10.1 概述

10.2 插入排序

10.3 快速排序

10.4 堆排序

10.5 归并排序

10.7 各种排序方法的综合比较

# 10.1 概述

一、排序的定义

二、内部排序和外部排序

三、内部排序方法的分类

# 一、什么是排序?

排序是计算机内 经常进行的一种操作,其目 的是将一组“ **无序”的记录序列调整为“ 有序”** 的记录序列。

例如： 将下列关键字序列

**52, 49, 80, 36, 14, 58, 61, 23, 97, 75**

调整为

**14, 23, 36, 49, 52, 58, 61 ,75, 80, 97**

一般情况下，

假设含$n$个记录的序列为$\{ R_1, R_2, \ldots, R_n \}$

其相应的关键字序列为 $\{ K_1, K_2, \ldots, K_n \}$

这些关键字相互之间可以进行比较，即在它们之间存在着这样一个关系：

$$K_{p1} \leqslant K_{p2} \leqslant \ldots \leqslant K_{pn}$$

按此固有关系将上式记录序列重新排列为

$$\{ R_{p1}, R_{p2}, \ldots, R_{pn} \}$$

的**操作**称作**排序**。

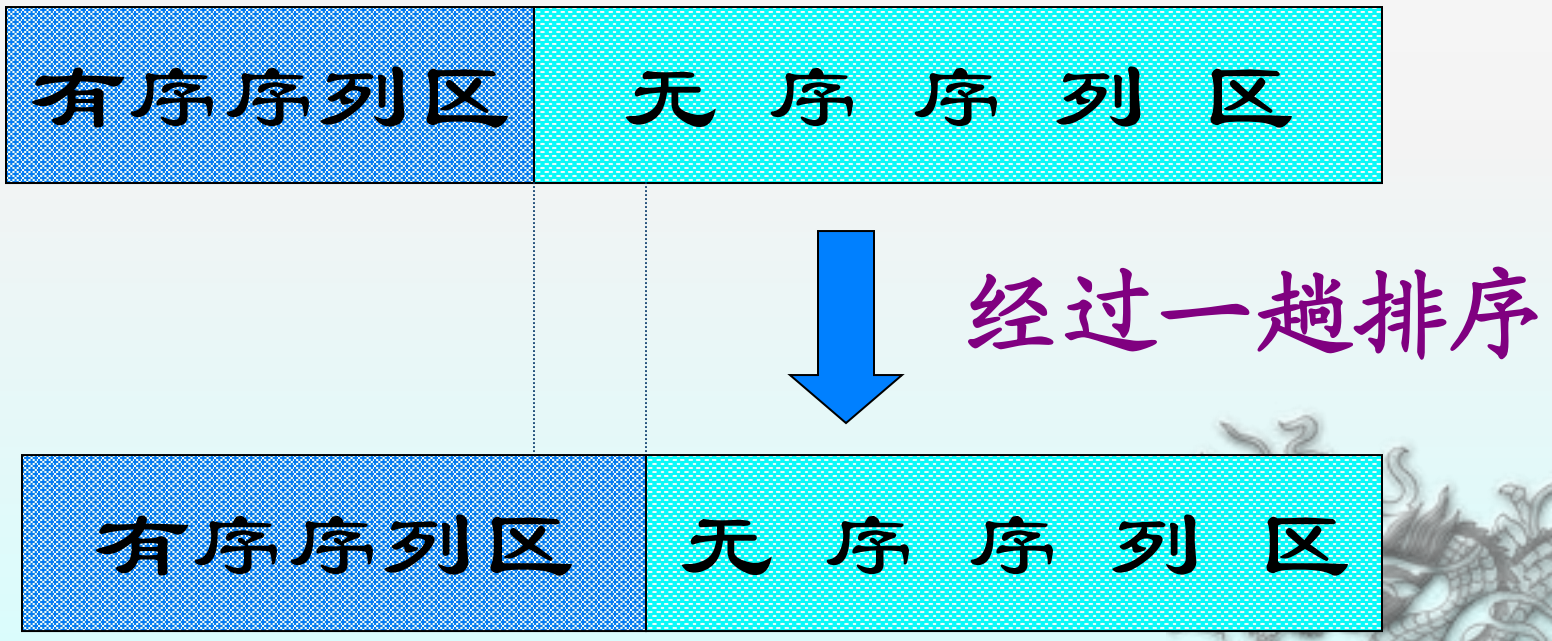
# 二、内部排序和外部排序

若整个排序过程**不需要访问外存**便能完成，则称此类排序问题为**内部排序**;

反之，若参加排序的记录数量很大，整个序列的排序过程不可能在内存中完成，则称此类排序问题为**外部排序**。

# 三、内 部排序的方法

内 部排序的过程是一个**逐步扩大记录的有序序列长度**的过程。



基于不同的“扩大”有序序列长度的方法，内部排序方法大致可分下列几种类型：

**插入类　　交换类　　选择类**

**归并类　　其它方法**

待排记录的数据类型定义如下：

```
#define MAXSIZE 20          // 待排顺序表最大长度
typedef  int  KeyType;  // 关键字类型为整数类型

typedef  struct {
   KeyType   key;            // 关键字项
    InfoType  otherinfo;  // 其它数据项
} RcdType;                        // 记录类型

typedef  struct {
    RcdType    r[MAXSIZE+1]; // r[0]闲置
    int              length;          // 顺序表长度
} SqList;                             // 顺序表类型
```

# 1. 插入类

将无序子序列中的一个或几个记录“**插入**”到有序序列中，从而增加记录的有序子序列的长度。

# 2. 交换类

通过“交换”无序序列中的记录从而得到其中关键字最小或最大的记录，并将它加入到有序子序列中，以此方法增加记录的有序子序列的长度。

# 3. 选择类

从记录的无序子序列中“选择”关键字最小或最大的记录，并将它加入到有序子序列中，以此方法增加记录的有序子序列的长度。

## 4. 归并类

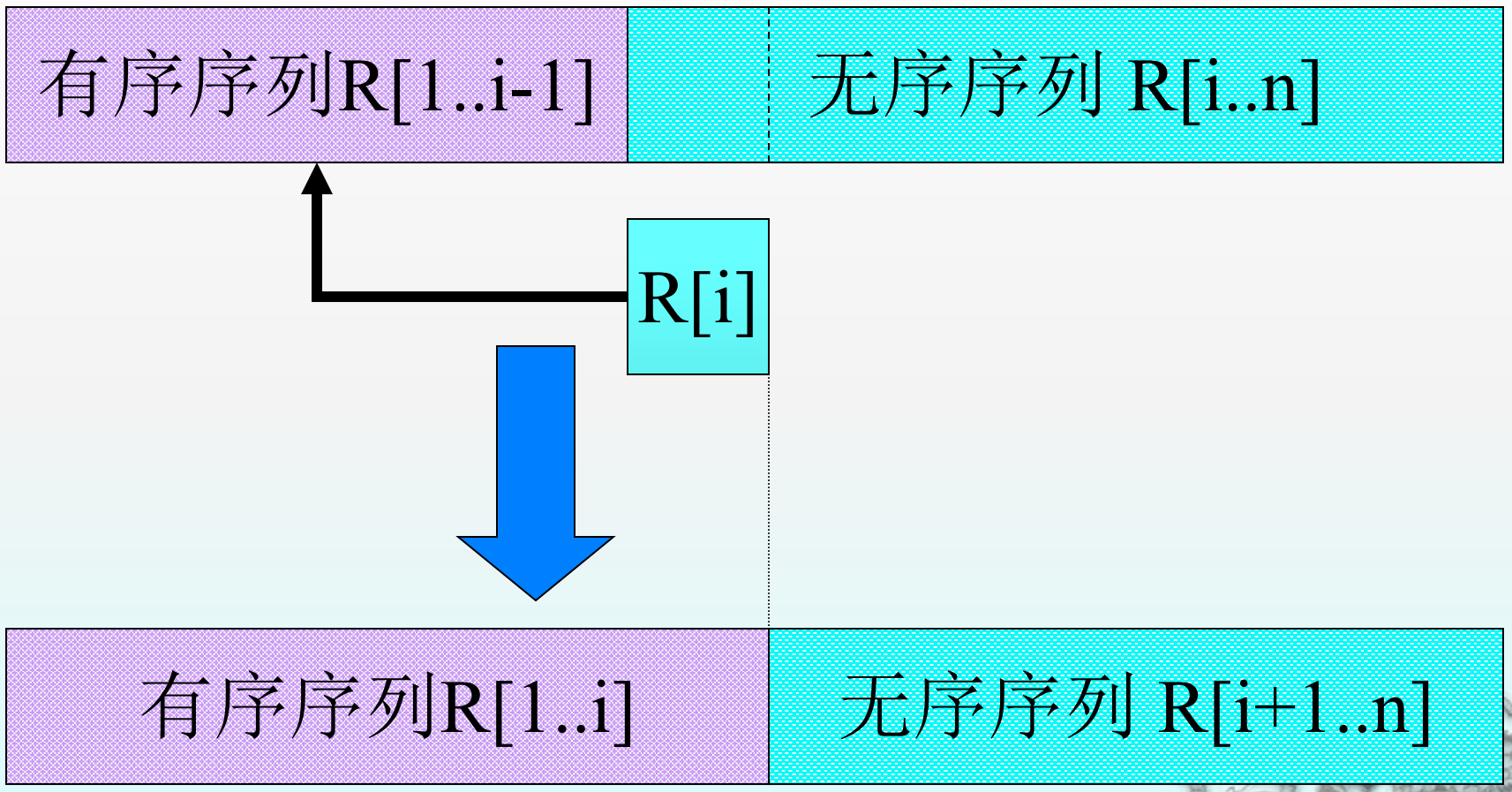
通过“归并”两个或两个以上的记录有序子序列，逐步增加记录有序序列的长度。

## 5. 其它方法

# 10.2

# 插入排序

# 一趟直接插入排序的基本思想:



实现“一趟插入排序”可分三步进行:

1. 在R[1..i-1]中查找R[i]的插入位置，
   R[1..j].key ≤ R[i].key < R[j+1..i-1].key;

2. 将R[j+1..i-1]中的所有记录均后移
   一个位置;

3. 将R[i]插入(复制)到R[j+1]的位置上。

# 不同的具体实现方法导致不同的算法描述

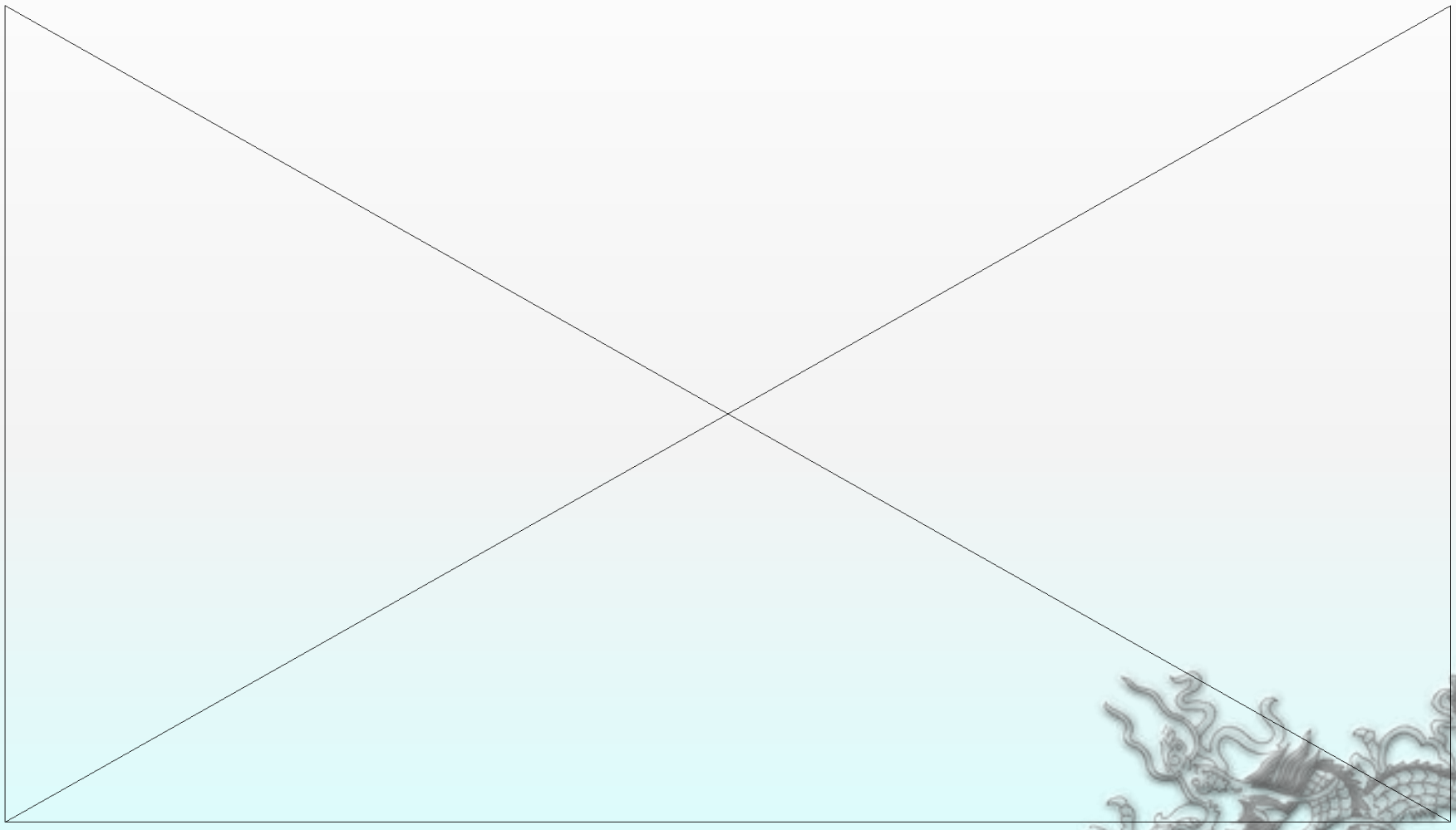
- 直接插入排序( 基于顺序查找)
- 折半插入排序( 基于折半查找)
- 表插入排序( 基于链表存储)
- 希尔 排序( 基于逐趟缩小增量)

# 一、直接插入排序

利用“ 顺序查找”实现

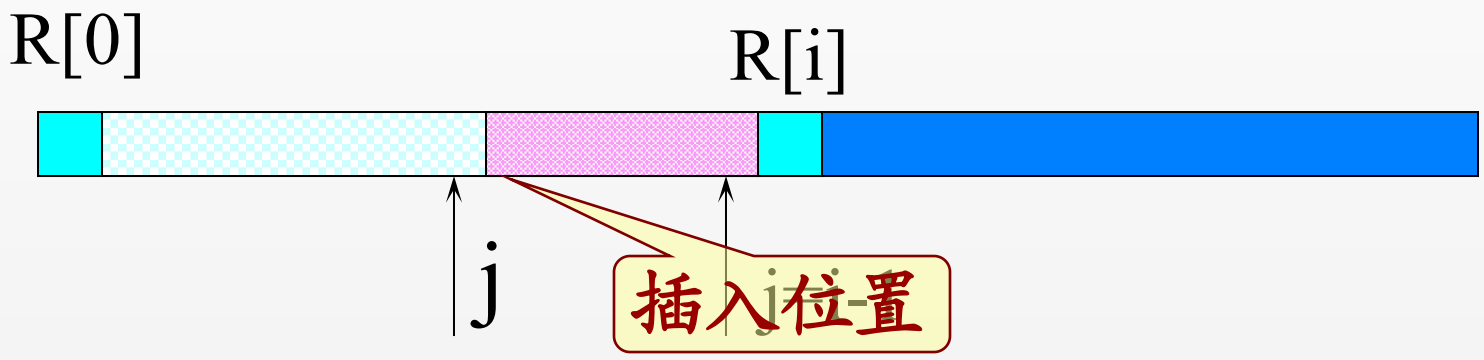
“ 在R[1..i-1]中 查找R[i]的插入位置”

例如



- 从R[i-1]起向前进行顺序查找，

  监视哨设置在R[0];




```
R[0] = R[i];            // 设置“ 哨兵”

for (j=i-1; R[0].key<R[j].key; --j);
                        // 从后往前找
```

循环结束表明R[i]的插入位置为 *j+1*

- 对于在查找过程中找到的那些关键字不小于R[i].key的记录，并在查找的同时实现记录向后移动;

```
for (j=i-1; R[0].key<R[j].key; --j);
```

**R[j+1] = R[j]**

- 上述循环结束后可以直接进行“插入”

- 令 i = 2，3，…, n,

  实现整个序列的排序。

```
for ( i=2;  i<=n;  ++i )
  if (R[i].key<R[i-1].key)
    { 在 R[1..i-1]中 查找R[i]的 插入位置;
      插入R[i] ;
    }
```

```
void InsertionSort ( SqList &L ) {
  // 对顺序表 L 作直接插入排序。
    for ( i=2; i<=L.length; ++i )
      if (L.r[i].key < L.r[i-1].key) {
        L.r[0] = L.r[i];              // 复制为监视哨
        for ( j=i-1; L.r[0].key < L.r[j].key;  -- j )
          L.r[j+1] = L.r[j];          // 记录后移
        L.r[j+1] = L.r[0];            // 插入到正确位置
      }
} // InsertSort
```

# 内部排序的时间分析:

实现内部排序的基本操作有两个:

(1)“比较”序列中两个关键字的大小;

(2)“移动”记录。

对于直接插入排序：

最好的情况（关键字在记录序列中顺序有序）：

“比较”的次数：

$$\sum_{i=2}^{n} 1 = n-1$$

“移动”的次数：

$$0$$

最坏的情况（关键字在记录序列中逆序有序）：

“比较”的次数：

$$\sum_{i=2}^{n}(i+1) = \frac{(n+4)(n-1)}{2}$$

“移动”的次数：

$$\sum_{i=2}^{n}(i+1) = \frac{(n+4)(n-1)}{2}$$

题1：直接插入排序在最好情况下的时间复杂度为____。

A. O(n)　　B. O($n\log_2 n$)　C. O($\log_2 n$)　D. O($n^2$)

题2：用插入排序对下面4个序列进行递增排序，元素比较次数最少的是_____。

A.94,32,40,90,80,46,21,69

B.32,40,21,46,69,94,90,80

C.21,32,46,40,80,69,90,94

D.90,69,80,46,21,32,94,40

题1：直接插入排序在最好情况下的时间复杂度为____。

A. O(n)　　B. O($n\log_2 n$)　C. O($\log_2 n$)　D. O($n^2$)

题2：用插入排序对下面4个序列进行递增排序，元素比较次数最少的是_____。

A.94,32,40,90,80,46,21,69

B.32,40,21,46,69,94,90,80

C.21,32,46,40,80,69,90,94

D.90,69,80,46,21,32,94,40

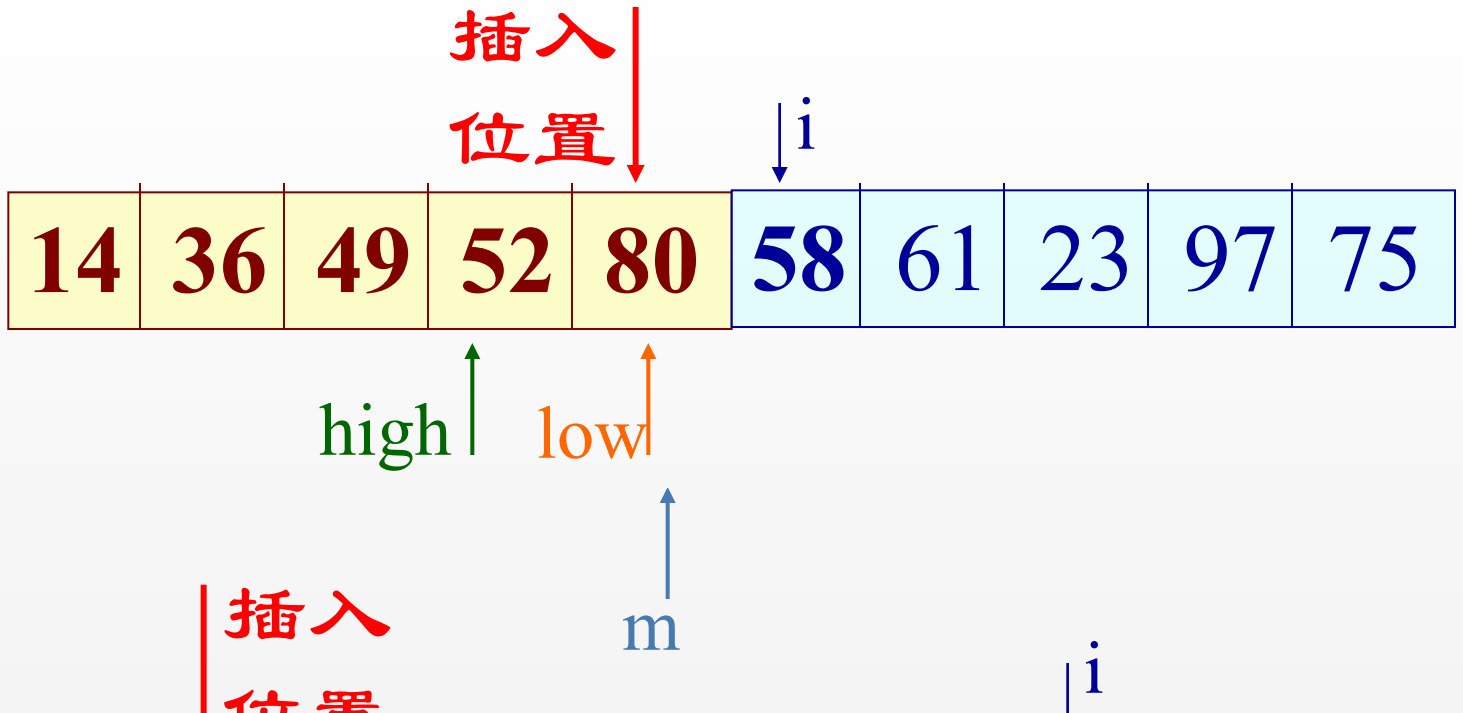
# 二、折半插入排序

因为 R[1..i-1] 是一个按关键字有序的有序序列，则可以利用**折半查找**实现“在R[1..i-1]中**查找**R[i]的插入位置”，如此实现的插入排序为**折半插入**排序。

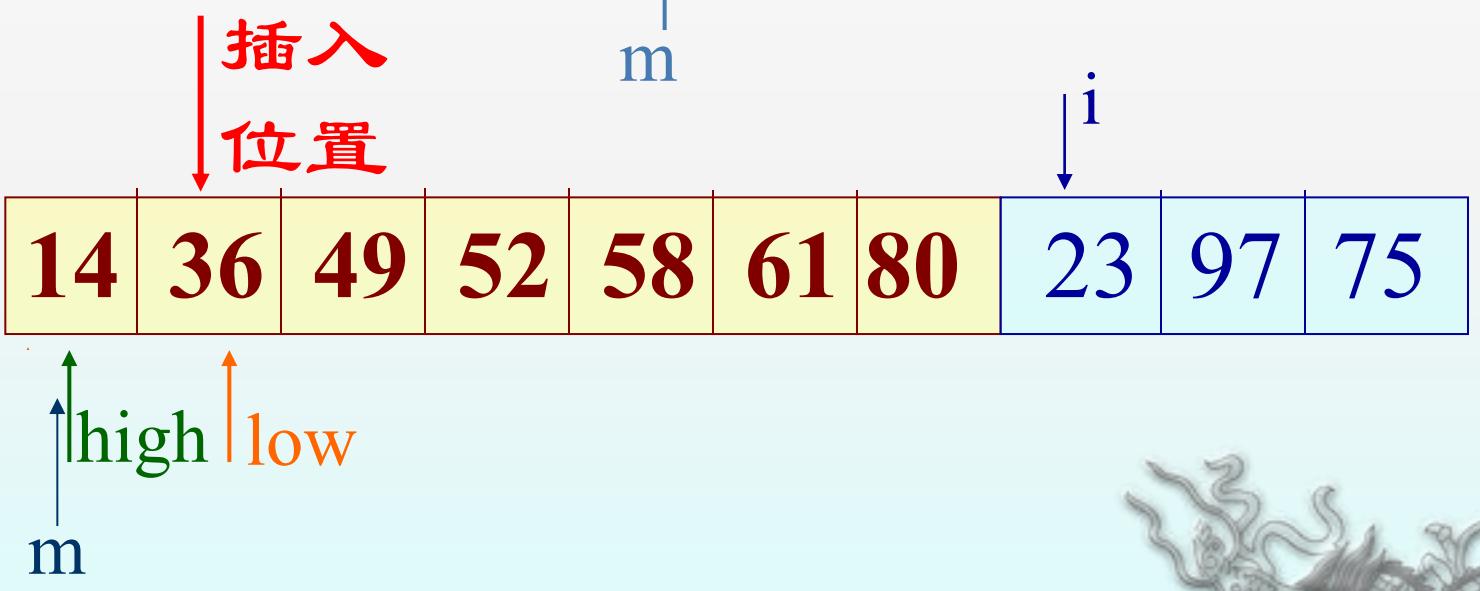
```
void BiInsertionSort ( SqList &L ) {
  for ( i=2; i<=L.length; ++i ) {
    L.r[0] = L.r[i];      // 将 L.r[i] 暂存到 L.r[0]
    在 L.r[1..i-1]中 折半查找插入位置;
    for ( j=i-1;  j>=high+1;  --j )
      L.r[j+1] = L.r[j];      // 记录后移
    L.r[high+1] = L.r[0];  // 插入
  } // for
} // BInsertSort
```

```
low = 1;   high = i-1;
while (low<=high) {
    m = (low+high)/2;           // 折半
    if (L.r[0].key < L.r[m].key)
            high = m-1;   // 插入点在低半区
    else  low = m+1; // 插入点在高半区
}
```

例如：


插入
位置
i
L.r
14 36 49 52 80 58 61 23 97 75
high
low
m

再如：


插入
位置
i
L.r
14 36 49 52 58 61 80 23 97 75
high
low
m

# 三、表插入排序

为了减少在排序过程中进行的“移动”记录的操作，必须改变排序过程中采用的存储结构。利用静态链表进行排序，并在排序完成之后，一次性地调整各个记录相互之间的位置，即将每个记录都调整到它们所应该在的位置上。

```
void LInsertionSort (Elem SL[ ],    int n){
  // 对记录序列SL[1..n]作表插入排序
  SL[0].key = MAXINT ;
  SL[0].next = 1;  SL[1].next = 0;
  for ( i=2; i<=n; ++i )
      for ( j=0, k = SL[0].next;SL[k].key<=
                SL[i].key ; j=k, k=SL[k].next )
    SL[j].next = i;  SL[i].next = k;
    // 结点i插入在结点j和结点k之间
}// LinsertionSort
```

# 如何在排序之后调整记录序列?

算法中 使用了 三个指针:

其中： p指示第i个记录的当 前位置

i指示第i个记录应在的位置

q指示第i+1个记录的当 前位置

```
void Arrange ( Elem SL[ ], int n ) {
  p = SL[0].next;   // p指示第一个记录的当 前位置
  for ( i=1; i<n; ++i ) {
    while (p<i)  p = SL[p].next;
    q = SL[p].next;     // q指示尚 未调整的 表尾
    if ( p!= i ) {
      SL[p]←→SL[i]; // 交换记录，使第i个记录到位
      SL[i].next = p;    // 指向被移走的记录
    }
    p = q;          // p指示尚 未调整的 表尾,
                    // 为找第 i+1个记录作准备
  }
} // Arrange
```

# 四、希尔排序(又称缩小增量排序)

基本思想：对待排记录序列先作“宏观”调整，再作“微观”调整。

所谓“宏观”调整，指的是，“跳跃式”的插入排序。

具体做法为：

将记录序列分成若干子序列，分别对每个子序列进行插入排序。

例如：将 n 个记录分成 d 个子序列：

{ R[1], R[1+d], R[1+2d], …, R[1+kd] }

{ R[2], R[2+d], R[2+2d], …, R[2+kd] }

…

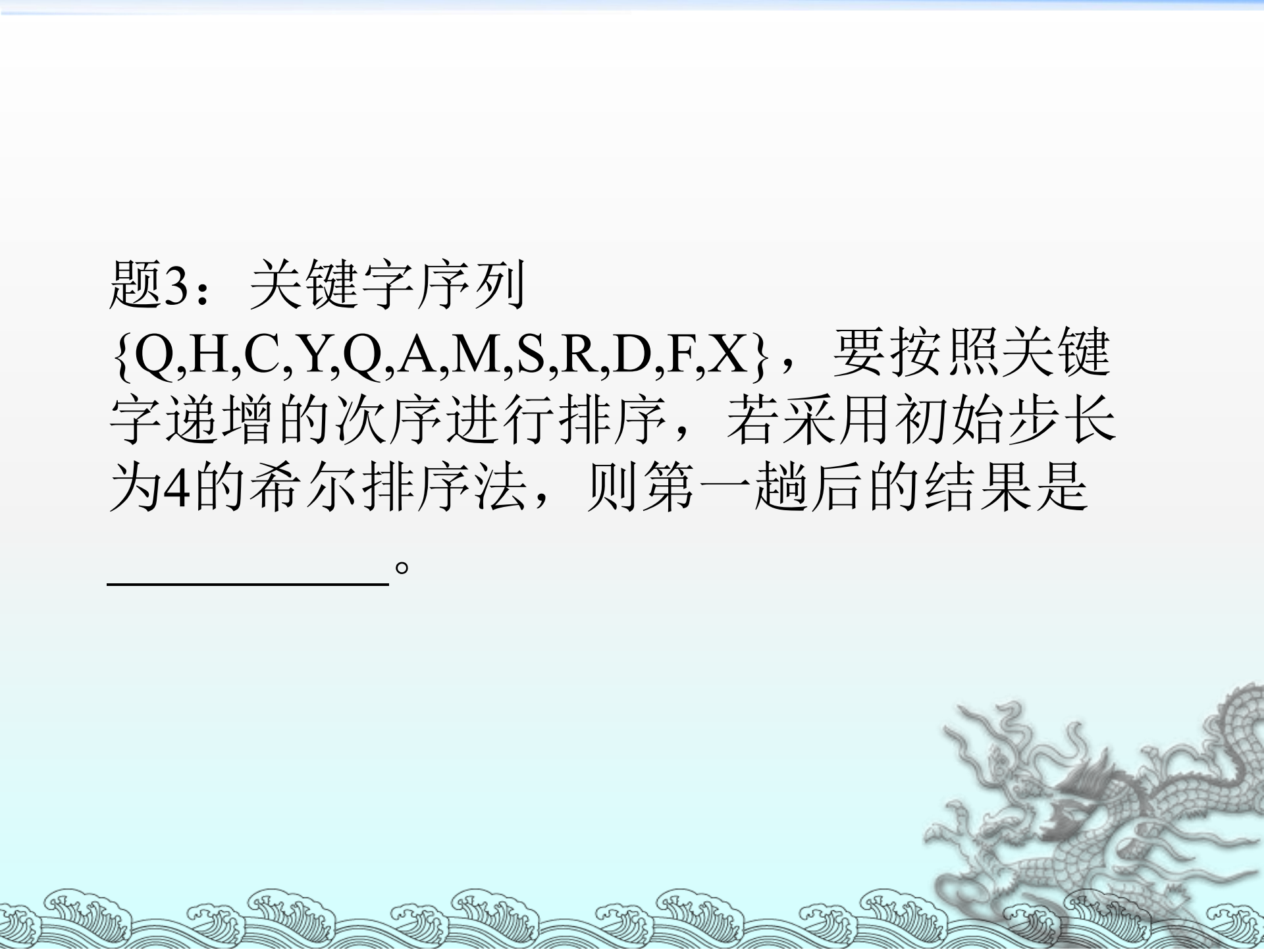
{ R[d], R[2d], R[3d], …, R[kd], R[(k+1)d] }

**其中，d 称为增量，它的值在排序过程中从大到小逐渐缩小，直至最后一趟排序减为 1。**

```
void ShellInsert ( SqList &L, int dk ) {
  for ( i=dk+1; i<=n; ++i )
    if ( L.r[i].key< L.r[i-dk].key) {
      L.r[0] = L.r[i];              // 暂存在R[0]
      for (j=i-dk;  j>0&&(L.r[0].key<L.r[j].key);
            j-=dk)
        L.r[j+dk] = L.r[j];  // 记录后移，查找插入位置
      L.r[j+dk] = L.r[0];                 // 插入
    } // if
} // ShellInsert
```

```
void ShellSort (SqList &L, int dlta[], int t)
{   // 增量为dlta[]的希尔 排序
    for (k=0; k<t; ++t)
        ShellInsert(L, dlta[k]);
            //一趟增量为dlta[k]的插入排序
} // ShellSort
```

题3：关键字序列{Q,H,C,Y,Q,A,M,S,R,D,F,X}，要按照关键字递增的次序进行排序，若采用初始步长为4的希尔排序法，则第一趟后的结果是_____。



# 10.3 快速排序

一、起泡排序

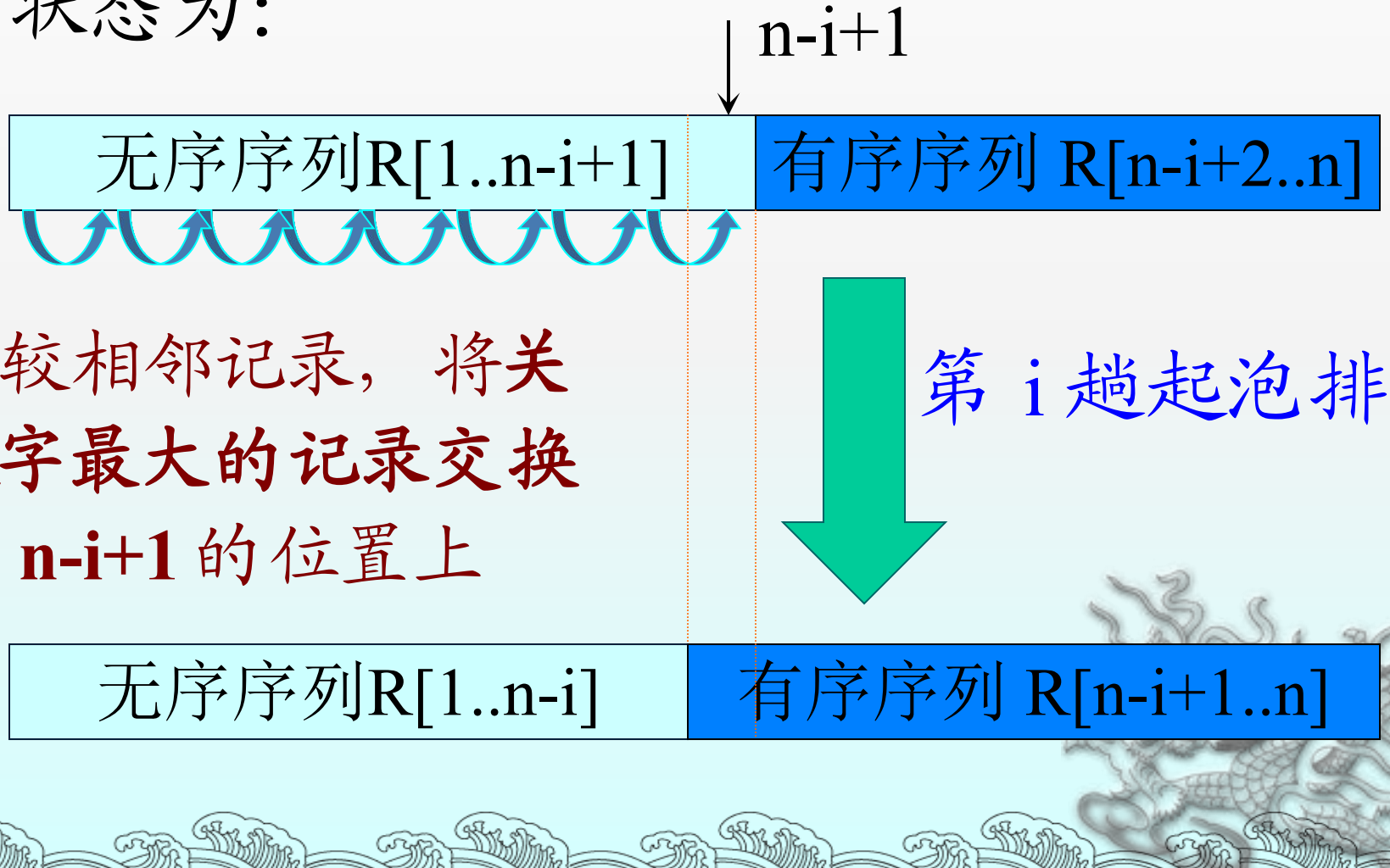
二、一趟快速排序

三、快速排序

四、快速排序的时间分析

# 一、起泡排序

假设在排序过程中，记录序列R[1..n]的状态为：



```
void BubbleSort(Elem R[ ], int n) {
    i = n;
   while (i >1) {
      lastExchangeIndex = 1;
      for (j = 1;  j < i;  j++)
          if (R[j+1].key < R[j].key) {
             Swap(R[j], R[j+1]);
             lastExchangeIndex = j;  //记下进行交换的记录位置
          } //if
      i = lastExchangeIndex; // 本趟进行过交换的
                             // 最后一个记录的位置
   } // while
} // BubbleSort
```

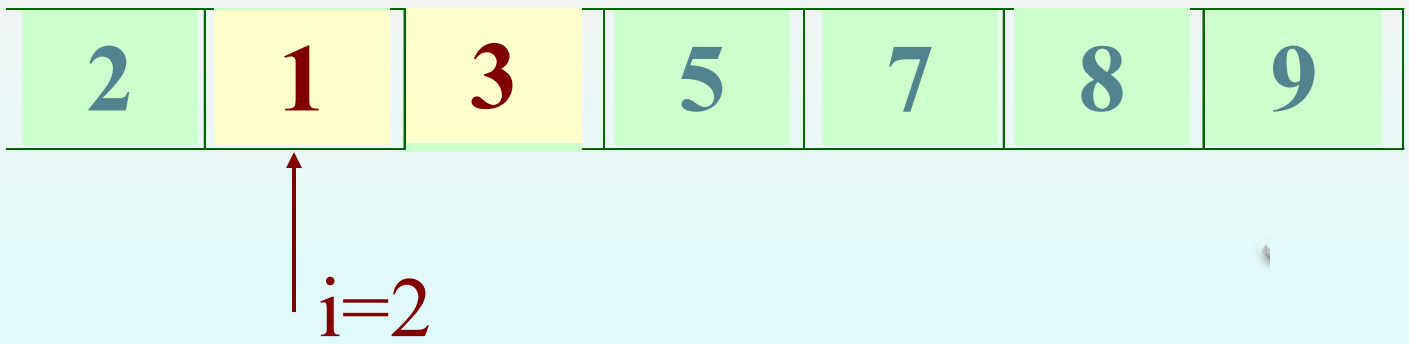
## 注意:

1. 起泡排序的结束条件为，

**最后一趟没有进行“交换记录”。**

2. 一般情况下，每经过一趟“起泡”，“i 减一”，但并不是每趟都如此。

## 例如:




**for** (j = 1; **j < i;** j++) **if** (R[j+1].key < R[j].key) …

# 时间分析：

最好的情况（关键字在记录序列中顺序有序）：

只需进行一趟起泡

“比较”的次数：

**n-1**

“移动”的次数：

**0**

最坏的情况（关键字在记录序列中逆序有序）：

需进行n-1趟起泡

“比较”的次数：

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

“移动”的次数：

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

题4：对数据序列{8,9,10,4,5,6,20,1,2}进行递增排序，采用每一趟冒出一个最小元素的冒泡排序算法，需要进行的趟数至少是_____。

A. 3　　B. 4　　C. 5　　D. 8

# 二、一趟快速排序( 一次划分)

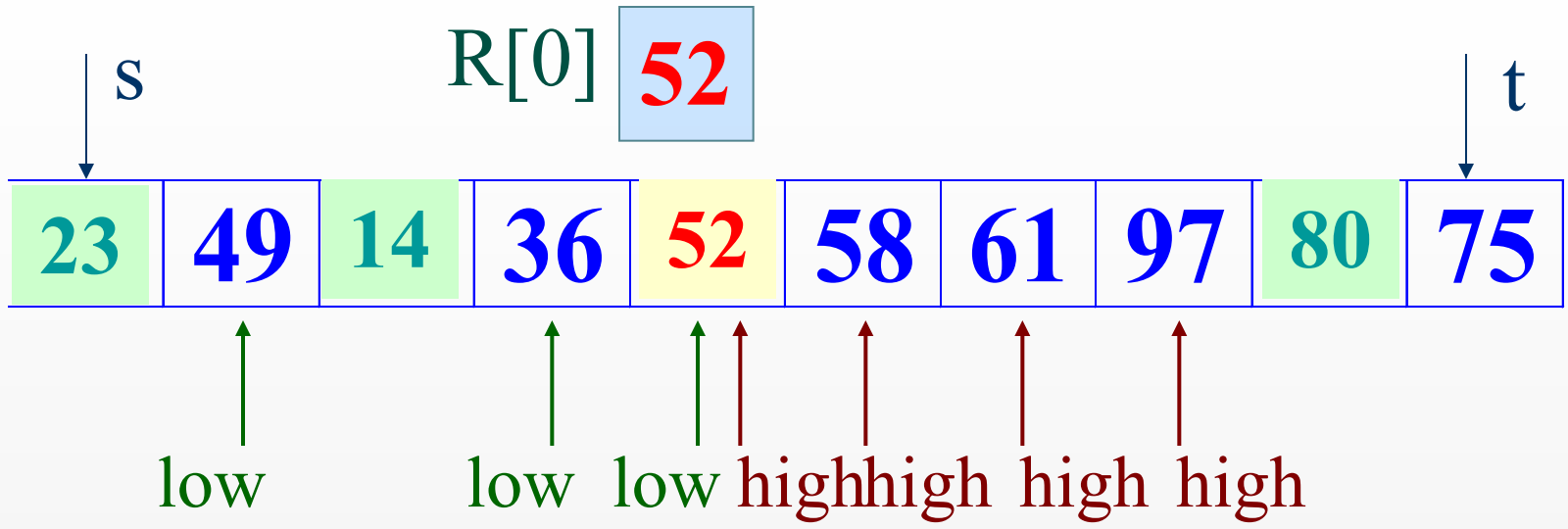
目 标: 找一个记录，以它的关键字作为“ 枢轴”，凡其关键字小于枢轴的记录均移动至该记录之前，反之，凡关键字大于枢轴的记录均移动至该记录之后。

致使一趟排序之后，记录的无序序列R[s..t]将分割成两 部分： R[s..i-1]和R[i+1..t]， 且

$$R[j].key \le R[i].key \le R[j].key$$

$(s \le j \le i-1)$ 枢轴 $(i+1 \le j \le t)$。

例如




**设 R[s]=52 为枢轴**

将 R[high].key 和 枢轴的关键字进行比较，要求R[high].key ≥ 枢轴的关键字

将 R[low].key 和 枢轴的关键字进行比较，要求R[low].key ≤ 枢轴的关键字

可见，经过“**一次划分**”，将关键字序列

52, 49, 80, 36, 14, 58, 61, 97, 23, 75

调整为：**23**, 49, **14**, 36, **(52)** 58, 61, 97, **80**, 75

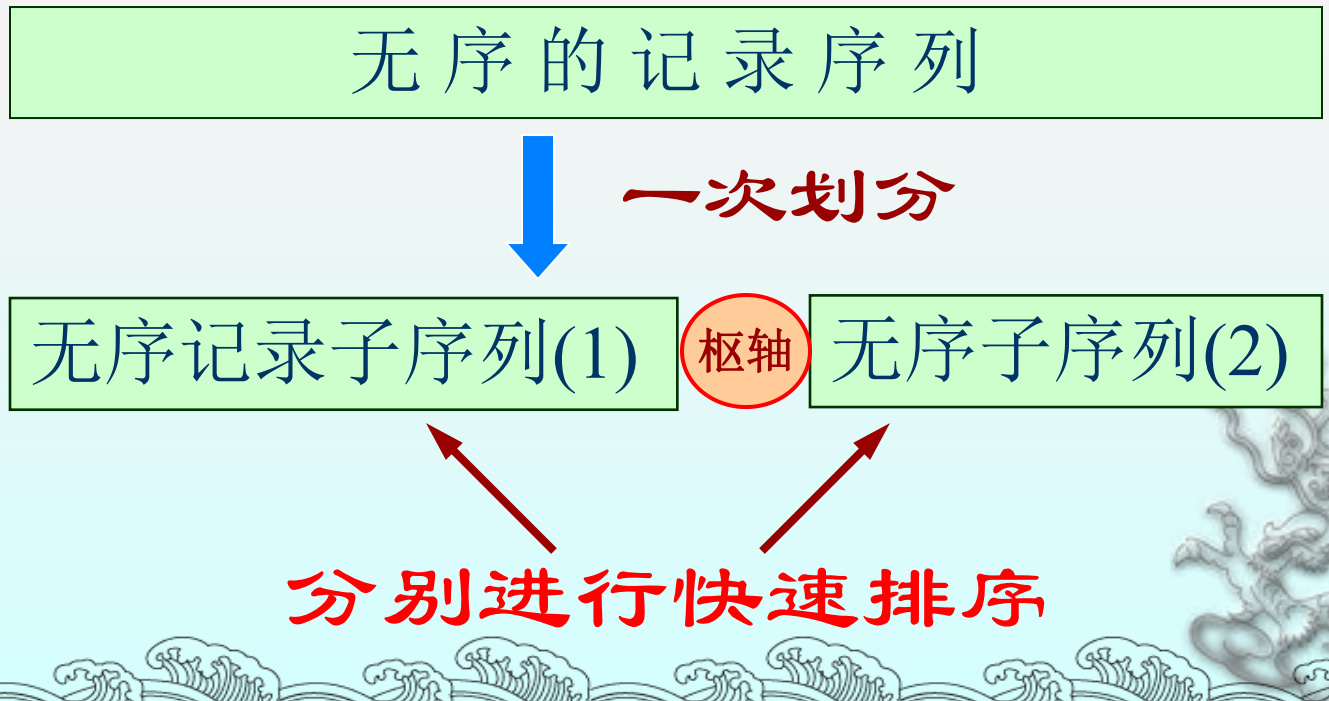
在调整过程中，设立了两个指针：**low**和**high**，它们的初值分别为：s和t，

之后逐渐减小**high**，增加**low**，并保证R[**high**].key≥52，和R[**low**].key≤52，否则进行记录的“交换”。

```
int Partition (RedType R[], int low, int high) {
    R[0] = R[low];  pivotkey = R[low].key;  // 枢轴
    while (low<high) {
        while(low<high&& R[high].key>=pivotkey)
            -- high;      // 从右向左搜索
        R[low] = R[high];
        while (low<high && R[low].key<=pivotkey)
            ++ low;      // 从左向右搜索
        R[high] = R[low];
    }
    R[low] = R[0];      return low;
}// Partition
```

# 三、快速排序

首先对无序的记录序列进行“**一次划分**”，之后**分别**对分割所得两个子序列“**递归**”**进行快速排序**。



```
void QSort (RedType & R[], int s, int t ) {
  // 对记录序列R[s..t]进行快速排序
  if (s < t-1) {                  // 长度大于1
    pivotloc = Partition(R, s, t);
                                  // 对 R[s..t] 进行一次划分
    QSort(R, s, pivotloc-1);
        // 对低子序列递归排序， pivotloc是枢轴位置
    QSort(R, pivotloc+1, t); // 对高子序列递归排序
  }
} // QSort
```

第一次调用函数 Qsort 时，待排序记录序列的上、下界分别为 1 和 L.length。

```
void QuickSort( SqList & L) {
  // 对顺序表进行快速排序
    QSort(L.r, 1, L.length);
} // QuickSort
```

# 四、快速排序的时间分析

假设一次划分所得枢轴位置 $i=k$，则对 $n$ 个记录进行快排所需时间：

$$T(n) = T_{pass}(n) + T(k-1) + T(n-k)$$

其中 $T_{pass}(n)$为对 $n$ 个记录进行一次划分所需时间。

若待排序列中记录的关键字是随机分布的，则 $k$ 取 1 至 $n$ 中任意一值的可能性相同。

由此可得快速排序所需时间的平均值为:

$$T_{avg}(n) = Cn + \frac{1}{n}\sum_{k=1}^{n}\left[T_{avg}(k-1) + T_{avg}(n-k)\right]$$

设 $T_{avg}(1) \leqslant b$

则可得结果:

$$T_{avg}(n) < (\frac{b}{2} + 2c)(n+1)\ln(n+1)$$

**结论:快速排序的时间复杂度为O(nlogn)**

若待排记录的初始状态为按关键字有序时，快速排序将蜕化为起泡排序，其时间复杂度为$O(n^2)$。

为避免出现这种情况，需在进行一次划分之前，进行“预处理”，即：

先对 R(s).key，R(t).key 和 R[⌊(s+t)/2⌋].key，进行相互比较，然后取关键字为“三者之中”的记录为枢轴记录。

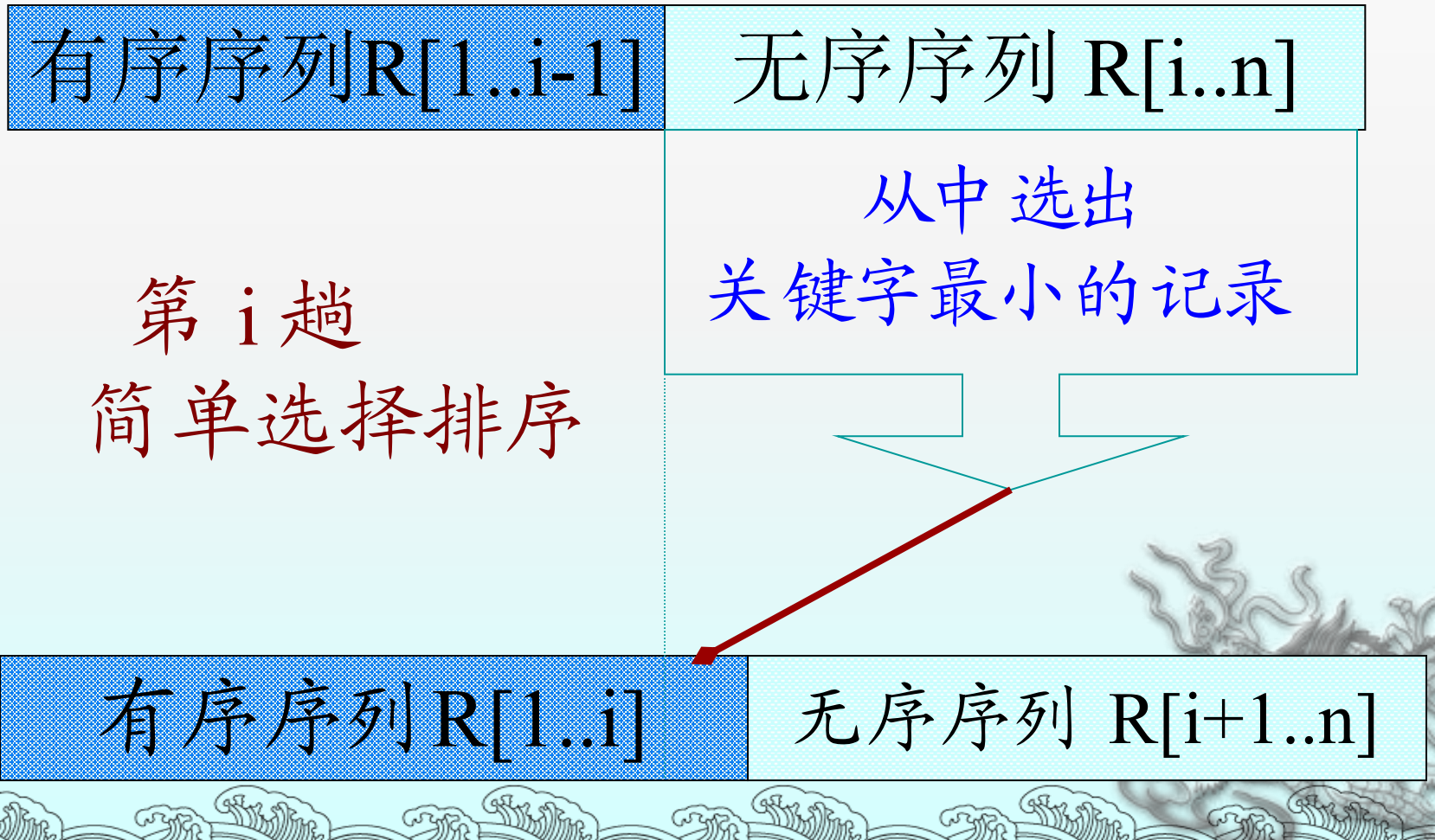
题4：已只序列{15,5,16,2,25,8,20,9,18,12}，给出采用快速排序算法对该序列做升序排序时的每一趟的结果。

# 10.4 堆排序

- 简单选择排序
- 堆排序

# 一、简单选择排序

假设排序过程中，待排记录序列的状态为：



简单选择排序的算法描述如下：

```
void SelectSort (Elem R[], int n ) {
  // 对记录序列R[1..n]作简单选择排序。
  for (i=1; i<n; ++i) {
          // 选择第 i 小的记录，并交换到位
    j = SelectMinKey(R, i);
          // 在 R[i..n] 中选择关键字最小的记录
    if (i!=j)  R[i]←→R[j];
                  // 与第 i 个记录交换
  }
} // SelectSort
```

# 时间性能分析

对 n 个记录进行简单选择排序，所需进行的 **关键字间的比较次数** 总计为：

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

**移动记录的次数**，最小值为 0，最大值为3(n-1)。

# 二、堆排序

## 堆的定义：

堆是满足下列性质的数列 $\{r_1, r_2, \ldots, r_n\}$：

$$\begin{cases} r_i \le r_{2i} \\ r_i \le r_{2i+1} \end{cases} \text{(小顶堆)} \quad \text{或} \quad \begin{cases} r_i \ge r_{2i} \\ r_i \ge r_{2i+1} \end{cases} \text{(大顶堆)}$$

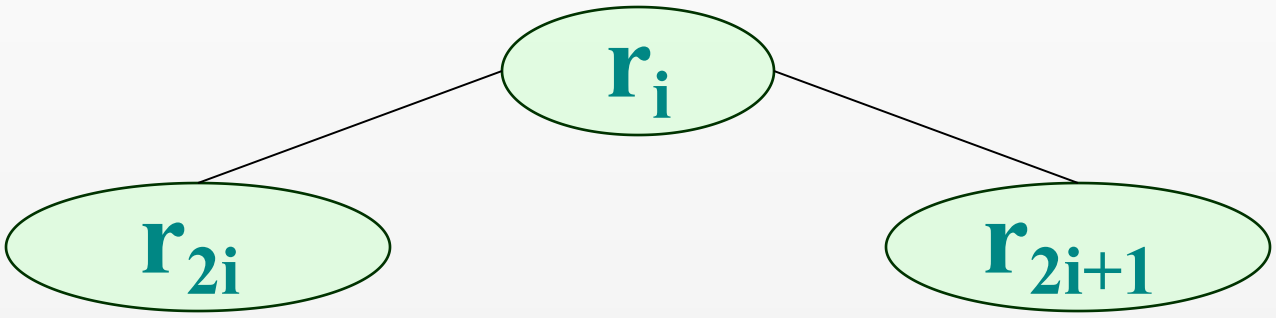
**例如：**

{12, 36, 27, 65, 40, 34, 98, 81, 73, 55, 49} 是小顶堆

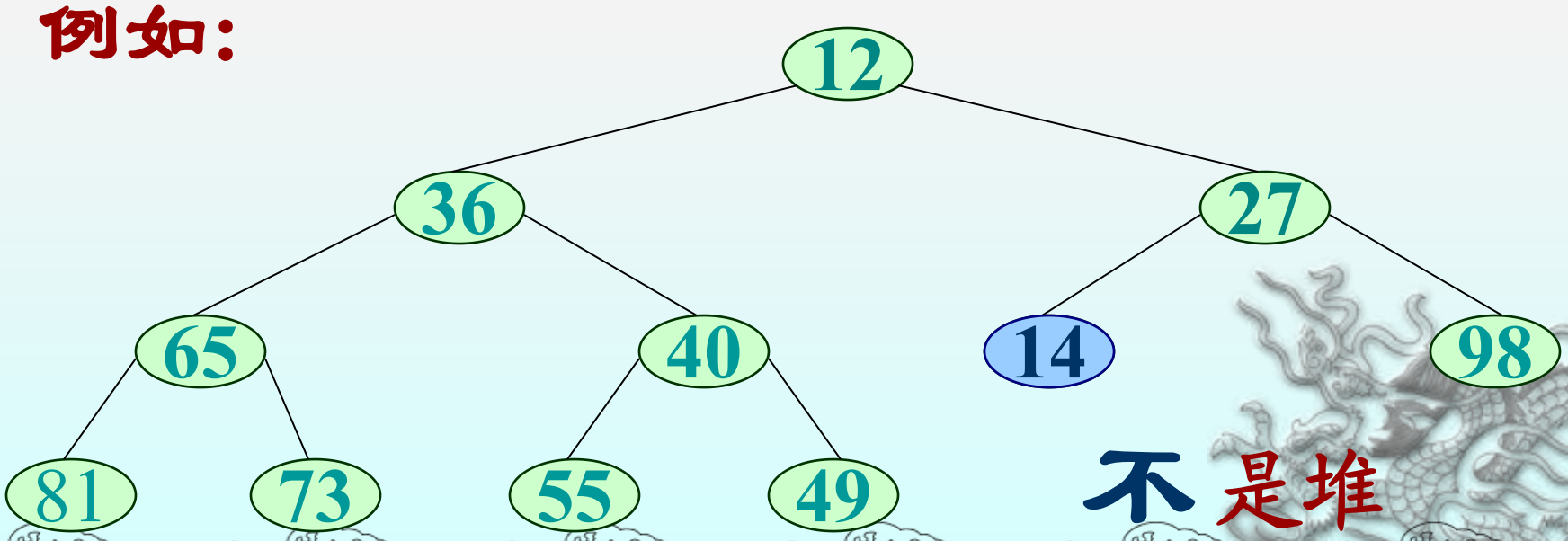
{12, 36, 27, 65, 40, **14**, 98, 81, 73, 55, 49} 不是堆

若将该数列视作完全二叉树，

则 $r_{2i}$ 是 $r_i$ 的左孩子； $r_{2i+1}$ 是 $r_i$ 的右孩子。




**例如：**




不是堆

堆排序即是利用堆的特性对记录序列进行排序的一种排序方法。

例如：

{ 40, 55, 49, 73, 12, 27, 98, 81, 64, 36 }

↓ 建大顶堆

{ 98, 81, 49, 73, 36, 27, 40, 55, 64, 12 }

↓ 交换 98 和 12

{ 12, 81, 49, 73, 36, 27, 40, 55, 64, 98 }

经过筛选 ↓ 重新调整为大顶堆

{ 81, 73, 49, 64, 36, 27, 40, 55, 12, 98 }

```
void HeapSort ( HeapType &H ) {
  // 对顺序表 H 进行堆排序
   for ( i=H.length/2;   i>0;   --i )
        HeapAdjust ( H.r, i, H.length );    // 建大顶堆
   for ( i=H.length; i>1; --i ) {
        H.r[1]←→H.r[i];
            // 将堆顶记录和当 前未经排序子序列
            //  H.r[1..i]中 最后一个记录相互交换
        HeapAdjust(H.r, 1, i-1);  // 对 H.r[1] 进行筛 选
   }
} // HeapSort
```

定义堆类型为：

**typedef** SqList HeapType;

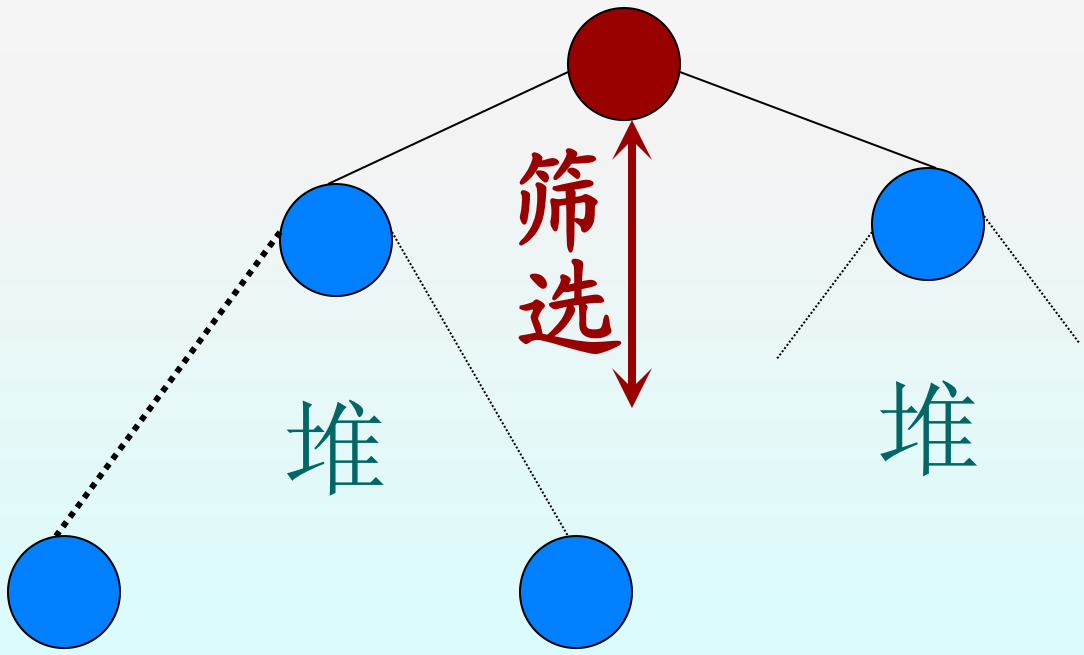
// 堆采用顺序表表示之

两个问题：

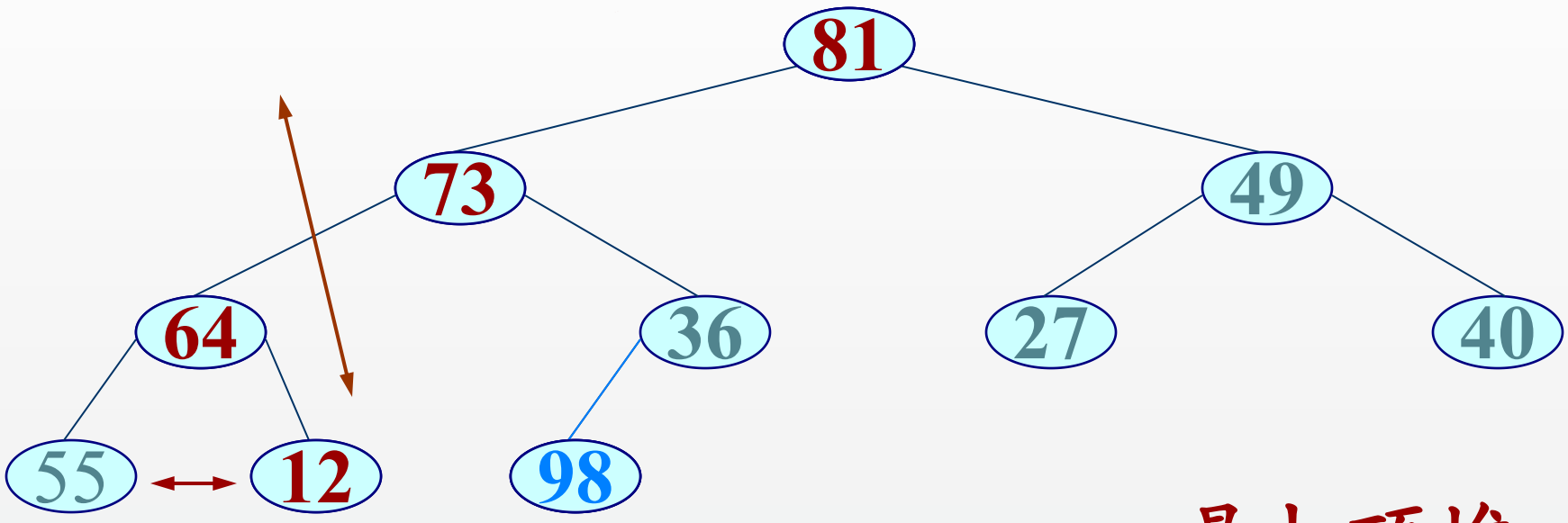
如何“ 建堆”？

如何“ 筛选”？

所谓“**筛选**”指的是，对一棵左/右子树均为堆的完全二叉树，“**调整**”**根结点**使整个二叉树也成为一个堆。



例如：




是大顶堆

但在 98 和 12 进行互换之后，它就不是堆了，

因此，需要对它进行“筛选”。

```
void HeapAdjust (RcdType &R[], int s, int m)
{  //已知 R[s..m]中记录的关键字除 R[s] 之外均
   //满足堆的特征，本函数自上而下调整 R[s] 的
   //关键字，使 R[s..m] 也成为一个大顶堆
   rc = R[s];   //暂存 R[s]
   for ( j=2*s; j<=m; j*=2 ) { //j 初值指向左孩子
      自上而下的筛选过程;
   }
   R[s] = rc;  //将调整前的堆顶记录插入到 s 位置
} // HeapAdjust
```

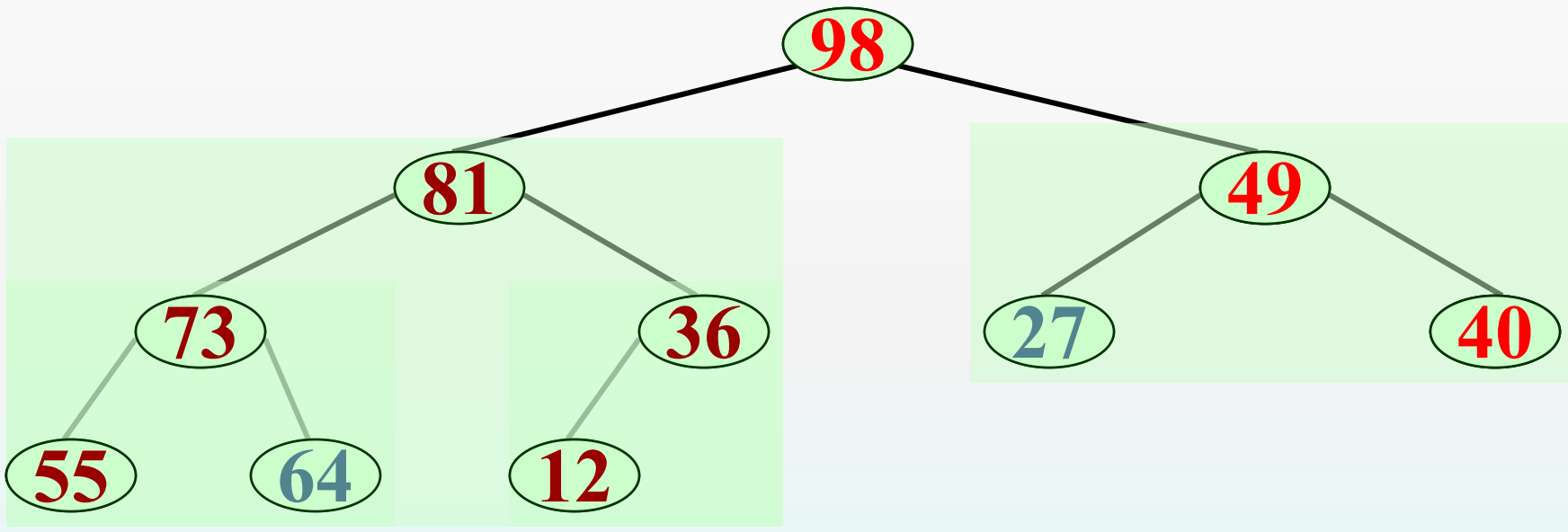
```
if ( j<m && R[j].key<R[j+1].key ) ++j;
        // 左/右“ 子树根” 之间先进行相互比较
        // 令 j 指示关键字较大记录的位置

if ( rc.key >= R[j].key ) break;
        // 再作“ 根” 和“ 子树根” 之间的比较,
        // 若“ >=”成立，则说明已找到 rc 的插
        // 入位置 s，不需要继续往下调整

R[s] = R[j];   s = j;
        // 否则记录上移，尚需继续往下调整
```

# 建堆是一个从下往上进行“筛选”的过程。

**例如：排序之前的关键字序列为**




现在，左/右子树都已经调整为堆，最后只要调整根结点，使整个二叉树是个“堆”即可。

## 堆排序的时间复杂度分析：

1. 对深度为 k 的堆，“筛选”所需进行的关键字比较的次数至多为 2(k-1)；

2. 对 $n$ 个关键字，建成深度为 $h(=\lfloor \log_2 n \rfloor+1)$ 的堆，所需进行的关键字比较的次数至多 $4n$；

3. 调整“堆顶” $n-1$ 次，总共进行的关键字比较的次数不超过

$$2(\lfloor \log_2(n-1) \rfloor+\lfloor \log_2(n-2) \rfloor+\ \ldots+\log_2 2) < 2n(\lfloor \log_2 n \rfloor)$$

因此，堆排序的时间复杂度为 $O(n\log n)$。

题5：给出关键字序列{12,13,11,18,60,15,7,20,25,100}的初始大顶堆。

# 10.5 归并排序

归并排序的过程基于下列**基本思想**进行:

将两个或两个以上的有序子序列“归并”为一个有序序列。

在内部排序中，通常采用的是2-路归并排序。即：将两个**位置相邻**的记录有序子序列

| 有序子序列 R[*l*..*m*] | 有序子序列 R[*m*+1..*n*] |
|---|---|

归并为**一个**记录的有序序列。

| 有 序 序 列 R[*l*..*n*] |
|---|

这个操作对顺序表而言，是轻而易举的。

```
void Merge (RcdType SR[], RcdType &TR[],
                                    int i, int m, int n) {
  // 将有序的记录序列 SR[i..m] 和 SR[m+1..n]
  //归并为有序的记录序列 TR[i..n]

  for (j=m+1, k=i;  i<=m && j<=n;  ++k)
    {                    // 将SR中 记录由小到大地并入TR
      if (SR[i].key<=SR[j].key)  TR[k] = SR[i++];
      else TR[k] = SR[j++];
    }
        …    …

} // Merge
```

```
if (i<=m) TR[k..n] = SR[i..m];
                    // 将剩余的 SR[i..m] 复制到 TR

if (j<=n) TR[k..n] = SR[j..n];
                    // 将剩余的 SR[j..n] 复制到 TR
```

# 归并排序的算法

如果记录无序序列 R[s..t] 的两部分

R[*s*..⌊(*s*+*t*)/2⌋] 和 R[⌊(*s*+*t*)/2⌋+1..*t*]

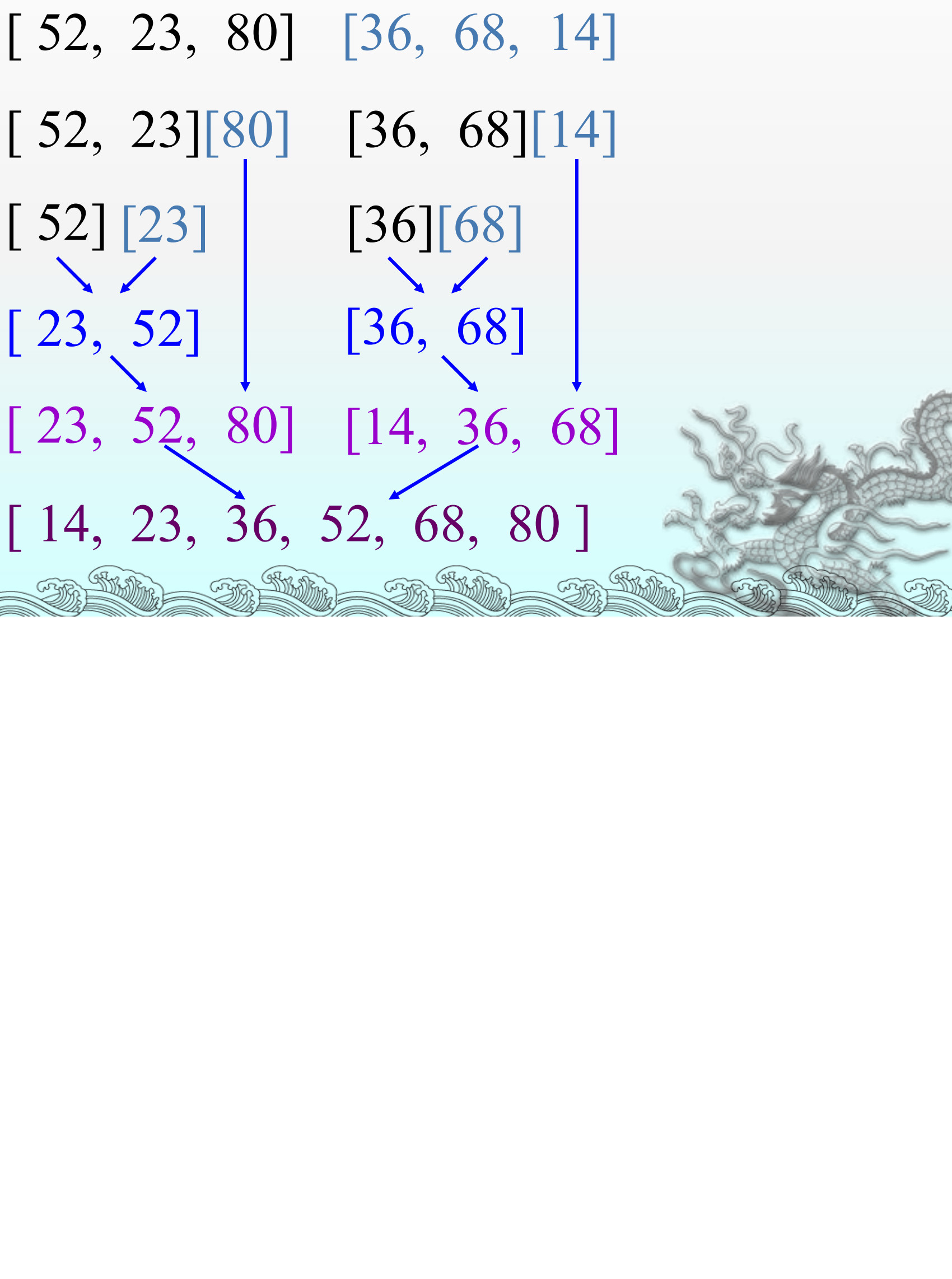
分别按关键字有序，

则利用上述归并算法很容易将它们归并成整个记录序列是一个有序序列。

由此，应该先分别对这两部分进行2-路归并排序。

## 例如：

52, 23, 80, 36, 68, 14 (s=1, t=6)



```
void Msort ( RcdType SR[],
                    RcdType &TR1[], int s, int t ) {
  // 将SR[s..t]归并排序为 TR1[s..t]
   if (s= =t) TR1[s]=SR[s];
   else
     {          …   …
      }
} // Msort
```

```
m = (s+t)/2;
        // 将SR[s..t]平分为SR[s..m]和SR[m+1..t]

Msort (SR, TR2, s, m);
    // 递归地将SR[s..m]归并为有序的TR2[s..m]
Msort (SR, TR2, m+1, t);
    //递归地SR[m+1..t]归并为有序的TR2[m+1..t]

Merge (TR2, TR1, s, m, t);
    // 将TR2[s..m]和TR2[m+1..t]归并到TR1[s..t]
```

```
void MergeSort (SqList &L) {
  // 对顺序表 L 作2-路归并排序
  MSort(L.r, L.r, 1, L.length);
} // MergeSort
```

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 $\lceil \log_2 n \rceil$ 趟。

题6：将两个各有n个元素的有序表归并成一个有序表，其最少的关键字比较次数是___。

A. n　　B. 2n-1　　C. 2n　　D.n-1

题7：已知序列{15,5,16,2,25,8,20,9,18,12}，采用二路归并排序法对该序列作升序排序时的每一趟的结果。

题8：对数据序列{15,9,7,8,20,-1,4}进行排序，进行一趟后数据的排序变为{9,15,7,8,20,-1,4}，则采用的是____算法。

A.简单选择排序　　B.冒泡排序

C.直接插入排序　　D.堆排序

题9：对一组数据{25,84,21,47,15,27,68,35,20}进行排序，前3躺的排序结果如下：

第一趟：20,15,21,25,47,27,69,35,84

第二趟：15,20,21,25,35,27,47,68,84

第三趟：15,20,21,25,27,35,47,68,84

则所采用的排序算法是_____。

A.简单选择排序　　B.希尔排序

C.二路归并排序　　D.快速排序

题10：序列{3,2,4,1,5,6,8,7}是第一趟递增排序后的结果，则采用的排序方法可能是_____。

A.快速排序　　B.冒泡排序

C.堆排序　　D.简单选择排序

# 10.6 基数排序

基数排序是一种采用对“多关键字排序”的思想来实现“单关键字排序”的内部排序算法。

## 一、多关键字的排序

♣2<♣3<♣4<♣5<♣6<♣7<♣8<♣9<♣10<♣J<♣Q<♣K<♣A<

♦2<♦3<♦4<♦5<♦6<♦7<♦8<♦9<♦10<♦J<♦Q<♦K<♦A<

♥2<♥3<♥4<♥5<♥6<♥7<♥8<♥9<♥10<♥J<♥Q<♥K<♥A<

♠2<♠3<♠4<♠5<♠6<♠7<♠8<♠9<♠10<♠J<♠Q<♠K<♠A

花色（♣<♦<♥<♠）优先于面值(2<3<...<A)

这就是多关键字排序。排序后形成的有序序列叫做词典有序序列。

对于上例两排序码的排序，可以先按花色排序（分为4堆），之后再按面值排序；也可以先按面值排序（“分配”为13堆后收集起来），再按花色排序（“分配”为4堆后收集起来）。

$n$ 个记录的序列 $\{R_1, R_2, \ldots, R_n\}$ 对关键字 $(K_i^0, K_i^1, \ldots, K_i^{d-1})$ 有序是指：

对于序列中任意两个记录 $R_i$ 和 $R_j(1 \leqslant i<j \leqslant n)$ 都满足下列(词典)有序关系：

$$(K_i^0, K_i^1, \ldots, K_i^{d-1}) < (K_j^0, K_j^1, \ldots, K_j^{d-1})$$

其中：$K^0$ 被称为“最主”位关键字

$K^{d-1}$ 被称为“最次”位关键字

实现多关键字排序通常有两种作法:

## 最高位优先MSD法

先对$K^0$进行排序，并按 $K^0$的不同值将记录序列分成若干子序列之后，分别对 $K^1$进行排序，…，依此类推，直至最后对最次位关键字排序完成为止。

## 最低位优先LSD法

先对 $K^{d-1}$进行排序，然后对 $K^{d-2}$ 进行排序，依此类推，直至对最主位关键字 $K^0$排序完成为止。

排序过程中不需要根据“前一个”关键字的排序结果，将记录序列分割成若干个(“前一个”关键字不同的)子序列。

例如: 学生记录含三个关键字: 系别、班号和班内的序列号，其中以系别为最主位关键字。

LSD的排序过程如下:

| | | | | | |
|---|---|---|---|---|---|
| 无序序列 | 3,2,30 | 1,2,15 | 3,1,20 | 2,3,18 | 2,1,20 |
| 对$K^2$排序 | 1,2,15 | 2,3,18 | 3,1,20 | 2,1,20 | 3,2,30 |
| 对$K^1$排序 | 3,1,20 | 2,1,20 | 1,2,15 | 3,2,30 | 2,3,18 |
| 对$K^0$排序 | 1,2,15 | 2,1,20 | 2,3,18 | 3,1,20 | 3,2,30 |

## 二、链式基数排序

假如多关键字的记录序列中，每个关键字的取值范围相同，则按LSD法进行排序时，可以采用“分配-收集”的方法，其好处是不需要进行关键字间的比较。

对于数字型或字符型的单关键字，可以看成是由多个数位或多个字符构成的多关键字，此时可以采用这种“分配-收集”的办法进行排序，称作基数排序法。

例如：对下列这组关键字

{209, 386, 768, 185, 247, 606, 230, 834, 539 }

首先按其“个位数”取值分别为 0, 1, …, 9 “分配”成 10 组，之后按从 0 至 9 的顺序将它们“收集”在一起;

然后按其“ 十位数” 取值分别为 0, 1, …, 9 “分配” 成 10 组，之后再按从 0 至 9 的顺序将它们“ 收集” 在一起；最后按其“ 百位数”重复一遍上述操作。

在计算机上实现基数排序时，为减少所需辅助存储空间，应采用链表作存储结构，即链式基数排序，具体作法为：

1．待排序记录以指针相链，构成一个链表；

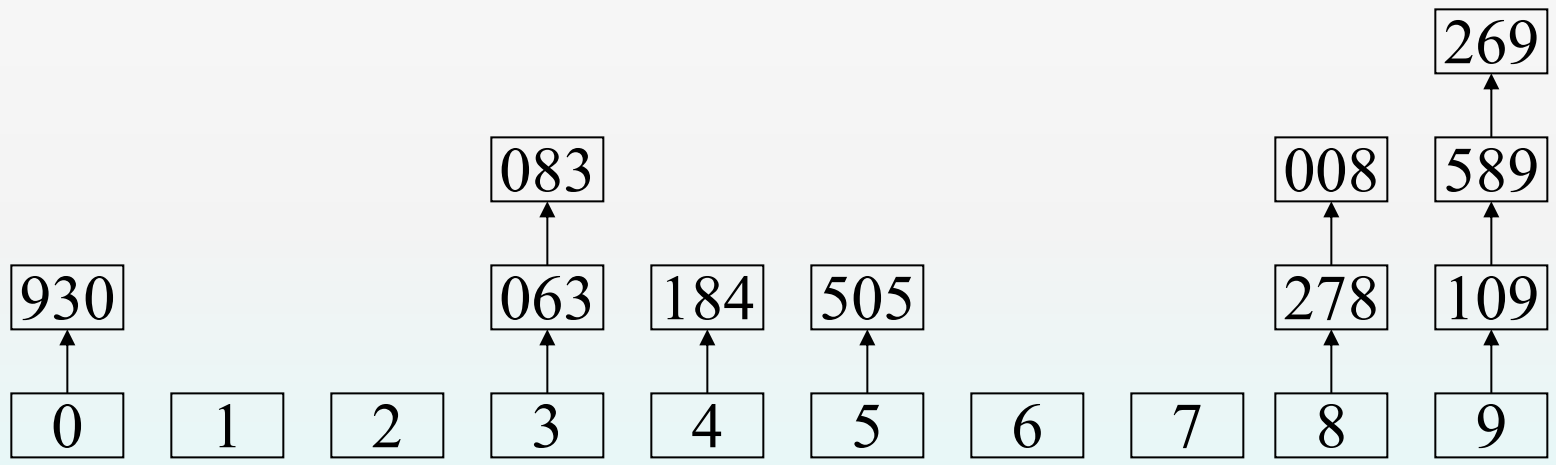
2．“ 分配” 时，按当前“ 关键字位”所取值，将记录分配到不同的“ 链队列” 中，每个队列中记录的“ 关键字位” 相同；

3．“ 收集”时，按当前关键字位取值从小到大将各队列首尾相链成一个链表；

4．对每个关键字位均重复 2 和 3 两步。

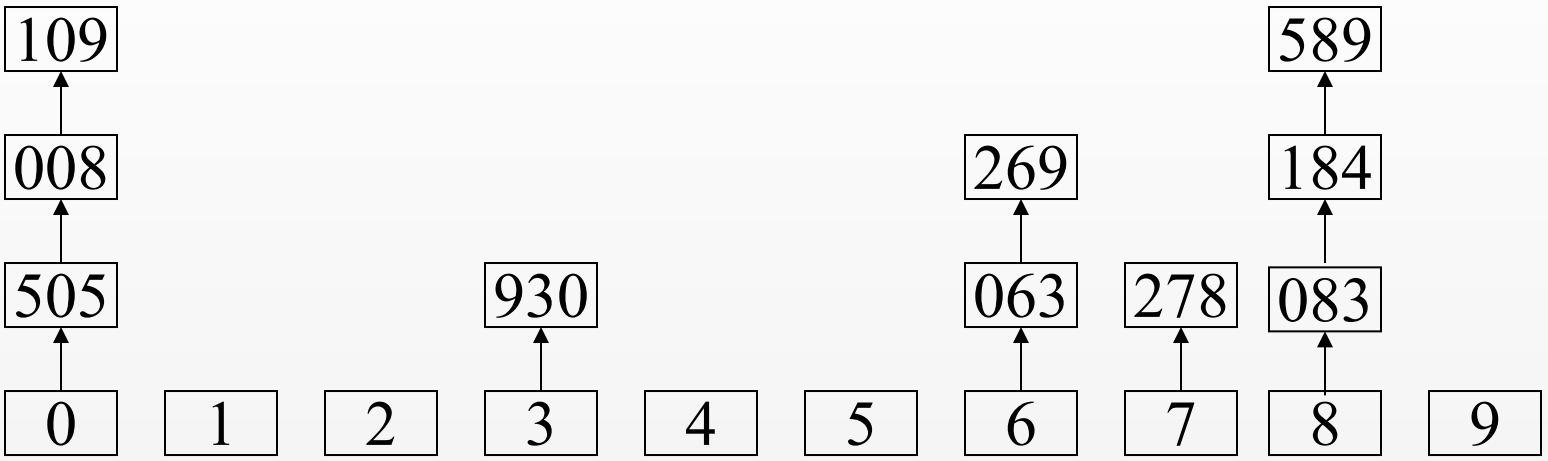
例:

278->109->063->930->589->184->505->269->008->083

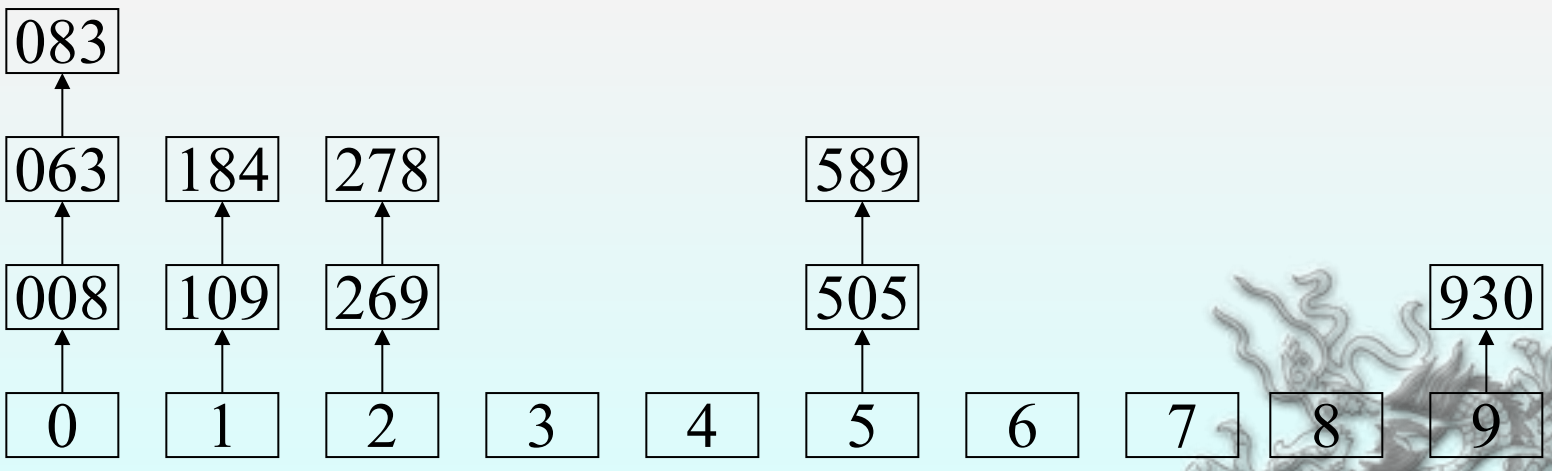



930->063->083->184->505->278->008->109->589->269

930->063->083->184->505->278->008->109->589->269




505->008->109->930->063->269->278->083->184->589




008->063->083->109->184->269->278->505->589->930

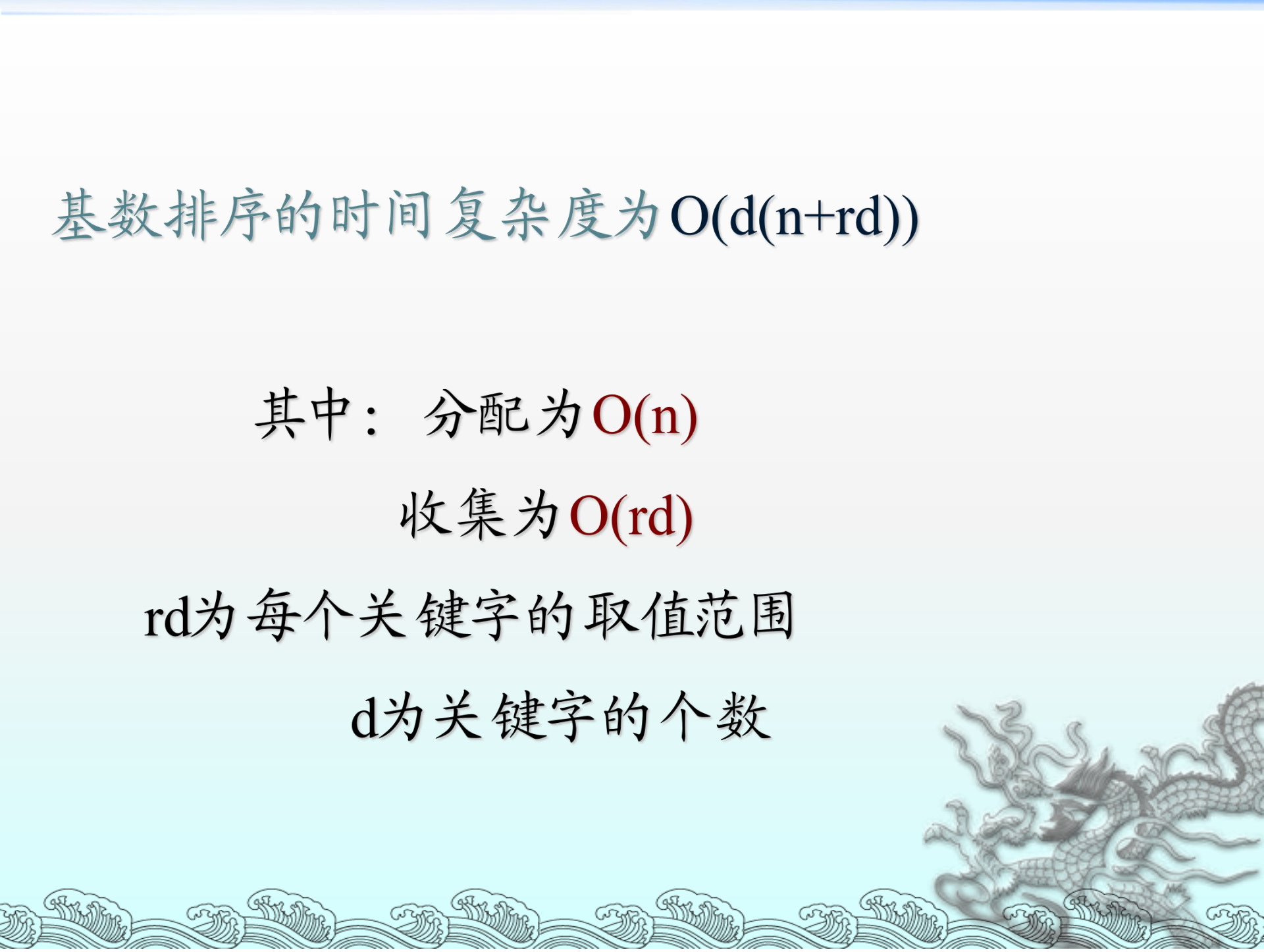
基数排序的时间复杂度为O(d(n+rd))

其中： 分配为O(n)

收集为O(rd)

rd为每个关键字的取值范围

d为关键字的个数



# 10.7

# 各种排序方法的综合比较

# 一、时间性能

## 1. 平均的时间性能

**时间复杂度为 O($n$log$n$):**

快速排序、堆排序和归并排序

**时间复杂度为 O($n^2$):**

直接插入排序、起泡排序和简单选择排序

**时间复杂度为 O(n):**

基数排序

2. 当待排记录序列按关键字顺序有序时

直接插入排序和起泡排序能达到$O(n)$的时间复杂度，

快速排序的时间性能蜕化为$O(n^2)$。

3. 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。

# 二、空间性能

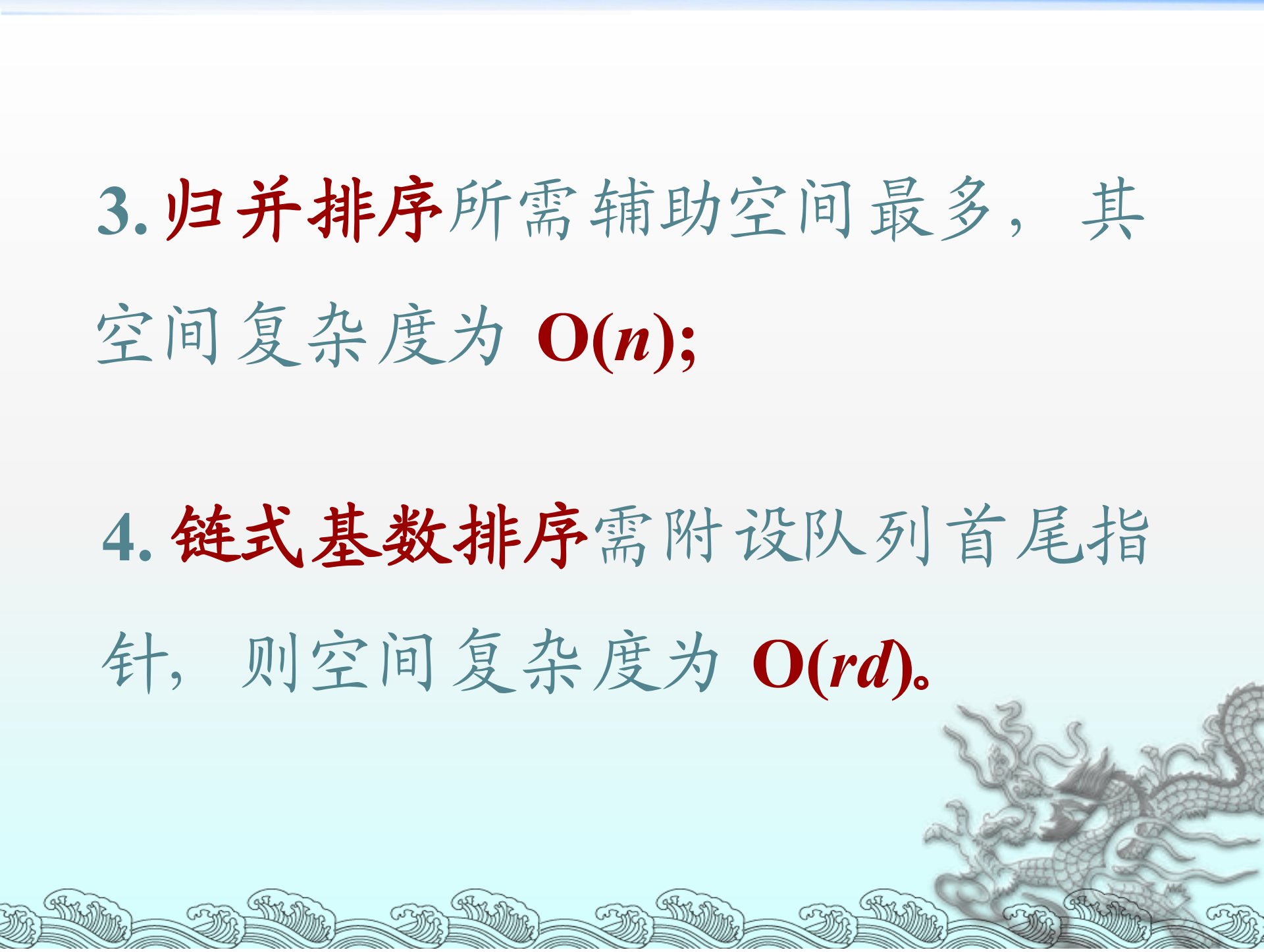
**指的是排序过程中所需的辅助空间大小**

1. 所有的**简单排序方法**(包括：直接插入、起泡和简单选择)和**堆排序**的空间复杂度**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*);**

4. **链式基数排序**需附设队列首尾指针，则空间复杂度为 **O(*rd*)。**



# 三、排序方法的稳定性能

**1.** 稳定的排序方法指的是，对于两个关键字相等的记录，它们在序列中的相对位置，在排序之前和经过排序之后，没有改变。

排序之前：{ · · · · · $R_i(K)$ · · · · · $R_j(K)$ · · · · · }

排序之后：{ · · · · · $R_i(K)$ $R_j(K)$ · · · · · · · · · }

**2.** 当对多关键字的记录序列进行LSD方法排序时，必须采用稳定的排序方法。

例如：

排序前 ( 56, 34, **47**, 23, 66, 18, 82, **47** )

若排序后得到结果

( 18, 23, 34, **47**, **47**, 56, 66, 82 )

则称该排序方法是**稳定**的;

若排序后得到结果

( 18, 23, 34, **47**, **47**, 56, 66, 82 )

则称该排序方法是**不稳定**的。

**3.** 对于不稳定的排序方法，只要能举出一个实例说明即可。

**例如：** 对 { **4**, 3, 4, 2 } 进行快速排序，

得到 { 2, 3, 4, **4** }

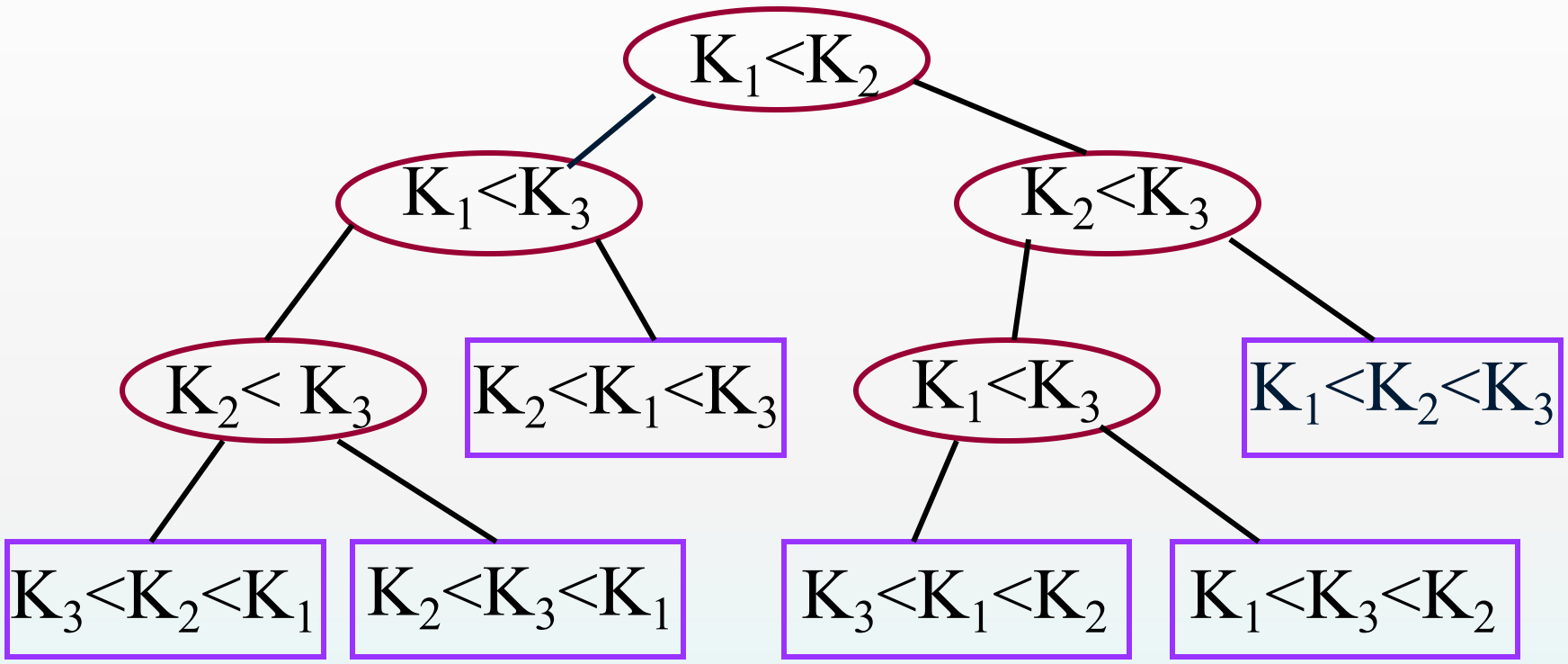
**4. 快速排序、堆排序和希尔排序是不稳定的排序方法。**

## 四、关于“排序方法的时间复杂度的下限”

本章讨论的各种排序方法，除基数排序外，其它方法都是**基于“比较关键字”进行排序的排序方法。**

可以证明，这类排序法**可能达到的最快的时间复杂度为O(*n*log*n*)**。（基数排序不是基于“比较关键字”的排序方法，所以它不受这个限制。）

例如:对三个关键字进行排序的判定树如下:




1．树上的每一次“比较”都是必要的

2．树上的叶子结点包含所有可能情况。

一般情况下，对$n$个关键字进行排序，可能得到的结果有$n!$种，由于含$n!$个叶子结点的二叉树的深度不小于$\lceil \log_2(n!) \rceil +1$，则对 $n$ 个关键字进行排序的比较次数至少是$\lceil \log_2(n!) \rceil \approx n\log_2 n$ (斯蒂林近似公式)。

所以，**基于“比较关键字”进行排序的**
**排序方法，可能达到的最快的时间复杂度为$O(n\log n)$。**

# 作业

- 1. 以关键码序列(503,087,512,061,908,170,897,275,653,426)为例，手工执行以下排序算法，写出每一趟排序结束时的关键码状态：
  (1) 直接插入排序；(2) 希尔排序(增量 d[1]=5 )；
  (3) 快速排序；(4) 堆排序；
  (5) 归并排序。

- 2. 不难看出，对长度为n的记录序列进行快速排序时，所需进行的比较次数依赖于这 n 个元素的初始排列。
  (1) n=7 时在最好情况下需进行多少次比较? 请说明理由。
  (2) 对 n=7 给出一个最好情况的初始排列实例。

- 3. 判别以下序列是否为堆（小顶堆或大顶堆）。如果不是，则把它调整为堆（要求记录交换次数最少）。
  (1) (100, 86, 48, 73, 35, 39, 42, 57, 66, 21);
  (2) (12, 70, 33, 65, 24, 56, 48, 92, 86, 33);
  (3) (103, 97, 56, 38, 66, 23, 42, 12, 30, 52, 06, 20);
  (4) (05, 56, 20, 23, 40, 38, 29, 61, 35, 76, 28, 100)。