第十章　内部排序

制作：　数据结构精品资源共享课课题组

南京审计大学 信息工程学院

2022.10

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$\{ R_1, R_2, \cdots, R_n \}$

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

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：

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

按此固有关系将上式记录序列重新排列为

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

的操作称作**排序**。

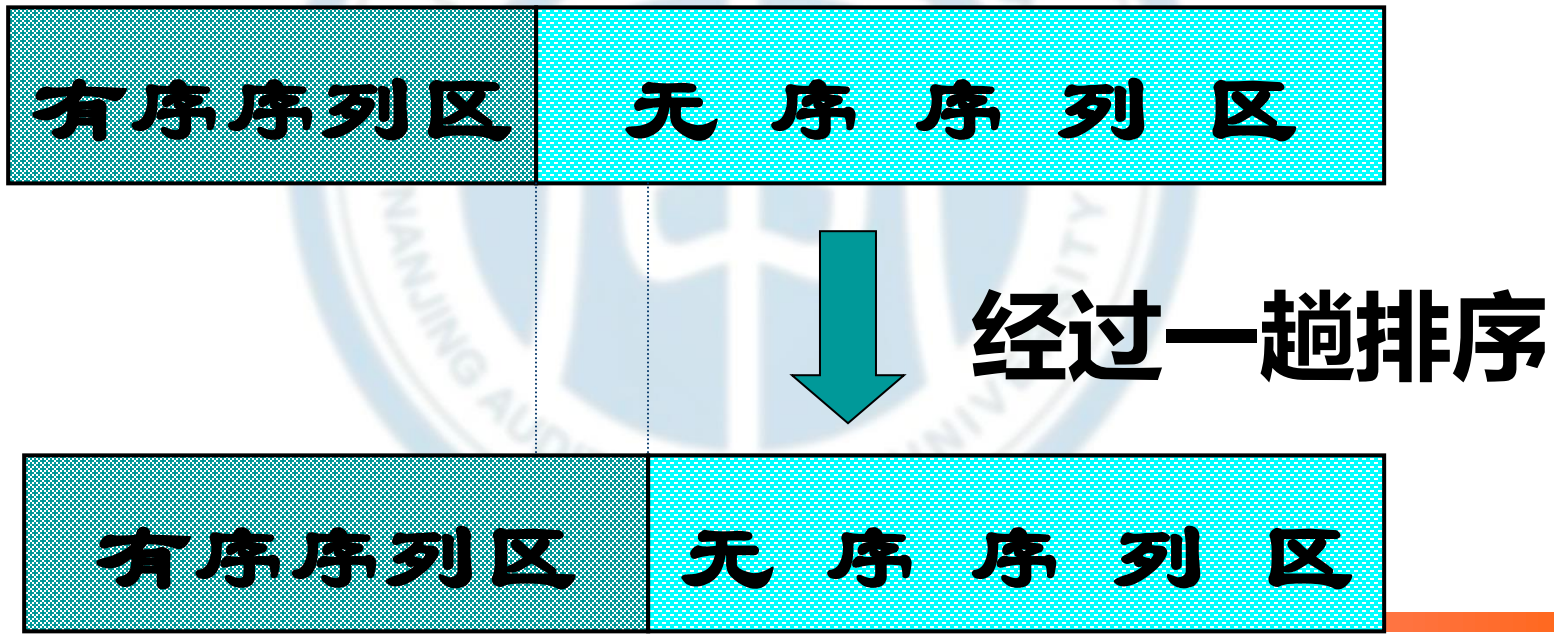
二、内部排序和外部排序

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。

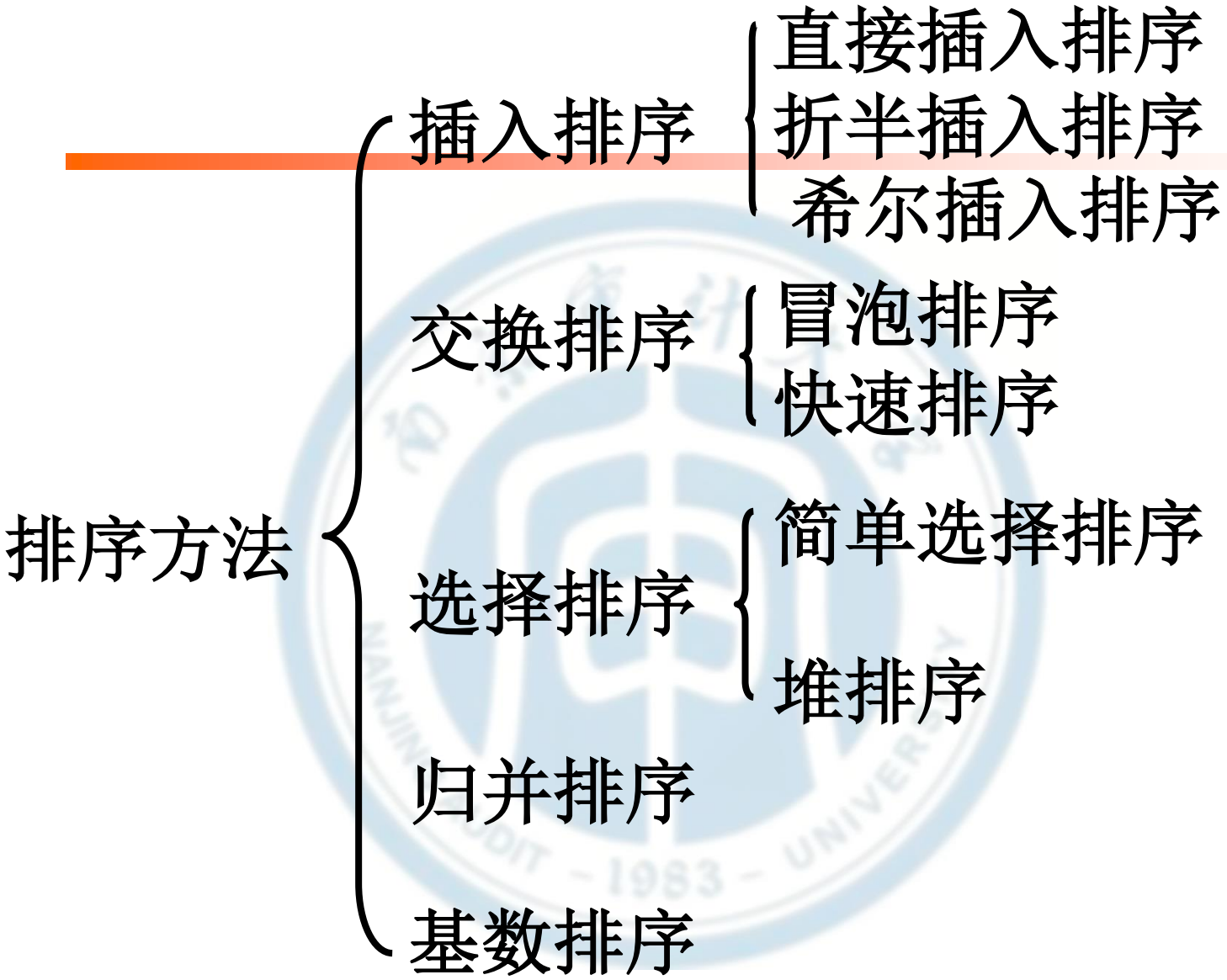


基于不同的“扩大” 有序序列长度

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


排序方法
插入排序
直接插入排序
折半插入排序
希尔插入排序
交换排序
冒泡排序
快速排序
选择排序
简单选择排序
堆排序
归并排序
基数排序

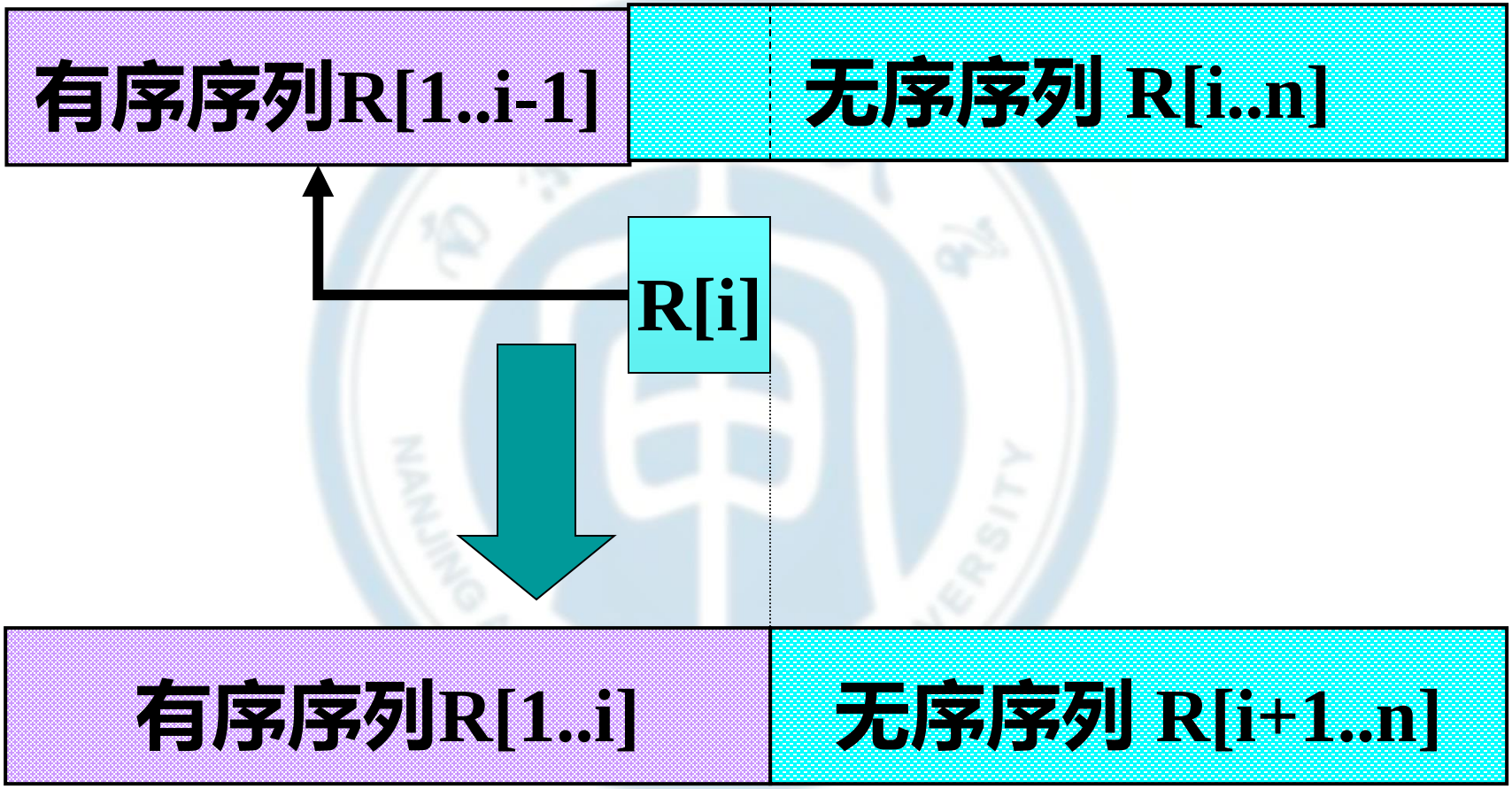
待排记录的数据类型定义如下：

```
typedef  int  KeyType;   // 关键字类型为整数类型

#define MAXSIZE  1000 // 待排顺序表最大长度
 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
```

10.2
插入排序

一趟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，

$R[1..j].key \le R[i].key < R[j+1..i-1].key$；

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)

希尔排序 (基于逐趟缩小增量)

一、直接插入排序

直接插入排序(InsertSort)

Cooling

R[ ]

输入待排序的记录数组R[1..n](数据之间用半角逗号隔开)

Clear Start

一、直接插入排序

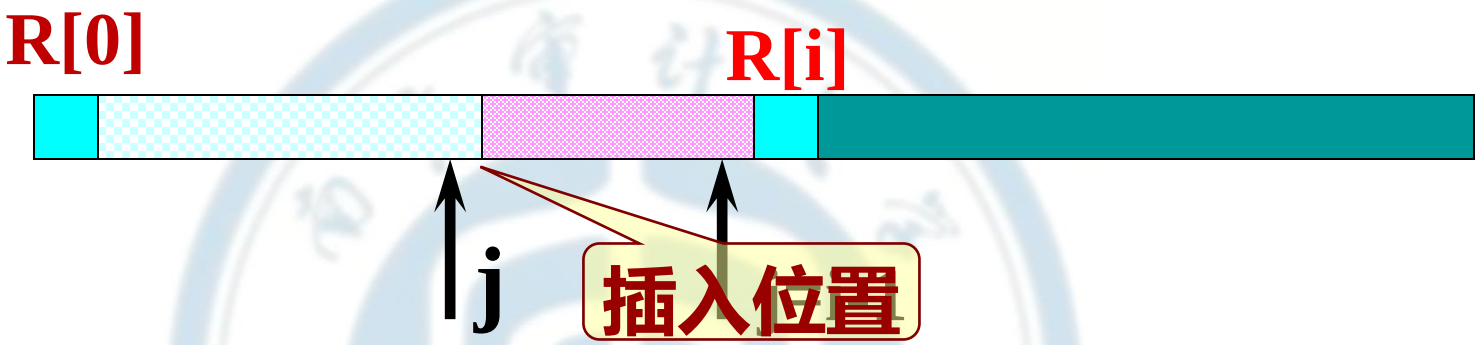
利用“顺序查找”实现

“在R[1..i-1]中查找R[i]的插入位置”

算法的实现要点：

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]
```

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
```

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：

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

“移动”的次数：

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

- 算法评价

- 时间复杂度：$T(n)=O(n^2)$
- 空间复杂度：$S(n)=O(1)$

二、折半插入排序

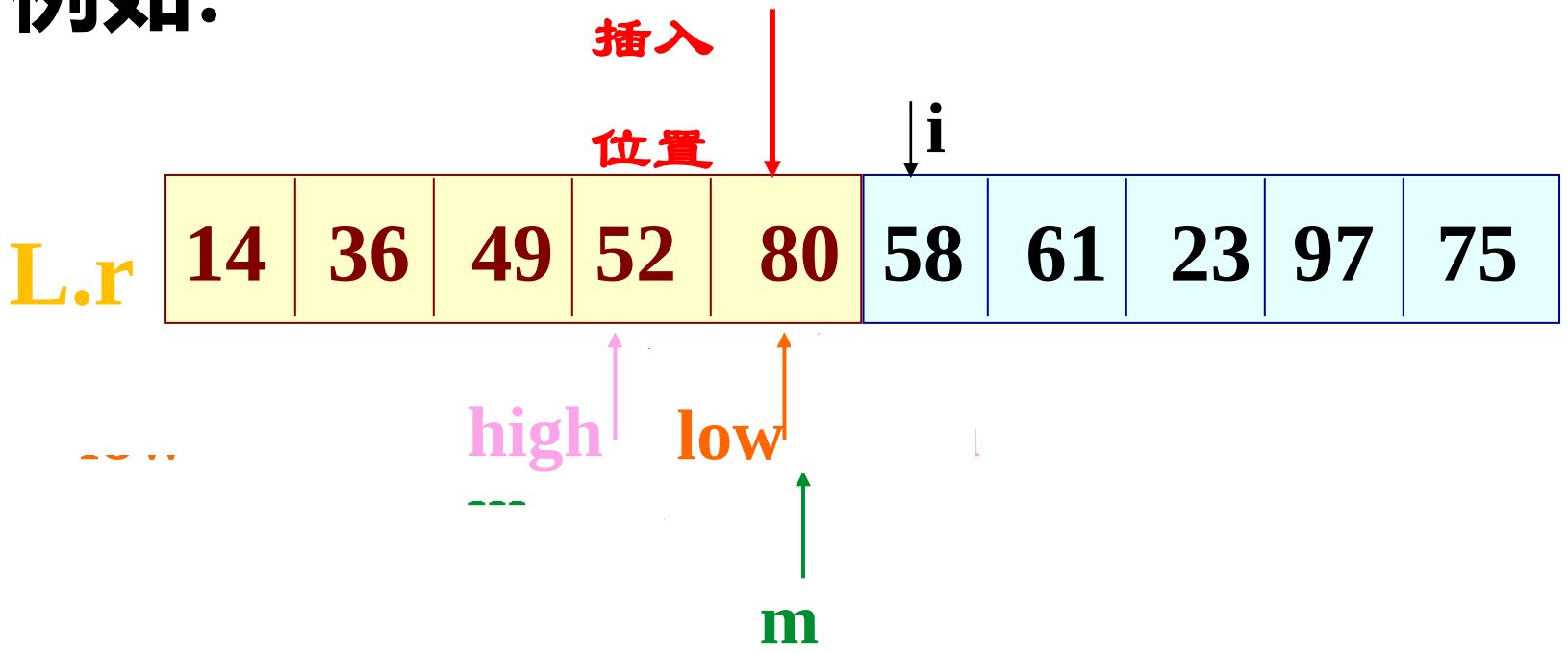
因为 R[1..i-1] 是一个按关键字有序的有序序列，则可以**利用折半查找实现“在R[1..i-1]中查找R[i]的插入位置”**，如此实现的插入排序为**折半插入**排序。

```
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：



– 算法评价

- 时间复杂度：$T(n)=O(n^2)$
- 空间复杂度：$S(n)=O(1)$

折半插入排序只能减少排序过程中关键字比较的时间，并不能减少记录移动的时间。

三、希尔排序（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。

对待排记录序列先做“ 宏观 ”调整，
再作“ 微观 ”调整。
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

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。

三、希尔排序（缩小增量排序）

希尔排序 bobo

初始关键字

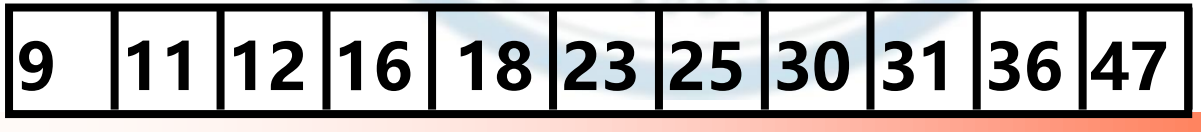
开始

例如：16 25 12 30 47 11 23 36 9 18 31

第一趟希尔排序，设增量 d=5

第二趟希尔排序，设增量 d=3

第三趟希尔排序，设增量 d=1



一趟希尔排序算法

```
void ShellInsert ( SqList &L, int dk){
    // 对顺序表L作一趟希尔插入排序。本算法是和一趟直接插入排序
    // 相比，作了如下修改：1.前后记录位置的增量是dk，不是1；
    // 2.r[0]只是暂存单元，不是哨兵。当j<=0时，插入位置已找到。
    for ( i=dk+1; i<=L.length; ++i )
        if (LT(L.r[i].key , L.r[i-dk].key )) {
                // 需将L.r[i]插入有序增量子表
            L.r[0] = L.r[i];
            for(j=i-dk; j>0&&LT(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
```

希尔排序的算法

```
void ShellSort (SqList &L, int dlta[], int t){
      // 按增量序列 dlta[0..t-1] 对顺序表L作希尔排序
   for (k=0; k<t; ++t)          // 一趟增量为 dlta[k] 的插入排序
      ShellInsert(L, dlta[k]);
} // ShellSort
```

希尔排序的时间复杂度和所取增量序列相关，例如已有学者证明，当增量序列为 $2^{t-k-1}(k=0,1,\ldots,t-1)$ 时，希尔排序的时间复杂度为 $O(n^{3/2})$。

希尔排序特点

- 子序列的构成**不是简单的“逐段分割”**，而是将**相隔某个增量的记录组成一个子序列。**
- 希尔排序可提高排序速度，因为
  - 分组后n值减小，n^2更小，而$T(n)=O(n^2)$，所以T(n)从总体上看是减小了；
  - 关键字较小的记录跳跃式前移，在进行最后一趟增量为1的插入排序时，序列已基本有序。
- 增量序列取法
  - **无除1以外的公因子；**
  - **最后一个增量值必须为1；**
  - 如……, 9, 5, 3, 2, 1或……, 31, 15, 7, 3, 1或……, 40, 13, 4, 1等。

10.3 交换排序

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：

一、起泡排序


冒泡排序(BubbleSort)
Cooling
R[
]
Clear
Start
输入待排序的记录数组R[1..n](数字间用半角逗号分隔)

算法实现1：

```
void BubbleSort(int R[ ],int n)
{   int i, j, flag,temp;
    for (i = n;i >= 2;i --)   //数组从下标1开始存储数据
    { flag = 0;               //用来标识本趟排序是否发生了交换
      for ( j = 2; j <= i; j ++)
          if (R[ j -1] > R[ j ])
              { temp = R[ j ];
                R[ j ] = R[ j -1];
                R[ j -1] = temp;
                flag = 1;
              }
      if ( flag == 0)
          return;
    }
}
```

起泡排序的结束条件为：最后一趟没有进行“交换”

算法实现2：

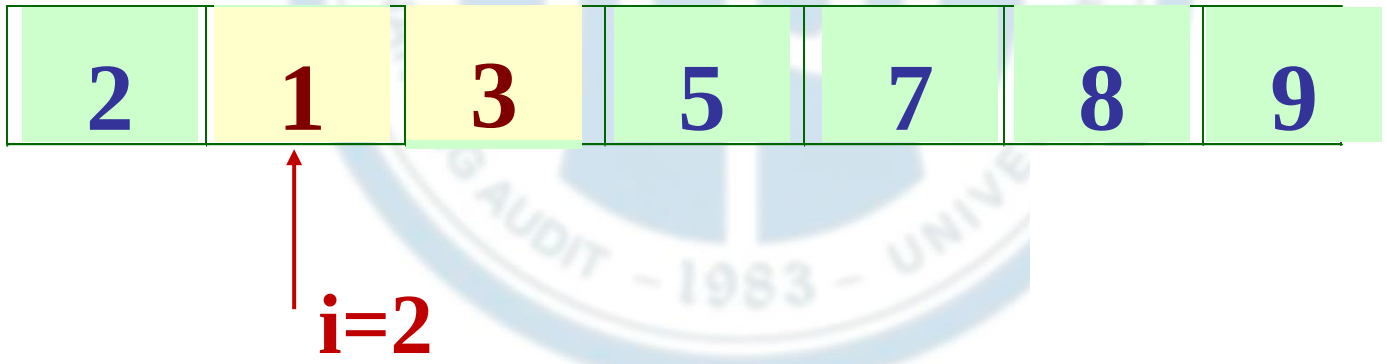
```
void BubbleSort(SqList &L) {
    i = L.length;
   while (i >1) {
      lastExchangeIndex = 1;
      for (j = 1;  j < i;  j++)
     if (L.r[j+1].key < L.r[j].key) {
        Swap(L.r[j], L.r[j+1]);
        lastExchangeIndex = j;  //记下进行交换的记录位置
     } //if
    i = lastExchangeIndex; // 本趟进行过交换的
   } // while                    // 最后一个记录的位置
} // 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 –1”，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：

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

例：双向冒泡排序

```
void DoubleBubble(int R[ ], int n)   //R[]中元素的存储从1~n-1
{   int i, j, temp, flag;
    i = 0;
    flag = 0;
    while ( flag = 0)
    {       flag = 1;
            for ( j = n - i - 1; j > i; j--)
             if (R[ j ] < R[ j - 1])
                { flag = 0;
                  temp = R[ j ];
                  R[ j ] = R[ j - 1];
                  R[ j - 1] = temp;
                }
            for ( j = i; j < n - 1; j++)
              if (R[ j ] > R[ j + 1])
                { flag = 0;
                  temp = R[ j ];
                  R[ j ] = R[ j - 1];
                  R[ j - 1] = temp;
                }
            i++;
    }
}
```

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

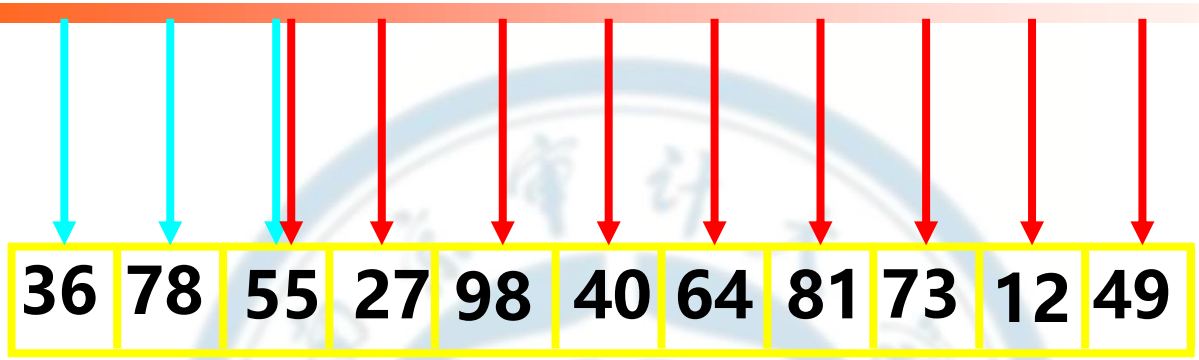
$$R[j].key \le R[i].key \le R[j].key$$

$$(s \le j \le i-1) \quad \text{枢轴} \quad (i+1 \le j \le t)$$

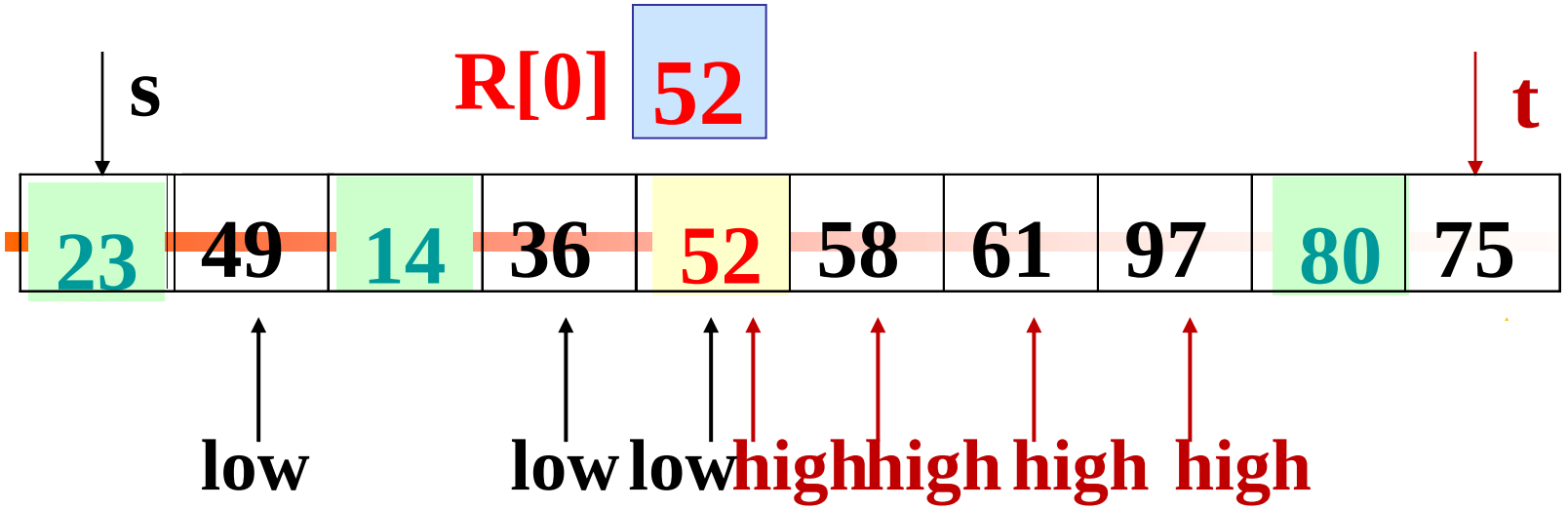
找一个记录，以它的关键字作为“枢轴”，凡其关键字小于枢轴的记录均移动至该记录之前，反之，凡关键字大于枢轴的记录均移动至该记录之后。递归对该枢轴的左右子序列执行相同操作

交换排序——快速排序

例如：



可以看出一趟快速排序后，将原来的序列以36为枢轴划分为两个子序列，左边的序列中的元素都小于等于它，右边的都大于等于它。接下来对两个子序列用同样的方法进行排序。经过几次这样的快速排序，最终得到一个有序的序列。



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

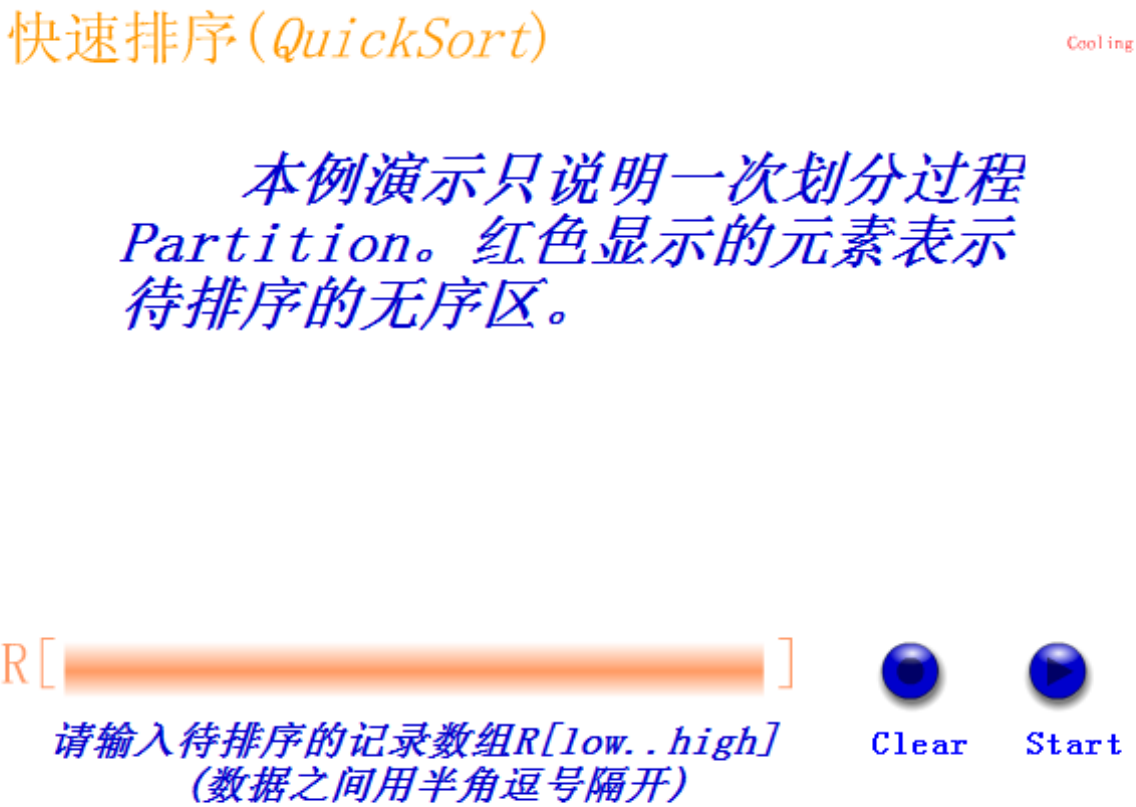
调整为：23, 49, 14, 36, (52) 58, 61, 97, 80, 75

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

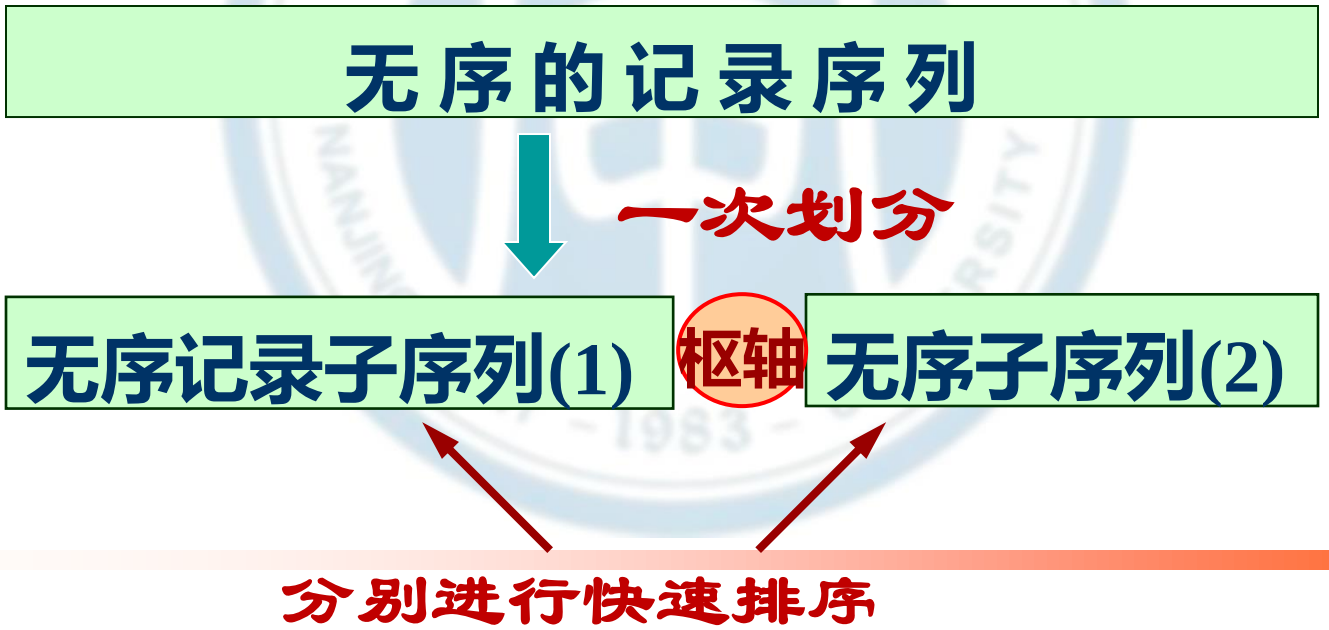
R[high].key≥52，和 R[low].key≤52,否则进行记录的“交换”。

三、快速排序


快速排序(QuickSort)
Cooling
本例演示只说明一次划分过程Partition。红色显示的元素表示待排序的无序区。
R[ ]
请输入待排序的记录数组R[low..high](数据之间用半角逗号隔开)
Clear
Start

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。



一趟快速排序的算法

```
int Partition ( SqList &L, int low, int high){
  // 交换顺序表L中子表r[low..high]中的记录，枢轴记录到位，并返回
  // 其所在位置，此时，在它之前（后）的记录均不大（小）于它。
  L.r[0] = L.r[low];                  //用子表的第一个记录作枢轴记录
  pivotkey = L.r[low].key;            // 枢轴记录关键字
  while (low < high) {                // 从表的两端交替地向中间扫描
      while (low < high && L.r[high].key >= pivotkey) --high;
      L.r[low] = L.r[high];     // 将比枢轴记录小的记录移到低端
      while (low < high && L.r[low].key <= pivotkey)  ++low;
      L.r[high] = L.r[low];         // 将比枢轴记录大的记录移到高端
  } // while
   L.r[low] = L.r[0];                 // 枢轴记录到位
  return low;                         // 返回枢轴位置
} // Partition
```

快速排序算法

```
void Qsort(SqList &L, int low, int high){
   //对顺序表L中的子序列L.r[low...high]作快速排序
   if (low < high) {                          //长度大于1
        pivotloc= Partition(L,low,high);      //将L.r[low...high]划分
        Qsort(L,low,pivotloc-1);              //对低子表排序
        Qsort(L, pivotloc+1,high);            //对高子表排序
   }
}// Qsort
void QuickSort(SqList &L){
   //对顺序表L作快速排序
   Qsort(L,1,L.length);
}// QuickSort
```

```
void QuickSort(int R[ ], int low, int high)
{  int temp;
   int i = low, j = high;
   if (low < high)
   {  temp = R[low];
      while (i != j)
      {  while (j > i && R[j] > temp)  j--;
         if (i < j)
         {  R[i] = R[j];
            i++;
         }
         while (i < j && R[i] < temp)  i++;
         if (i < j)
         {  R[j] = R[i];
            j--;
         }
      }
      R[i] = temp;
      QuickSort(R, low, i-1);
      QuickSort(R, i+1, high);
   }
}
```

快速排序算法实现2：

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 $T_{avg}(1) \le b$

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。

- 算法分析
  - 时间复杂度
    - 最好情况（每次总是选到中间值作枢轴）
      $T(n)=O(n\log_2 n)$
    - 最坏情况（每次总是选到最小或最大元素作枢轴）
      $T(n)=O(n^2)$
  - 空间复杂度：需栈空间以实现递归
    - 最坏情况：$S(n)=O(n)$
    - 一般情况：$S(n)=O(\log_2 n)$
  - 对随机的关键字序列，快速排序是**目前被认为是最好的排序方法。**

10.4 选择排序

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：

一、简单选择排序

直接选择排序(*StraightSelectionSort*) Cooling

简单选择排序算法

```
void SelectSort (SqList &L){
      // 对顺序表L作简单选择排序
   for (i=1; i<L.length; ++i) {
     // 选择第 i 小的记录，并交换到位
        j = i;
        for ( k=i+1; k<=L.length; k++ )
   // 在L.r[i..L.length]中选择关键字最小的记录
                if ( LT( L.r[k].key , L.r[j].key )) j =k ;
        if ( i!=j ) L.r[j] ↔ L.r[i];    // 与第 i 个记录互换
   }// for
} // SelectSort
```

时间复杂度$T(n)=O(n^2)$

算法评价

- 时间复杂度
  - 记录移动次数
    - 最好情况：0
    - 最坏情况：3(n-1)
  - 比较次数：$\sum_{i=1}^{n-1}(n-i)=\frac{n(n-1)}{2}$
- 空间复杂度：S(n)=O(1)

二、堆排序

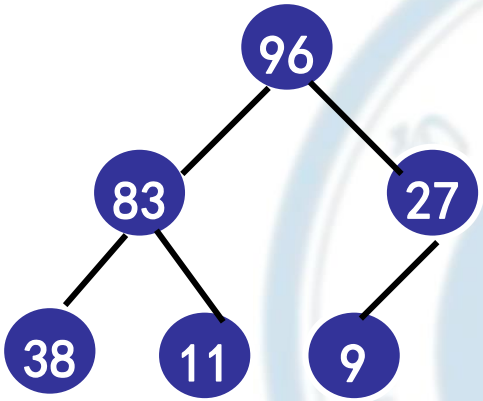
堆的定义：

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

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

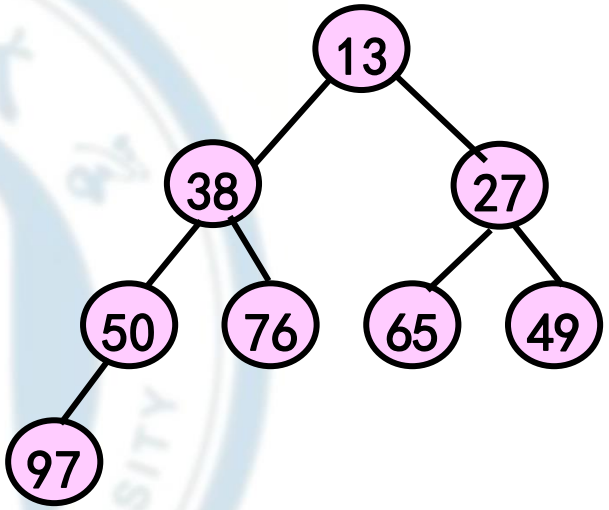
- 若将此序列对应的一维数组看成是一个完全二叉树，则r_i为二叉树的根结点，r_{2i}和r_{2i+1}分别为r_i的“左子树根”和“右子树根”。

（96，83，27，38，11，9）



大顶堆

（13，38，27，50，76，65，49，97）



小顶堆

- **堆排序**
  - 将无序序列建成一个堆，得到关键字最小（或最大）的记录；输出堆顶的最小（大）值后，使剩余的n-1个元素重又建成一个堆，则可得到n个元素的次小值；重复执行，得到一个有序序列的过程。
- 堆排序需解决的两个问题：
  - 如何由一个无序序列建成一个堆？（初始建堆）
  - 如何在输出堆顶元素之后，调整剩余元素，使之成为一个新的堆？

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 }

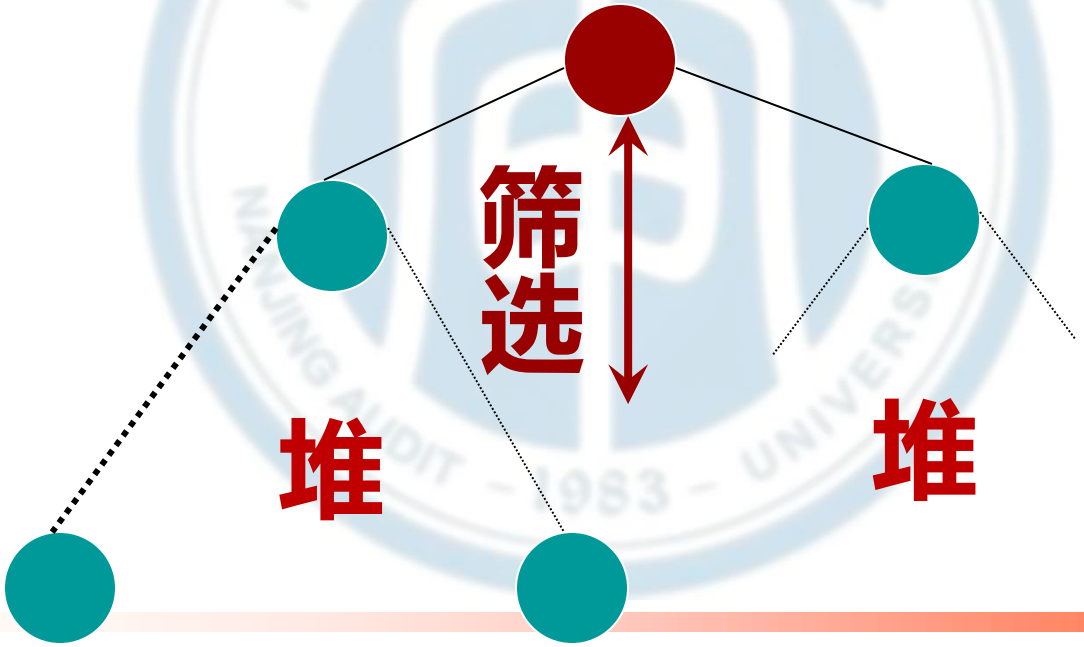
重新调整为大顶堆

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

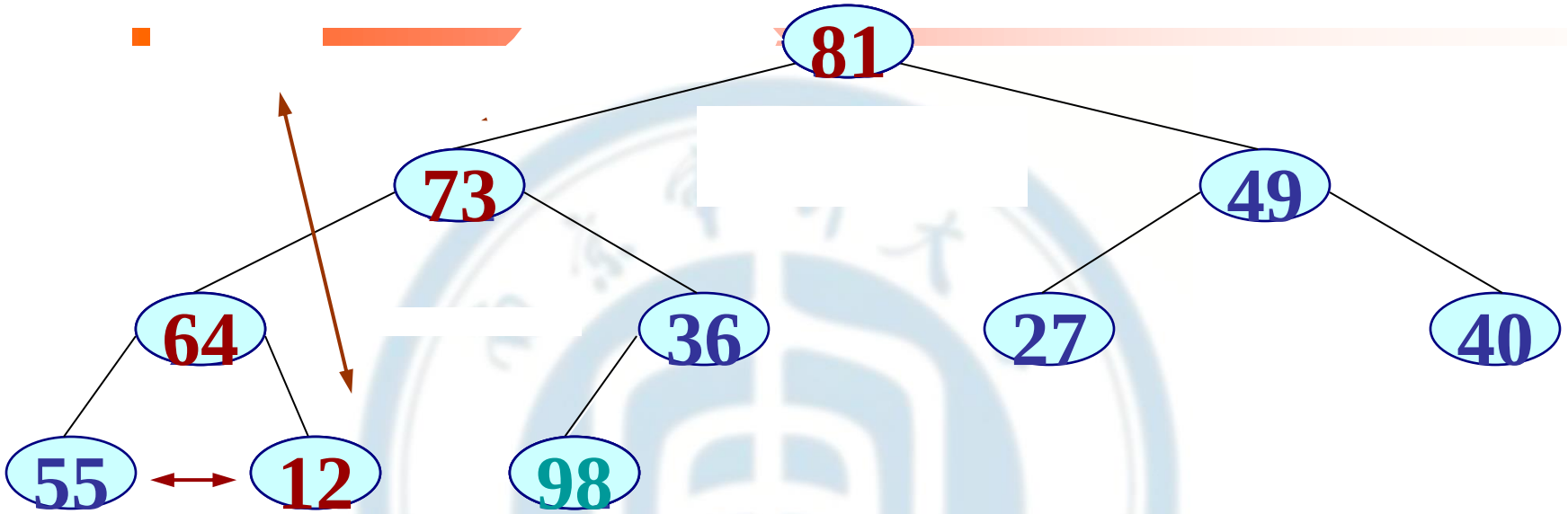
- 第二个问题解决方法——筛选（以大顶堆为例）

  - 输出堆顶元素之后，以堆中最后一个元素替代之；
  - 然后将根结点值与左、右子树的根结点值进行比较，并与其中大者进行交换；
  - 重复上述操作，直至叶子结点，将得到新的堆，称这个从堆顶至叶子的调整过程为“筛选”。

“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

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”。

堆排序

堆排序(*HeapSort*)

Cooling

堆排序是一种树型选择排序。待排序列R[1..n]可视为一棵完全二叉树。堆排序的具体方法：

（1）用筛选法依次将以R[n/2]、R[n/2-1]、…R[1]为根结点的二叉树调整为大根堆。

（2）设大根堆为R[1..i]，将堆顶记录R[1]与堆序列最后一个记录R[i]交换位置，形成新的有序区R[i..n]和待调整为堆的序列R[1..i-1]。

（3）若待调整序列的记录个数大于1，则对以R[1]为根的二叉树进行一次筛选，调整为大根堆，重复第二步骤。

堆排序结束后，R[1..n]即为一有序序列。在演示中，实线连接的结点表示无序结点，虚线连接的结点表示有序结点！

R[ ]

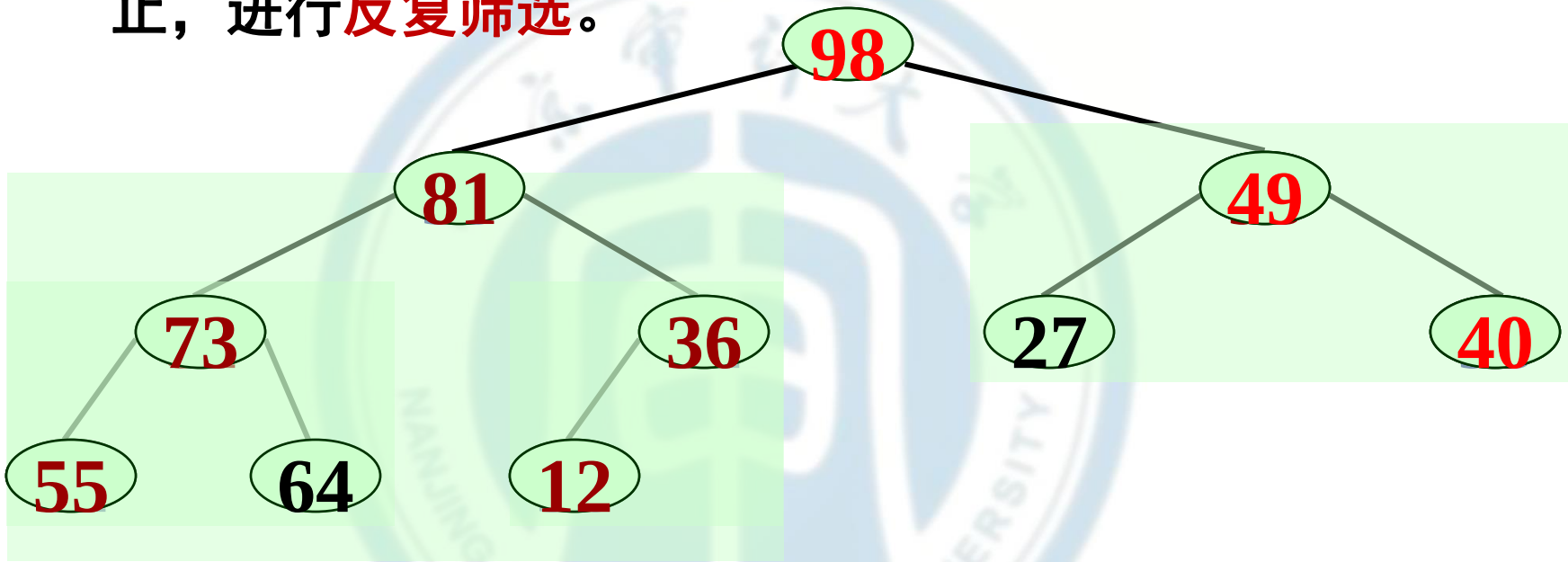
Clear Start

输入待排序的记录数组R[1..n],数据个数小于16.每个数据小于1000。(数据之间用半角逗号隔开)

筛选算法

```
void HeapAdjust (SqList &H, int s, int m){
   //使H.r[s..m]成为一个大顶堆
    rc = H.r[s];                          // 暂存根结点的记录
   for(j=2*s;j<=m;j*=2 ){//沿关键字较大的孩子结点向下筛选
        if ( j<m && LT(H.r[j].key,H.r[j+1].key ))  ++j;
   // j 为关键字较大的孩子记录的下标
        if (!LT(rc.key, H.r[j].key )) break;
                                      // 不需要调整，跳出循环
        H.r[s] = H.r[j]; s = j;  // 将大关键字记录往上调
   }// for
   H.r[s] = rc;                       // 回移筛选下来的记录
} // HeapAdjust
```

- 第一个问题解决方法
  - 从无序序列的第⌊n/2⌋个元素（即此无序序列对应的完全二叉树的最后一个非终端结点）起，至第一个元素止，进行反复筛选。



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。

```
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
```

• 算法评价

- 时间复杂度：T(n)=O(nlogn)
  - 筛选算法：最多从第1层筛到最底层，为完全二叉树的深度

    $\lfloor \log_2 n \rfloor + 1$;
  - 初始建堆：调用筛选算法$\lfloor n/2 \rfloor$次；
  - 重建堆：调用筛选算法 n-1 次。
- 空间复杂度：S(n)=O(1)
- 记录较少时，不提倡使用。

10.5 归并排序

- 归并
  - 将两个或两个以上的有序表组合成一个新的有序表。
- 2-路归并排序
  - 排序过程
    - 设初始序列含有n个记录，则可看成n个有序的子序列，每个子序列长度为1；
    - 两两合并，得到⌊n/2⌋个长度为2或1的有序子序列；
    - 再两两合并，……如此重复，直至得到一个长度为n的有序序列为止。

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 SR[*l*..*m*]	有序子序列 SR[*m*+1..*n*]

归并为一个记录的有序序列。

有 序 序 列 TR[*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++];
         }
    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
} // Merge
```

归并排序

归并排序(*MergeSort*)（只演示二路归并算法）

```
void Merge(Seqlist R,int low,int m,int high)
{ //将两个有序的子文件R[low..m]和R[m+1..high]
  //归并成一个有序的子文件R[low..high]
  int i=low,j=m+1,p=0; //置初始值
  RecType *R1; //R1是局部向量，若p定义为此类型指针速度更快
  R1=(RecType*)malloc((high-low+1)*sizeof(RecType));
  if(! R1) //申请空间失败
     error("Insufficient memory available!");
     while(i<=m&&j<=high)
                  //两个子文件非空时取其小者输出到R1[p]上
          R1[p++]=(R[i].key<=R[j].key)?R[i++]:R[j++];
     while(i<=m) //若第1个子文件非空，则复制剩余记录到R1中
          R1[p++]=R[i++];
     while(j<=high) //若第2个子文件非空，则复制剩余记录到R1中
          R1[p++]=R[j++];
     for(p=0,i=low;i<=high;p++,i++)
          R[i]=R1[p]; //归并完成后将结果复制回R[low..high]
}//Merge
```

Cooling

R[low..m]

[ ]

R[m+1..high]

[ ]

请输入待排序的记录数组R[low..high]
(数据之间用半角逗号隔开)

Clear Start

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。

```
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]
```

归并排序算法 10.14

```
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
```

- **算法评价**
  - **时间复杂度：** T(n)=O(nlogn)
    - 每趟两两归并需调用merge算法⌈n/2h⌉次（h为有序段长度）；
    - 整个归并要进行⌈$\log_2 n$⌉趟。
  - **空间复杂度：** S(n)=O(n)

10.6 基数排序

基数排序是一种借助“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。

多关键字的排序

链式基数排序

一、多关键字的排序

n 个记录的序列 $\{R_1, R_2, \cdots, R_n\}$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 $R_j(1 \le i < j \le n)$ 都满足下列(词典)有序关系：

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

- 多关键字排序

例　对52张扑克牌按以下次序排序：

两个关键字：花色（ ♣<♦<♥<♠ ）

面值（2<3<……<A）

并且“花色”地位高于“面值”

♣2<♣3<……<♣A<♦2<♦3<……<♦A<

♥2<♥3<……<♥A<♠2<♠3<……<♠A

1 多关键字排序方法

- **最高位优先法（MSD）：**

  先对最高位关键字K^0排序，将序列分成若干子序列，每个子序列有相同的K^0值；然后让每个子序列对次关键字K^1（如面值）排序，又分成若干更小的子序列；依次重复，直至就每个子序列对最低位关键字k^{d-1}排序；最后将所有子序列依次连接在一起成为一个有序序列。

- **最低位优先法(LSD)：**

  从最低位关键字k^{d-1}起进行排序，然后再对高一位的关键字排序，……依次重复，直至对最高位关键字K^0排序后，便成为一个有序序列。

待排序的原始序列：

278,109,063,930,589,184,505,269,008,083

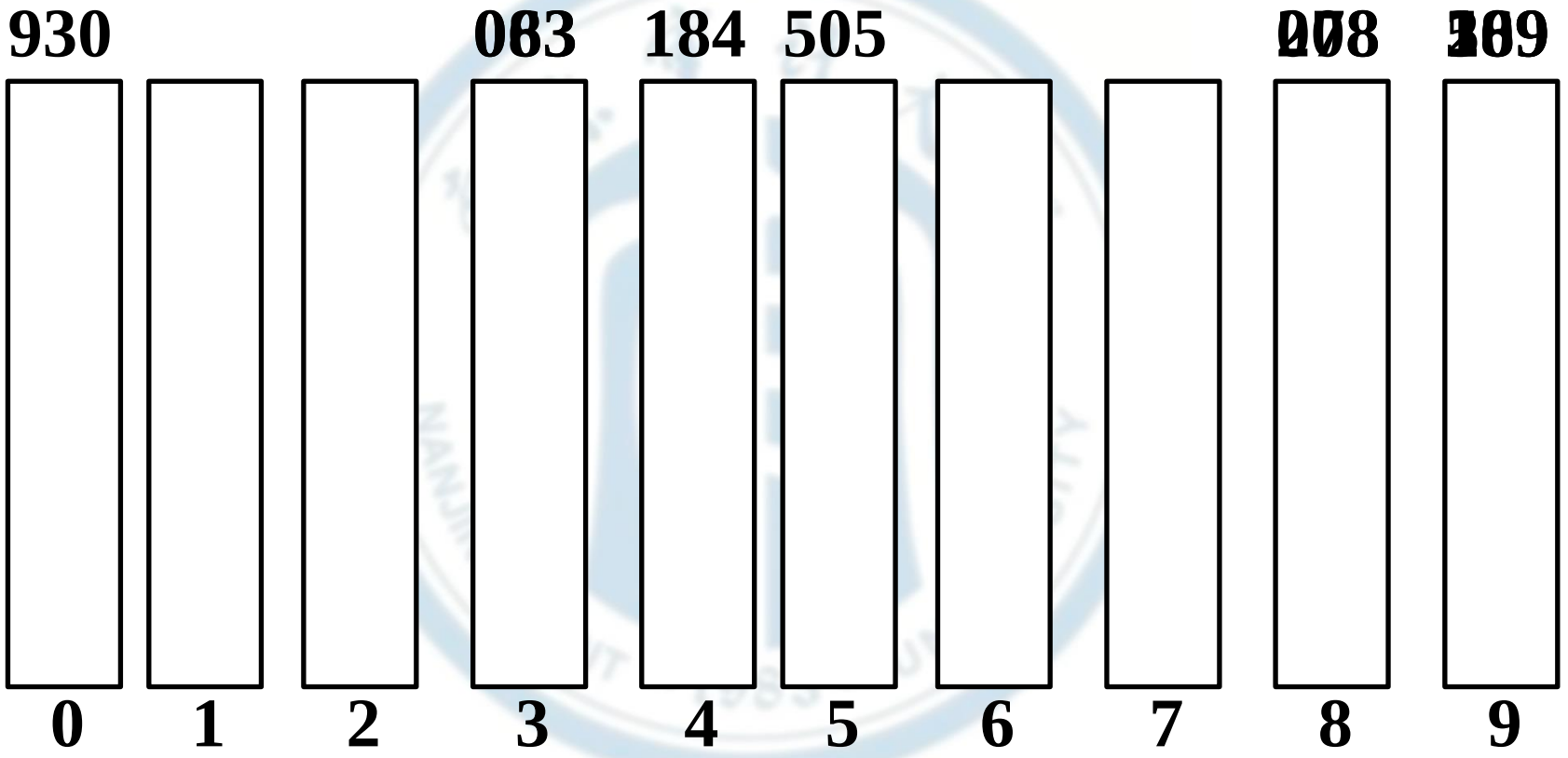
由于每个元素的每一位都是由“数字”组成，数字的范围是0~9，所以可以准备十个**队列**来盛放元素。

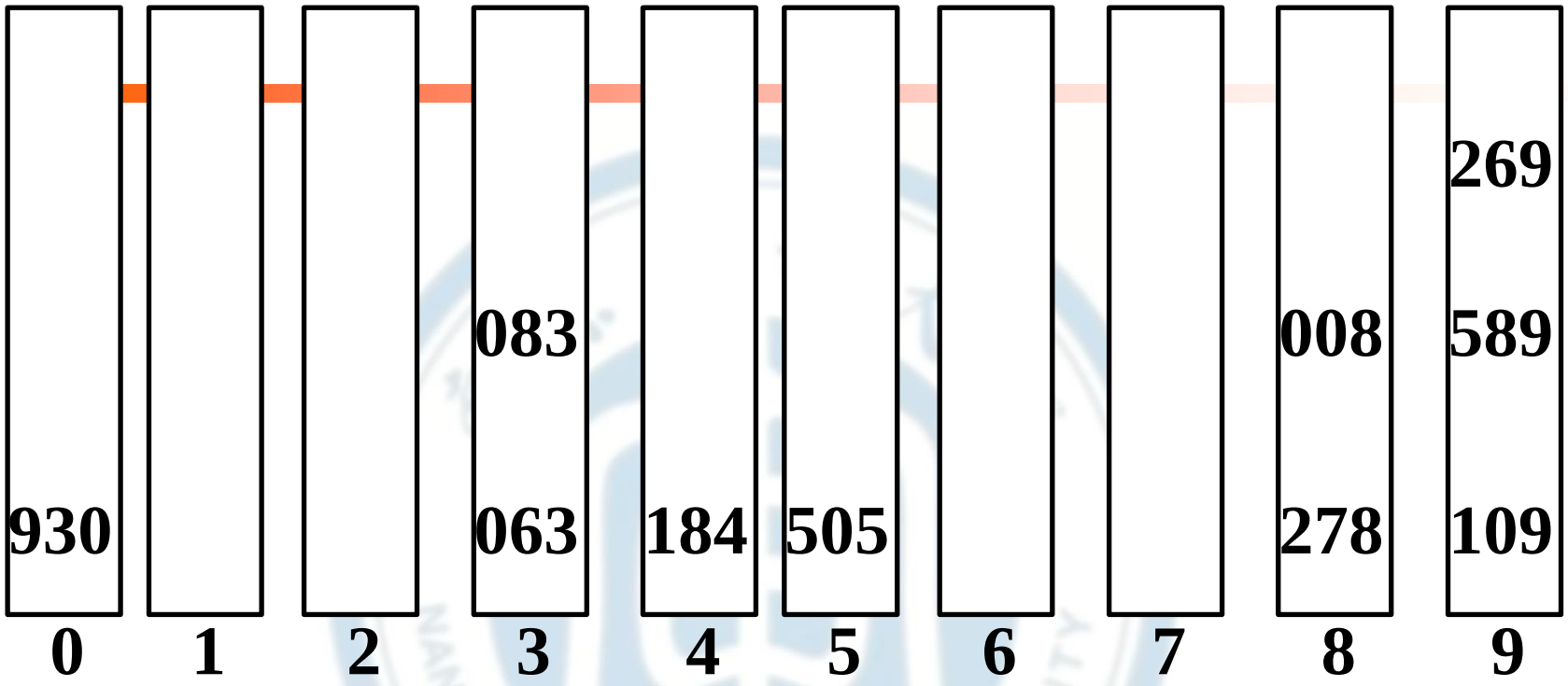
如果带排序的序列某一位非数字，如学号的首位是大写的英文字符，那么就需要**在排最高位时**，准备26个队列。

待排序的原始序列：

278,109,063,930,589,184,505,269,008,083

第一趟排序



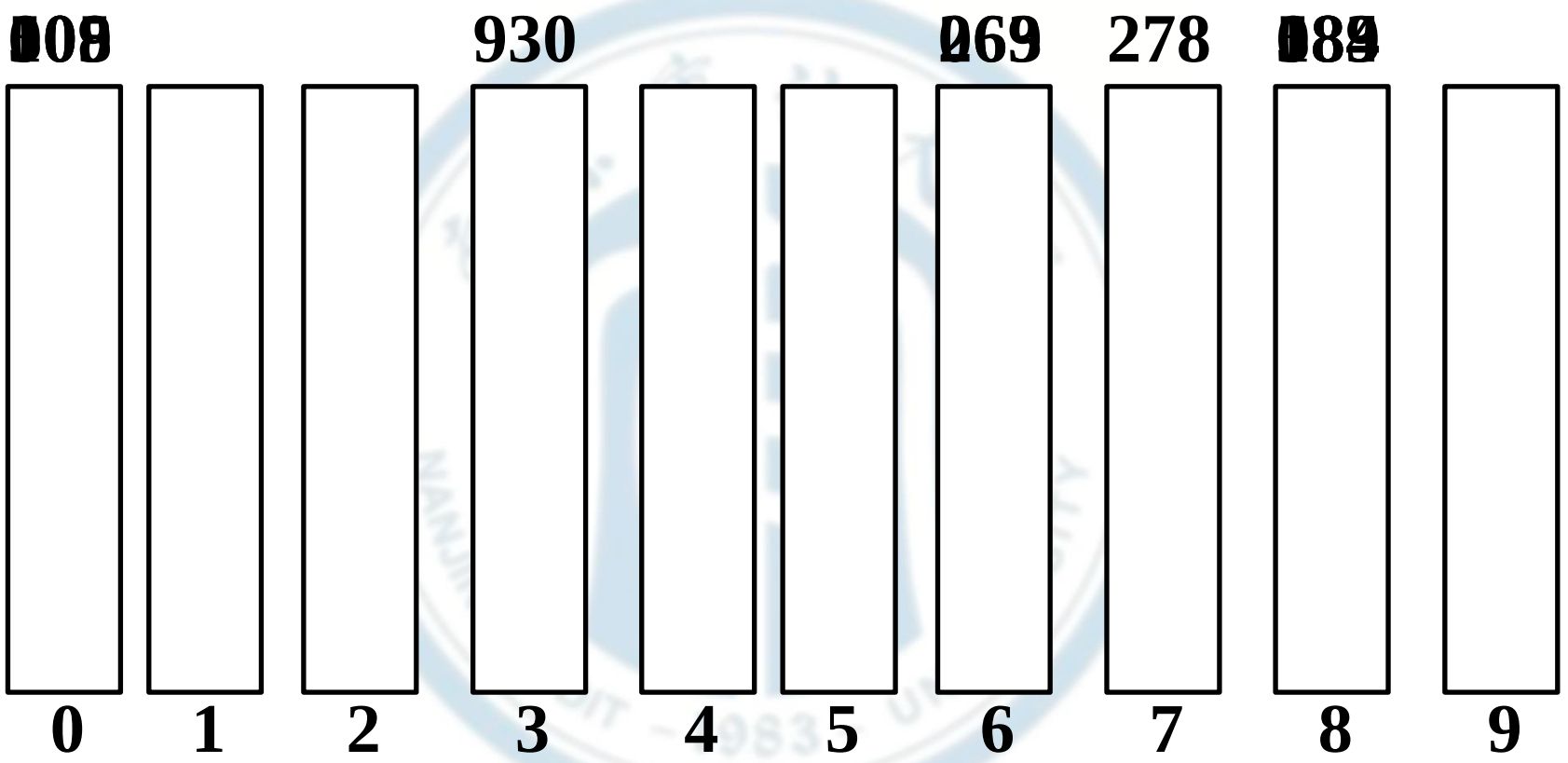


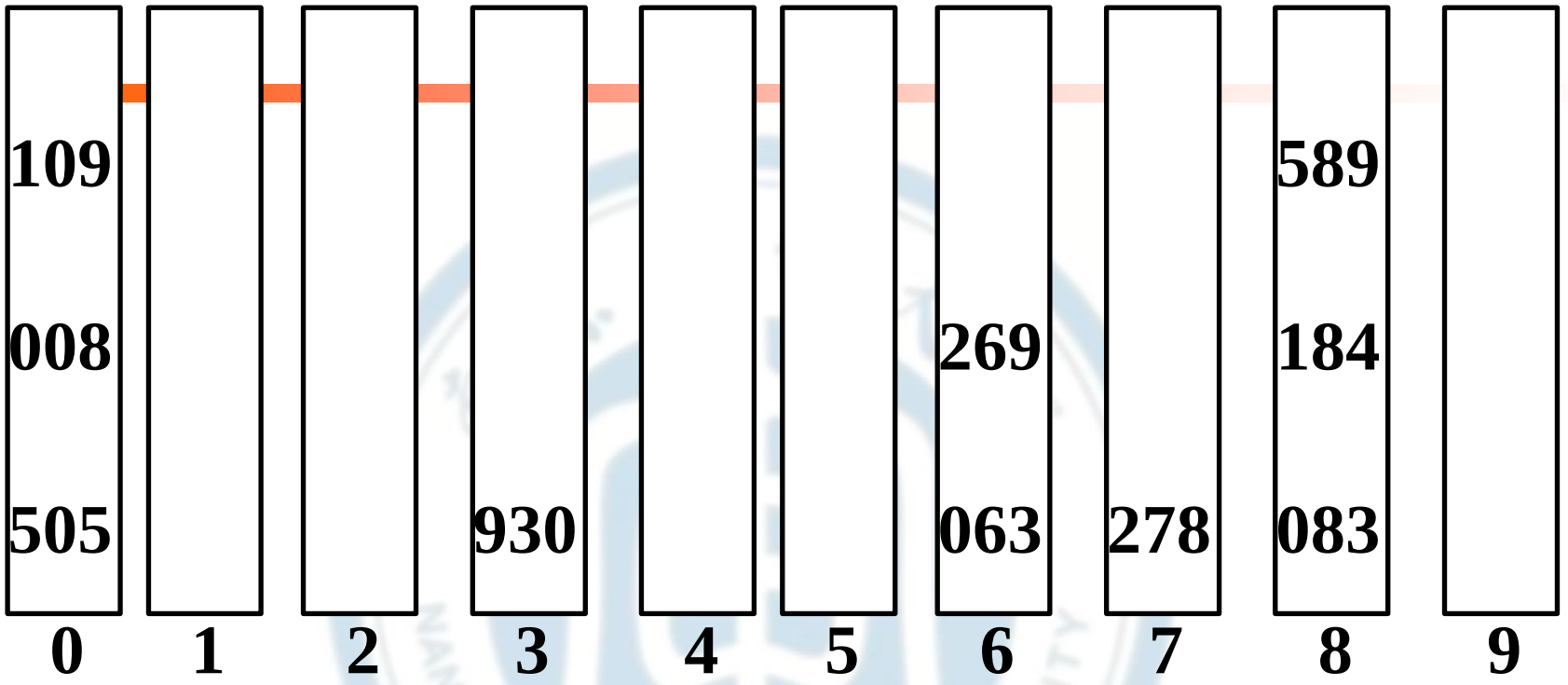
第一趟排序后的输出序列：

930, 063, 083, 184, 505, 278, 008,109, 589, 269

第二趟排序前序列：

930, 063, 083, 184, 505, 278, 008,109, 589, 269



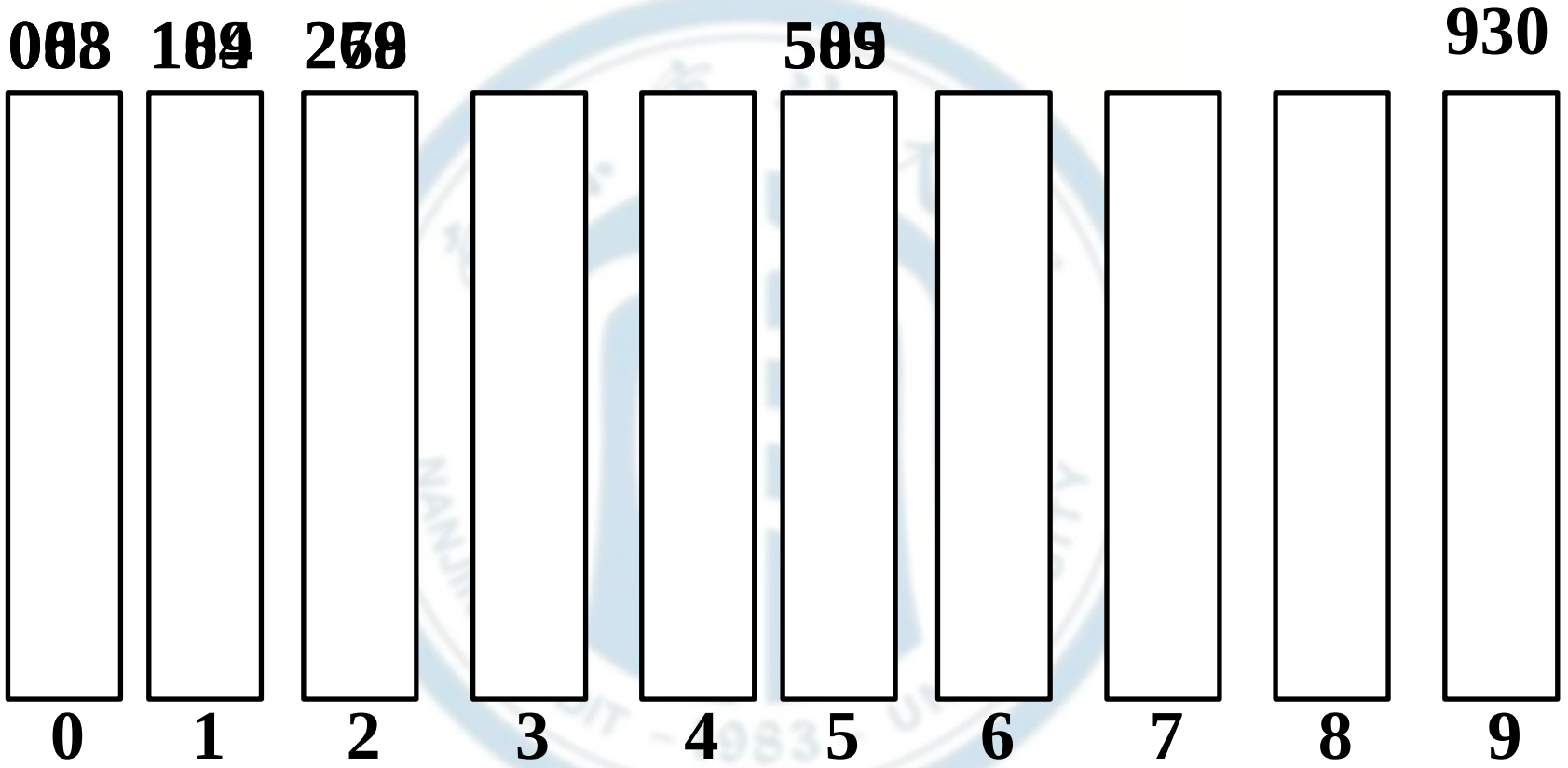


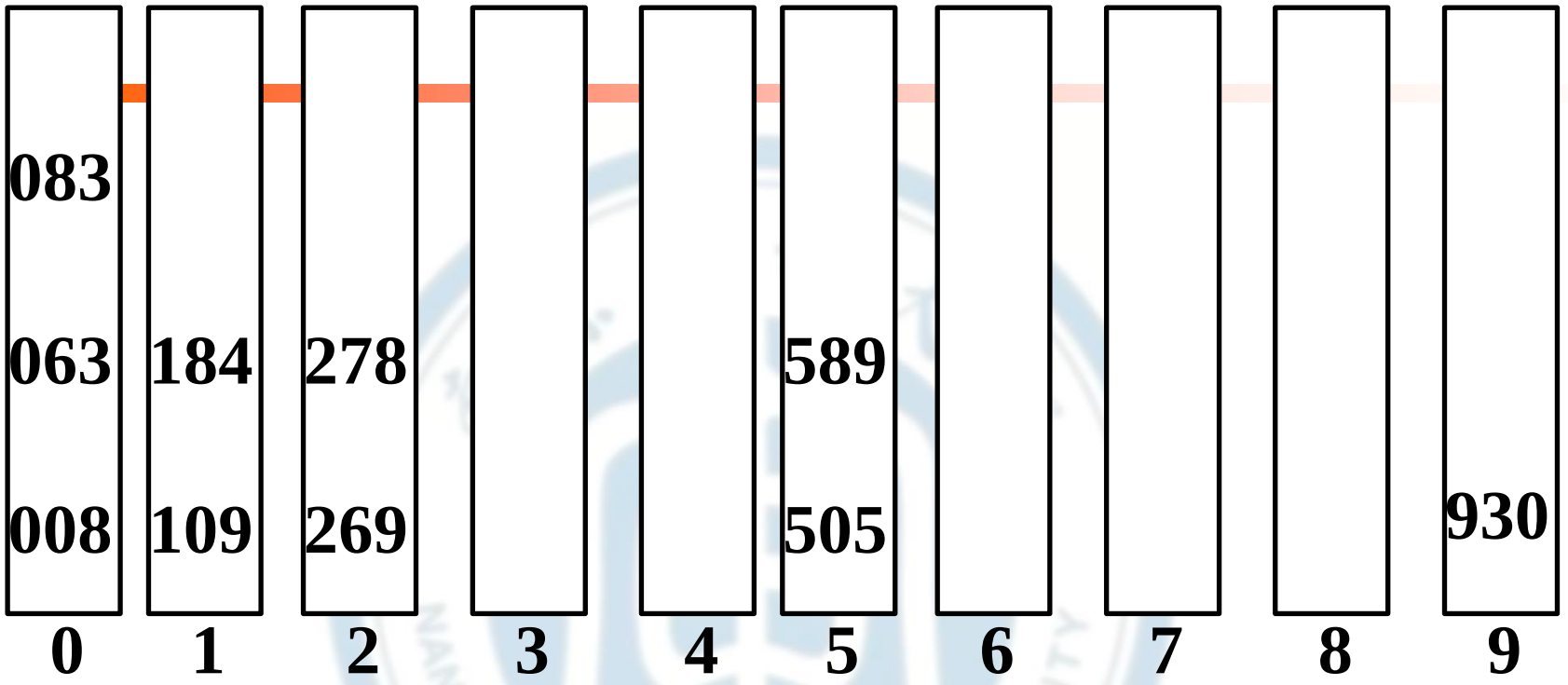
第二趟排序后的输出序列：

505, 008,109, 930, 063, 269,278, 083, 184, 589

第三趟排序前序列：

505, 008,109, 930, 063, 269,278, 083, 184, 589





第三趟排序后的输出序列：

008, 063, 083, 109, 184, 269, 278, 505, 589, 930

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

- MSD与LSD不同特点
  - 按MSD排序，必须将序列逐层分割成若干子序列，然后对各子序列分别排序。
  - 按LSD排序，**不必分成子序列，对每个关键字都是整个序列参加排序；**并且可不通过关键字比较，而通过若干次分配与收集实现排序。

基数排序(*RadixSort*)

Cooling

基数排序是箱排序的改进和推广。设文件中任一记录R[i]的关键字均由d个分量K_i^0 K_i^1 K_i^{d-1}构成，则基数排序的基本思想是：从高到低依次对K^j(j=d-1, d-2, . . . , 0)进行箱排序。在d趟箱排序中，所需的箱子数就是基数rd，这就是“基数排序”名称的由来。本例演示的基数为10，d为2。

请输入待排序的记录数组R，数据个数小于13，每个数据小于100。(数据之间用半角逗号隔开)

R[ ]

Clear Start

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。

例如：

p→369→367→167→239→237→138→230→139

进行第一次分配

f[0]→230←r[0]

f[7]→367→167→237←r[7]

f[8]→138←r[8]

f[9]→369→239→139←r[9]

进行第一次收集

p→230→367→167→237→138→368→239→139

p→230→367→167→237→138→368→239→139

进行第二次分配

f[3]→230→237→138→239→139←r[3]

f[6]→367→167→368←r[6]

进行第二次收集

p→230→237→138→239→139→367→167→368

p→230→237→138→239→139→367→167→368

进行第三次分配

f[1]→138→139→167← r[1]

f[2]→230→237→239←r[2]

f[3]→367→368←r[3]

进行第三次收集之后便得到记录的有序序列

p→138→139→167→230→237→239→367→368

- 算法评价

  - 时间复杂度：

    - 分配：T(n)=O(n)
    - 收集：T(n)=O(rd)

      T(n)=O(d(n+rd))

其中：n——记录数

d——关键字数

rd——关键字取值范围

  - 空间复杂度：S(n)=2rd个队列指针+n个指针域空间

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 $O(d(n+rd))$:

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

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. 简单选择排序、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。）

本章小结

一般来说，在选择排序方法时，可有下列几种选择：

- **若待排序的记录个数n值较小（例如n<30），则可选用插入排序法，但若记录所含数据项较多，所占存储量大时，应选用选择排序法。**
- **若待排序的记录个数n值较大时，应选用快速排序法。但若待排序记录关键字有“有序”倾向时，就慎用快速排序，而宁可选用堆排序或归并排序，而后两者的最大差别是所需辅助空间不等。**
- **快速排序和归并排序在n值较小时的性能不及直接插入排序，因此在实际应用时，可将它们和插入排序“混合”使用。如在快速排序划分子区间的长度小于某值时，转而调用直接插入排序；或者对待排记录序列先逐段进行直接插入排序，然后再利用“归并操作”进行两两归并直至整个序列有序为止。**

- 基数排序的时间复杂度为$O(d \times n)$，因此特别适合于待排记录数 n 值很大，而关键字“位数 d ”较小的情况。并且还可以调整“基数” (如将基数定为100或1000等）以减少基数排序的趟数d的值。
- 一般情况下，对单关键字进行排序时，所用的排序方法是否稳定无关紧要。但当按"最次位优先"进行多关键字排序时(除第一趟外)必须选用稳定的排序方法。