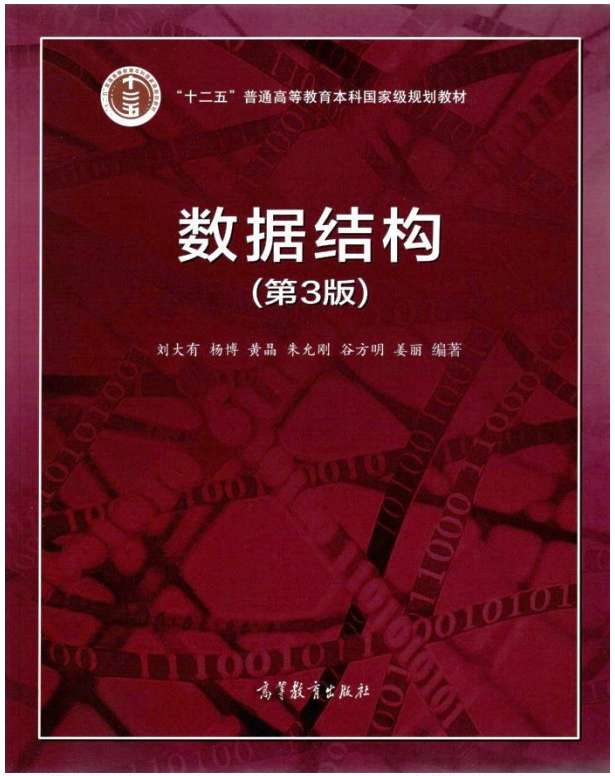
think.create.solve



归并排序及其他

- 归并排序
- 排序算法时间下界
- 分布排序
- 外排序方法简介

数据之法 结构之美 算法之道

zhuyungang@jlu.edu.cn

Last updated on 2022.11

归并排序算法提出者

John von Neumann

冯·诺依曼

美国科学院院士

普林斯顿大学教授

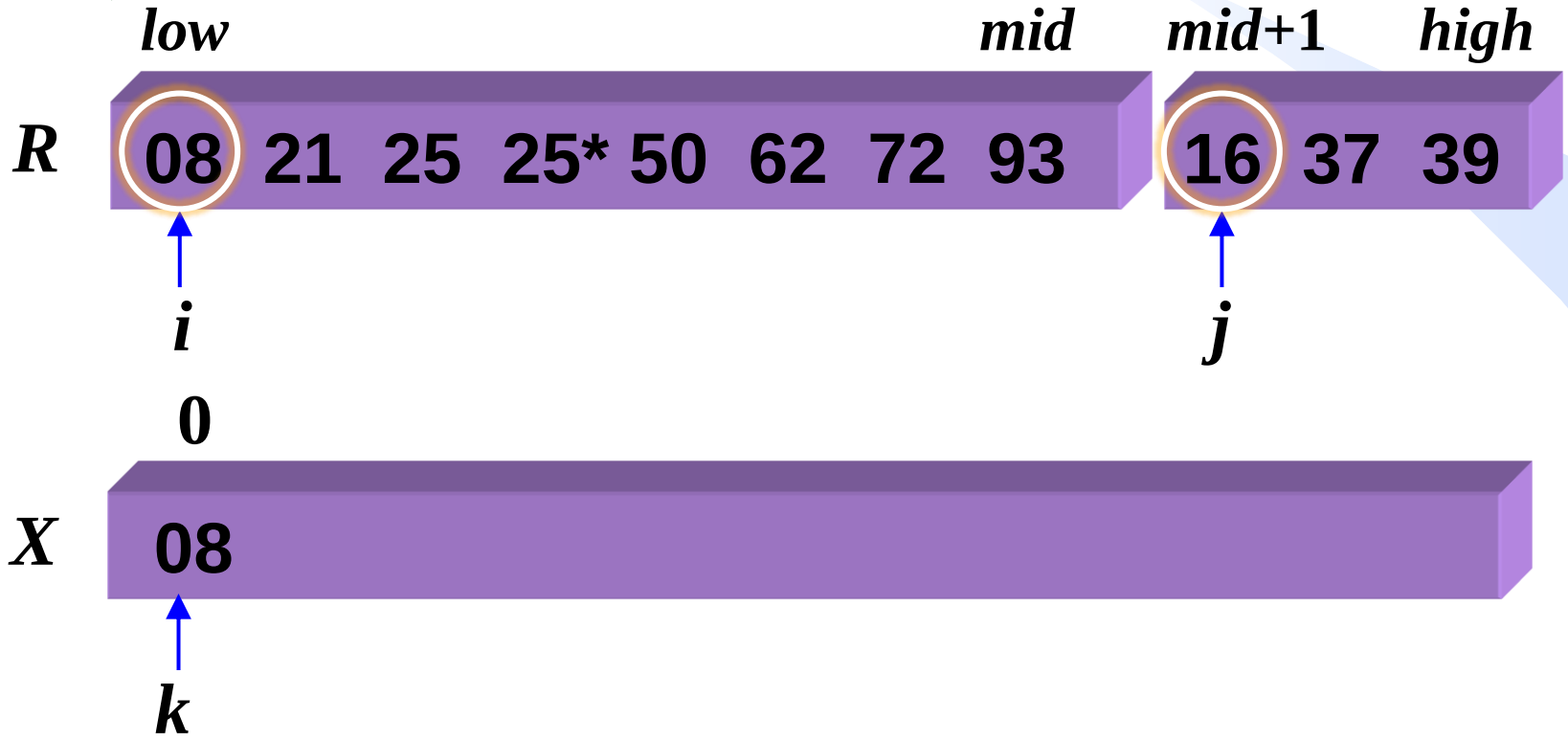
现代计算机之父

If people do not believe that mathematics is simple, it is only because they do not realize how complicated life is.

如果有人不相信数学是简单的，那是因为他们没有意识到生活有多复杂。

两个有序文件合并成一个大的有序文件

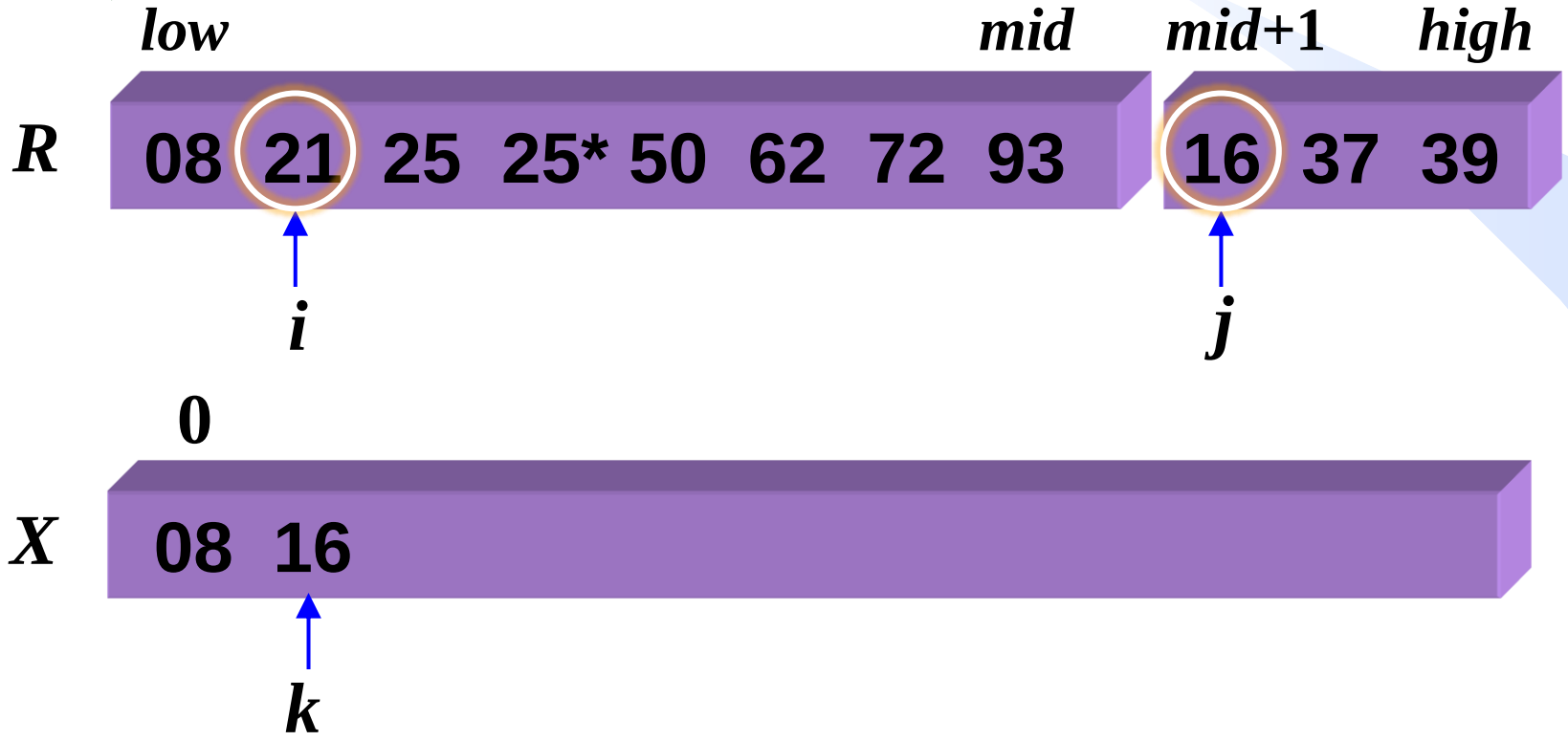
```
void Merge(int R[],int low, int mid, int high){
/*将两个相邻的有序数组(Rlow,Rlow+1,…,Rmid)和(Rmid+1,Rmid+2,…,Rhigh)
合并成一个有序数组*/
```



- 通过 *i* 和 *j* 扫描两个文件，R[*i*]与R[*j*]的关键词较小者放入*X*[*k*]中
- 当一个文件已扫描完毕后，将另一个文件中的剩余部分直接放入*X*中

两个有序文件合并成一个大的有序文件

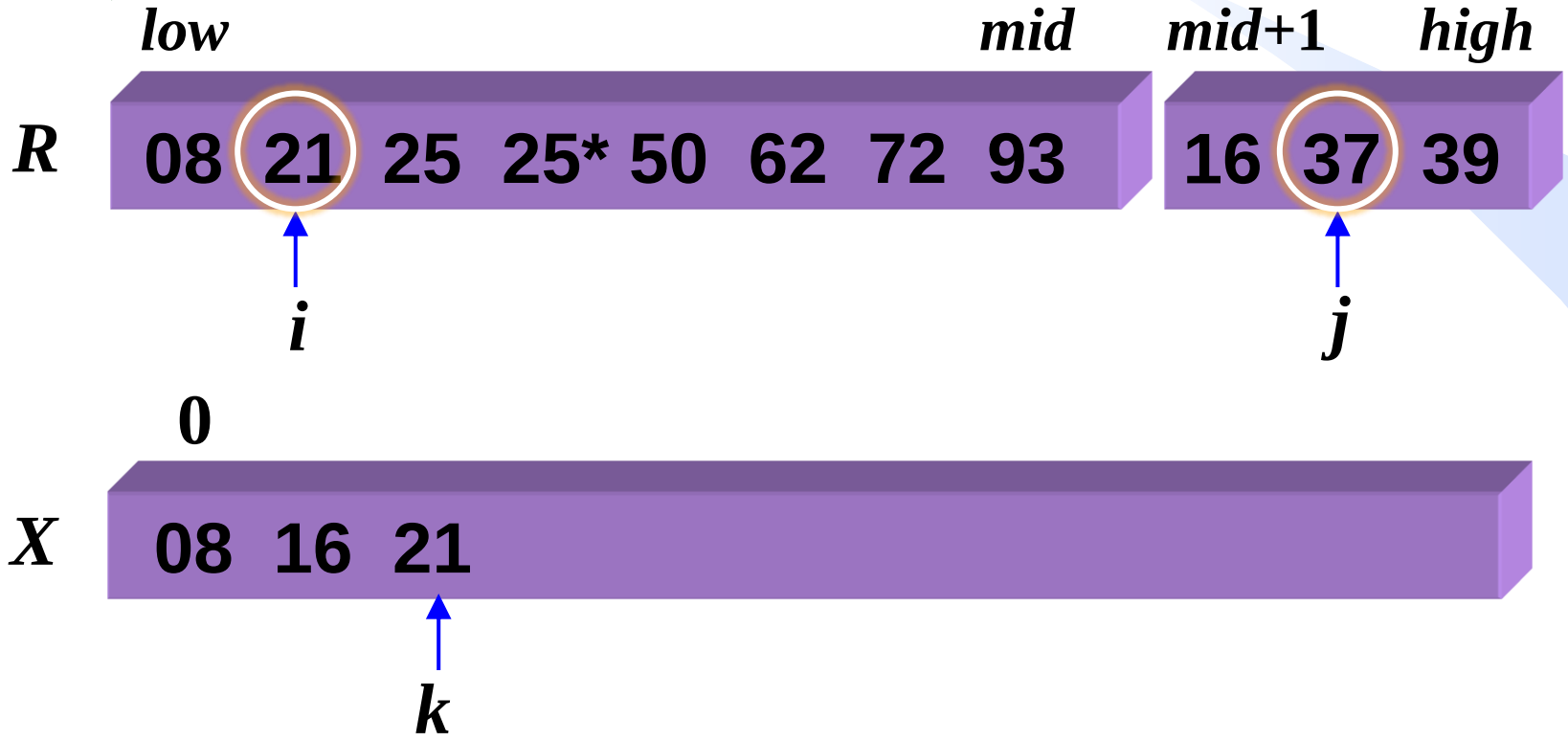
```
void Merge(int R[],int low, int mid, int high){
/*将两个相邻的有序数组(Rlow,Rlow+1,…,Rmid)和(Rmid+1,Rmid+2,…,Rhigh)
合并成一个有序数组*/
```



- 通过 *i* 和 *j* 扫描两个文件，R[*i*]与R[*j*]的关键词较小者放入*X*[*k*]中
- 当一个文件已扫描完毕后，将另一个文件中的剩余部分直接放入*X*中

两个有序文件合并成一个大的有序文件

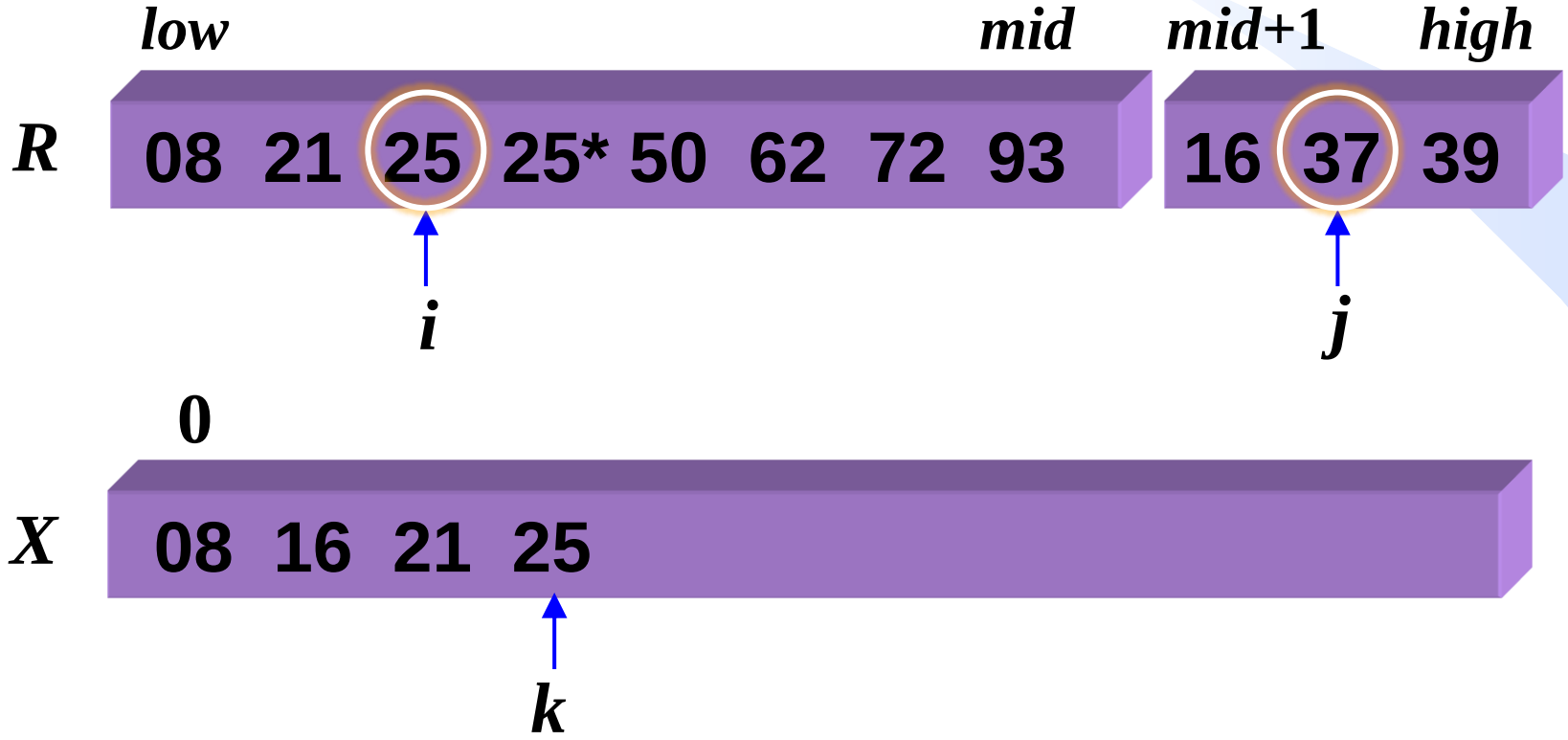
```
void Merge(int R[],int low, int mid, int high){
/*将两个相邻的有序数组(Rlow,Rlow+1,…,Rmid)和(Rmid+1,Rmid+2,…,Rhigh)
合并成一个有序数组*/
```



- 通过 *i* 和 *j* 扫描两个文件，R[*i*]与R[*j*]的关键词较小者放入*X*[*k*]中
- 当一个文件已扫描完毕后，将另一个文件中的剩余部分直接放入*X*中

两个有序文件合并成一个大的有序文件

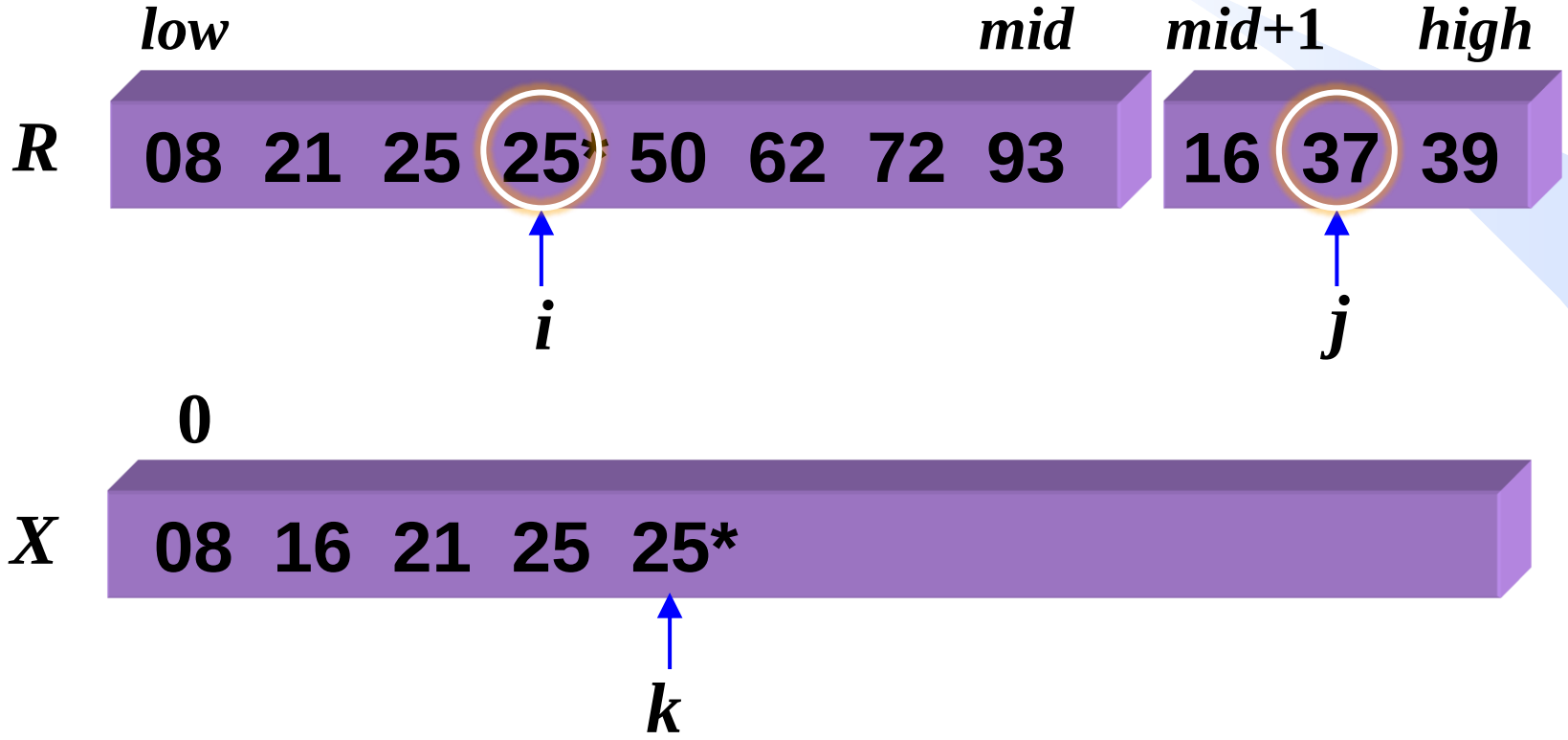
```
void Merge(int R[],int low, int mid, int high){
/*将两个相邻的有序数组(Rlow,Rlow+1,…,Rmid)和(Rmid+1,Rmid+2,…,Rhigh)
合并成一个有序数组*/
```



- 通过 *i* 和 *j* 扫描两个文件，R[*i*]与R[*j*]的关键词较小者放入*X*[*k*]中
- 当一个文件已扫描完毕后，将另一个文件中的剩余部分直接放入*X*中

两个有序文件合并成一个大的有序文件

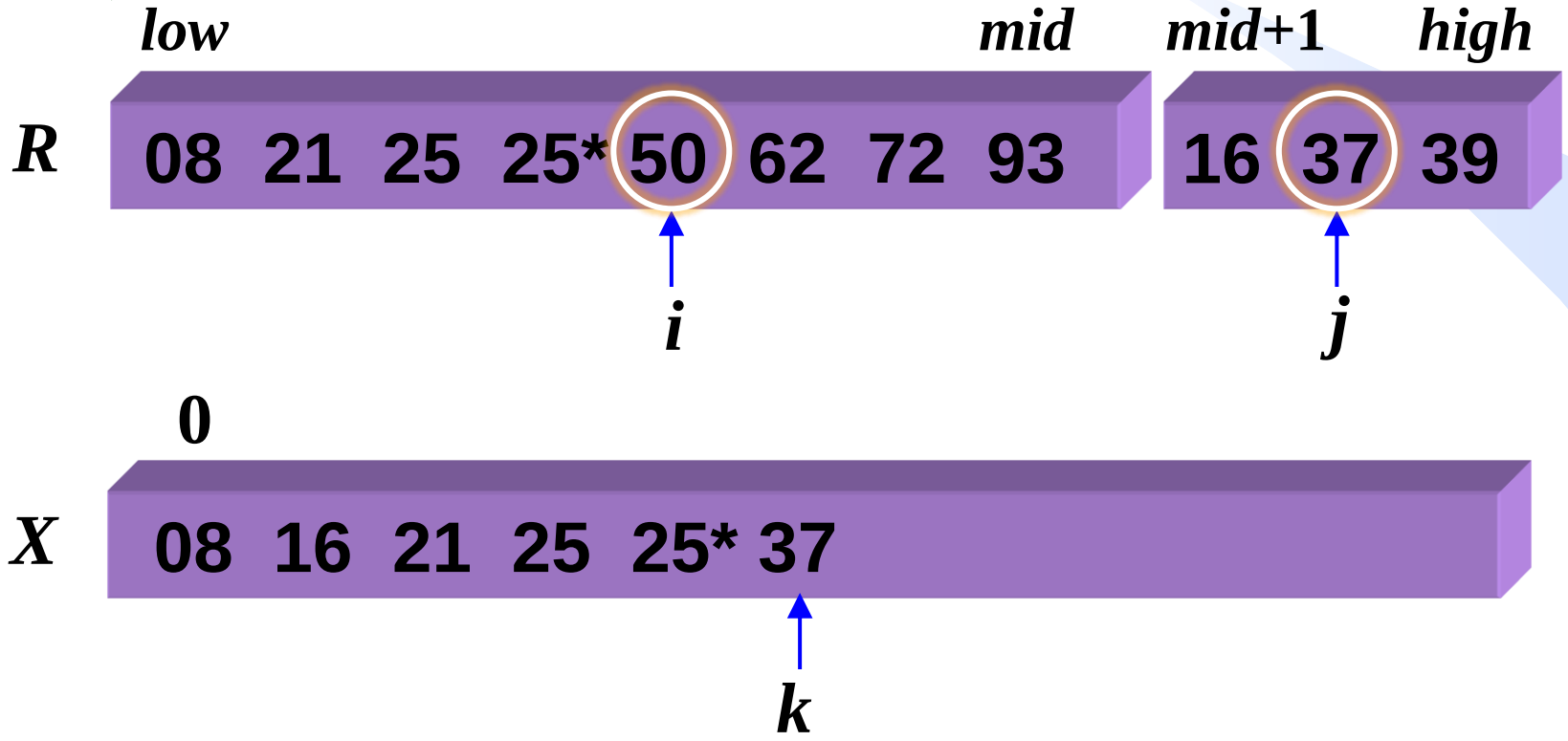
```
void Merge(int R[],int low, int mid, int high){
/*将两个相邻的有序数组(Rlow,Rlow+1,…,Rmid)和(Rmid+1,Rmid+2,…,Rhigh)
合并成一个有序数组*/
```



- 通过 *i* 和 *j* 扫描两个文件，R[*i*]与R[*j*]的关键词较小者放入*X*[*k*]中
- 当一个文件已扫描完毕后，将另一个文件中的剩余部分直接放入*X*中

两个有序文件合并成一个大的有序文件

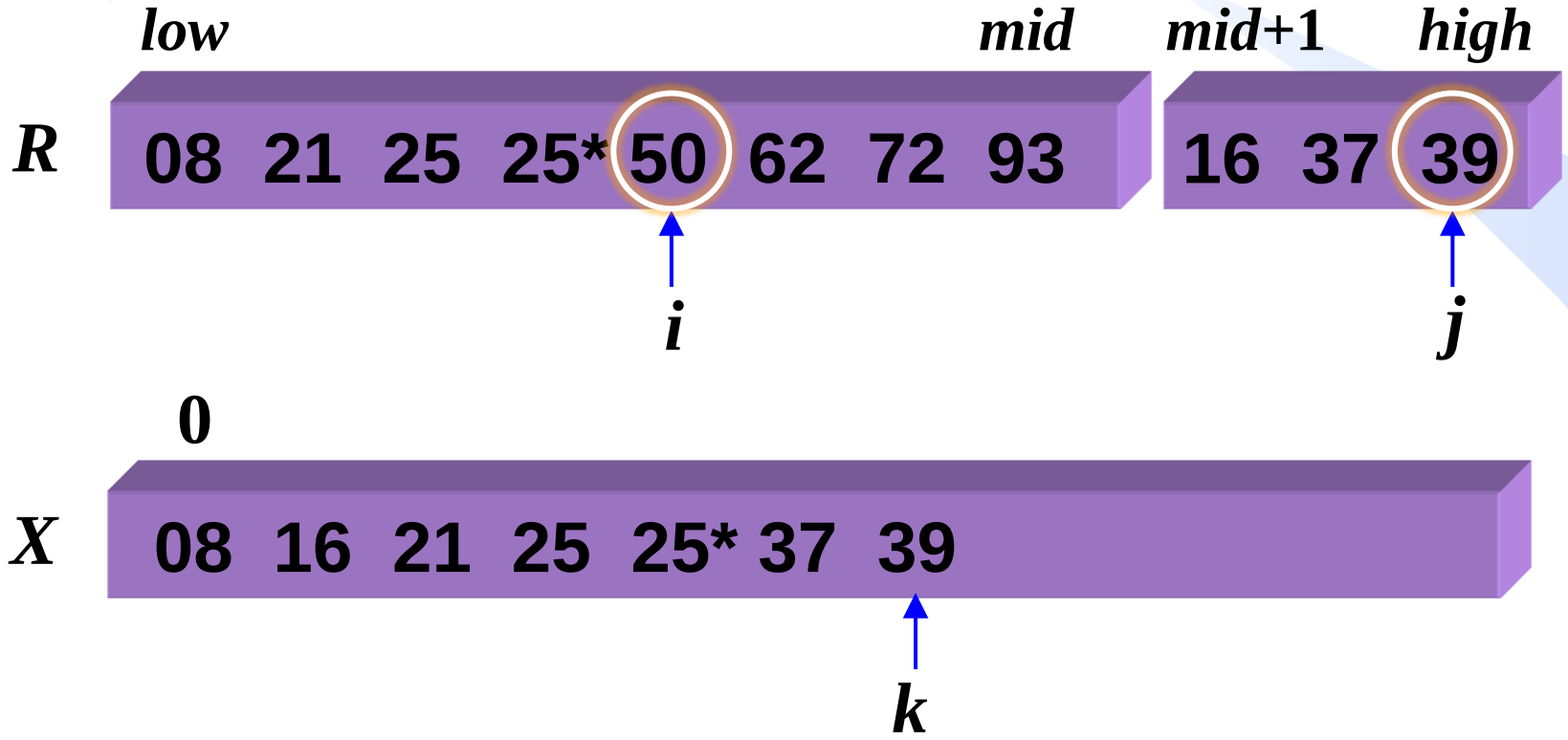
```
void Merge(int R[],int low, int mid, int high){
/*将两个相邻的有序数组(Rlow,Rlow+1,…,Rmid)和(Rmid+1,Rmid+2,…,Rhigh)
合并成一个有序数组*/
```



- 通过 *i* 和 *j* 扫描两个文件，R[*i*]与R[*j*]的关键词较小者放入*X*[*k*]中
- 当一个文件已扫描完毕后，将另一个文件中的剩余部分直接放入*X*中

两个有序文件合并成一个大的有序文件

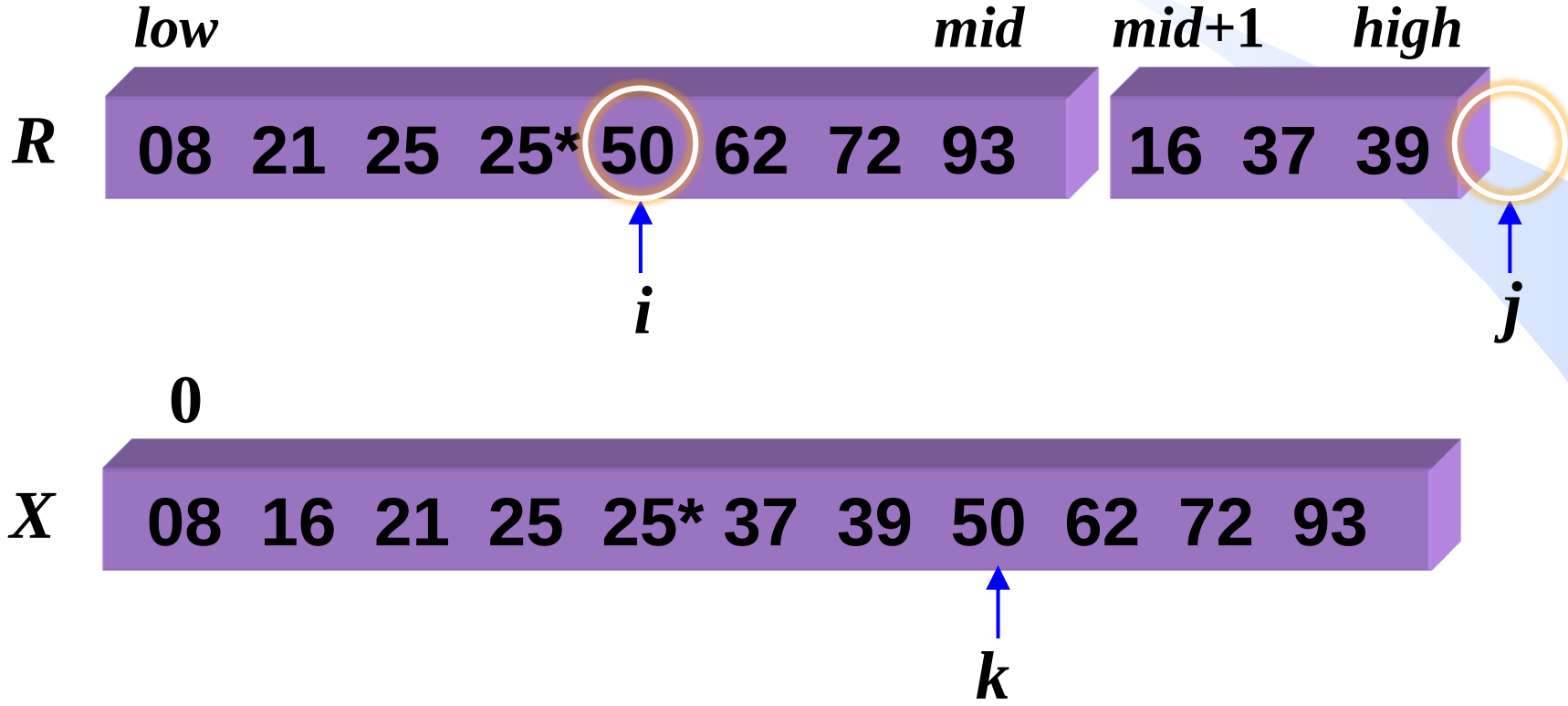
```
void Merge(int R[],int low, int mid, int high){
/*将两个相邻的有序数组(Rlow,Rlow+1,…,Rmid)和(Rmid+1,Rmid+2,…,Rhigh)
合并成一个有序数组*/
```



- 通过 *i* 和 *j* 扫描两个文件，R[*i*]与R[*j*]的关键词较小者放入*X*[*k*]中
- 当一个文件已扫描完毕后，将另一个文件中的剩余部分直接放入*X*中

两个有序文件合并成一个大的有序文件

```
void Merge(int R[],int low, int mid, int high){
/*将两个相邻的有序数组(Rlow,Rlow+1,…,Rmid)和(Rmid+1,Rmid+2,…,Rhigh)
合并成一个有序数组*/
```



- 通过 *i* 和 *j* 扫描两个文件，R[*i*]与R[*j*]的关键词较小者放入*X*[*k*]中
- 当一个文件已扫描完毕后，将另一个文件中的剩余部分直接放入*X*中

两个有序文件合并成一个大的有序文件

```
void Merge(int R[],int low, int mid, int high){
/*将两个相邻的有序数组(R_low,R_low+1,…,R_mid)和(R_mid+1,R_mid+2,…,R_high)
合并成一个有序数组*/
    int i=low, j=mid+1, k=0;
    int *X=new int[high-low+1];
    while(i<=mid  && j<=high)
        if(R[i]<=R[j]) X[k++]=R[i++];
        else X[k++]=R[j++];
    while(i<=mid) X[k++]=R[i++]; //复制余留记录
    while(j<=high) X[k++]=R[j++];
    for(int k=0,i=low; i<=high; i++,k++) //将X拷贝回R
        R[i]=X[k];
    delete []X;
}
```

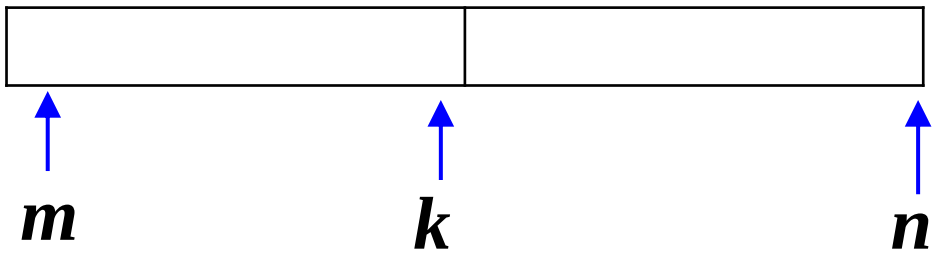
时间复杂度O(*n*)
空间复杂度O(*n*)

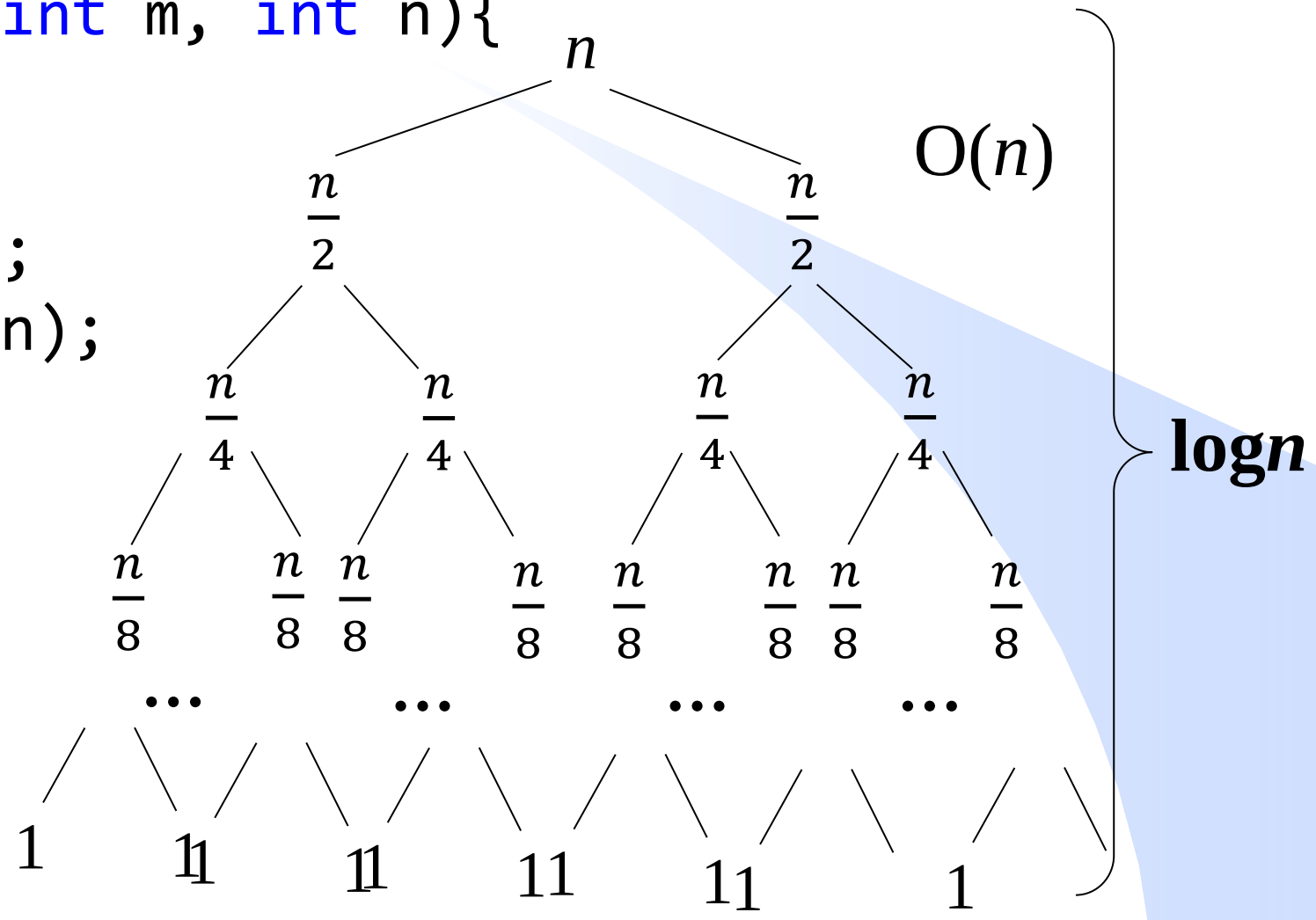
归并排序（递归形式）

分治法

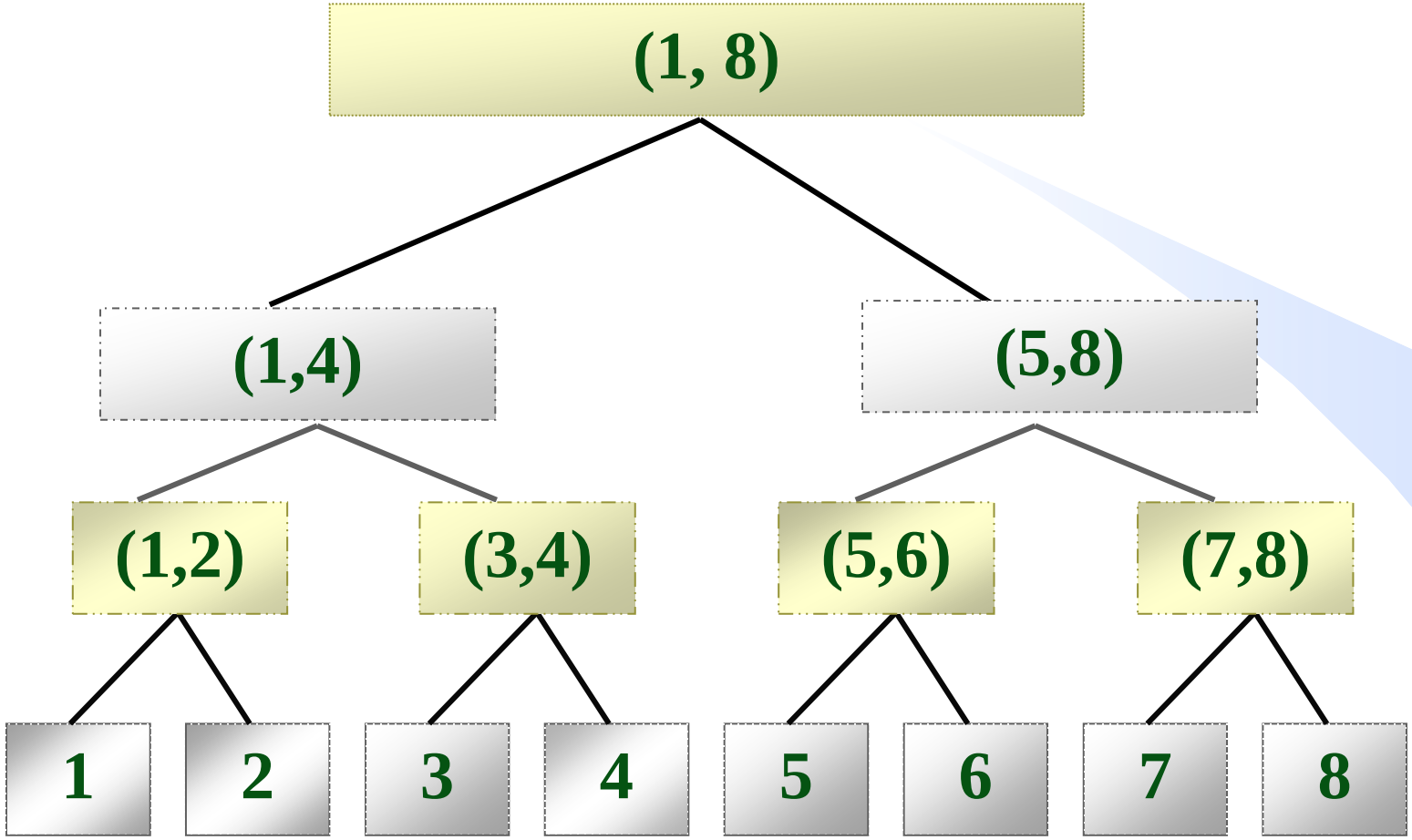
```
void MergeSort(int R[], int m, int n){
    if(m < n){
        int k = (m+n)/2;
        MergeSort(R, m, k);
        MergeSort(R, k+1, n);
        Merge(R, m, k, n);
    }
}
```

时间复杂度O(*nlogn*)
空间复杂度O(*n*)



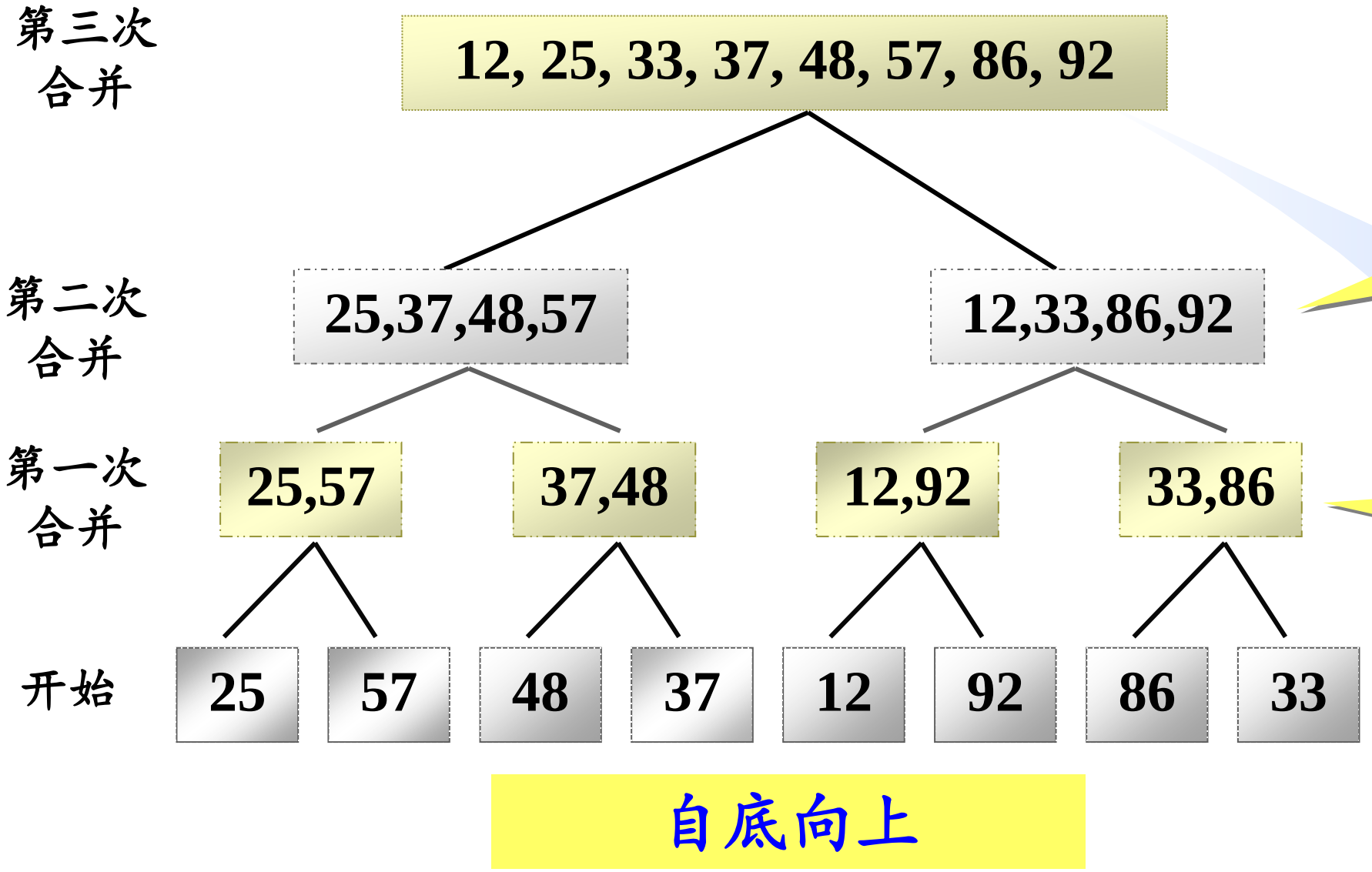


归并排序递归过程示例（对$R_1 \ldots R_8$排序）



自顶向下

非递归归并排序过程示例



对长度为4的子数组两两合并

对长度为2的子数组两两合并

对长度为1的子数组两两合并

自底向上

归并排序——非递归形式

```
void MergeSort(int R[], int n){
    for(int L=1; L<n; L*=2)
        Mpass (R, n, L);
}
```

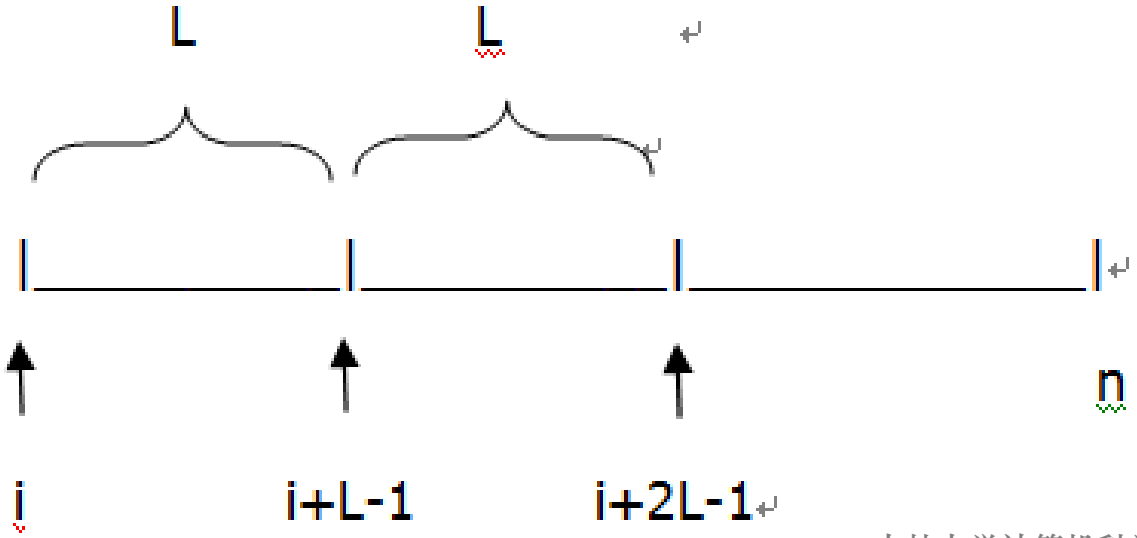
执行一趟合并过程，将数组R中长度为L的所有子数组两两合并

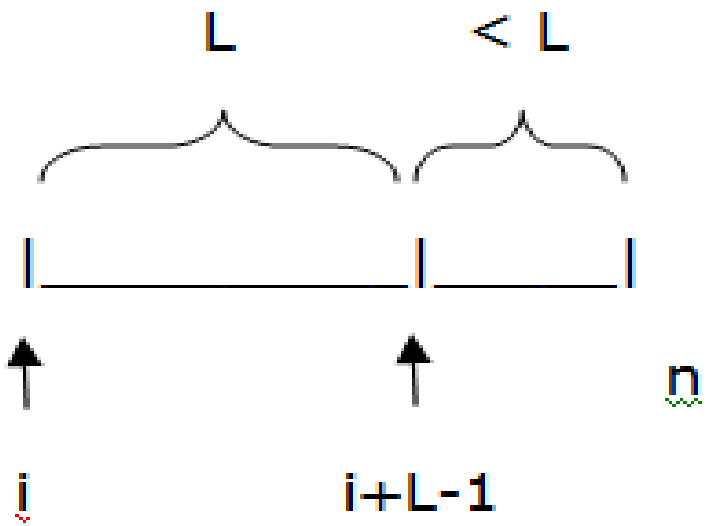
```
void Mpass(int R[],int n,int L){
   //合并相邻的两个长度为L的子数组
   for(int i=1; i+2*L-1≤n; i+=2*L)
      Merge(R, i, i＋L-1, i＋2*L-1);

   //处理余留的长度小于2*L的子数组
   if(i+L-1 < n)
      Merge(R, i, i+L-1, n);    //L<剩余部分长度<2L
}
```

执行一趟合并过程，将数组R中长度为L的所有子数组两两合并

时间复杂度O(n)





归并排序——非递归形式

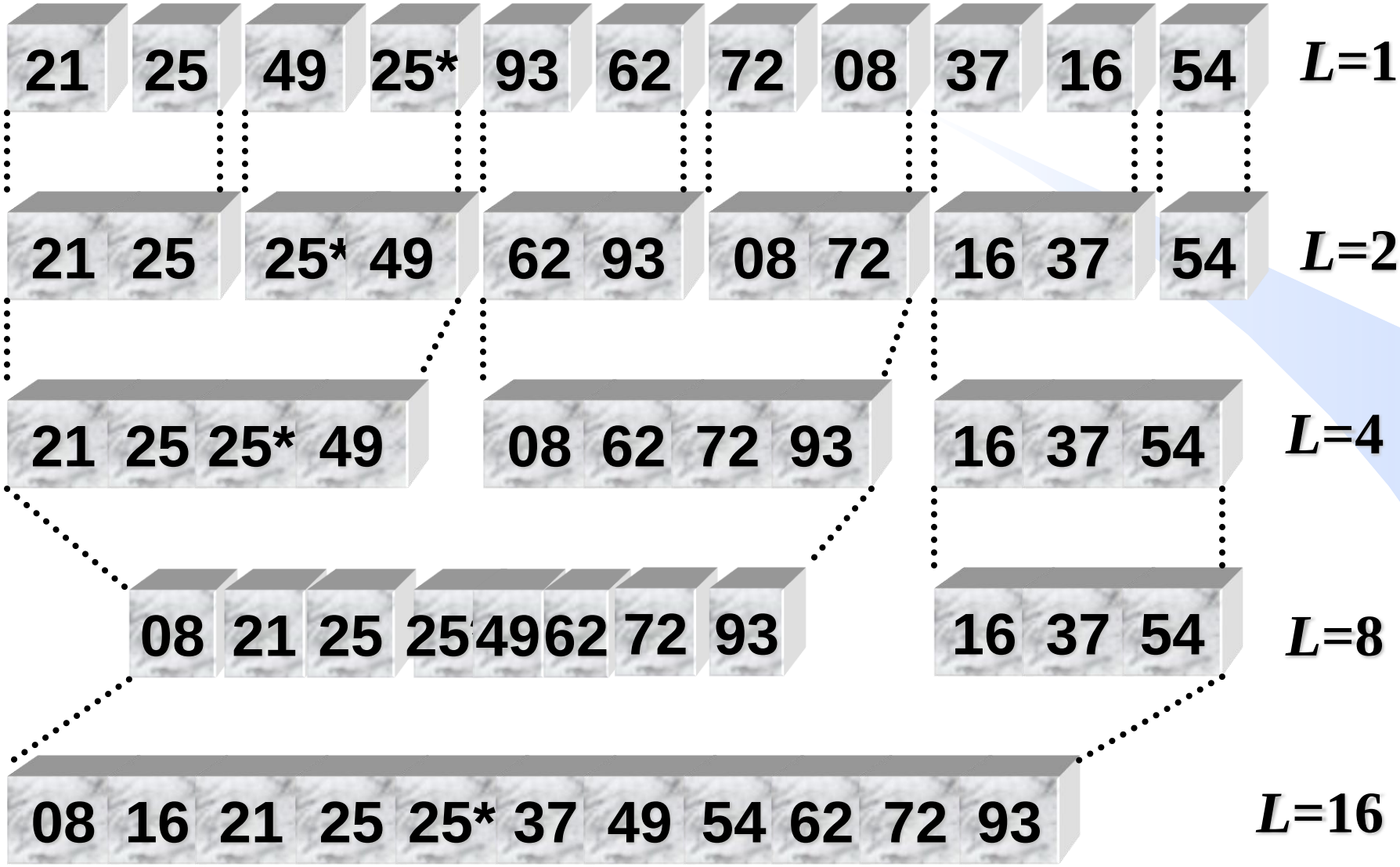
```
void MergeSort(int R[], int n){
    for(int L=1; L<n; L*=2)
        Mpass (R, n, L);
}
```

时间复杂度
O(*n*log*n*)

稳定性：若关键词相等，先放i指向的元素

```
void Merge(int R[],int low, int mid, int high){
/*将两个相邻的有序数组(R_low,R_low+1,…,R_mid)和(R_mid+1,R_mid+2,…,R_high)
合并成一个有序数组*/
    int i=low, j=mid＋1, k=0;
    int *X=new int[high-low+1];
    while(i<=mid  && j<=high)
        if(R[i]<=R[j]) X[k++]=R[i++];
        else X[k++]=R[j++];
    while(i<=mid) X[k++]=R[i++]; //复制余留记录
    while(j<=high) X[k++]=R[j++];
    for(int k=0,i=low; i<=high; i++,k++) //将X拷贝回R
        R[i]=X[k];
    delete []X;
}
```

归并排序算法及稳定性示例



归并排序算法总结

- 时间复杂度O(*n*log*n*)
- 空间复杂度O(*n*)
- 稳定性：稳定
- 最快的稳定性算法

归并排序优化策略

- 问题：当数据量非常小时，若仍然采用分治策略，效率不高。
- 优化策略：对于非常小的数据集，以及前几次合并动作，调用直接插入排序算法。

- 问题：Merge操作基于元素移动，当元素比较大时，赋值操作会比较费时。
- 优化策略：将数组存储改为链表存储，这样记录移动就变为指针移动了。

课下思考

对单链表进行排序，哪种排序算法最适合？【腾讯、华为、阿里、字节跳动、百度、快手、美团、谷歌、微软、苹果面试题】

思考

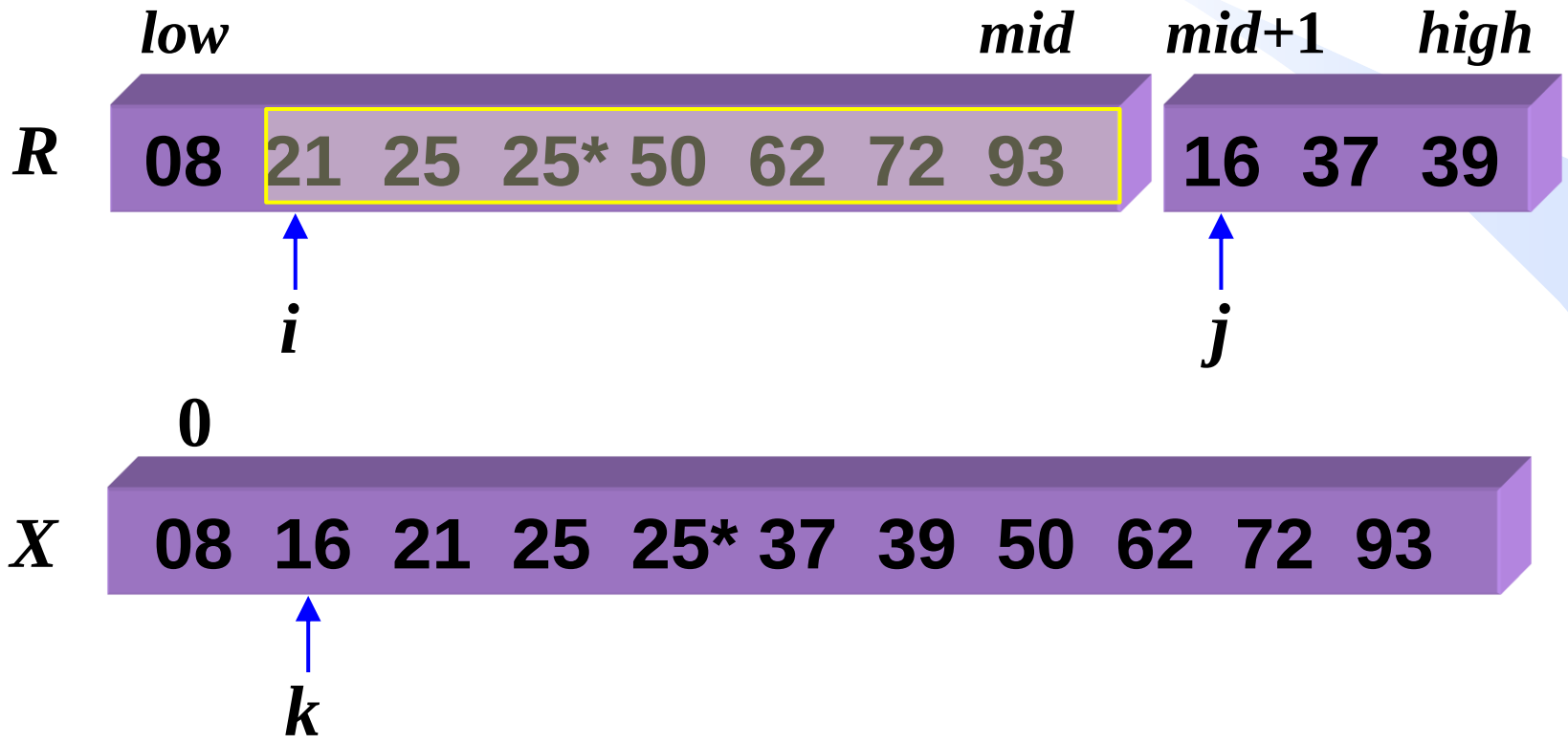
求数组中逆序对的个数【腾讯、阿里、字节跳动、百度、小米、美团、谷歌、京东、滴滴面试题】

```
int cnt=0;
for(int i=1;i<=n;i++)
    for(int j=i+1;j<=n;j++)
        if(R[i]>R[j]) cnt++;
```

时间复杂度$O(n^2)$

两个有序文件合并成一个大的有序文件

```
void Merge(int R[],int low, int mid, int high){
/*将两个相邻的有序数组(Rlow,Rlow+1,…,Rmid)和(Rmid+1,Rmid+2,…,Rhigh)合并成一个有序数组*/
```



R[*i*]>R[*j*] ⇔ R[*i*]…R[*mid*] >R[*j*] ⇔ 逆序对增加*mid*-*i*+1个

```
void Merge(int R[],int low, int mid, int high){
/*将两个相邻的有序数组(Rlow,Rlow+1,…,Rmid)和(Rmid+1,Rmid+2,…,Rhigh)
合并成一个有序数组*/
    int i=low, j=mid+1, k=0, cnt=0;
    int *X=new int[high-low+1];
    while(i<=mid  && j<=high)
        if(R[i]<=R[j]) X[k++]=R[i++];
        else X[k++]=R[j++], cnt+=mid-i+1;
    while(i<=mid) X[k++]=R[i++]; //复制余留记录
    while(j<=high) X[k++]=R[j++];
    for(int k=0,i=low; i<=high; i++,k++) //将X拷贝回R
        R[i]=X[k];
    delete []X;
    return cnt;
}
```

合并排序时，每次调用Merge时，都把其算出的*cnt*累加起来

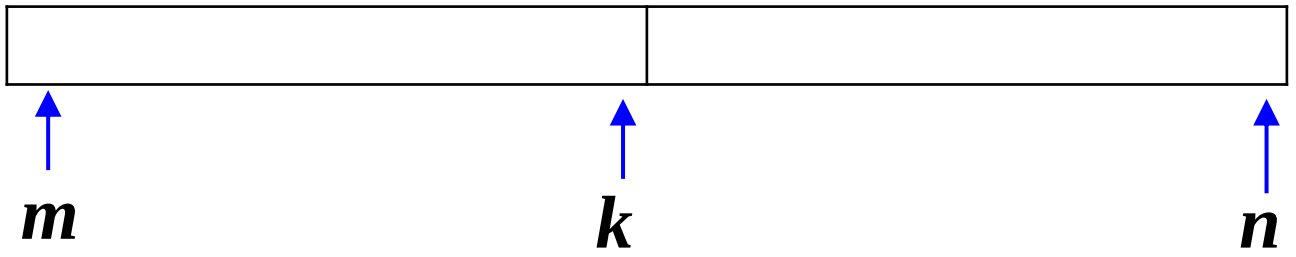
基于分治法的排序

分治法的思想：将一个输入规模为n的问题分解为k个规模较小的子问题，这些子问题互相独立且与原问题相同，然后递归地求解这些子问题，最后用适当的方法将各子问题的解合并成原问题的解。

Divide and Conquer

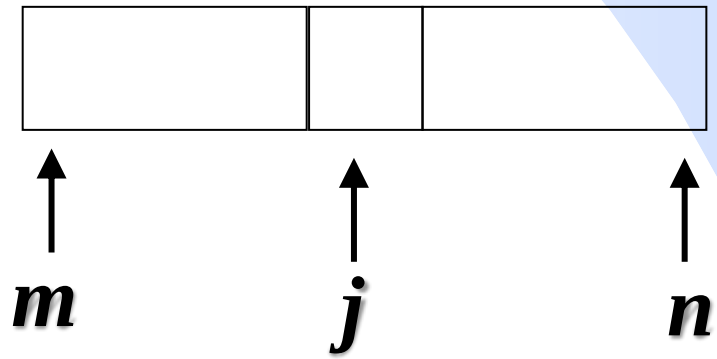
合并排序（递归形式）

```
void MergeSort(int R[], int m, int n){
    if(m < n){
        int k = (m+n)/2;
        MergeSort(R, m, k);
        MergeSort(R, k+1, n);
        Merge(R, m, k, n);
    }
}
```



快速排序

```
void QuickSort(int R[], int m, int n){
    if(m < n){
        int j=Partition(R, m, n);
        QuickSort(R, m, j-1);
        QuickSort(R, j+1, n);
    }
}
```

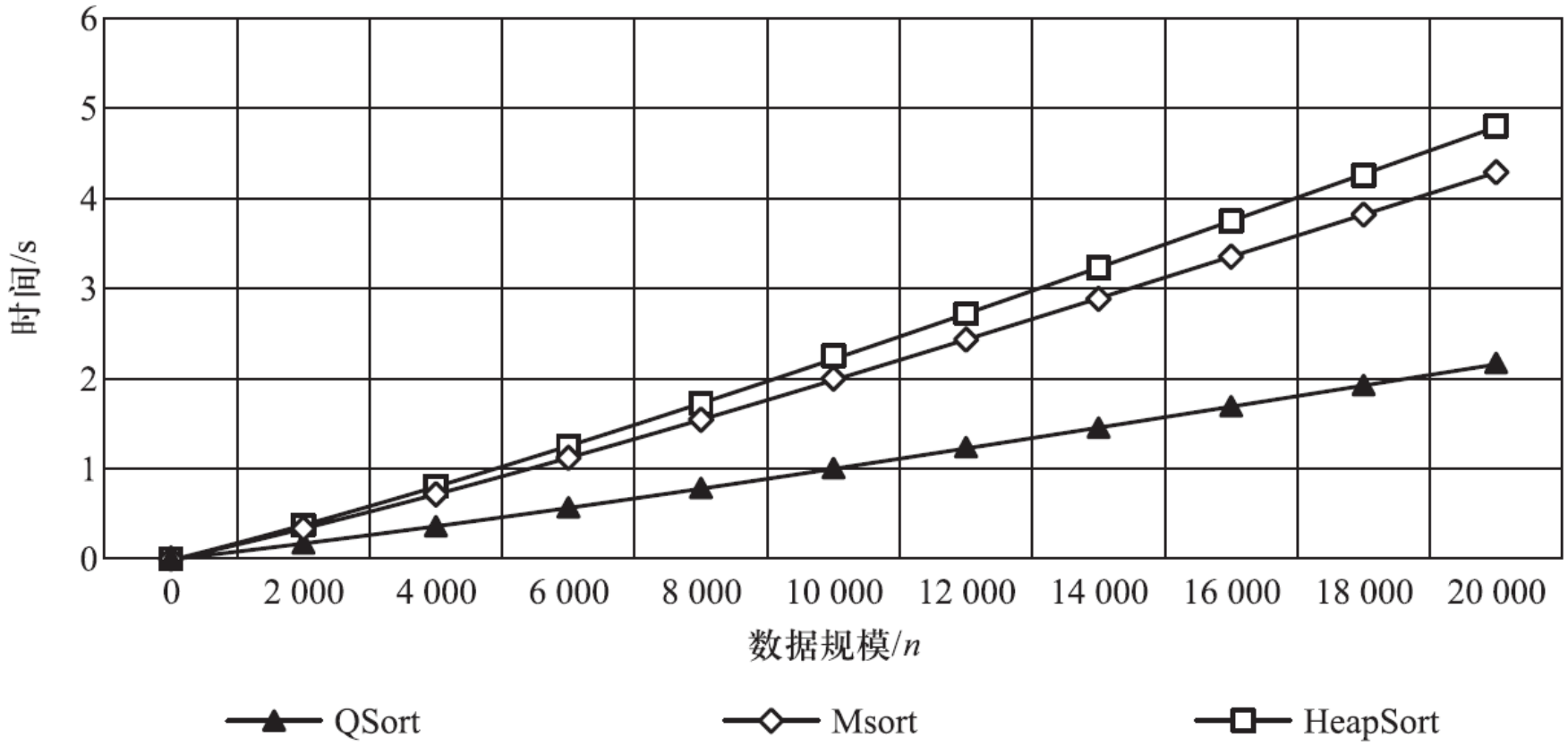


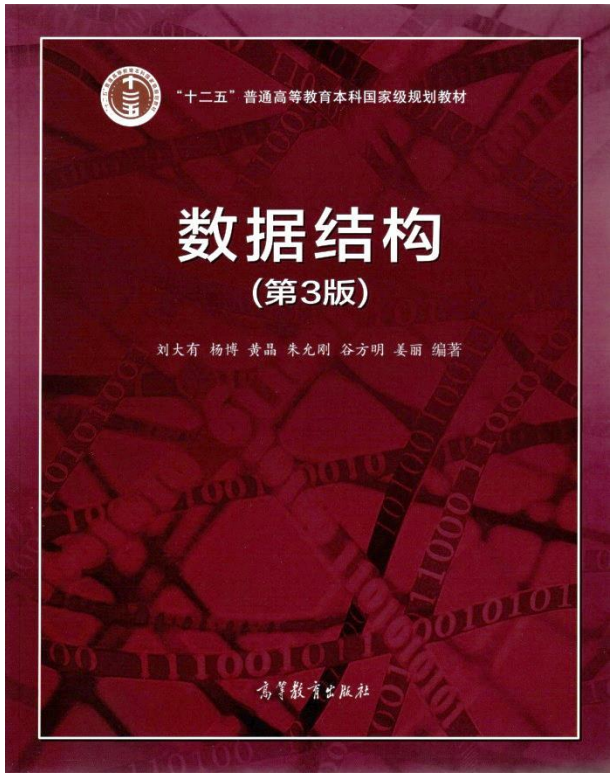
- 分治排序包括三个步骤：“分”，将记录划分为若干子部分，一般为二分，可以证明多分策略效果不会优于二分策略；“治”，对子部分递归排序；“组”，将子部分处理后的结果整合在一起。
- 由于“治”是递归调用过程，分治排序的重点在“分”和“组”两步，而实际上具体的算法往往侧重其中的某一步，如：快速排序侧重在“分”，在“分”的过程中调整了两个子集的元素，而“组”过程无需任何操作。合并排序侧重在“组”，分的过程就是简单的二分。

基于关键词比较的排序算法对比

排序方法	最好时间	平均时间	最坏时间	辅助空间	稳定性
直接插入	O(n)	$O(n^2)$	$O(n^2)$	O(1)	稳定
直接选择	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	不稳定
冒泡	O(n)	$O(n^2)$	$O(n^2)$	O(1)	稳定
希尔		$O(n^{1.25})$	$O(n^2)$	O(1)	不稳定
快速	O(nlogn)	O(nlogn)	$O(n^2)$	O(logn)	不稳定
堆	O(nlogn)	O(nlogn)	O(nlogn)	O(1)	不稳定
合并	O(nlogn)	O(nlogn)	O(nlogn)	O(n)	稳定

O(*n*log*n*)排序算法实验对比





合并排序及其他

- 合并排序
- **排序算法时间下界**
- 分布排序
- 外排序方法简介

朱允刚
zhuyungang@jlu.edu.cn

Last updated on 2021.12.6

基于关键词比较的排序算法时间下界

下界 问题的输入大小为n，若针对该问题的算法的时间复杂性下界为$L(n)$，则不存在解决该问题的算法，其时间复杂性小于$L(n)$.

K_1, K_2, K_3 有6种可能排序结构。

$K_1<K_2<K_3$

$K_3<K_2<K_1$

$K_1<K_3<K_2$

$K_3<K_1<K_2$

$K_2<K_1<K_3$

$K_2<K_3<K_1$

$K_1 : K_2$

$K_1<K_2<K_3$

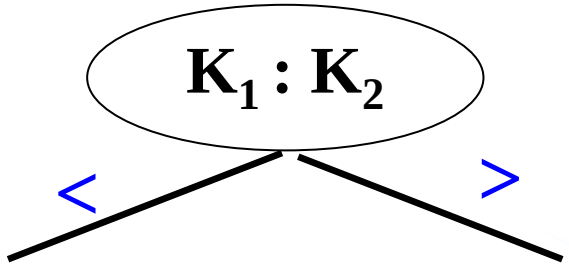
$K_3<K_2<K_1$

$K_1<K_3<K_2$

$K_3<K_1<K_2$

$K_2<K_1<K_3$

$K_2<K_3<K_1$



$K_1<K_2<K_3$

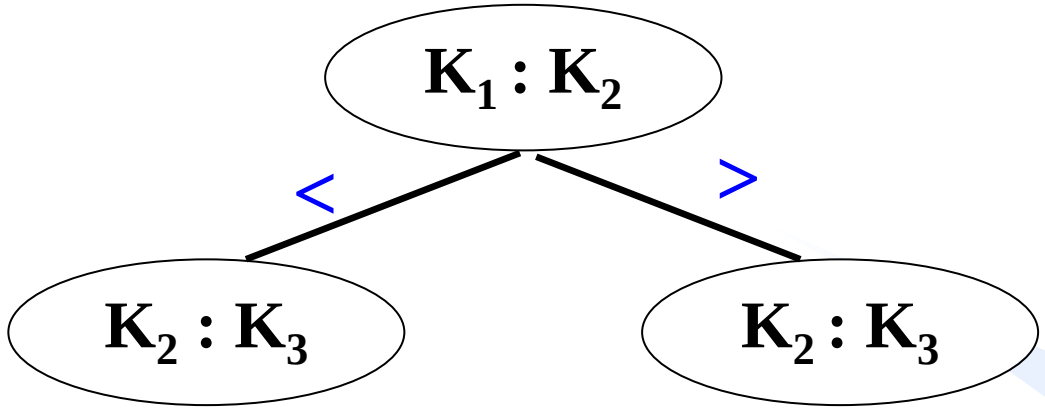
$K_3<K_2<K_1$

$K_1<K_3<K_2$

$K_3<K_1<K_2$

$K_2<K_1<K_3$

$K_2<K_3<K_1$



$K_1<K_2<K_3$

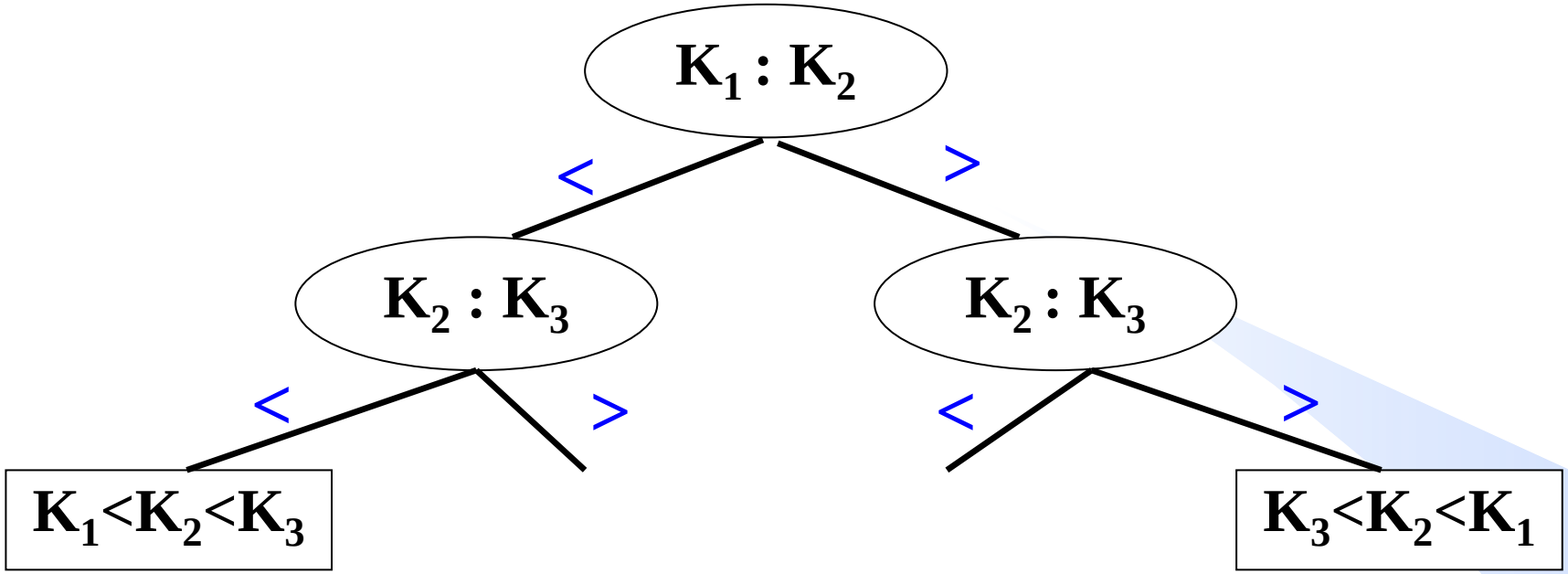
$K_3<K_2<K_1$

$K_1<K_3<K_2$

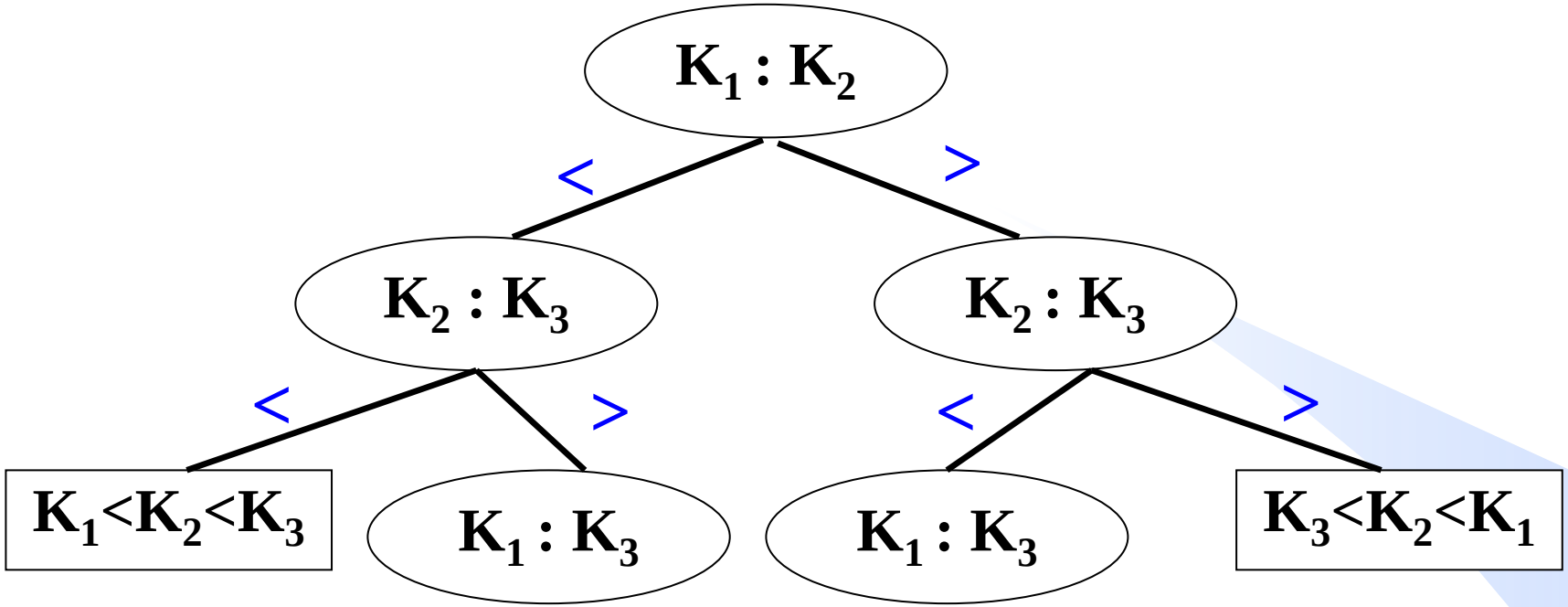
$K_3<K_1<K_2$

$K_2<K_1<K_3$

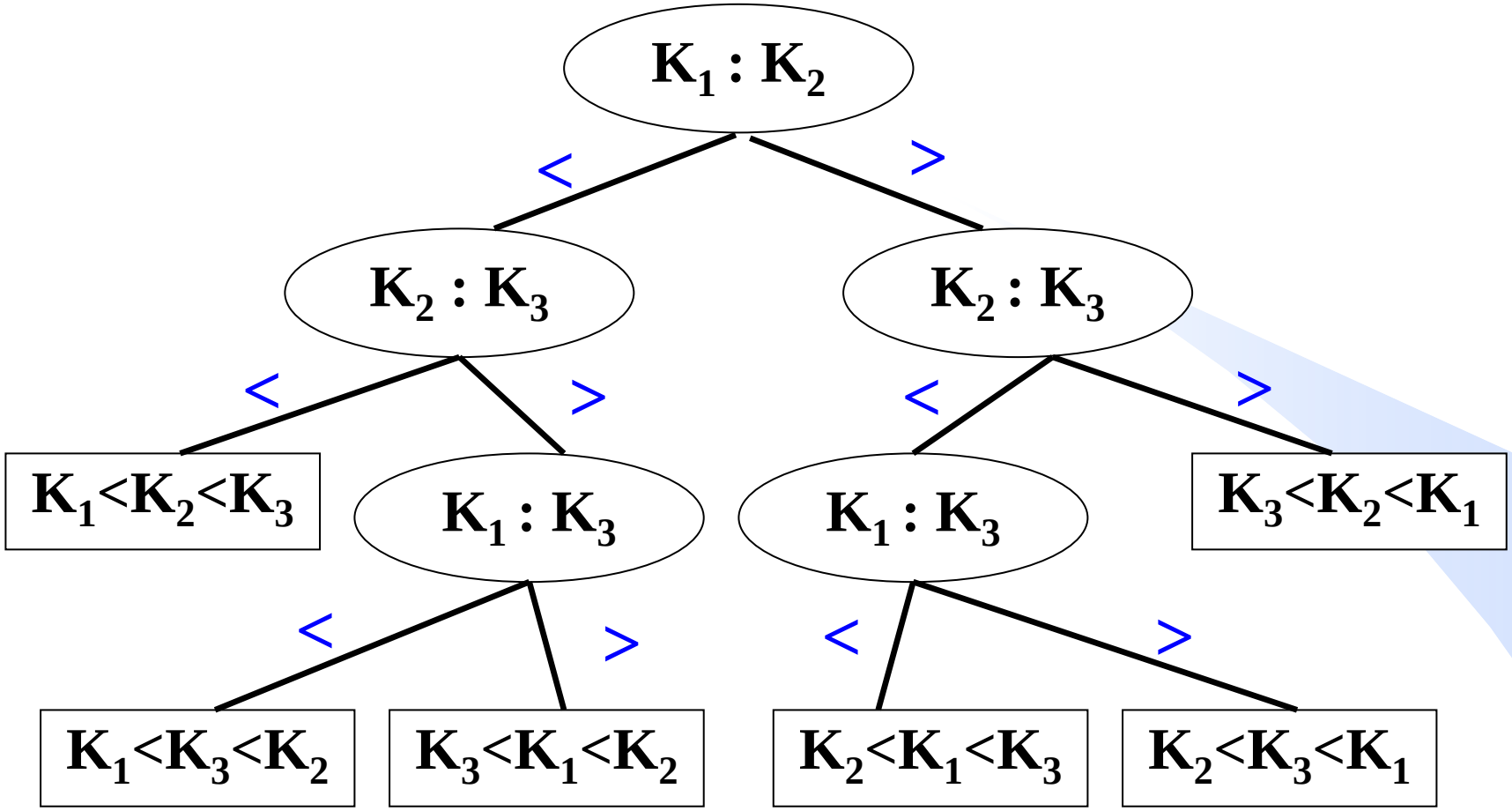
$K_2<K_3<K_1$



$K_1<K_3<K_2$ $K_3<K_1<K_2$ $K_2<K_1<K_3$ $K_2<K_3<K_1$

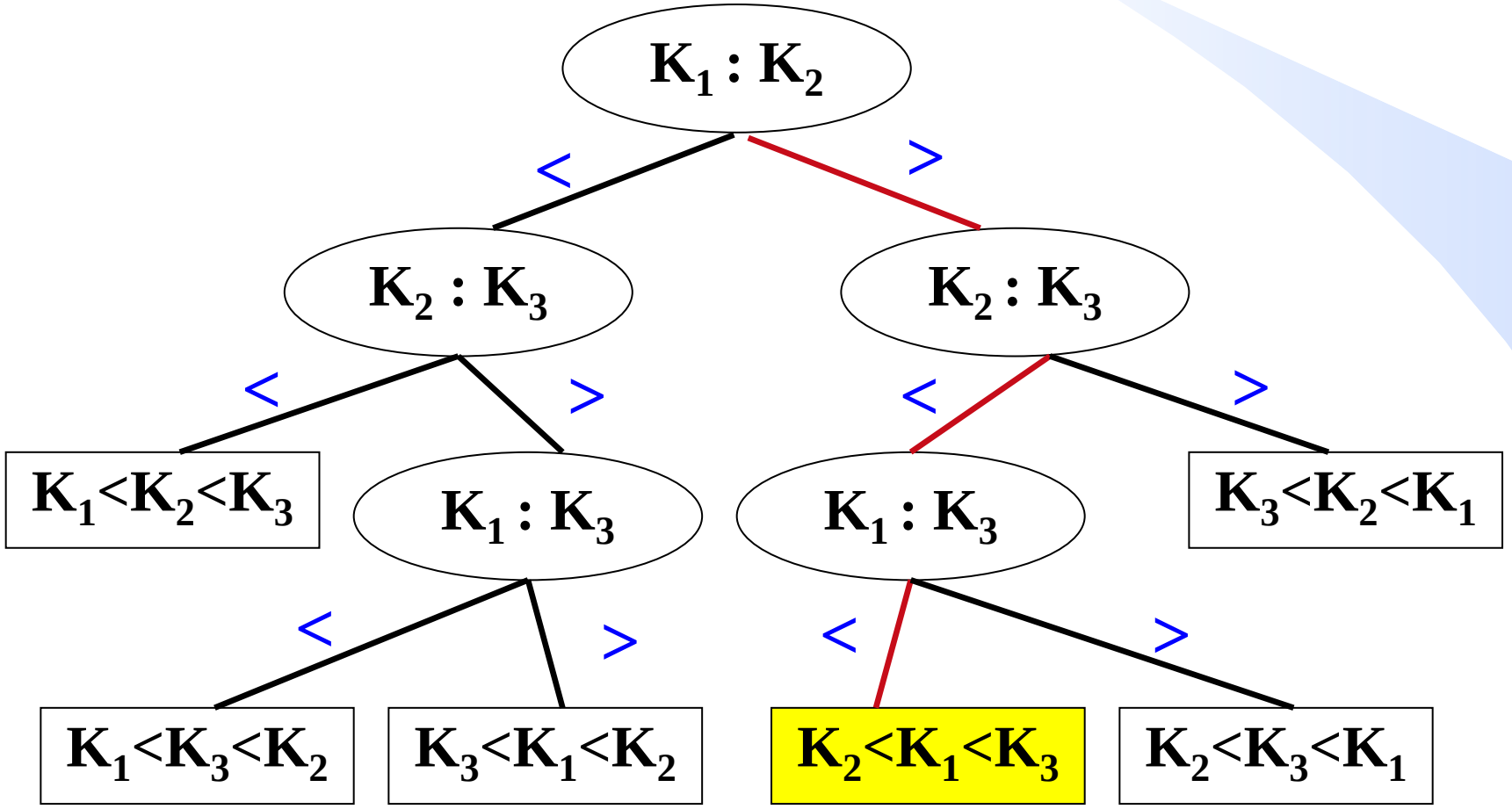


$K_1<K_3<K_2$ $K_3<K_1<K_2$ $K_2<K_1<K_3$ $K_2<K_3<K_1$


K1 : K2
<
>
K2 : K3
K2 : K3
<
>
<
>
K1<K2<K3
K1 : K3
K1 : K3
K3<K2<K1
<
>
<
>
K1<K3<K2
K3<K1<K2
K2<K1<K3
K2<K3<K1

三个元素的排序判定树

- 每一种排列看成一种输入，共有$n!$种输入。
- 对于每一种输入，排序算法所执行的比较次数至少为从根到该排列的路径长度。
- 判定树的高度是排序算法在最坏情况下关键词的比较次数



- n个元素的排列数为$n!$，故n个元素的排序判定树至少有$n!$个长方形结点（叶结点）
- 因高度为h的二叉树最多有2^h个叶结点，即叶结点个数小于等于2^h，又当前实际的叶结点个数为$n!$，故：$n! \leq 2^h$

即$2^h \geq n!$，$h \geq \log_2(n!)$,

$$\log(n!) = \log n + \log(n-1) + \log(n-2) + \ldots + \log 2 + \log 1$$

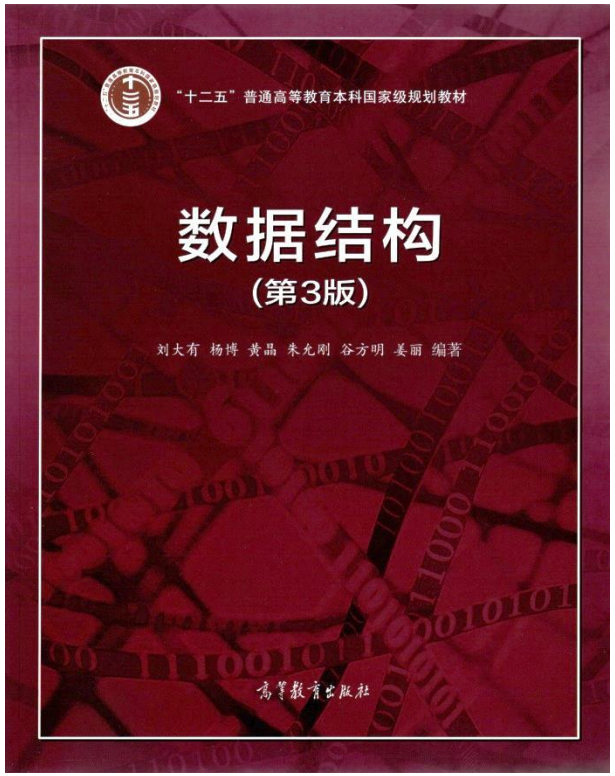
$$\geq \log n + \log(n-1) + \log(n-2) + \ldots + \log(n/2)$$

$$\geq \frac{n}{2}\log\frac{n}{2}$$

$$\geq \frac{n}{2}(\log n - \log 2) = \frac{n}{2}(\log n - 1) = \frac{n}{2}\log n - \frac{n}{2}$$

$$= \Omega(n \log n)$$

- 说明基于比较的排序算法的时间复杂性下界是$\Omega(n\log_2 n)$，即任何基于关键词比较的排序算法在最坏情况下的比较次数至少为$\Omega(n\log_2 n)$。
- 也可证明关键词的平均比较次数至少为$\Omega(n\log_2 n)$次。



合并排序及其他

- 合并排序
- 排序算法时间下界
- **分布排序**
- 外排序方法简介

朱允刚
zhuyungang@jlu.edu.cn

Last updated on 2021.12.6

基数分布

假定记录 R_1，R_2，…，R_n相应的关键词K_1，K_2，…，K_n都有如下的形式：

$$K_i = (K_{i,p}, K_{i,p-1}, \cdots, K_{i,1})$$

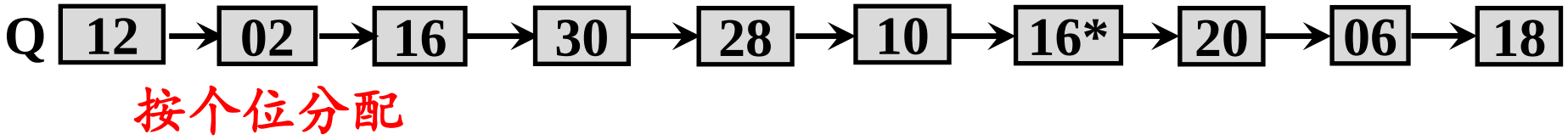
排序是按照关键词的字典序，由小到大排列。其中，字典序规定如下：

$$K_i = (K_{i,p}, \cdots, K_{i,1}) < K_j = (K_{j,p}, \cdots, K_{j,1})$$

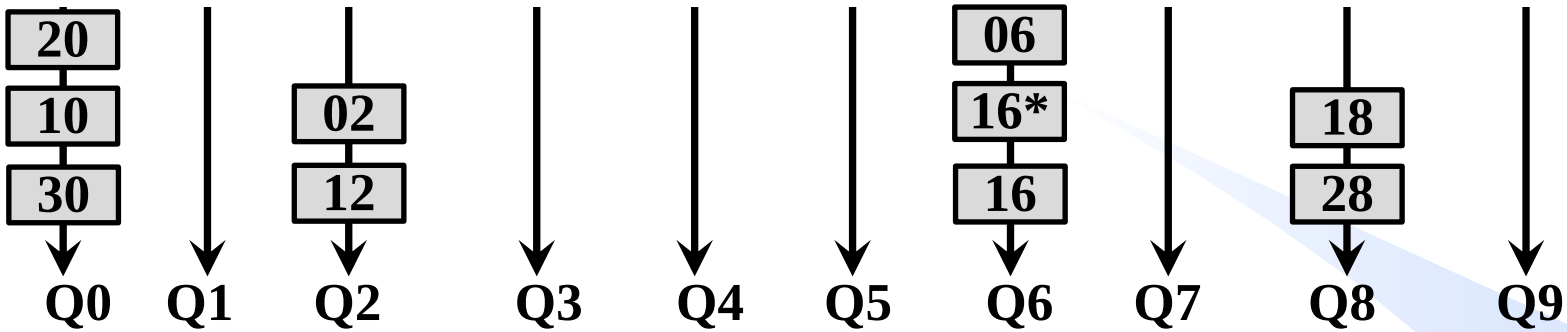
当且仅当存在$t \le p$，使得当$s > t$时

$$K_{i,s} = K_{j,s} \text{ 而 } K_{i,t} < K_{j,t} .$$

首先按最低位排序，然后按下一个次低位排序，…，
最后按最高位排序 。

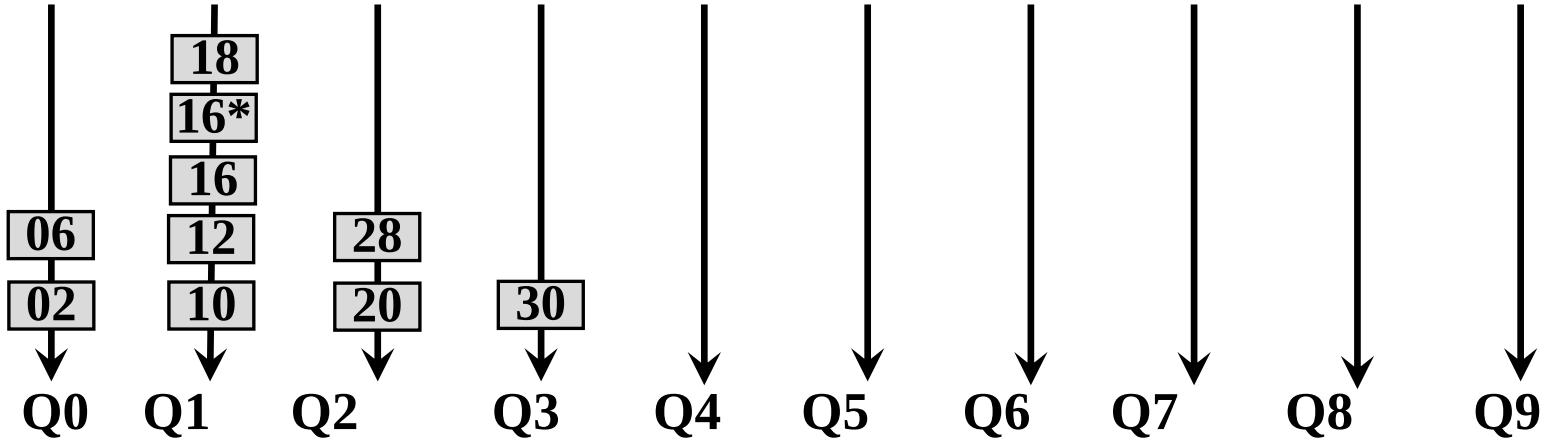

Q
12
02
16
30
28
10
16*
20
06
18

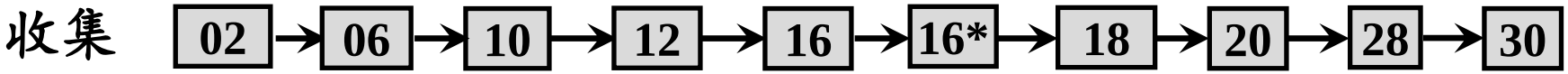
按个位分配


20
10
30
02
12
06
16*
16
18
28
Q0
Q1
Q2
Q3
Q4
Q5
Q6
Q7
Q8
Q9

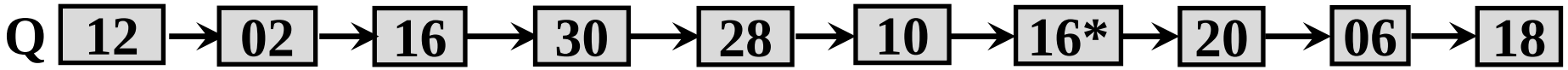

收集
30
10
20
12
02
16
16*
06
28
18

按十位分配

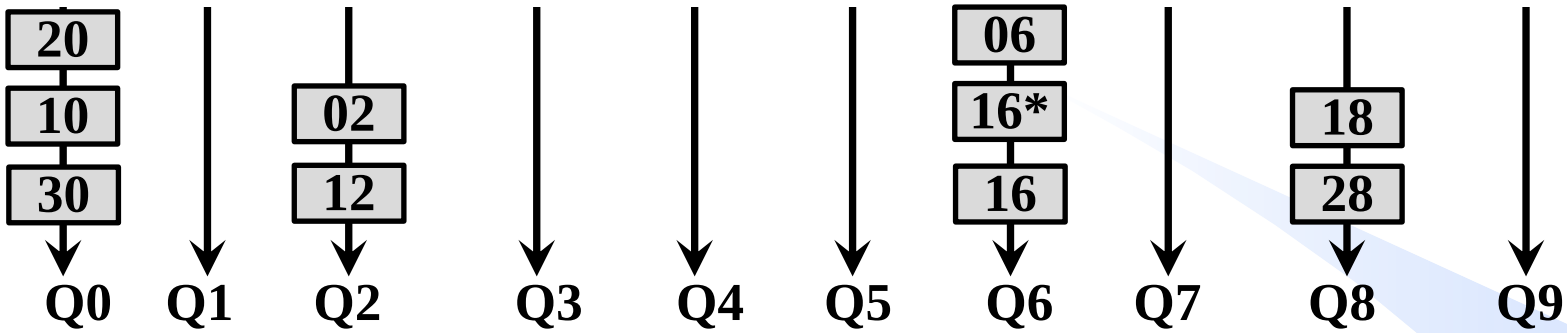

06
02
18
16*
16
12
10
28
20
30
Q0
Q1
Q2
Q3
Q4
Q5
Q6
Q7
Q8
Q9

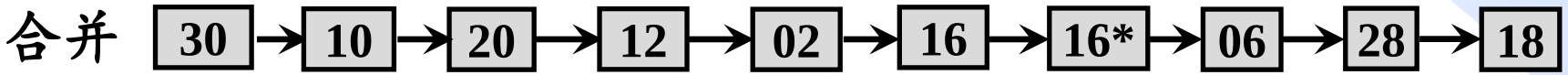

收集
02
06
10
12
16
16*
18
20
28
30

算法RadixSort (Q, n, p, r) // 最低次序位法的基数排序算法, *Q*为队列，*p*是关键词的位数，*r*为基数
用*LINK*域链接形成队列*Q*

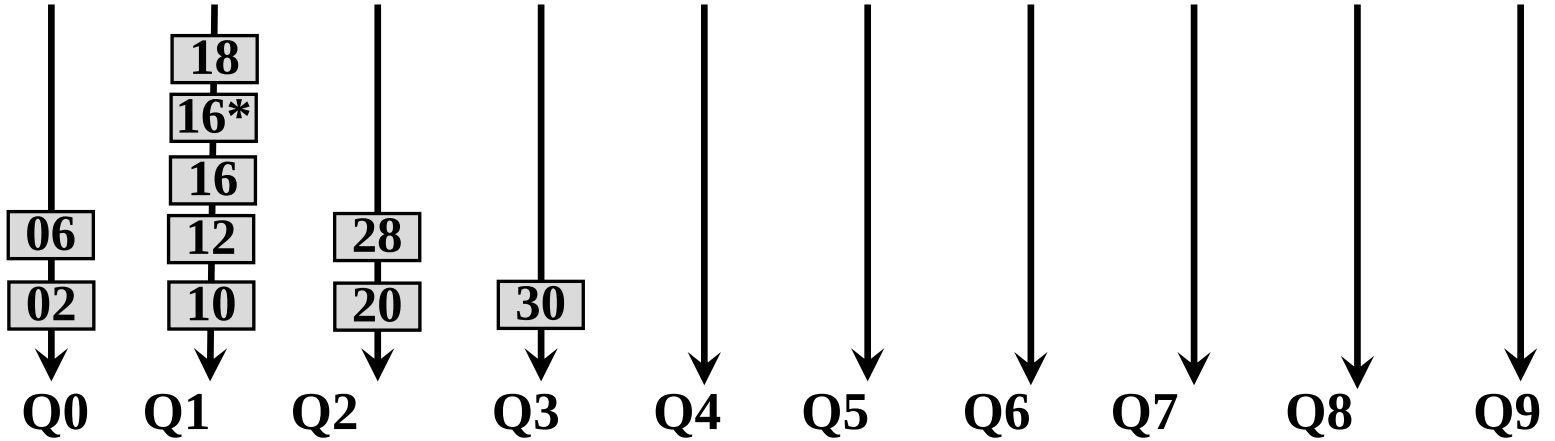


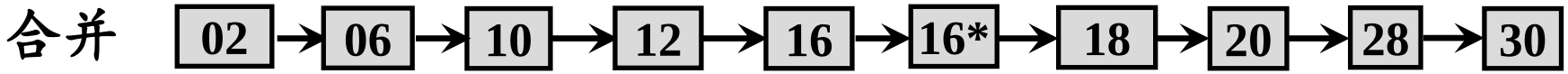
按最低位分配





按次低位分配



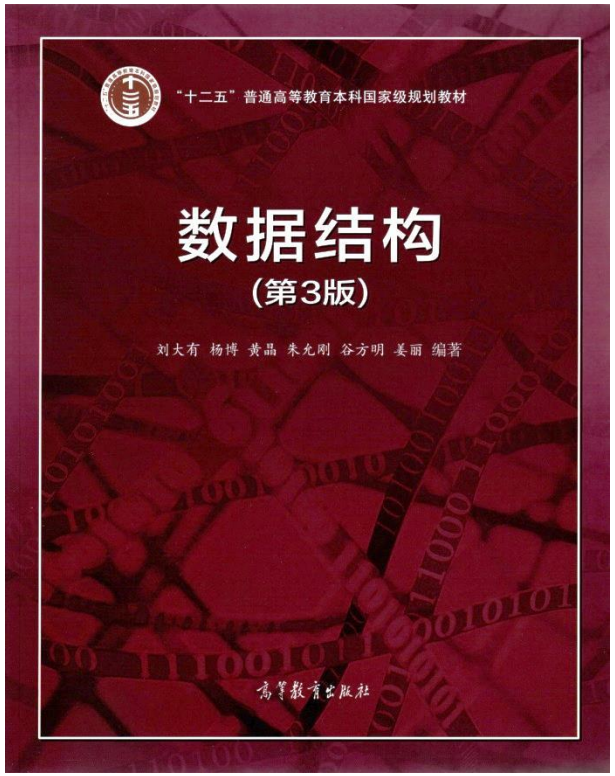


```
算法RadixSort (Q, n, p, r) // 最低次序位法的基数排序
算法, Q为队列，p是关键词的位数，r为基数
用LINK域链接形成队列Q
//关键词低位到高位分别按基数排序
FOR j=1 TO p DO
(     把队列Q0, Q1, …, Qr–1清空 .
      WHILE  NOT( QEmpty(Q) ) DO
      (     R⇐Q .  X←KEY(R).
            Kj←X的第j位.
            QKj ⇐X
      )
      合并Q0, Q1, …, Qr–1形成新的Q
) ▌
```

时间复杂度

O(np)

X的第j位是几，就把X放入第几个队列



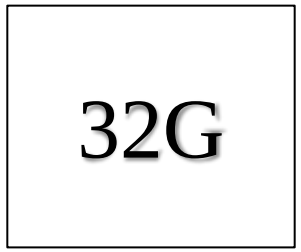
合并排序及其他

- 合并排序
- 排序算法时间下界
- 分布排序
- **外排序方法简介**

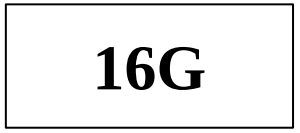
朱允刚
zhuyungang@jlu.edu.cn

Last updated on 2021.12.6

外排序方法简介

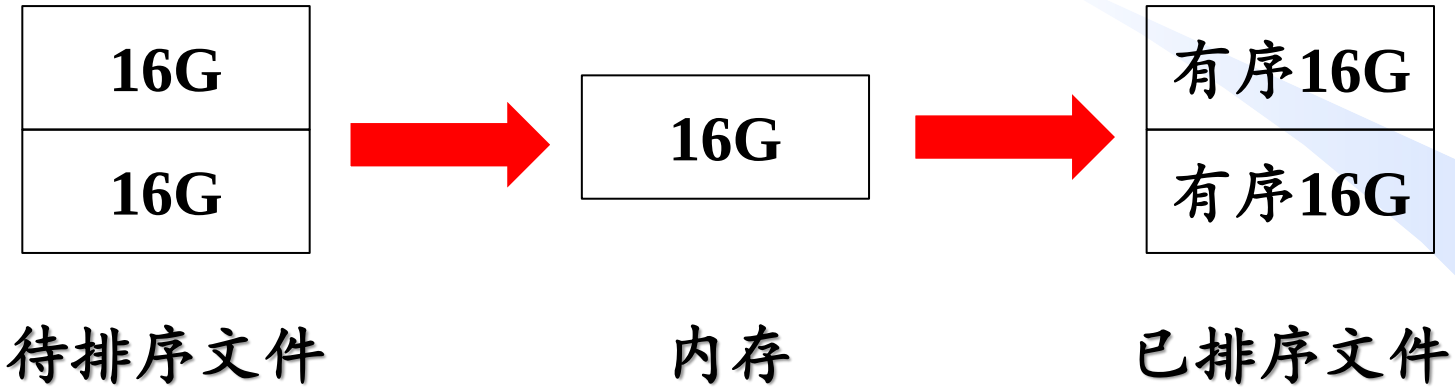


待排序文件

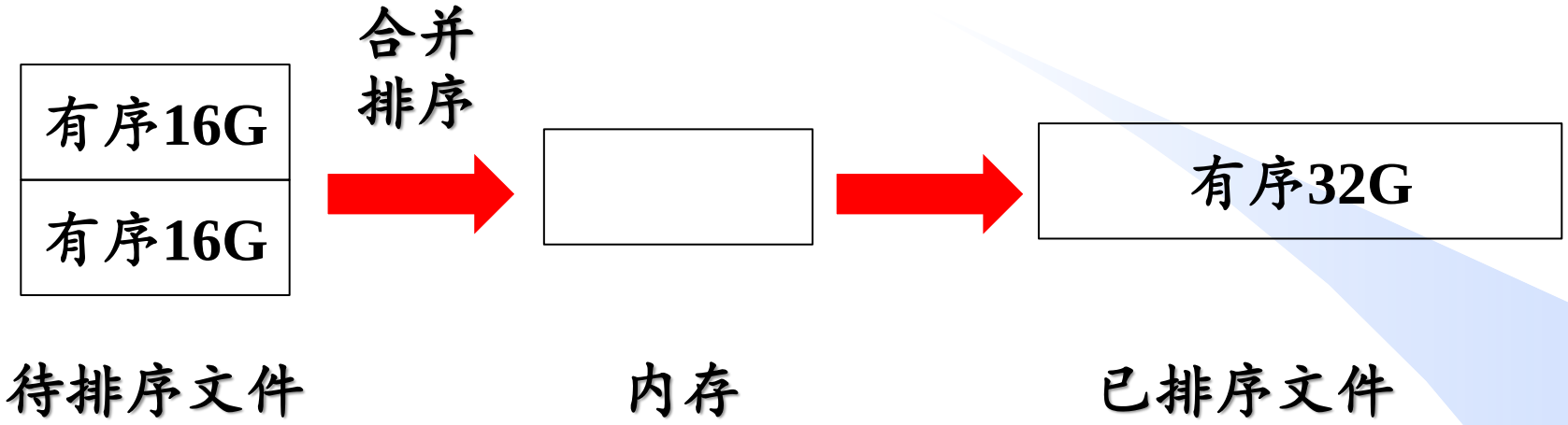


内存

外排序方法简介



外排序方法简介



练习

对10TB的数据文件进行排序，应使用的方法是_____。【2016年考研题全国卷】

A.希尔排序　　B.堆排序

C.快速排序　　D.合并排序

讨 论

按时间代价分类，内排序算法大致分为三大类：

① 简单排序算法，它们一般都比较简单，容易实现，但时间复杂度相对较高，即最坏情况下时间复杂度均为$O(n^2)$，也有一些改进的算法，如Shell排序；

② $O(n\log_2 n)$类算法，以分治策略算法为主，采用堆结构也可实现，在以比较运算为时间复杂度衡量基准的前提下，此类算法是最优的策略，其平均情况下时间复杂度均为$O(n\log_2 n)$；

③ 线性时间算法，不以关键词比较为基础，有较低的时间代价(即$O(n)$)，但是需要对数据集有一定先验知识，比如数据分布于哪个区间内等等。

练习

下列排序方法中，每趟排序结束都至少能确定一个元素最终位置的方法是_____。【考研题全国卷】

I.直接选择排序 II.希尔排序 III.快速排序

IV.堆排序 V.合并排序

[A] I,III,IV [B] I,III,V [C] II,III,IV [D] III,IV,V

练习

下列哪个算法可能出现下列情况：在最后一趟开始之前，所有的元素都不在其最终的位置上。

[A] 堆排序 [B] 冒泡排序 [C] 插入排序 [D] 快速排序

2 3 4 5 6 ← 1