第十章 内部排序

10.1 概述

一. 排序

- **排序**（Sorting）：将一个数据元素（或记录）的任意序列，重新排列成一个按关键字有序的序列。
  - **正序**：数据记录是按照关键字非递减的有序排列。
  - **逆序**：数据记录是按照关键字非递增的有序排列。

10.1 概述

一. 排序

- 按照排序过程中所涉及的存储器的不同可分为：
  - **内部排序**：待排序序列完全存放在内存中，适合数量不大的数据元素的排序。
  - **外部排序**：是指待排序的数据元素非常多，以至于它们必须存储在外部存储器上，排序过程中需要在内外存之间多次交换数据才能进行。

10.1 概述

一. 排序

- 排序方法的稳定性

如果在记录序列中有两个记录r[i]和r[j]，它们的关键字相同（key[i] = key[j]），且在排序之前，记录r[i]位于r[j]前面，

- 如果在排序之后，记录r[i]仍在记录r[j]的前面，则称这个排序方法是稳定的，也就是说，对于两个关键字相等的记录，它们在序列中的相对次序，在排序前、后没有改变。
- 反之，若相对次序发生变化，称这个排序方法是不稳定的

例如，关键字序列 (56, 34, 46, 23, 66, 18, 82, 46)

若排序后得到结果 (18, 23, 34, 46, 46, 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, 46, 46, 56, 66, 82)，则称该排序方法是不稳定的。

10.1 概述

一. 排序

- 排序常用的数据结构是顺序表
- 排序基本操作
  - 关键字比较：比较两个关键字的大小
  - 记录移动：将记录从一个位置移动至另一个位置

10.1 概述

一. 排序

排序算法的性能评价

- **时间复杂度：**是衡量排序算法好坏的最重要的标志。高效率的排序算法应该是具有尽可能少的关键字比较次数（KCN）和尽可能少的记录移动次数（RMN）。

- **空间复杂度：**辅助存储空间是除了存放待排序所占用的存储空间之外，执行算法所需要的其他存储空间。

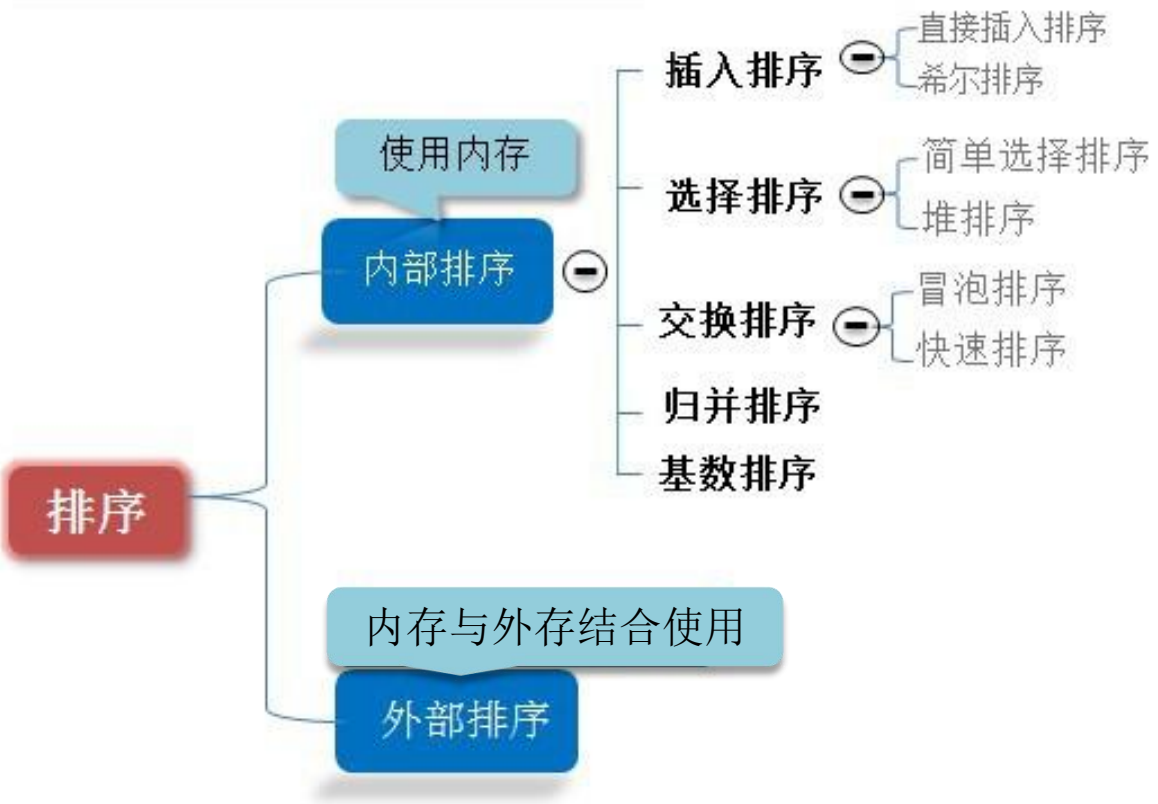
当排序算法中使用的额外内存空间与要排序数据元素的个数n无关时，其空间复杂度为O(1)，大多数排序算法的空间复杂度是O(1)，也有一些时间复杂度性能好的排序算法，其空间复杂度会达到O(n)。

- **稳定性：**稳定的排序算法是应用问题所希望的。

10.1 概述

一. 排序

- **内部排序**：插入排序、交换排序、选择排序、归并排序和基数排序。



10.2 插入排序

一. 直接插入排序

- 直接插入排序是最简单的排序方法
- 算法设计：

将待排序的记录R_i，插入到已排好序的记录表R_1，R_2，…，R_{i-1}中，得到一个新的、记录数增加1的有序表。 直到所有的记录都插入完为止。

10.2 插入排序

一. 直接插入排序

- 算法流程
  - 当插入第i(i≥1)个元素时，前面的r[0]，r[1]，...，r[i-1]已经排好序。
  - 用r[i]的关键字与r[i-1]，r[i-2]，...的关键字顺序进行比较(和顺序查找类似)，如果r[i]小于r[x]，则将r[x]向后移动(插入位置后的记录向后顺移)
  - 找到插入位置，将r[i]插入

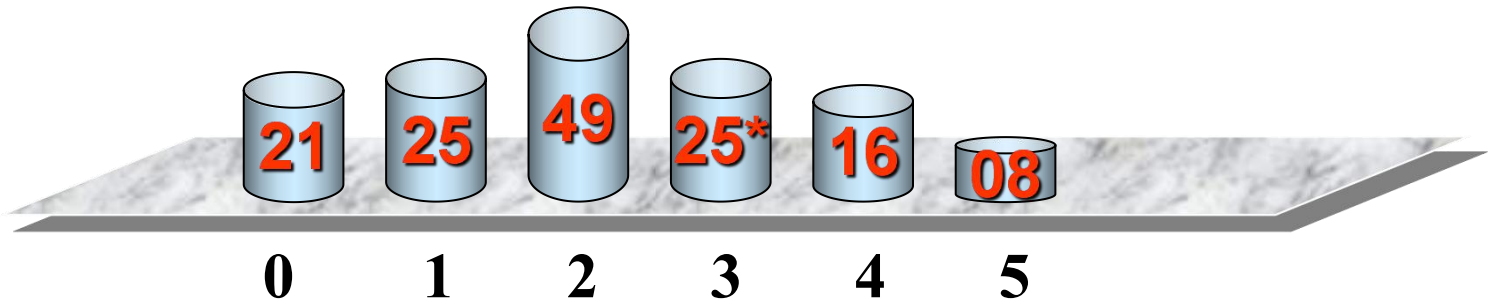
10.2 插入排序

一. 直接插入排序

- 举例

已知待排序的一组记录的初始序列为：

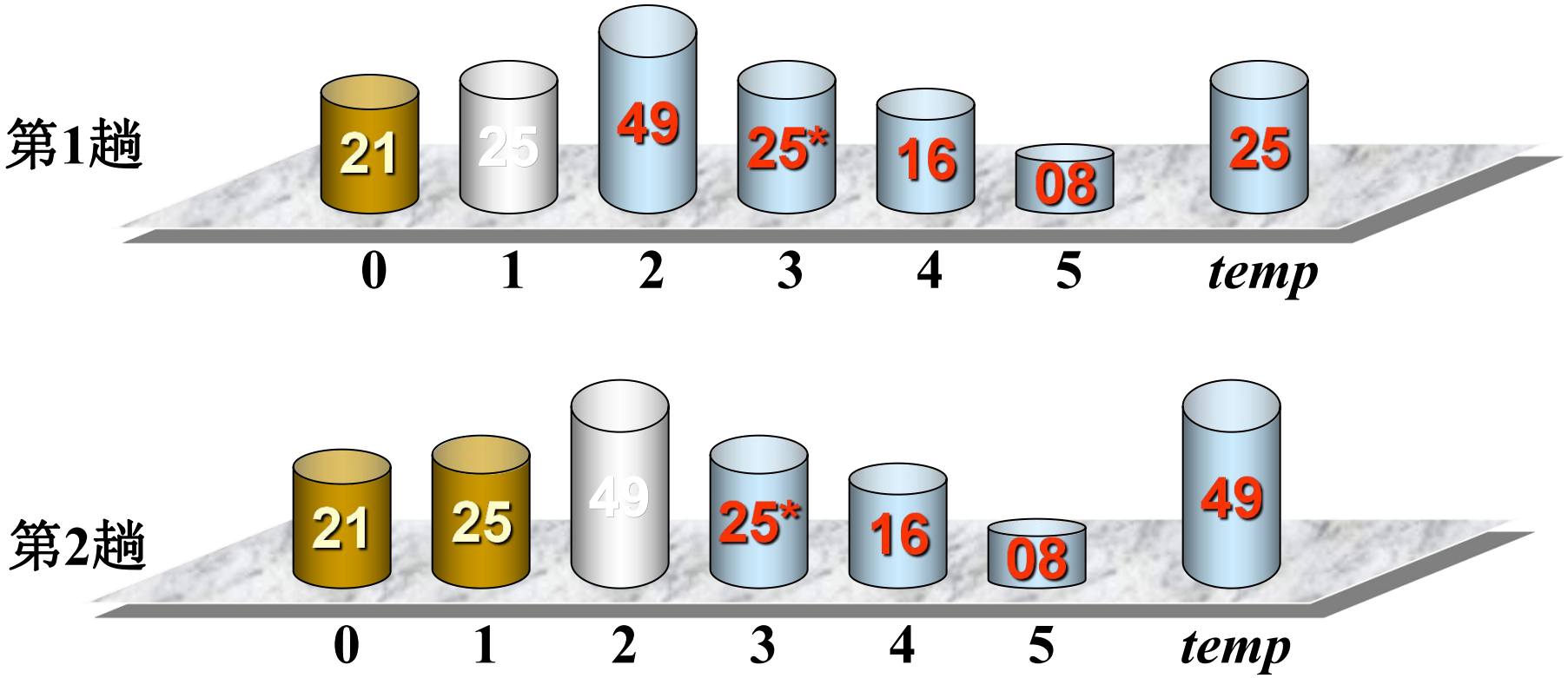
21，25，49，25*，16，08



10.2 插入排序

一. 直接插入排序

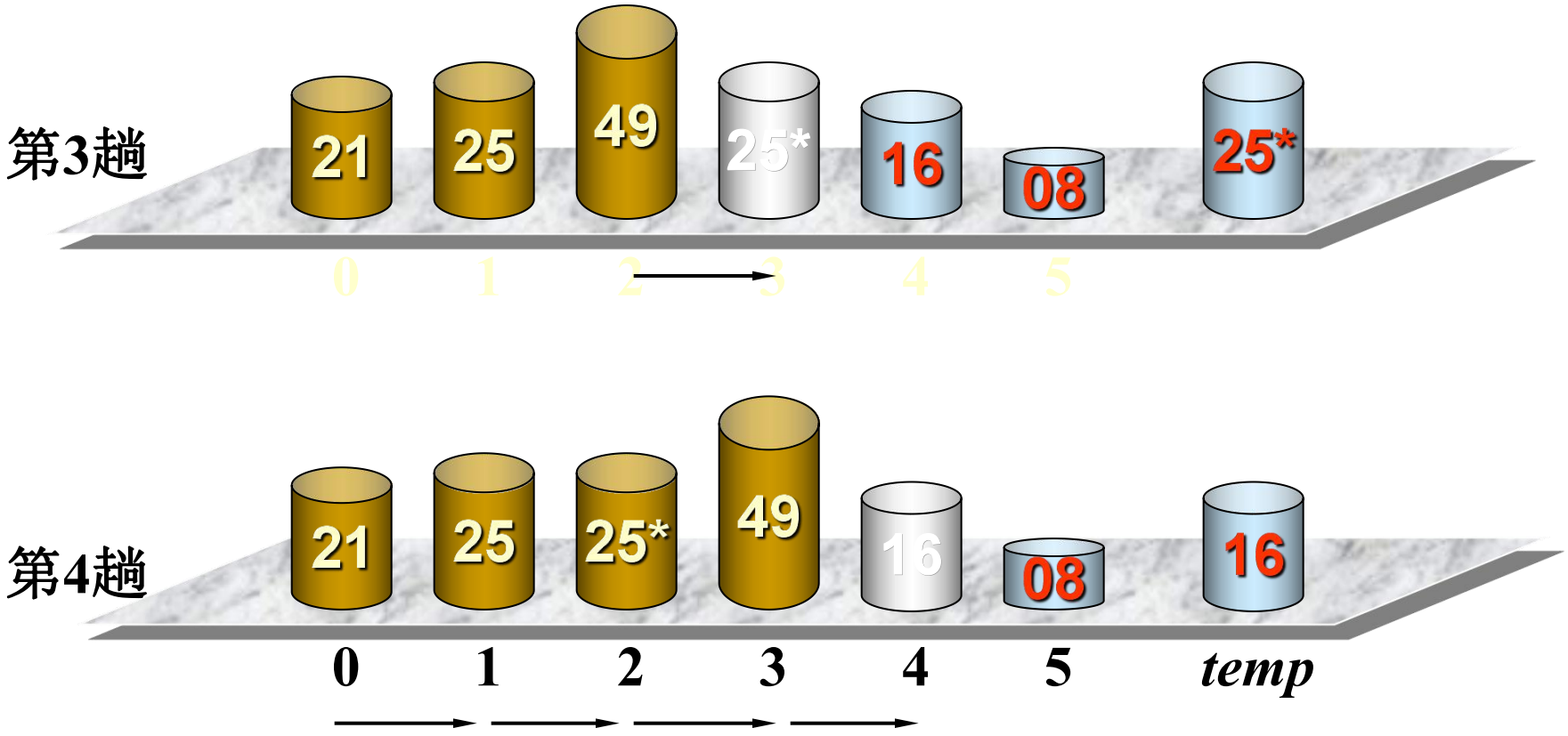
- 插入过程



10.2 插入排序

一. 直接插入排序

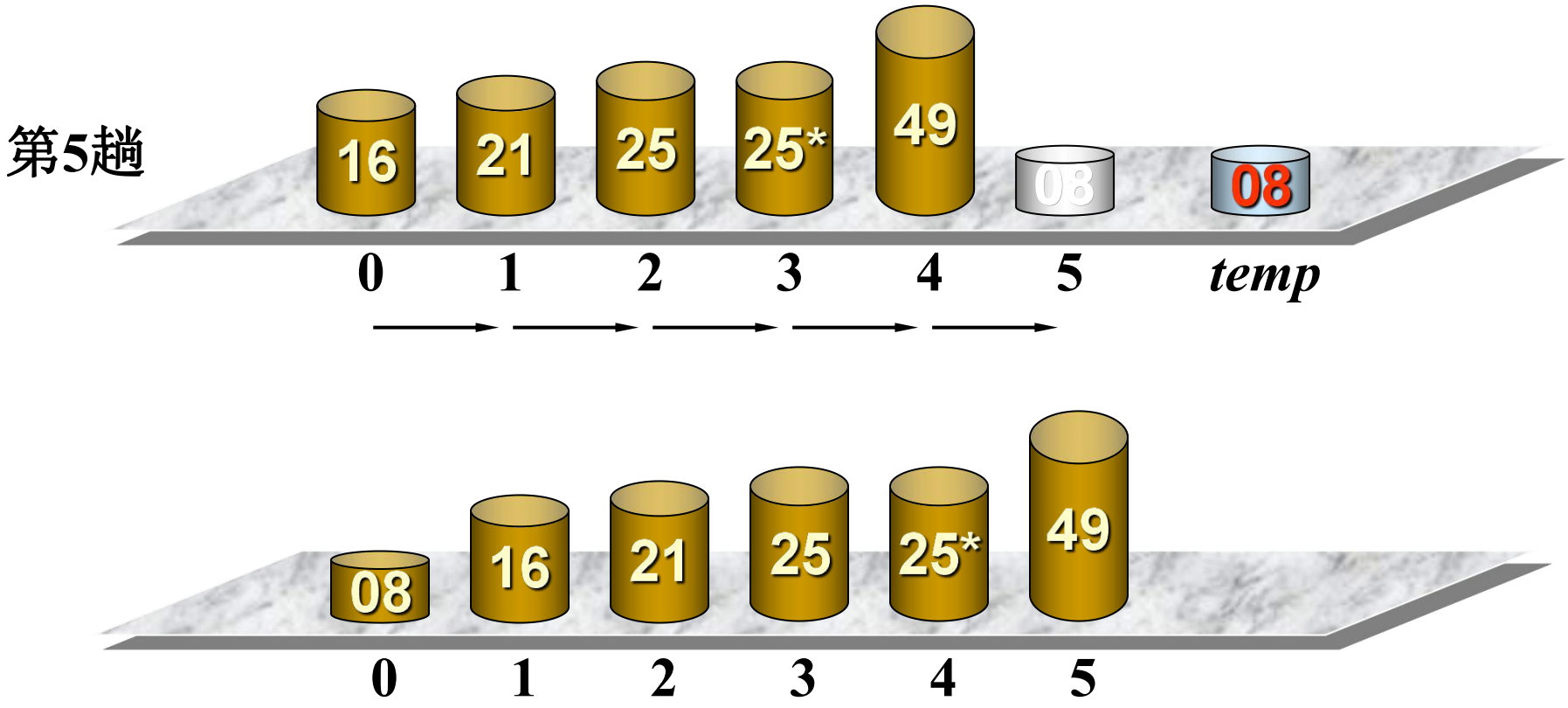
- 插入过程



10.2 插入排序

一. 直接插入排序

- 插入过程



10.2 插入排序

一. 直接插入排序

- 算法实现

```
void InsertSort (int k[ ], int n )
{
     for ( i = 1; i < n; i++ ) //n-1次搜索
     { temp = k[i];
       for ( j = i;  j > 0; j-- )//在有序表中从后向前依次比较
       {
         if ( temp < k[j-1] )
              k[j] = k[j-1];    //向后移动，空出插入位置
         else break;
       }
       k[j] = temp;   //确定插入
    }
}
```

10.2 插入排序

一. 直接插入排序

- 算法分析：
  - 关键字比较次数和记录移动次数与记录的初始排列有关
  - 最好的情况（关键字在记录序列中是正序）

    比较的次数：$\sum_{i=1}^{n-1} 1 = n - 1$　　移动的次数： 0

    最好 O(n)

  - 最坏的情况（关键字在记录序列中是逆序）：

    比较的次数：$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$

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

    最坏 $O(n^2)$

- 平均时间复杂度 $O(n^2)$

10.2 插入排序

一. 直接插入排序

- 直接插入排序的性能评价
  - 时间复杂度为$O(n^2)$，空间复杂度为$O(1)$
  - 是一种稳定的排序方法
  - 最大的优点是简单，在记录数较少时，是比较好的办法

练习

一. 已知数列为4、8、5*、7、5、3、2、9、6，采用直接插入排序，对数列进行从小到大的排序，要求写出排序过程。

10.2 插入排序

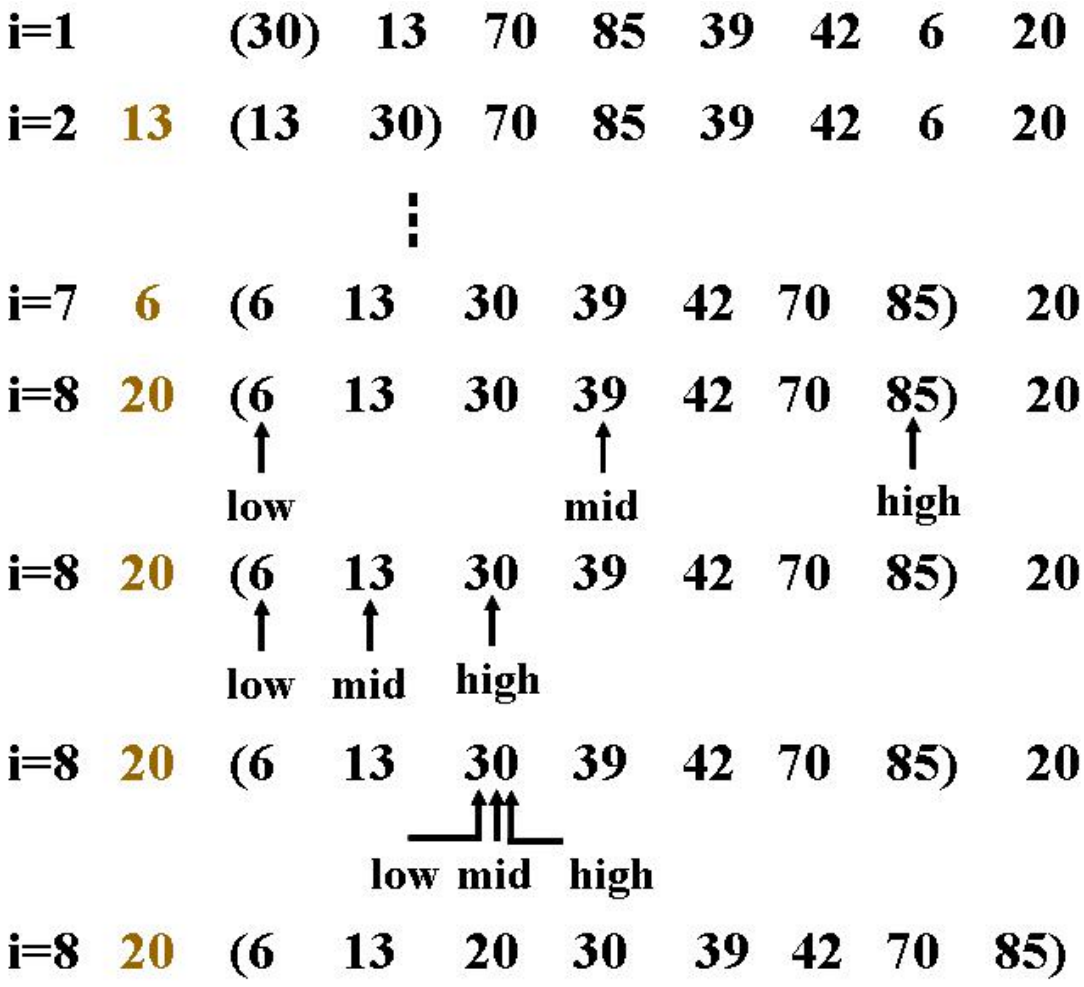
二. 折半插入排序

- 折半插入排序的基本操作是在一个有序表中进行查找和插入
  - 在查找记录插入位置时，采用折半查找算法
  - 折半查找可以减少关键字间的比较次数，比顺序查找快

10.2 插入排序

二. 折半插入排序

- 示例，设有一组关键字 30, 13, 70, 85, 39, 42, 6, 20



10.2 插入排序

二. 折半插入排序

- 折半插入排序算法实现

```
void BInsertSort(SqList &L) {   // 对顺序表L作折半插入排序
  int i,j,high,low,m;
  for (i=2; i<=L.length; ++i)
   {
     L.r[0] = L.r[i];          // 将L.r[i]暂存到L.r[0]
     low = 1;   high = i-1;
     while ( low<=high )       // 在r[low..high]中折半查找有序插入的位置
     {
       m = (low+high)/2;                                  // 折半
       if ( LT(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
     }//end while
     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
   }//end for
}
```

10.2 插入排序

二. 折半插入排序

- 折半插入排序在查找比较上性能比直接插入排序好，但需要移动的记录次数与直接插入排序相同，所以折半插入排序的时间复杂度为$O(n^2)$
- 折半插入排序的折半插入排序是一种稳定的排序方法

练习

一. 已知数列为4、8、5*、7、5、3、2、9、6，采用折半插入排序，对数列进行从小到大的排序，要求写出排序过程。

10.2 插入排序

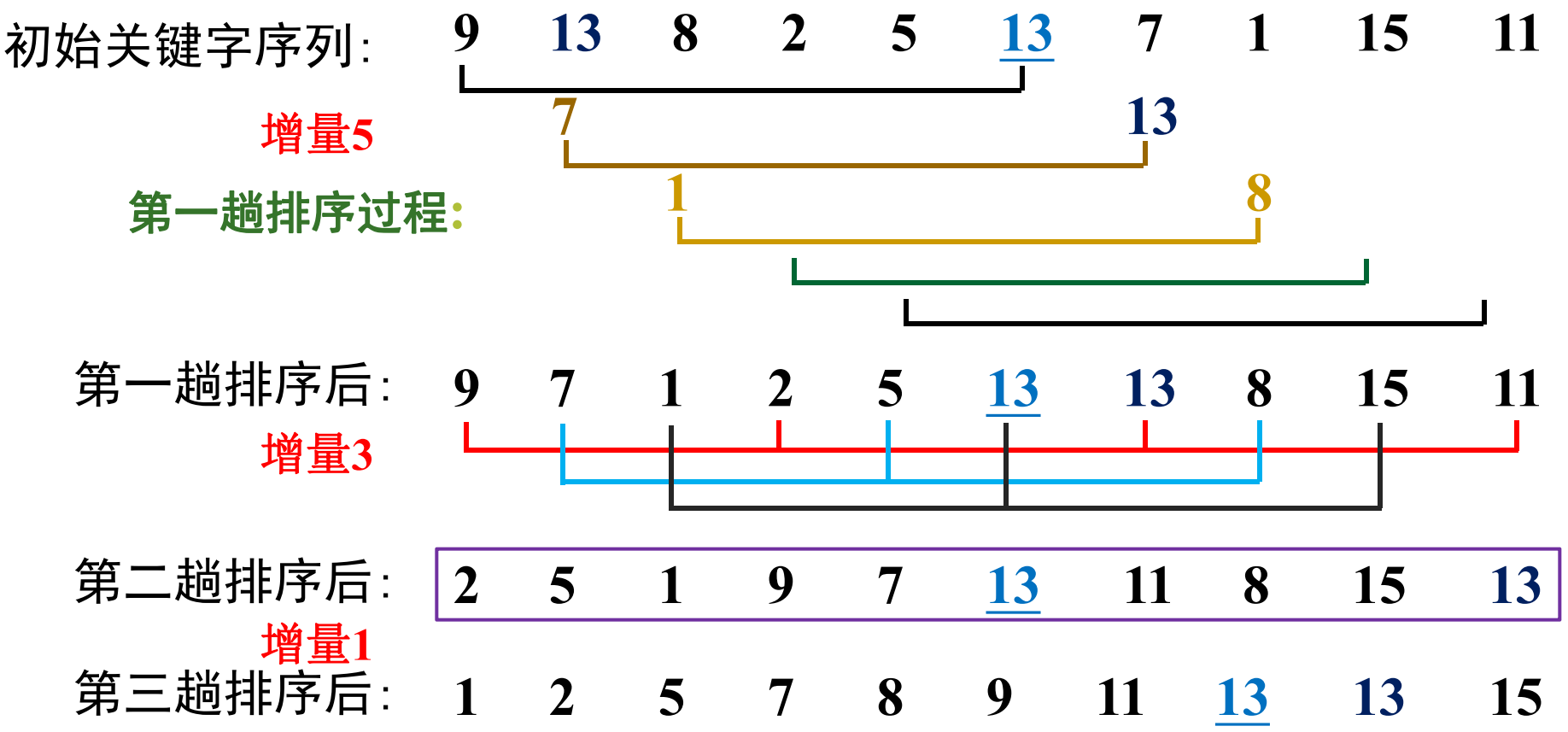
三. 希尔排序

- 希尔排序又称为“缩小增量排序”，是先将待排序列分成若干子序列（由相邻某个“增量”的记录组成的）分别进行插入排序，待整个序列基本有序时，再对全体记录进行一次直接插入排序。
- 特点：子序列的构成不是简单的“逐段分割”，而是将相邻某个“增量”的记录组成一个子序列。
- 优势：利用了插入排序的简单性，而且克服了插入排序每次只能交换相邻两个记录的缺点。

10.2 插入排序

三. 希尔排序

- 举例



10.2 插入排序

三. 希尔排序

- 算法过程：
  - 首先取一个整数 gap（gap< n）作为增量，将全部记录分为 gap 个子序列，所有距离为 gap 的记录放在同一个子序列中，在每一个子序列中分别进行直接插入排序，这样一次分组和排序过程称为一趟希尔排序。
  - 然后缩小增量gap，重复上述的子序列划分和排序工作，直到最后当 gap = 1时，将所有记录放在同一个序列中排序为止。

    存在一个递减gap序列， $gap_k > gap_{k-1} > \ldots > gap_1=1$

10.2 插入排序

三. 希尔排序

- 算法实现

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

10.2 插入排序

三. 希尔排序

- 算法实现

```
void ShellInsert( SqList &L, int dk)
{ // 1. 前后记录位置的增量是dk，而不是1
  // 2. r[0]只是暂存单元，当j<=0时，插入位置已找到
  int i,j;
  for (i=dk+1; i<=L.length; ++i)
 {
    if ( LT(L.r[i].key, L.r[i-dk].key) ) // 需要做交换
    {  L.r[0] = L.r[i];                    // 暂存在L.r[0]
       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
    }
}
```

10.2 插入排序

三. 希尔排序

- 算法实现：
  - 增量 gap 序列的取法

一般是没有除1以外的公因子的系列数值，并且最后一个增量值必须为1。

Shell 提出取 $gap_k = \lfloor n/2 \rfloor$

$gap_{k-1} = \lfloor gap_k / 2 \rfloor$

……

$gap_1 = 1$

10.2 插入排序

三. 希尔排序

- 算法分析：
  - 开始时 gap 的值较大，子序列中的记录较少，排序速度较快；gap 值逐渐变小，子序列中记录个数逐渐变多，但由于关键字较小的记录已跳跃式前移，在进行最后一趟增量为1的排序时，序列已基本有序，所以排序速度仍然很快，是直接插入排序算法的一种更高效的改进版本。

例如：有10元素要排序。

直接插入排序：

时间约为 $10^2=100$

希尔排序：

gap=5 ：分为5 组，时间约为$5\times2^2=20$

gap=2 ：分为2 组，时间约为$2\times5^2=50$

gap=1 ：分为1 组，基本有序，时间约为10

总时间约为 20+50+10=80

10.2 插入排序

三. 希尔排序

- 评价：
  - 希尔排序的分析比较复杂，其执行时间依赖于增量序列
  - 是直接插入排序算法的一种更高效的改进版本
  - 希尔排序是一种不稳定的排序方法

练习

一. 已知数列为4、8、5*、7、5、3、2、9、6，采用希尔排序（增量用n重复整除2直到1），对数列进行从小到大的排序，要求写出排序过程。

10.3 快速排序

一. 起泡排序

- 算法设计

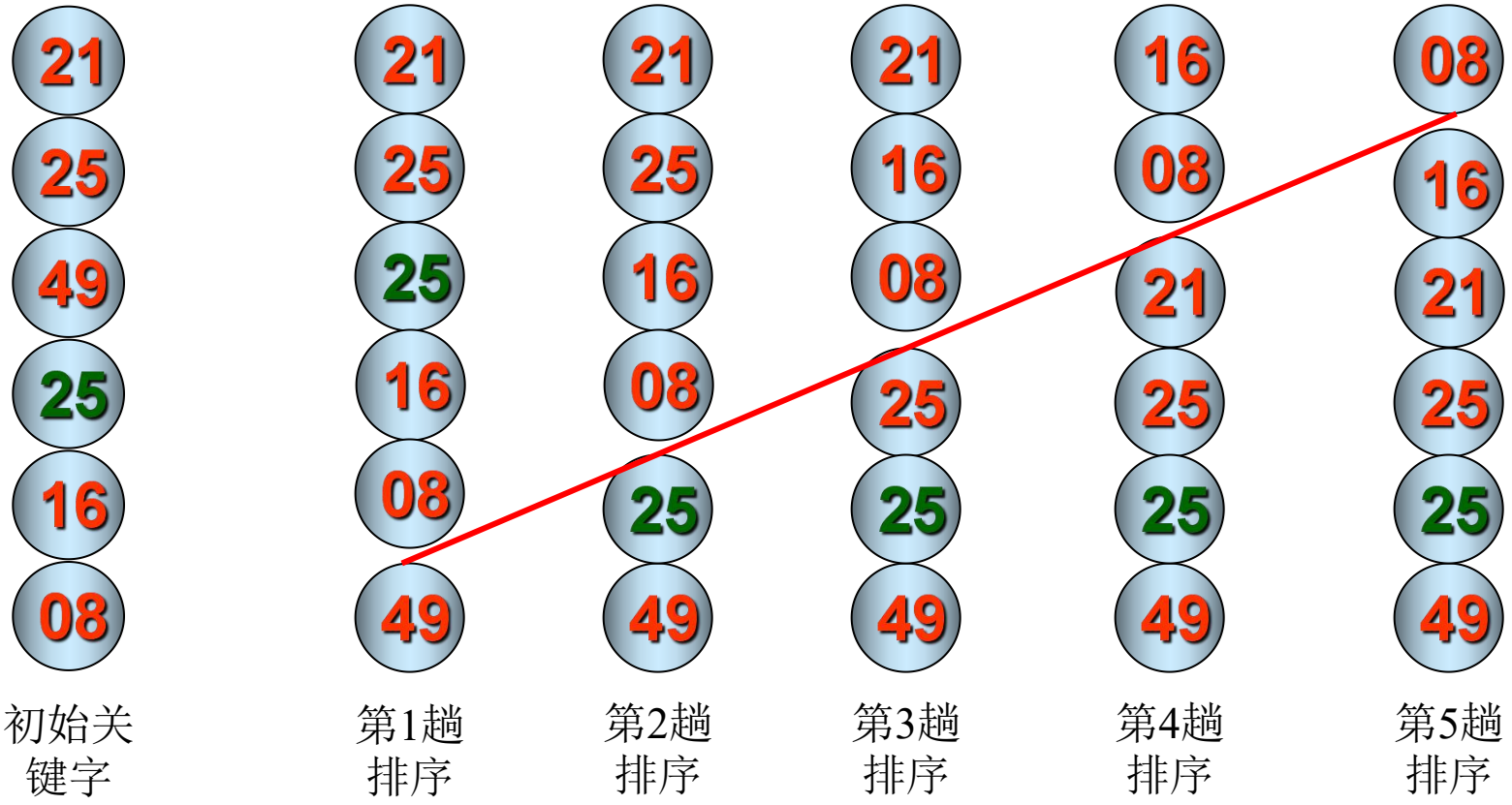
是一类基于交换的排序，系统地交换逆序对中的记录，直到不再有这样的逆序对为止。其中最基本的是起泡排序(冒泡排序，**Bubble Sort**)

- 设待排序记录序列中的记录个数为n
- 一般地，第i趟起泡排序对前n−i+1个记录排序
- 依次比较相邻两个记录的关键字，如果发生逆序，则交换之
- 其结果是这n−i+1个记录中，关键字最大的记录被交换到第n−i+1的位置上，称为一趟起泡排序，最多作n−1趟排序。

10.3 快速排序

一. 起泡排序

- 举例



10.3 快速排序

一. 起泡排序

- 算法过程
  - i=1，为第一趟排序，关键字最大的记录将被交换到最后一个位置
  - i=2，为第二趟排序，关键字次大的记录将被交换到第n-1个位置
  - 依此类推……
  - 关键字小的记录不断上浮(起泡)，关键字大的记录不断下沉(每趟排序中“最大”的记录一直沉到底)

10.2 插入排序

一. 起泡排序

- 算法实现

```
void Bubble_Sort(Sqlist *L )
{
      for ( j=0; j<L->length; j++ )      //  共有n-1趟排序
      {
          flag=FALSE ;
          for ( k=1;  k < L->length-j;  k++ ) //  一趟起泡
          {    if (  LT(L->R[k+1].key, L->R[k].key ) )
               {    L->R[0] = L->R[k] ;
                    L->R[k] = L->R[k+1] ;
                    L->R[k+1] = L->R[0] ;
                    flag= TRUE;   //  发生交换
               }
          }
          if  ( flag==FALSE )  break ;  //  全程无交换
      }
}
```

10.3 快速排序

一. 起泡排序

- 性能分析
  - 最好情况：在记录的初始序列正序时，此算法只执行一趟起泡，做n-1次关键字比较，不移动记录
  - 最坏情况：记录的初始序列逆序时，要执行n-1趟冒泡，第i趟做n-i次关键字比较，执行n-i次记录交换，比较次数KCN和交换次数RCN共计：

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

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

- 起泡排序的时间复杂度为$O(n^2)$
- 起泡排序是一种稳定的排序方法

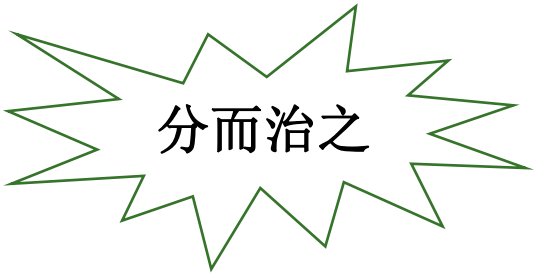
练习

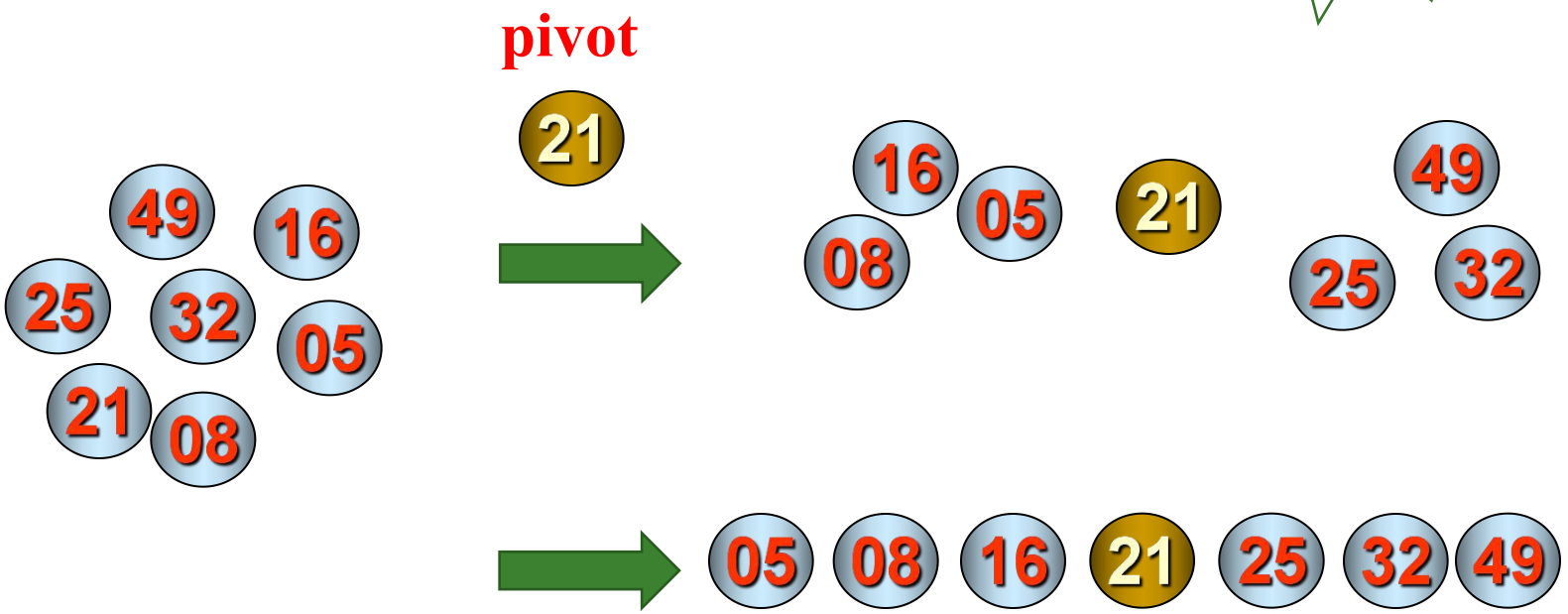
一. 已知数列为 4、8、5*、7、5、3、2、9、6 ，采用起泡排序，对数列进行从小到大的排序，要求写出排序过程。

10.3 快速排序

二. 快速排序

- 举例





10.3 快速排序

二. 快速排序

- 排序思想：

任取待排序记录序列中的某个记录作为基准(也称为枢轴、基准或支点记录，pivot)，通过一趟排序，将待排序记录以基准为界分割成独立的两部分，其中基准前面的记录的关键字均比基准的关键字小，而其后记录的关键字均比基准关键字大，再分别对这两部分记录采用同样方法分别进行下一趟排序，以达到整个序列有序。

10.3 快速排序

二. 快速排序

- 算法设计：
  - 任取待排序记录序列中的某个记录(例如取第一个记录)作为基准，按照该记录的关键字大小，将整个记录序列划分为左右两个子序列：
  - 左侧子序列中所有记录的关键字都小于或等于基准记录的关键字
  - 右侧子序列中所有记录的关键字都大于基准记录的关键字
  - 基准记录则排在这两个子序列中间(这也是该记录最终应放置的位置)
  - 然后分别对这两个子序列重复施行上述方法，直到所有的记录有序为止。

10.3 快速排序

二. 快速排序

- 算法设计：
  - 一趟快速排序

从序列的两端交替扫描各个记录，将关键字小于基准关键字的记录依次放置到序列的前边；而将关键字大于基准关键字的记录从序列的最后端起，依次放置到序列的后边，直到扫描完所有的记录。

设置指针low，high，初值分别为第1个和最后一个记录的位置。

10.3 快速排序

二. 快速排序

- 举例

初始关键字，比较

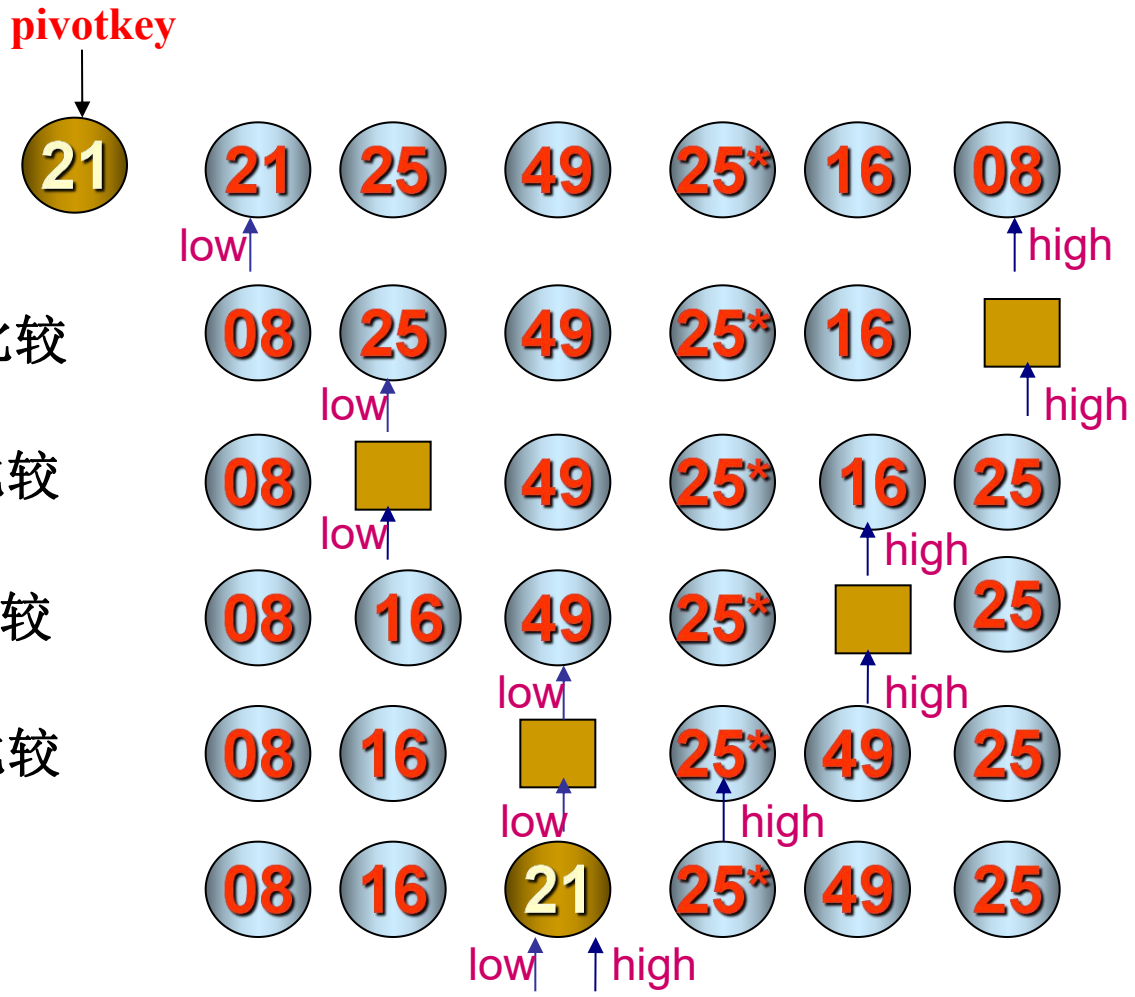
一次交换，low+1，比较

二次交换，high-1，比较

三次交换，low+1，比较

四次交换，high-1，比较

不交换，high-1=low，
完成一趟排序



10.3 快速排序

二. 快速排序

- 举例

完成一趟排序 

分别进行快速排序 

有序序列 

10.3 快速排序

二. 快速排序

- 算法过程：
  - 取序列第一个记录为枢轴记录，其关键字为Pivotkey
  - 指针low指向序列第1个记录位置
  - 指针high指向序列最后一个记录位置
  - 一趟排序(某个子序列)过程

    1. 从high指向的记录开始,向前找到第一个关键字的值小于Pivotkey的记录,将其放到low指向的位置,low+1
    2. 从low指向的记录开始,向后找到第一个关键字的值大于Pivotkey的记录,将其放到high指向的位置,high-1
    3. 重复1,2，直到low=high，将枢轴记录放在low(high)指向的位置

  - 对枢轴记录前后两个子序列执行相同的操作，直到每个子序列都只有一个记录为止

10.3 快速排序

二. 快速排序

- 程序实现

```
//算法10.7，对顺序表L中的子序列L.r[low..high]进行快速排序
void QSort(SqList &L, int low, int high)
{   int pivotloc;
    if (low < high)                  // 长度大于1
    {
        pivotloc = Partition(L, low, high);  // 将L.r[low..high]一分为二，
pivotloc是枢轴位置
        QSort( L, low, pivotloc-1 );      // 对低子表递归排序
        QSort( L, pivotloc+1, high );     // 对高子表递归排序
    }
} // QSort
```

10.3 快速排序

二. 快速排序

- 算法实现

```
/* 算法10.6(a) */
// 交换顺序表L中子序列L.r[low..high]的记录，使枢轴记录到位，
// 并返回其所在位置，此时，在它之前（后）的记录均不大（小）于它
int Partition( SqList &L, int low, int high )
{
  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
  {
    // 将比枢轴记录小的记录交换到低端
    while ( low<high && L.r[high].key >= pivotkey ) --high;
    swap（ L.r[low] ， L.r[high]）;
    // 将比枢轴记录大的记录交换到高端
    while ( low<high && L.r[low].key <= pivotkey ) ++low;
    swap（ L.r[low] ， L.r[high]）;
  }
  return low;                  // 返回枢轴所在位置
}
```

10.3 快速排序

二. 快速排序

- 算法实现

```
int Partition( SqList &L, int low, int high )
{
  temp = 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
  {
      // 将比枢轴记录小的记录移到低端
      while (low<high && L.r[high].key>=pivotkey)   --high;
      L.r[low] = L.r[high];
      // 将比枢轴记录大的记录移到高端
      while (low<high && L.r[low].key<=pivotkey)    ++low;
      L.r[high] = L.r[low];
  }
  L.r[low] = temp;              // 枢轴记录到位
  return low;                   // 返回枢轴位置
}
```

10.3 快速排序

二. 快速排序

算法分析：

- 最好情形，在n个元素的序列中，对一个枢轴记录定位所需时间为 O(n)。若设 T(n) 是对 n 个元素的序列进行排序所需的时间，而且每次对一个枢轴记录正确定位后，正好把序列划分为长度相等的两个子序列，此时，总的计算时间为：

$$
\begin{aligned}
T(n) &\le cn + 2T(n/2) \qquad \text{// c 是一个常数} \\
&\le cn + 2(cn/2 + 2T(n/4)) = 2cn + 4T(n/4) \\
&\le 2cn + 4(cn/4 + 2T(n/8)) = 3cn + 8T(n/8) \\
&\quad \cdots\cdots \\
&\le cn\log_2 n + nT(1) = O(n\log_2 n)
\end{aligned}
$$

时间复杂度为 $O(n\log_2 n)$

10.3 快速排序

二. 快速排序

算法分析：

- 最坏情况下，即待排序记录序列已经按其关键字从小到大排好序，
  - 每次划分只得到一个比上一次少一个记录的子序列
  - 必须经过n-1 趟才能把所有记录定位，而且第 i 趟需要经过 n-i 次关键字比较才能找到第 i 个记录的安放位置，总的关键字比较次数将达到

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

时间复杂度为$O(n^2)$

10.3 快速排序

二. 快速排序

算法分析：

- 快速排序算法是递归调用，系统内用堆栈保存递归参数，当每次划分比较均匀时，栈的最大深度为$[\log_2 n]+1$，所以，快速排序的空间复杂度是：$S(n)=O(\log_2 n)$

10.3 快速排序

二. 快速排序

算法分析：

- 实验结果表明：就平均计算时间而言，快速排序是所有内部排序方法中最好的一个，时间复杂度是$O(n\log_2 n)$
- 快速排序是一种不稳定的排序方法

练习

一. 已知数列为4、8、5*、7、5、3、2、9、6，若采用快速排序对数列进行从小到大的排序，枢轴记录选择序列的首记录，要求写出每趟排序后的结果。