需的比较次数总是 n-i 次，此处假定整个待排序对象序列有 n 个对象。因此，总的关键字比较次数为

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

简单选择排序 (Simple Selection Sort)

- 对象的移动次数与对象序列的初始排列有关。当这组对象的初始状态是按其关键字从小到大有序的时候，对象的移动次数$RMN = 0$，达到最少。
- **最坏情况**是每一趟都要进行交换，总的对象移动次数为$RMN = 3(n\text{-}1)$。
- 直接选择排序是一种**不稳定**的排序方法。

排序方法

- Insertion Sort (直接插入、希尔排序)
- Exchange sort (冒泡排序、快速排序)
- Selection Sort (简单选择排序、**堆排序**) ★
- Merge Sort (归并排序)
- Radix Sort (基数排序)

堆排序

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。

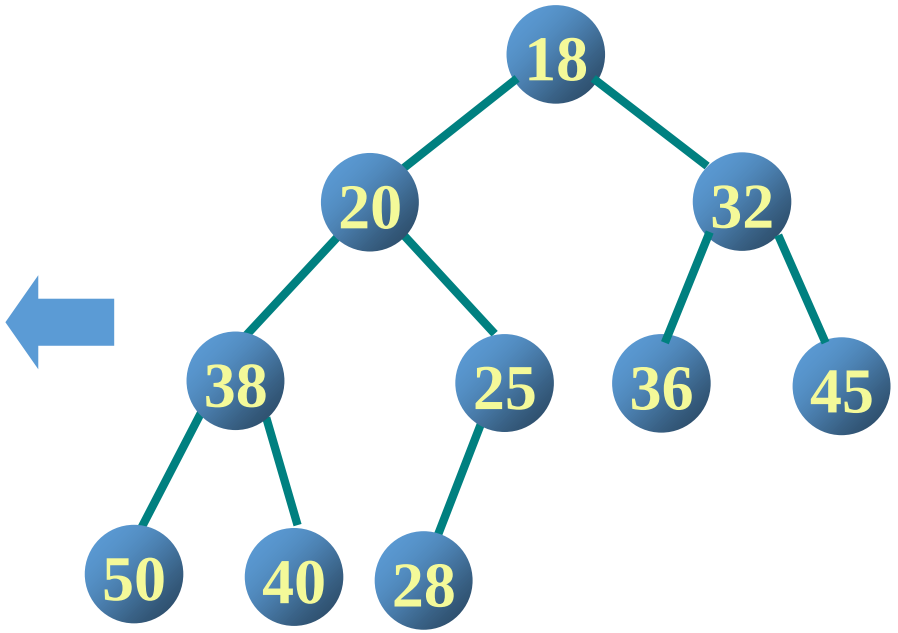
减少关键码间的比较次数

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）。

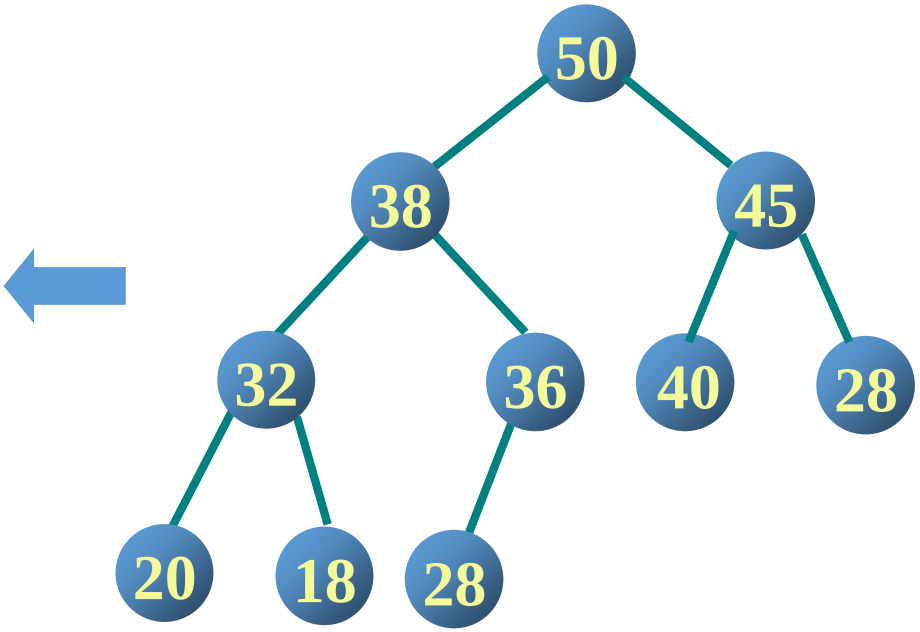
1. 小根堆的根结点是所有结点的最小者。
2. 较小结点靠近根结点，但不绝对。



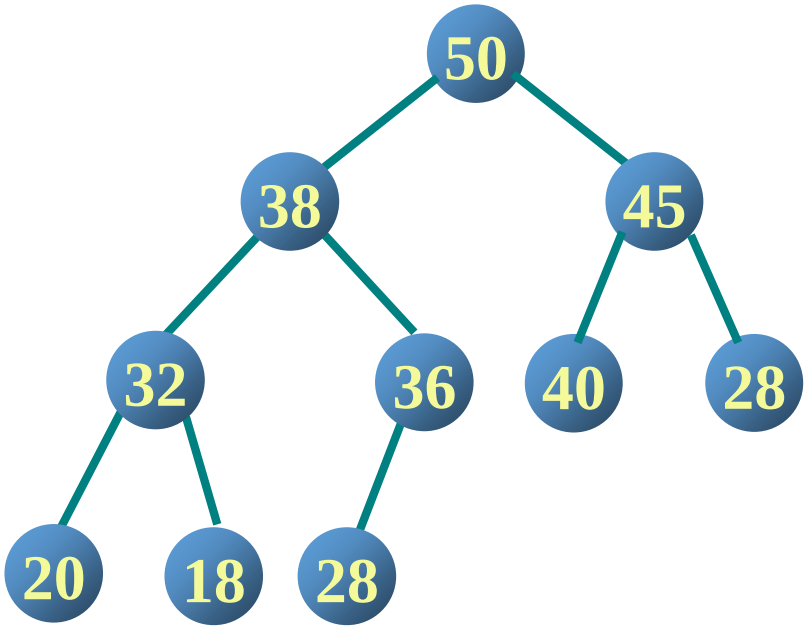
堆的定义

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，但不绝对。



堆和序列的关系



采用顺序存储

1	2	3	4	5	6	7	8	9	10
50	38	45	32	36	40	28	20	18	28

将堆用顺序存储结构来存储，则堆对应一组序列。

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。

需解决的关键问题?

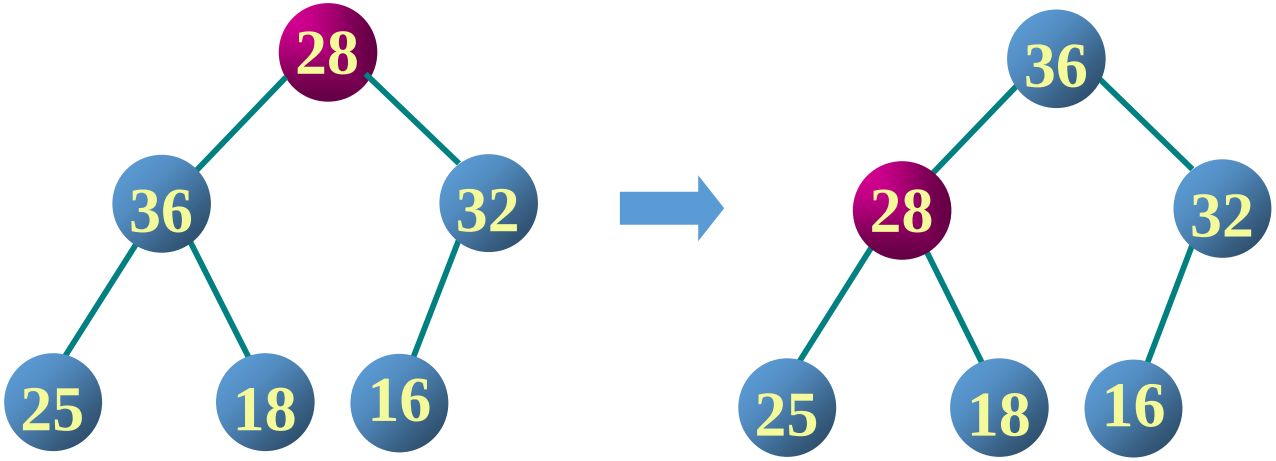
(1)如何由一个无序序列建成一个堆（即初始建堆）？

(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

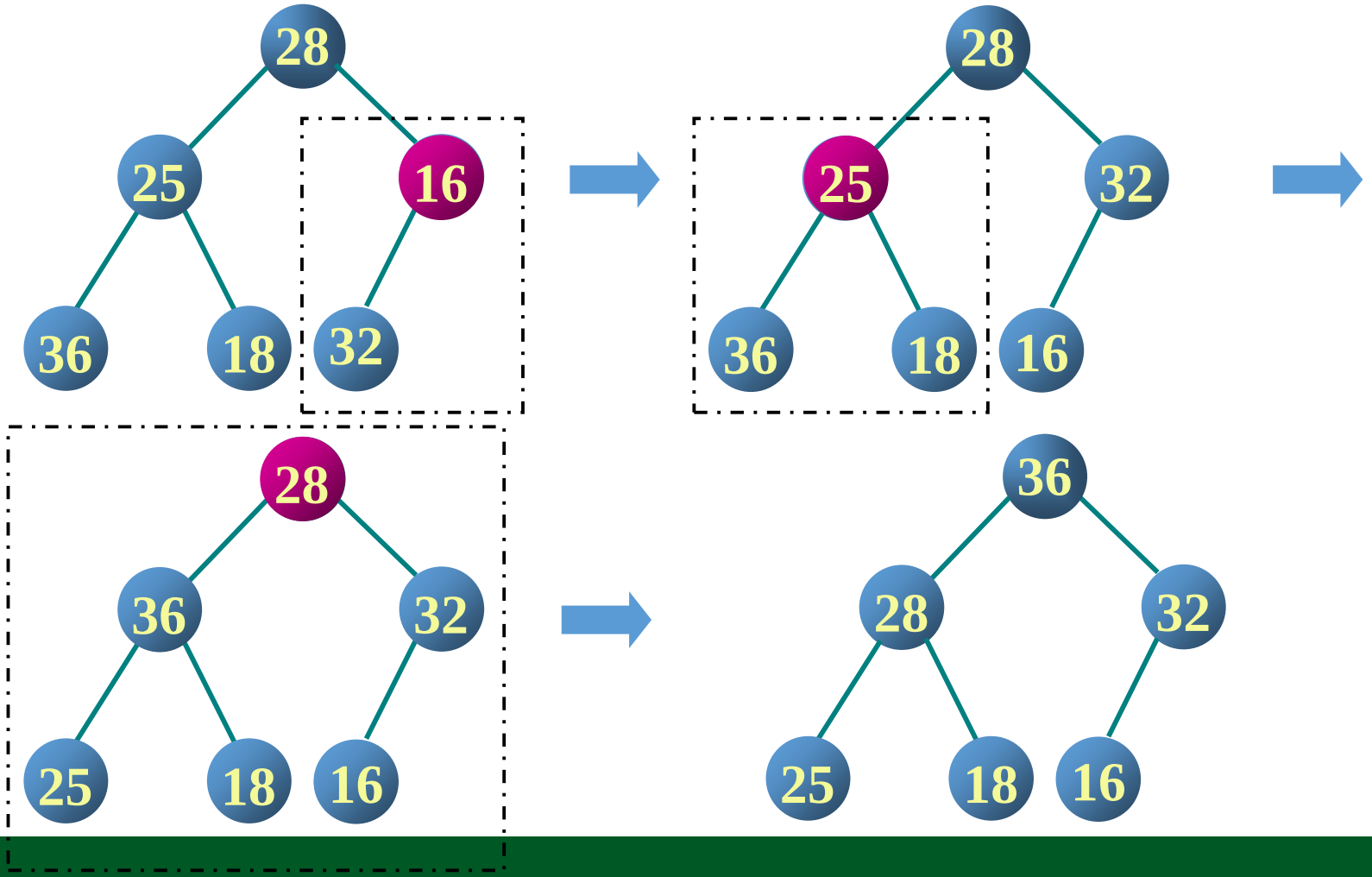
堆调整：在一棵完全二叉树中，根结点的左右子树均是堆，如何调整根结点，使整个完全二叉树成为一个堆？



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;
   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;
      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？



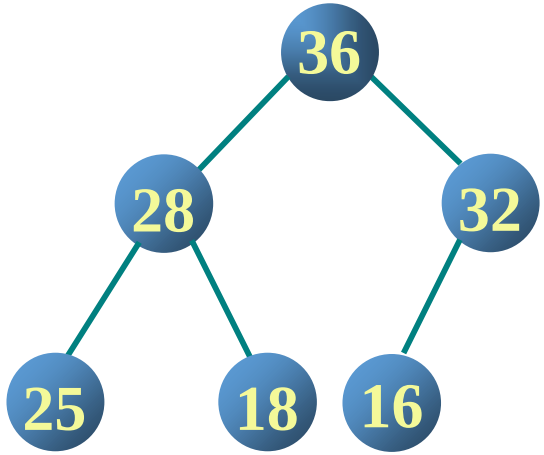
关键问题(1)：如何由一个无序序列建成一个堆？

算法描述：

```
for (i=n/2; i>=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。

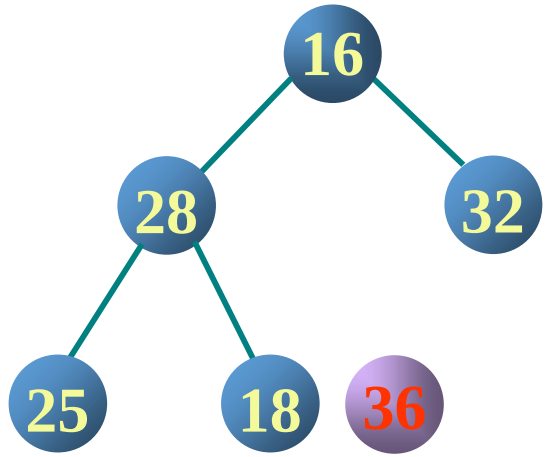
关键问题(2)：如何处理堆顶记录？



对应

1	2	3	4	5	6
36	28	32	25	18	16

交换



对应

1	2	3	4	5	6
16	28	32	25	18	36

关键问题(2)：如何处理堆顶记录？

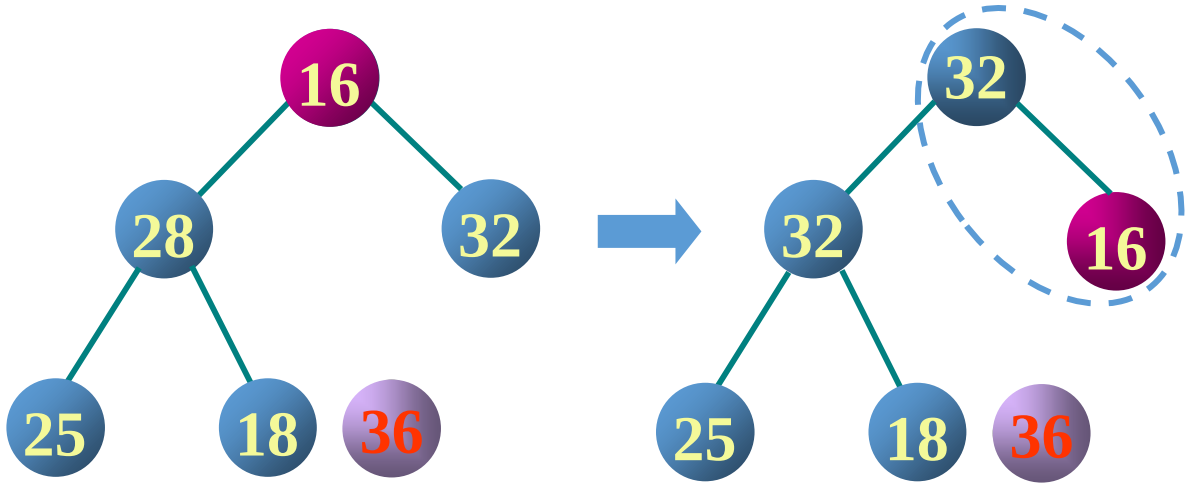
解决方法：

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]←→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$O(n\log_2 n)$时间；

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

因此整个时间复杂度为$O(n\log_2 n)$，这是堆排序的最好、最坏和平均的时间代价。

排序方法

- Insertion Sort (直接插入、希尔排序)
- Exchange sort (冒泡排序、快速排序)
- Selection Sort (简单选择排序、堆排序)
- Merge Sort (**归并排序**) ★
- Radix Sort (基数排序)

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。

需解决的关键问题?

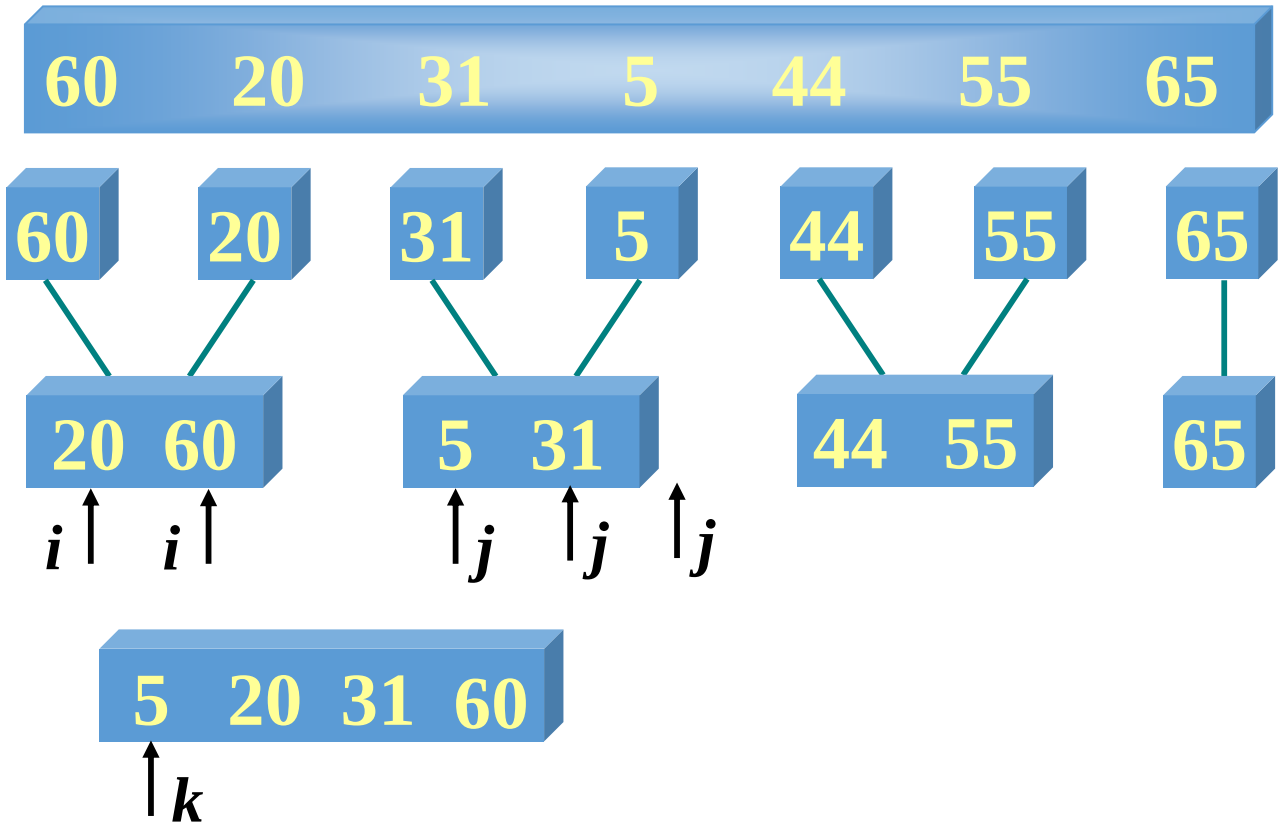
(1)如何将两个有序序列合成一个有序序列？

(2)怎样完成一趟归并？

(3)如何控制二路归并的结束？

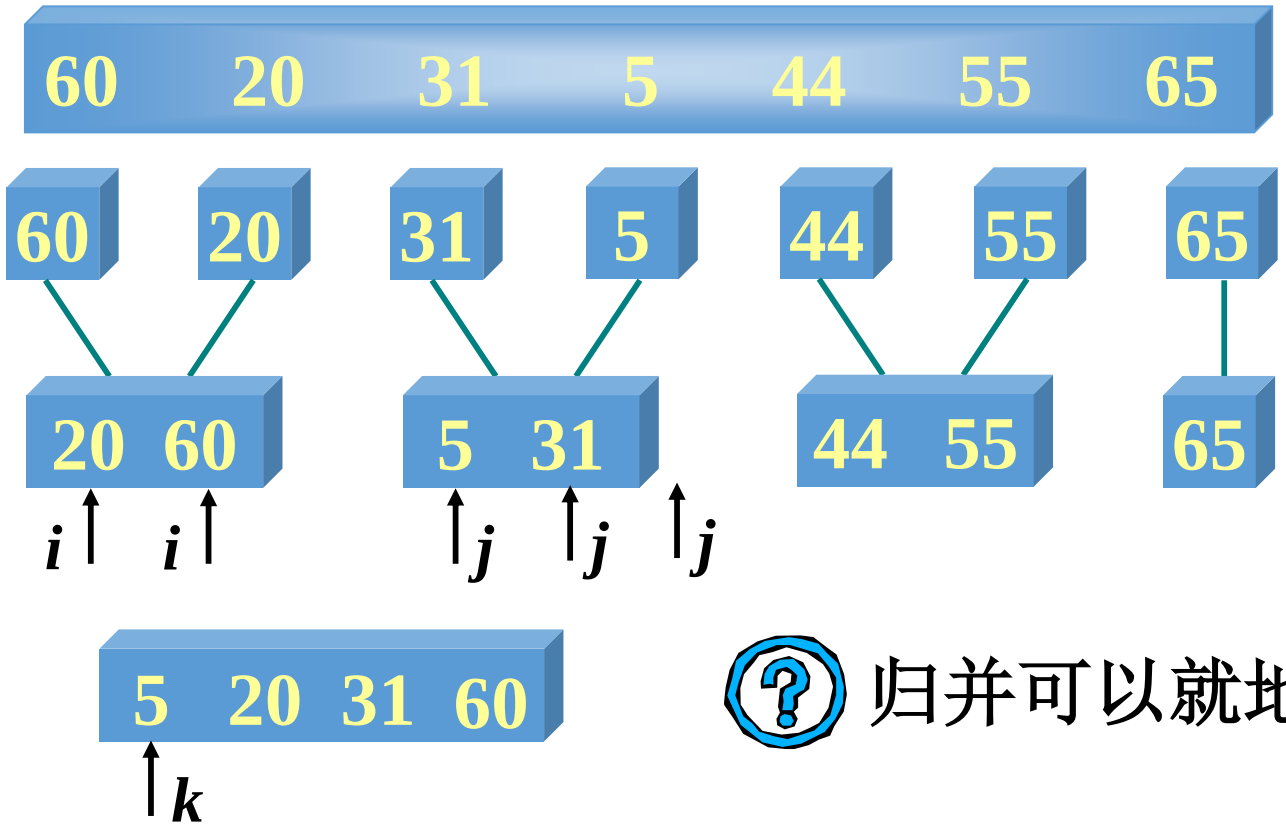
归并排序

关键问题(1)：如何将两个有序序列合成一个有序序列？



归并排序

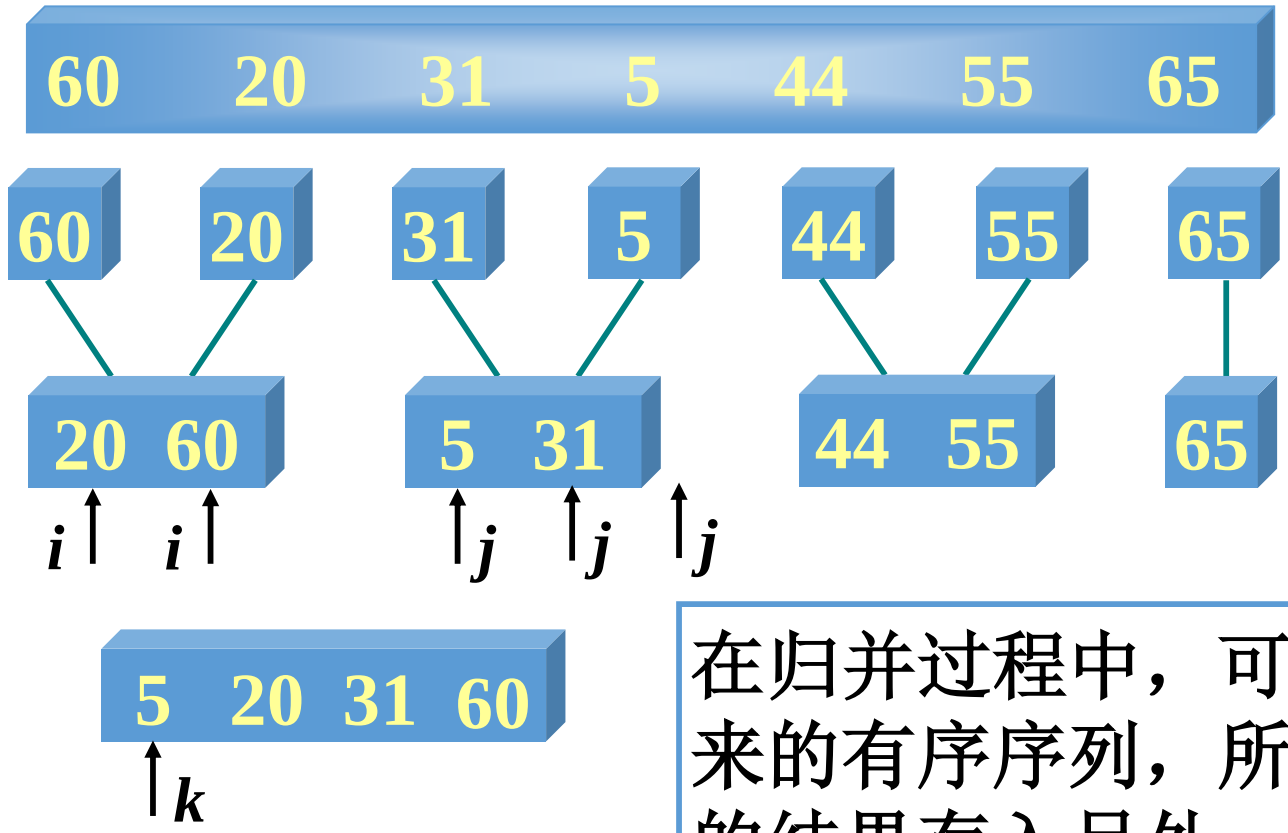
关键问题(1)：如何将两个有序序列合成一个有序序列？



归并可以就地进行吗?

归并排序

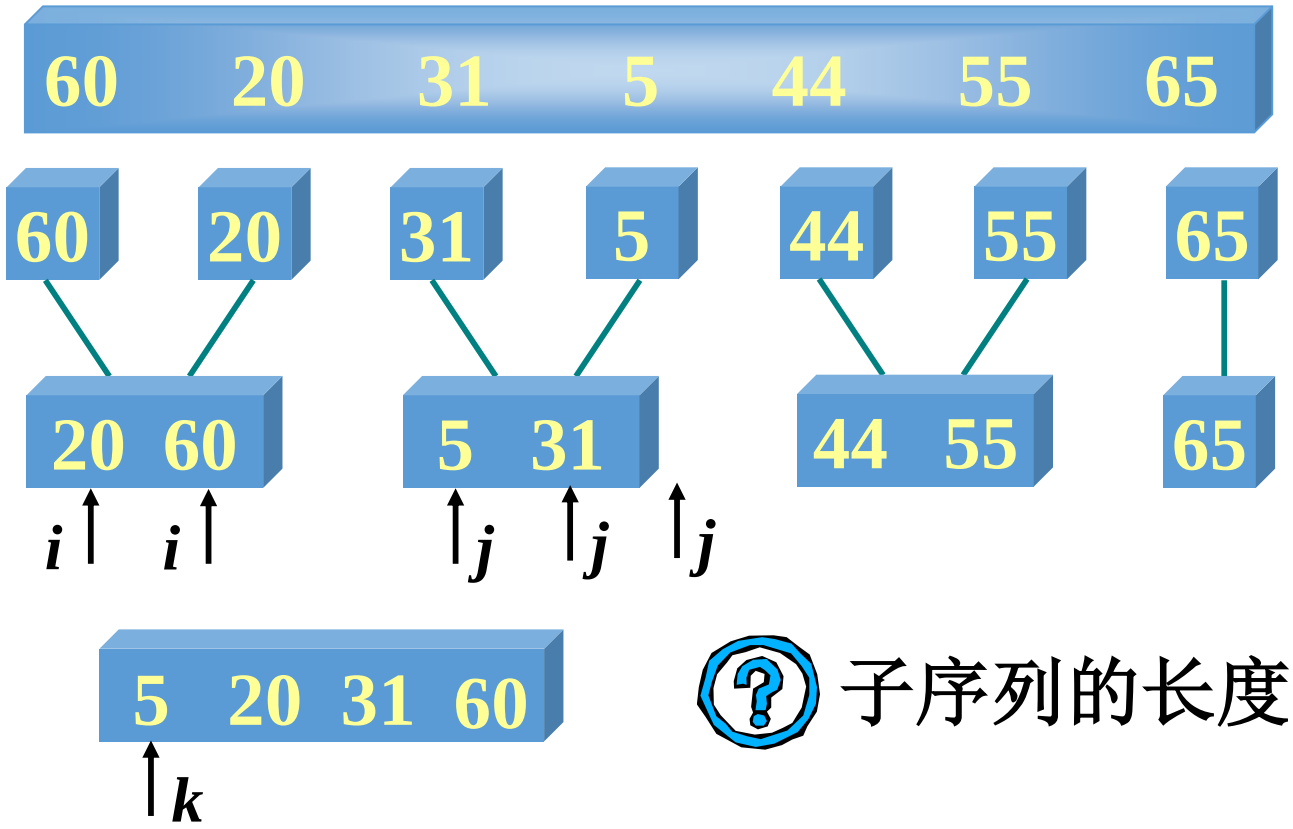
关键问题(1)：如何将两个有序序列合成一个有序序列？



在归并过程中，可能会破坏原来的有序序列，所以，将归并的结果存入另外一个数组中。

归并排序

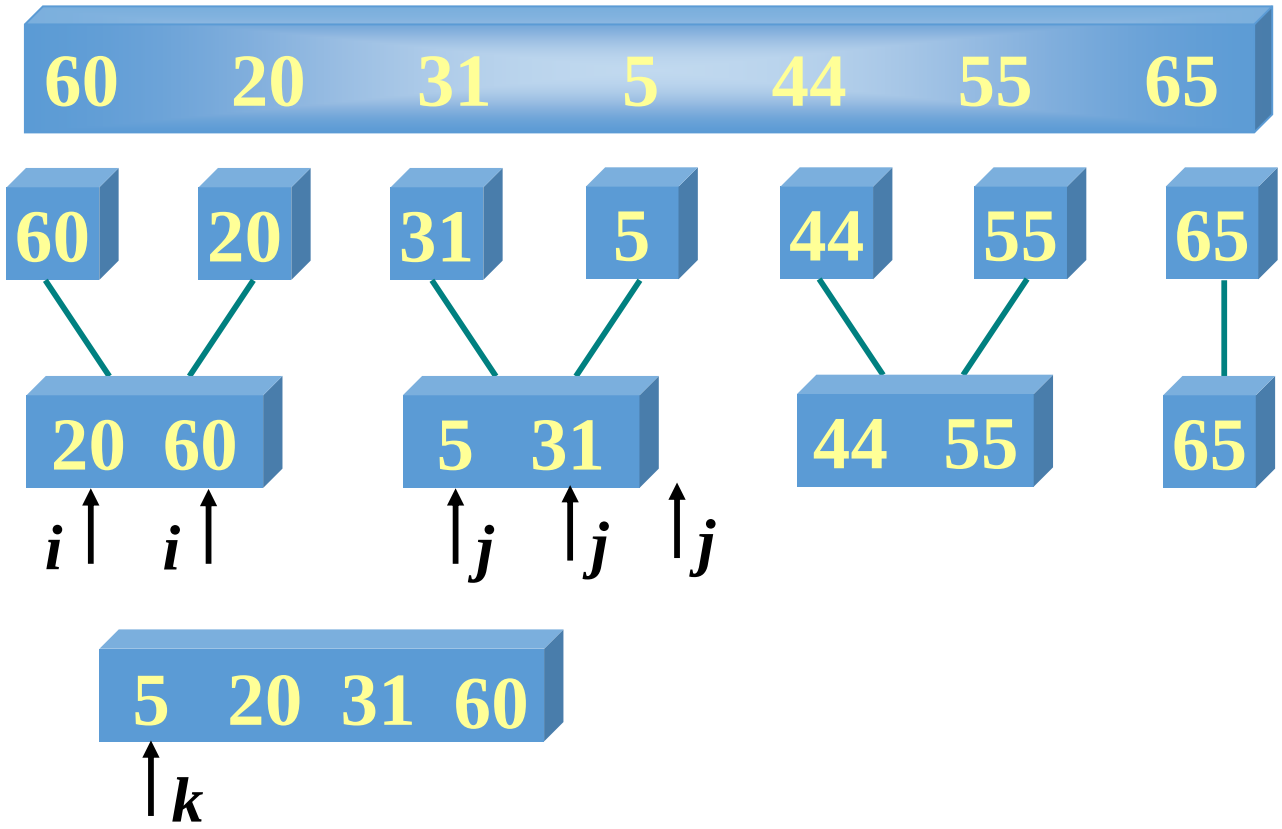
关键问题(1)：如何将两个有序序列合成一个有序序列？



子序列的长度一定相等吗?

归并排序

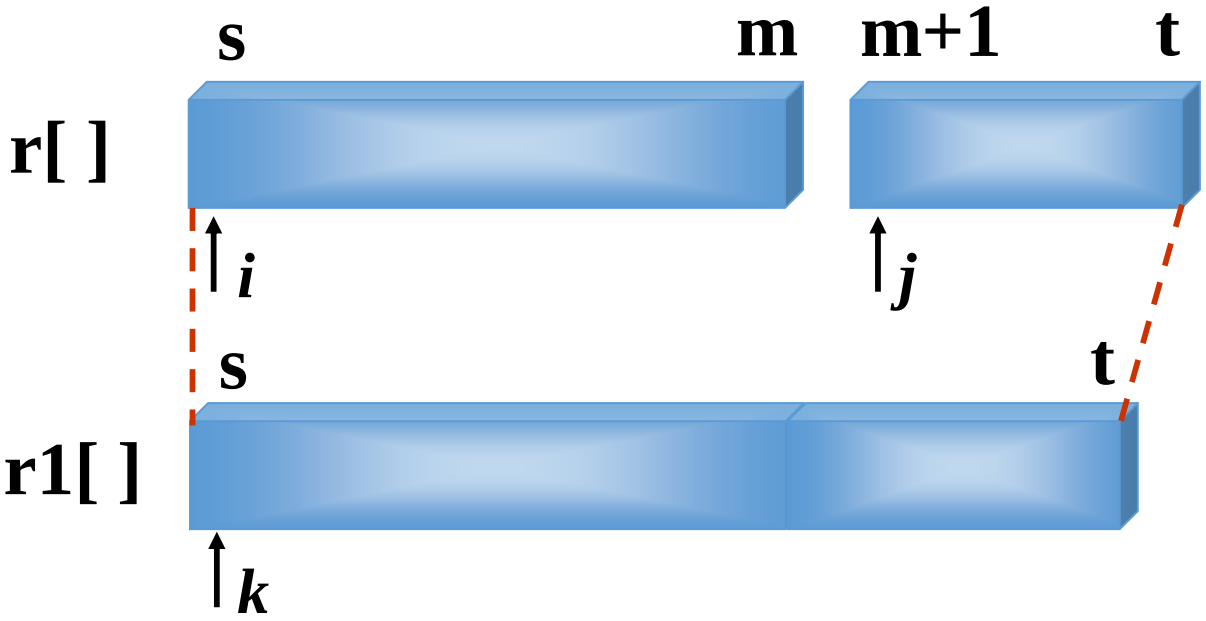
关键问题(1)：如何将两个有序序列合成一个有序序列？



归并排序

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] ~ r[m]和r[m+1] ~ r[t]，归并成一个有序序列r1[s] ~ r1[t]



归并排序

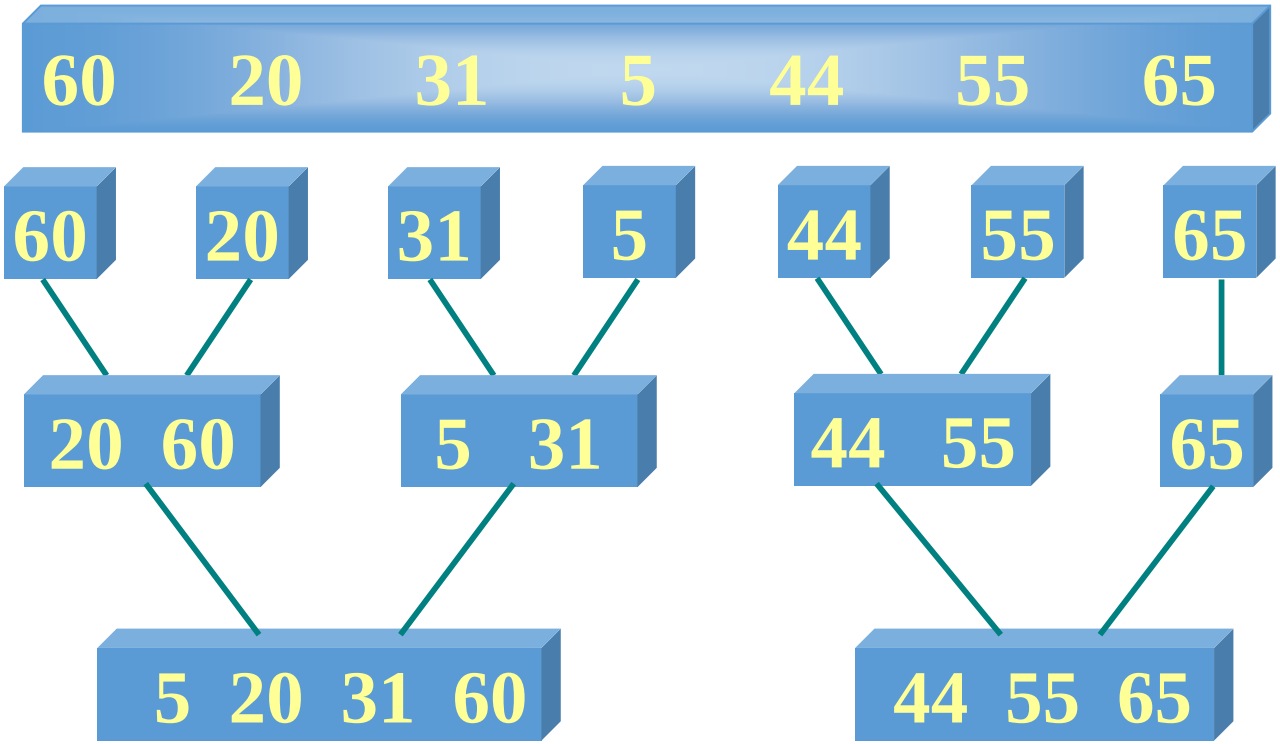
关键问题(1)：如何将两个有序序列合成一个有序序列？

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)
   {
      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
}
```

归并排序

关键问题(2)：怎样完成一趟归并？



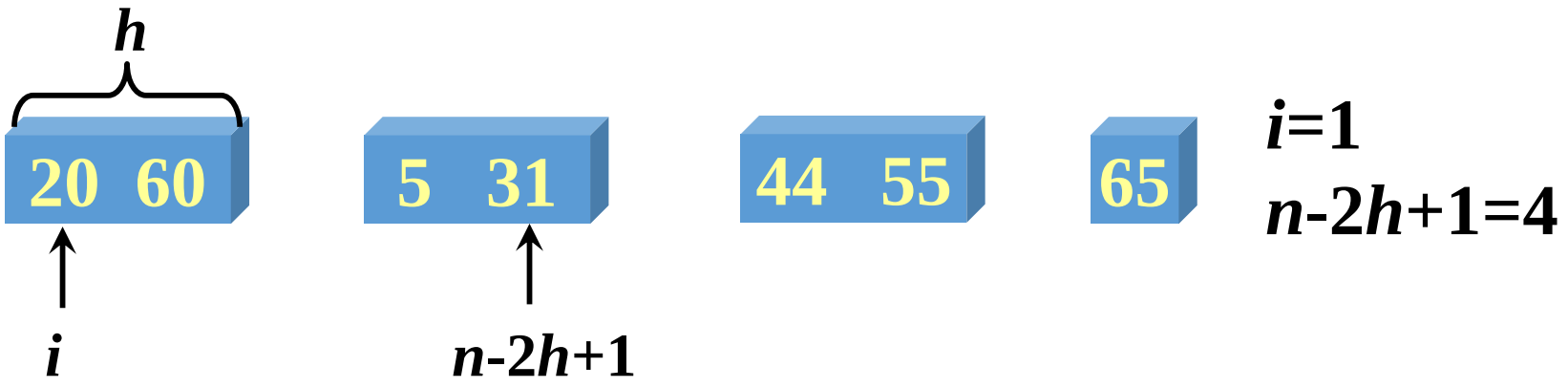
在一趟归并中，除最后一个有序序列外，其它有序序列中记录的个数相同，用长度*h*表示。

归并排序

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2*h*，在归并过程中，有以下三种情况：

①若*i*≤*n*-2*h*+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2*h*，准备进行下一次归并；



归并排序

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2*h*，在归并过程中，有以下三种情况：

①若$i \le n-2h+1$，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2*h*，准备进行下一次归并；

算法描述：

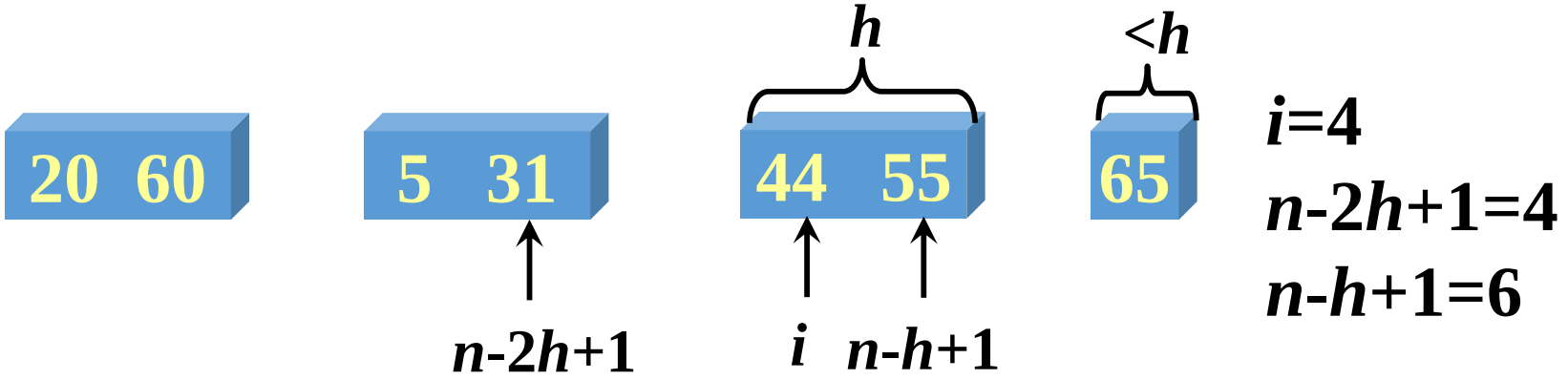
```
while (i≤n-2h+1)
{
    Merge (r, r1, i, i+h-1, i+2*h-1);
    i+=2*h;
}
```

归并排序

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。



归并排序

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：

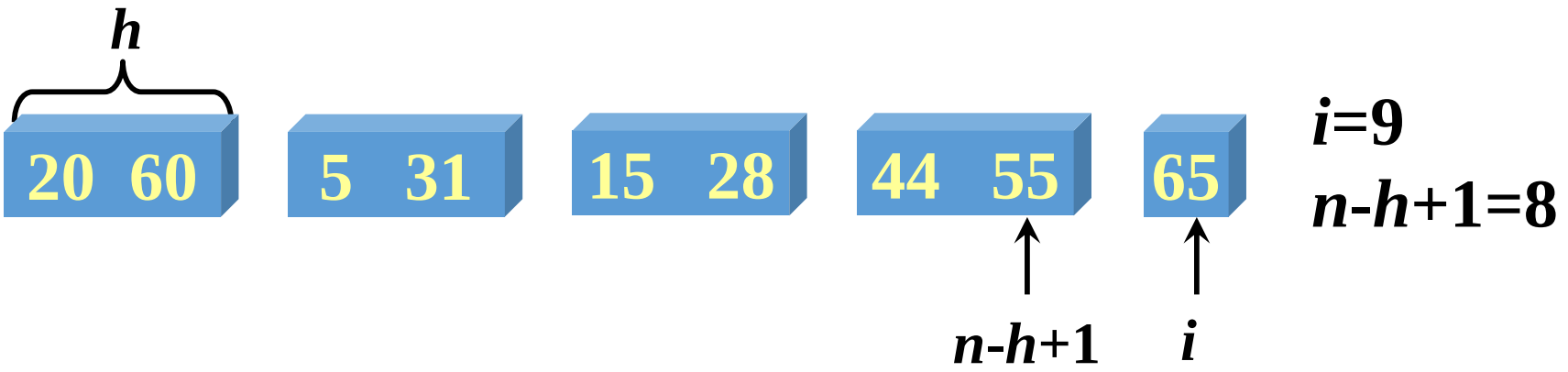
```
if (i<n-h+1) Merge (r, r1, i, i+h-1, n);
```

归并排序

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≥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。



归并排序

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≥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>=n-h+1)
    for (k=i; k<=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];
}
```

归并排序

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。

归并排序

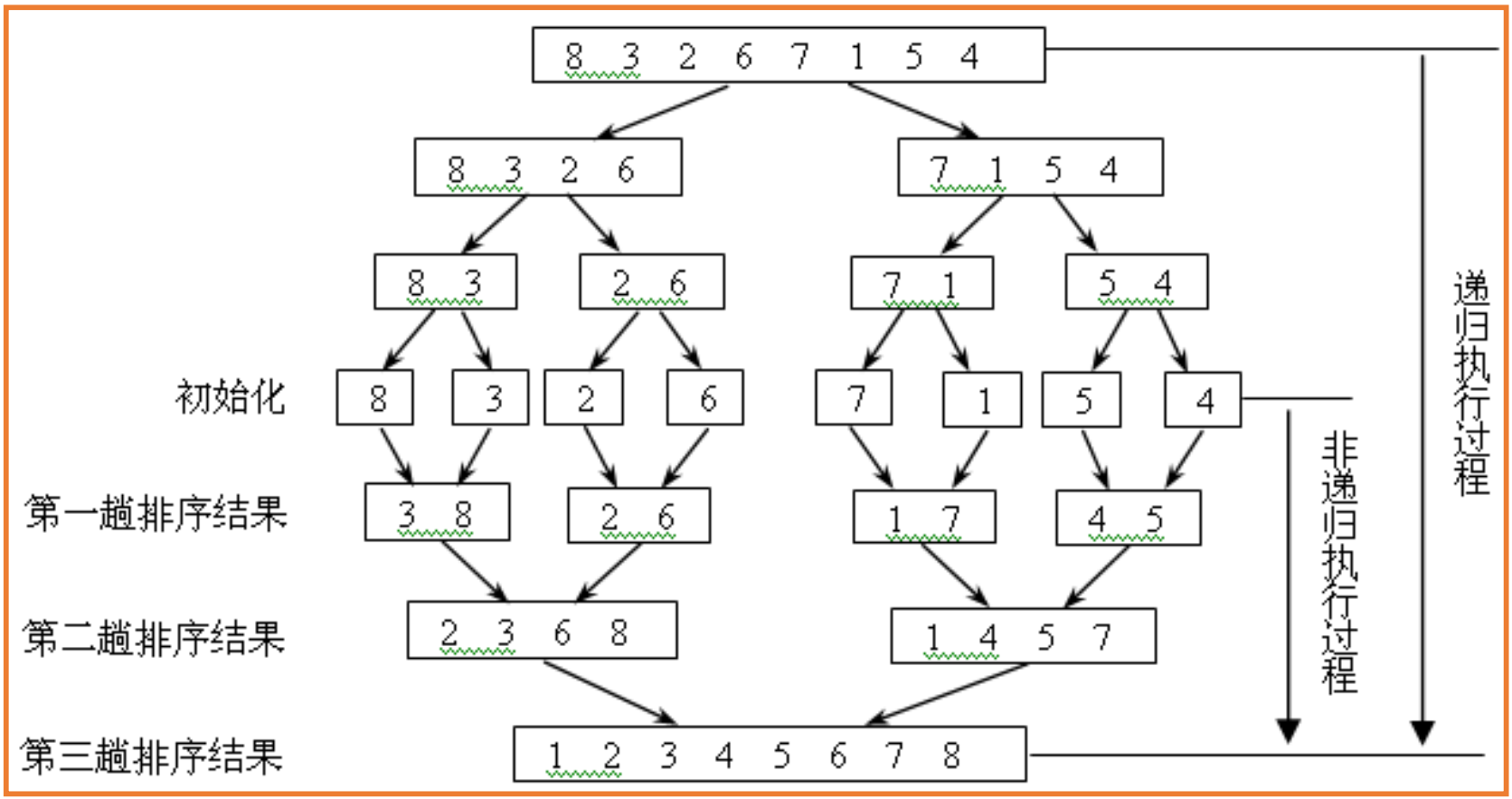
关键问题(3)：如何控制二路归并的结束？

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;
    }
}
```

归并排序

二路归并的递归实现



二路归并的递归实现

```
void MergeSort2(int r[ ], int r1[ ], int s, int t)
{
    if (s==t) r1[s]=r[s];                    //递归出口
    else {
        m=(s+t)/2;
        Mergesort2(r, r1, s, m);          //归并排序前半个子序列
        Mergesort2(r, r1, m+1, t);        //归并排序后半个子序列
        Merge(r1, r, s, m, t);            将两个已排序的子序列归并
  }
}
```

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]~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]~r1[n]中，这需要$O(n)$时间。整个归并排序需要进行$\lceil \log_2 n \rceil$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$O(n)$。

- 归并排序是一个稳定的排序方法。

排序方法

- Insertion Sort (直接插入、希尔排序)
- Exchange sort (冒泡排序、快速排序)
- Selection Sort (简单选择排序、堆排序)
- Merge Sort (归并排序)
- Radix Sort (**基数排序**) ★

基数排序 (Radix Sort)

- 基数排序(Radix Sorting) 又称为**桶排序**或**数字排序**：按待排序记录的关键字的组成成分(或“位”)进行排序。
- 基数排序和前面的各种内部排序方法完全不同，不需要进行关键字的比较和记录的移动。
- 基数排序是采用“**分配**”与“**收集**”的办法，用对多关键字进行排序的思想实现对单关键字进行排序的方法。

基数排序 (Radix Sort)

- **多关键字排序**

- 设有n个记录$\{R_1, R_2, \ldots, R_n\}$， 每个记录$R_i$的关键字是由若干项(数据项)组成，即记录$R_i$的关键字Key是若干项的集合： $\{K_i^1, K_i^2, \ldots, K_i^d\}(d>1)$ 。

- 记录$\{R_1, R_2, \ldots, R_n\}$有序的，指的是$\forall i, j \in [1,n]$，$i<j$ ，若记录的关键字满足：

$$\{K_i^1, K_i^2, \ldots K_i^d\} < \{K_j^1, K_j^2, \ldots K_j^d\}$$ ，

基数排序 (Radix Sort)

- 多关键字排序思想

- 先按第一个关键字K^1进行排序，将记录序列分成若干个子序列，每个子序列有相同的K^1值；然后分别对每个子序列按第二个关键字K^2进行排序，每个子序列又被分成若干个更小的子序列；如此重复，直到按最后一个关键字K^d进行排序。

- 最后，将所有的子序列依次联接成一个有序的记录序列，该方法称为**最高位优先(Most Significant Digit first)**。

- 另一种方法正好相反，排序的顺序是从最低位开始，称为**最低位优先(Least Significant Digit first)**。

基数排序 (Radix Sort)

- 链式基数排序
- 若记录的关键字由若干确定的部分(又称为 “位”)组成，每一位(部分)都有确定数目的取值。对这样的记录序列排序的有效方法是基数排序。
- 设有n个待排序记录$\{R_1, R_2, \ldots, R_n\}$， (单)关键字是由d位(部分)组成，每位有r种取值，则关键字R[i].key可以看成一个d元组： R[i].key=$\{K_i^1, K_i^2, \ldots, K_i^d\}$ 。
- 基数排序可以采用前面介绍的MSD或LSD方法。
- 以下以LSD方法讨论链式基数排序。

基数排序 (Radix Sort)

- 排序思想

(1) 首先以静态链表存储n个待排序记录，头结点指针指向第一个记录结点；

(2) 一趟排序的过程是：

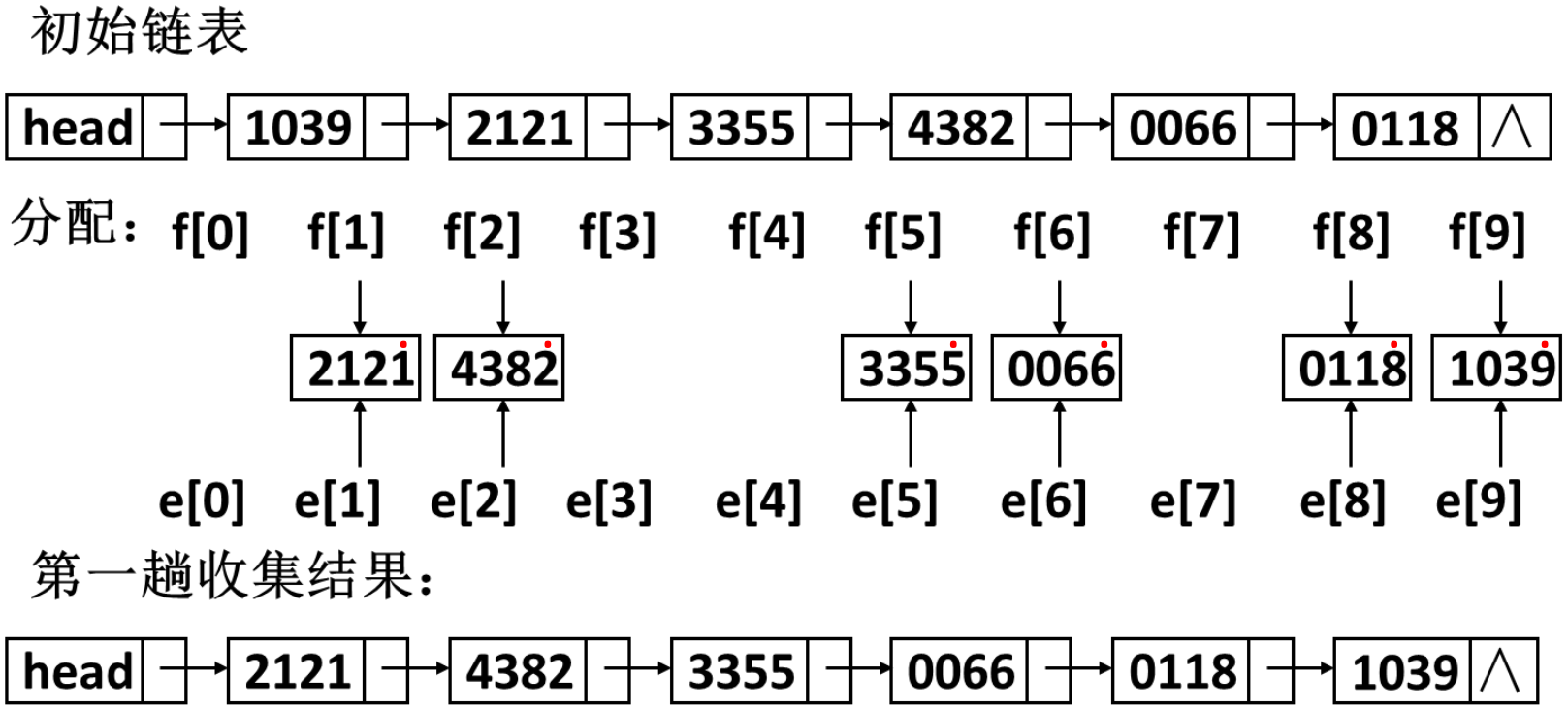
① **分配**： 按K^d值的升序顺序，改变记录指针，将链表中的记录结点分配到r个链表(桶)中，每个链表中所有记录的关键字的最低位(K^d)的值都相等，用f[i]、e[i]作为第i个链表的头结点和尾结点；

② **收集**：改变所有非空链表的尾结点指针，使其指向下一个非空连表的第一个结点，从而将r个链表中的记录重新链接成一个链表；

(3) 如此依次按K^{d-1}, K^{d-2}, … K^1分别进行，共进行d趟排序后排序完成。

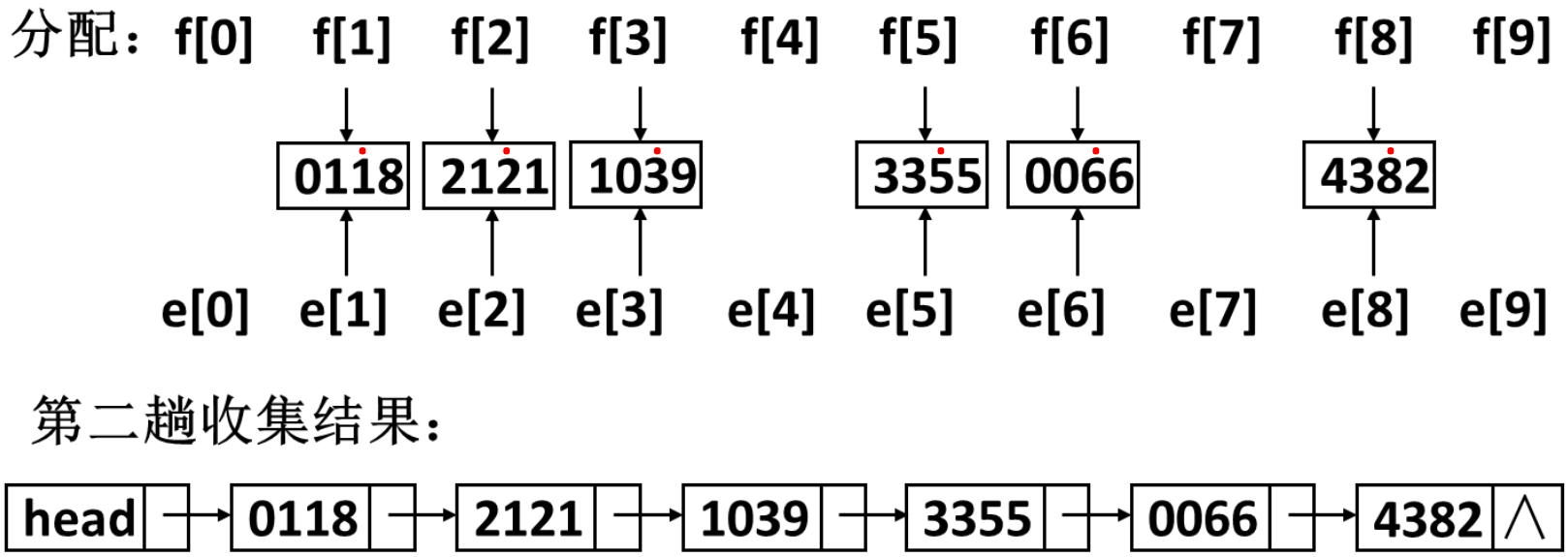
基数排序 (Radix Sort)

- 设有关键字序列为1039, 2121, 3355, 4382, 66, 118的一组记录，采用链式基数排序的过程如下图所示。



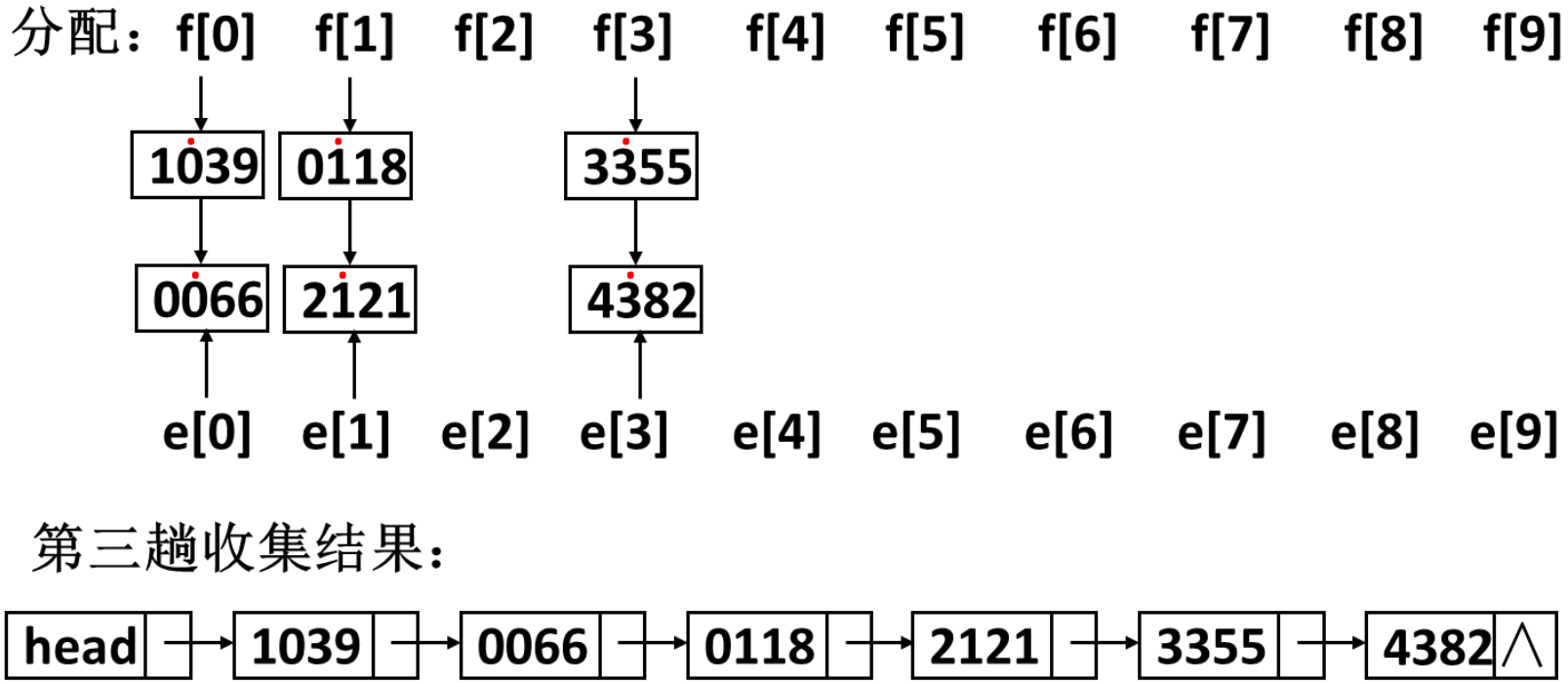
以LSD方法进行链式基数排序的过程

基数排序 (Radix Sort)



以LSD方法进行链式基数排序的过程

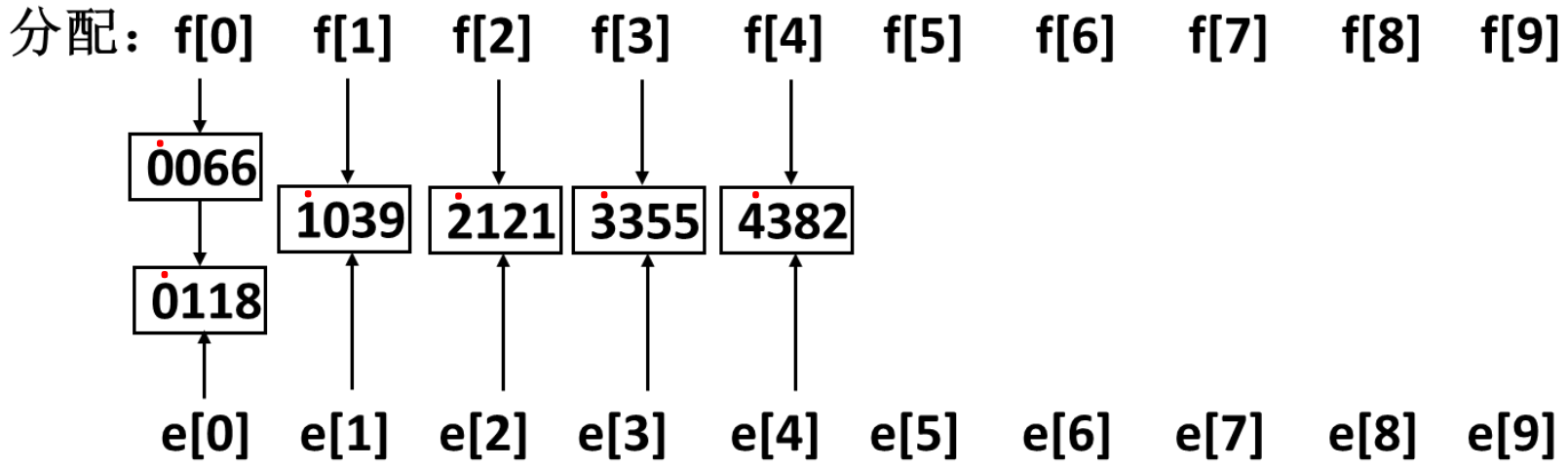
基数排序 (Radix Sort)



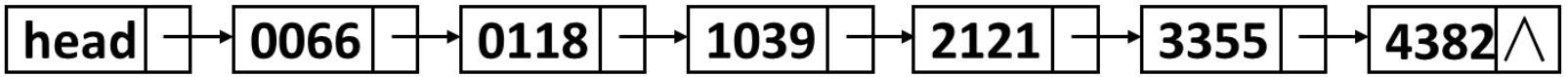
第三趟收集结果：

以LSD方法进行链式基数排序的过程

基数排序 (Radix Sort)



第四趟收集结果：



以LSD方法进行链式基数排序的过程

基数排序 (Radix Sort)

- 算法分析

设有n个待排序记录，关键字位数为 d，每位有 r 种取值。则排序的趟数是 d；在每一趟中：

- 链表初始化的时间复杂度：O(r) ；
- 分配的时间复杂度：O(n) ；
- 分配后收集的时间复杂度：O(r) ；

则链式基数排序的时间复杂度为： O(d(n+r))

在排序过程中使用的辅助空间是：2r 个链表指针， n 个指针域空间，则空间复杂度为：O(n+r)

基数排序是稳定的。

各种内部排序的比较

各种内部排序按所采用的基本思想(策略)可分为：插入排序、交换排序、选择排序、归并排序和基数排序，它们的基本策略分别是：

1 **插入排序**：依次将无序序列中的一个记录，按关键字值的大小插入到已排好序一个子序列的适当位置，直到所有的记录都插入为止。具体的方法有：直接插入和希尔排序。

2 **交换排序**：对于待排序记录序列中的记录，两两比较记录的关键字，并对反序的两个记录进行交换，直到整个序列中没有反序的记录偶对为止。具体的方法有：冒泡排序、快速排序。

各种内部排序的比较

3 选择排序：不断地从待排序的记录序列中选取关键字最小的记录，放在已排好序的序列的最后，直到所有记录都被选取为止。具体的方法有：简单选择排序、堆排序。

4 归并排序：利用“归并”技术不断地对待排序记录序列中的有序子序列进行合并，直到合并为一个有序序列为止。

5 基数排序：按待排序记录的关键字的组成成分(“位”)从低到高 (或从高到低)进行。每次是按记录关键字某一“位”的值将所有记录分配到相应的桶中，再按桶的编号依次将记录进行收集，最后得到一个有序序列。

各种内部排序的比较

- 各种内部排序方法的性能比较如下表

方法	平均时间	最坏所需时间	附加空间	稳定性
直接插入	$O(n^2)$	$O(n^2)$	$O(1)$	稳定的
希尔排序	$O(n^{1.3})$		$O(1)$	不稳定的
直接选择	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定的
堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	不稳定的
冒泡排序	$O(n^2)$	$O(n^2)$	$O(1)$	稳定的
快速排序	$O(n\log_2 n)$	$O(n^2)$	$O(\log_2 n)$	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	$O(d(n+r))$	$O(d(n+r))$	$O(n+r)$	稳定的

各种内部排序的比较

- 选取排序方法的主要考虑因素
  - 待排序的记录数目 n；
  - 每个记录的大小；
  - 关键字的结构及其初始状态；
  - 是否要求排序的稳定性；
  - 语言工具的特性；
  - 存储结构的初始条件和要求；
  - 时间复杂度、空间复杂度和开发工作的复杂程度的平衡点等。