第10章 内部排序

- 概述
- 插入排序
- 交换排序
- 选择排序
- 归并排序
- 小结

概 述

排序：将一组杂乱无章的记录按一定的规律顺次排列起来。

关键字(*key*)：通常数据记录有多个属性域，即多个数据成员组成，其中有一个属性域可用来区分记录，作为排序依据。该域即为关键字。

主关键字：如果在待排序记录序列中各个记录的关键字互不相同，这种关键字即主关键字。按照主关键字进行排序，排序的结果是唯一的。

次关键字:待排序记录序列中有些记录的关键字可能相同，这种关键字称为次关键字。按照次关键字进行排序，排序的结果可能不唯一。

排序算法的稳定性: 如果在记录序列中有两个记录$R_{[i]}$和$R_{[j]}$，它们的关键字 $key_{[i]} == key_{[j]}$，且在排序之前，记录$R_{[i]}$排在$R_{[j]}$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，否则称这个排序方法是不稳定的。

内排序与外排序：内排序是指在排序期间数据记录全部存放在内存的排序；外排序是指在排序期间全部记录个数太多，不能同时存放在内存，必须根据排序过程的要求，不断在内、外存之间移动的排序。

排序的时间开销：排序的时间开销是衡量算法好坏的最重要的标志。排序的时间开销可用算法执行中的记录比较次数与记录移动次数来衡量。各节给出算法运行时间代价的大略估算一般都按平均情况进行估算。对于那些受记录关键字序列初始排列及记录个数影响较大的，需要按最好情况和最坏情况进行估算。

算法执行时所需的附加存储：评价算法好坏的另一标准。

- 待排序记录的存储方式：
  - **以一维数组作为存储结构，排序时必须实际移动记录；**
  - **以链表(动态链表或静态链表)作为存储结构，排序时不用物理移动记录，只需修改指针；**
  - **有的排序方法难于在链表上实现，且需要避免排序过程中的记录移动，可将待排序记录本身存储在一组地址连续的存储单元内，同时附设一个指示记录存储位置的地址向量，排序过程中不移动记录，而移动地址向量中这些记录的地址。**

待排序记录的数据类型说明：

```
struct Record{
    int  key;
    …;
};

Record r[n];
```

第十章 内部排序

- 概述
- 插入排序
- 交换排序
- 选择排序
- 归并排序
- 小结

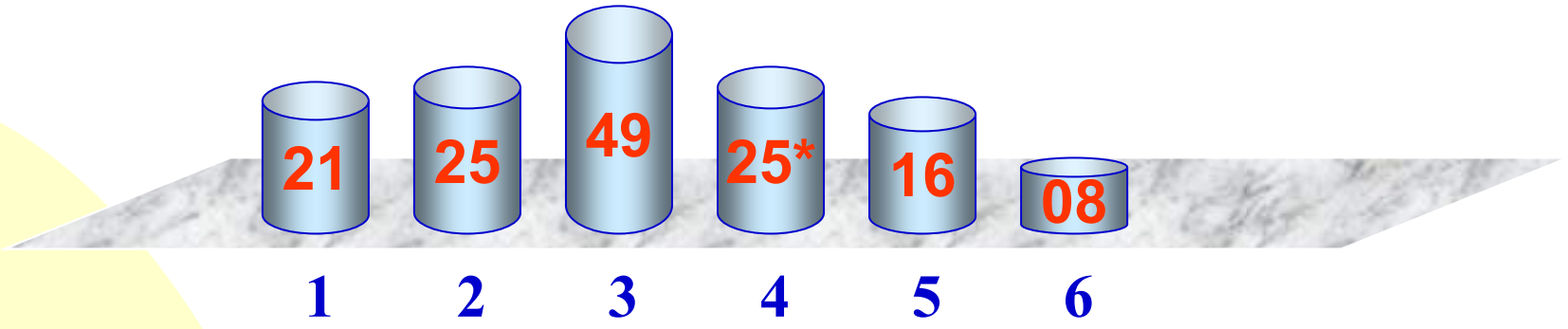
插入排序 (Insert Sorting)

基本方法：每步将一个待排序的记录，按其关键字大小，插入到前面已经排好序的一组记录的适当位置上，直到记录全部插入为止。

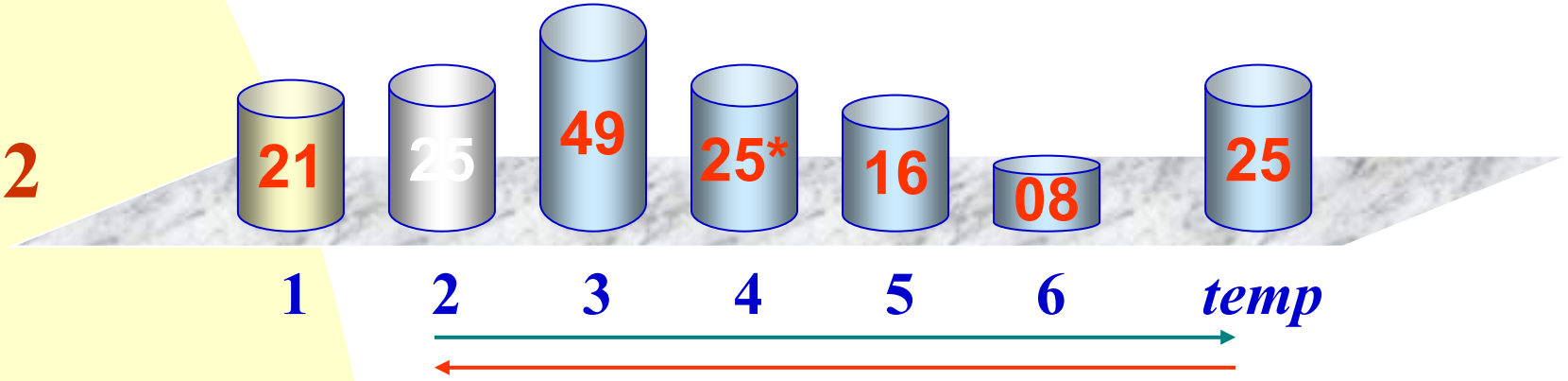
直接插入排序 (Insert Sort)

直接插入排序的基本思想是：当插入第i ($i \geq 1$) 个记录时，前面的R[1], R[2], …, R[i-1]已经排好序。这时，用R[i]的关键字与R[i-1], R[i-2], …的关键字顺序进行比较，找到插入位置即将R[i]插入，原来位置上的记录向后顺移。

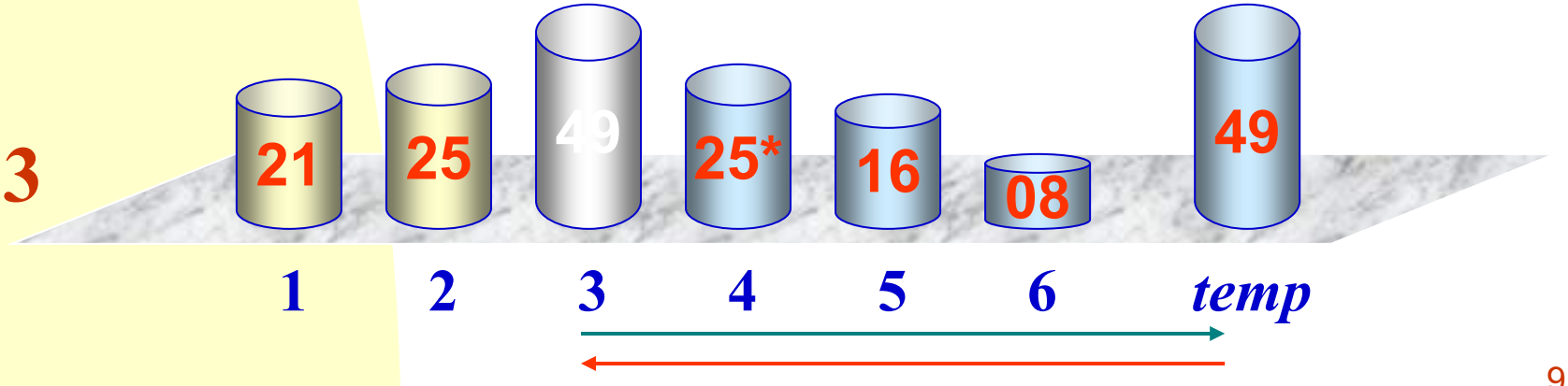
各趟排序结果

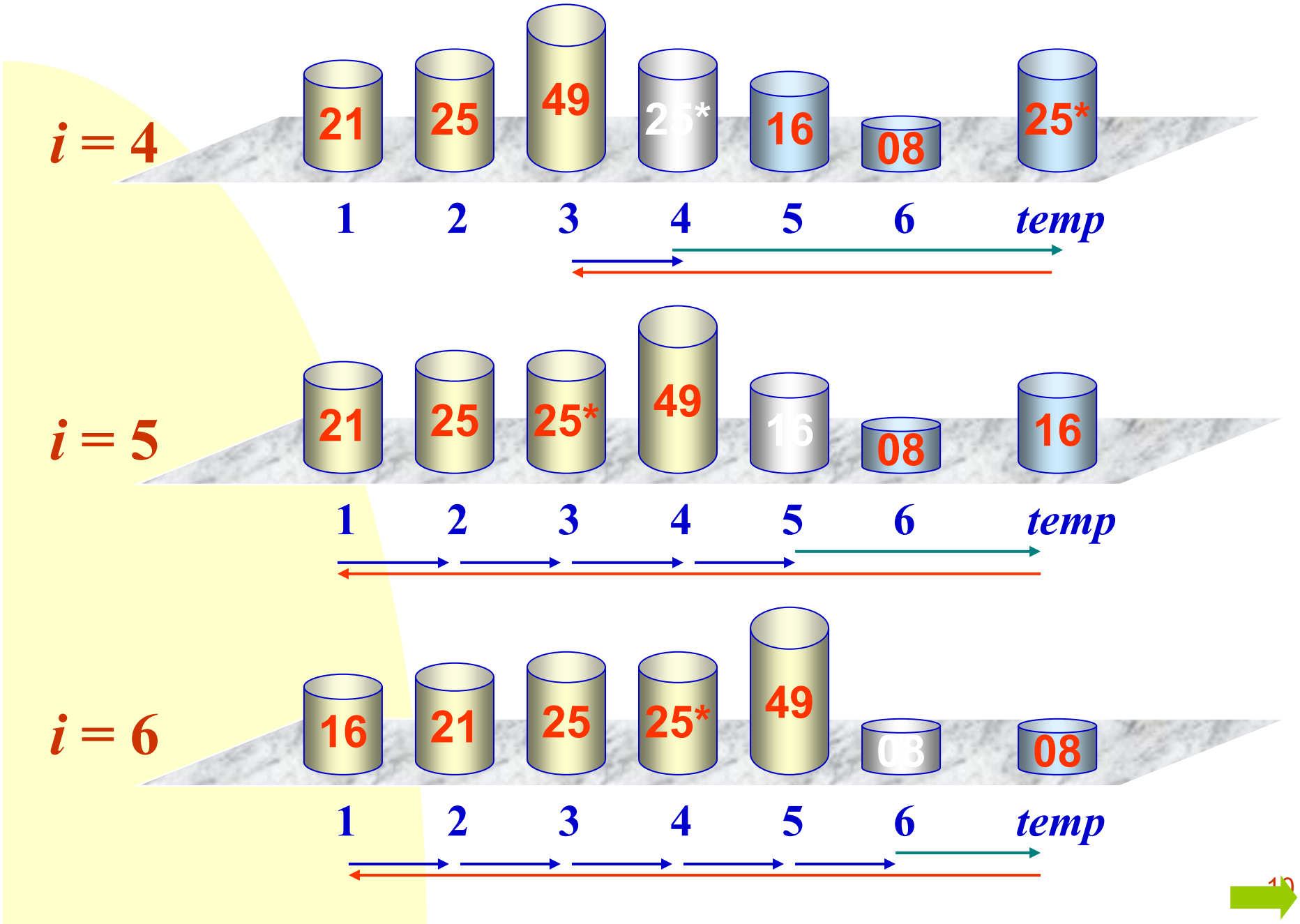

21
25
49
25*
16
08
1
2
3
4
5
6

$i = 2$

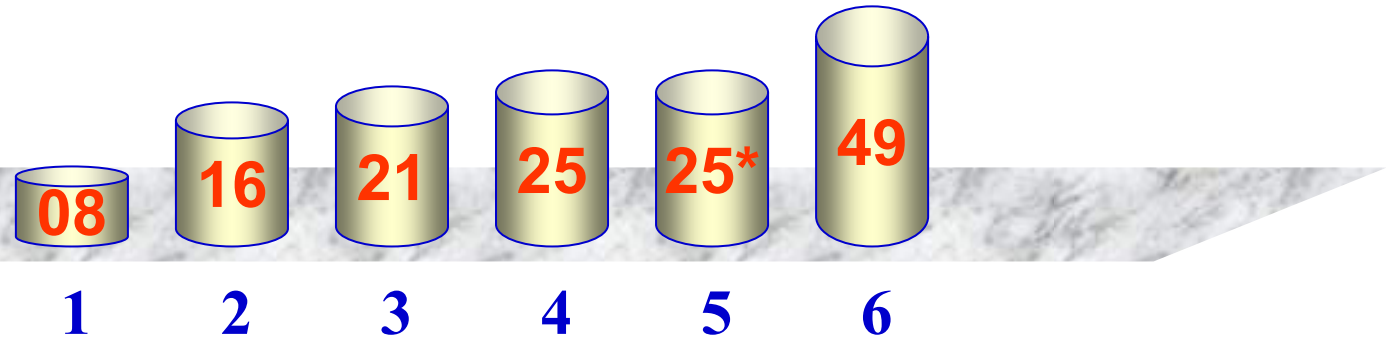

21
25
49
25*
16
08
25
1
2
3
4
5
6
temp

$i = 3$


21
25
49
25*
16
08
49
1
2
3
4
5
6
temp

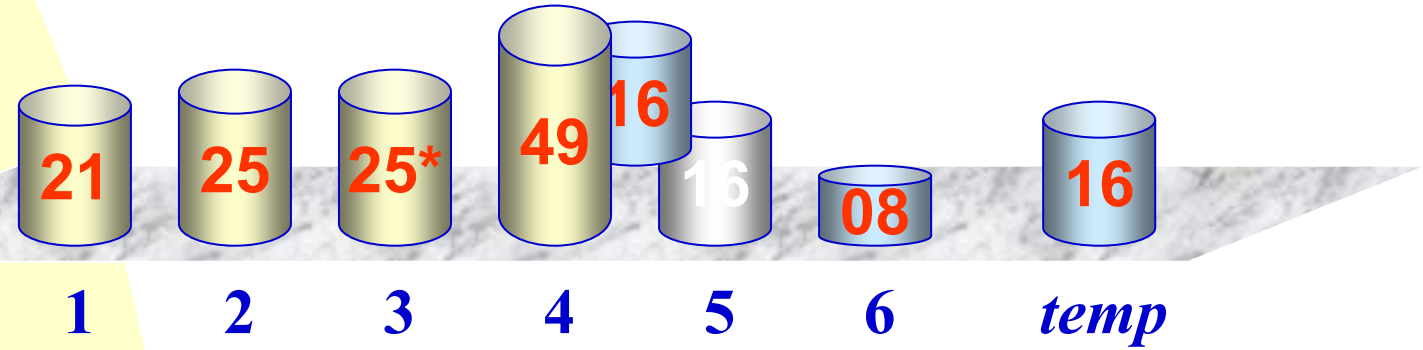

i = 4
21
25
49
25*
16
08
25*
1
2
3
4
5
6
temp
i = 5
21
25
25*
49
16
08
16
1
2
3
4
5
6
temp
i = 6
16
21
25
25*
49
08
08
1
2
3
4
5
6
temp

完成

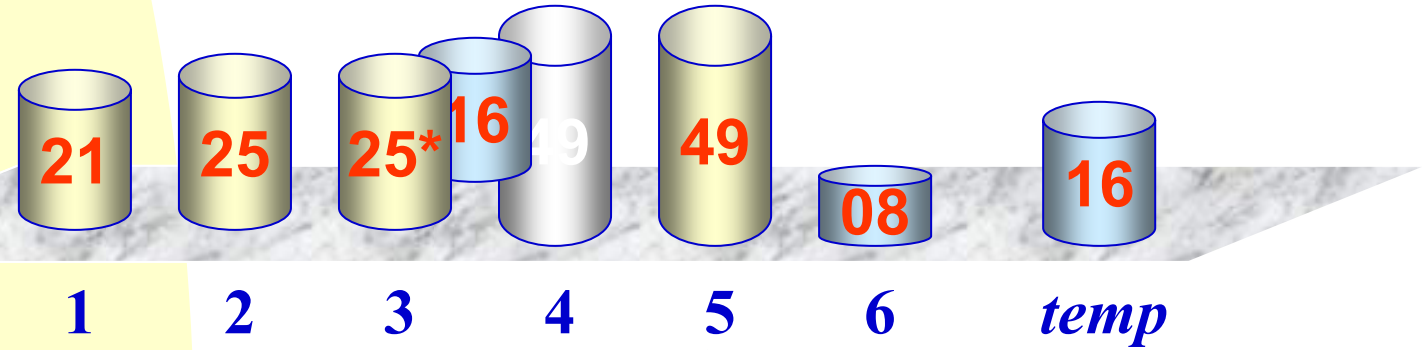


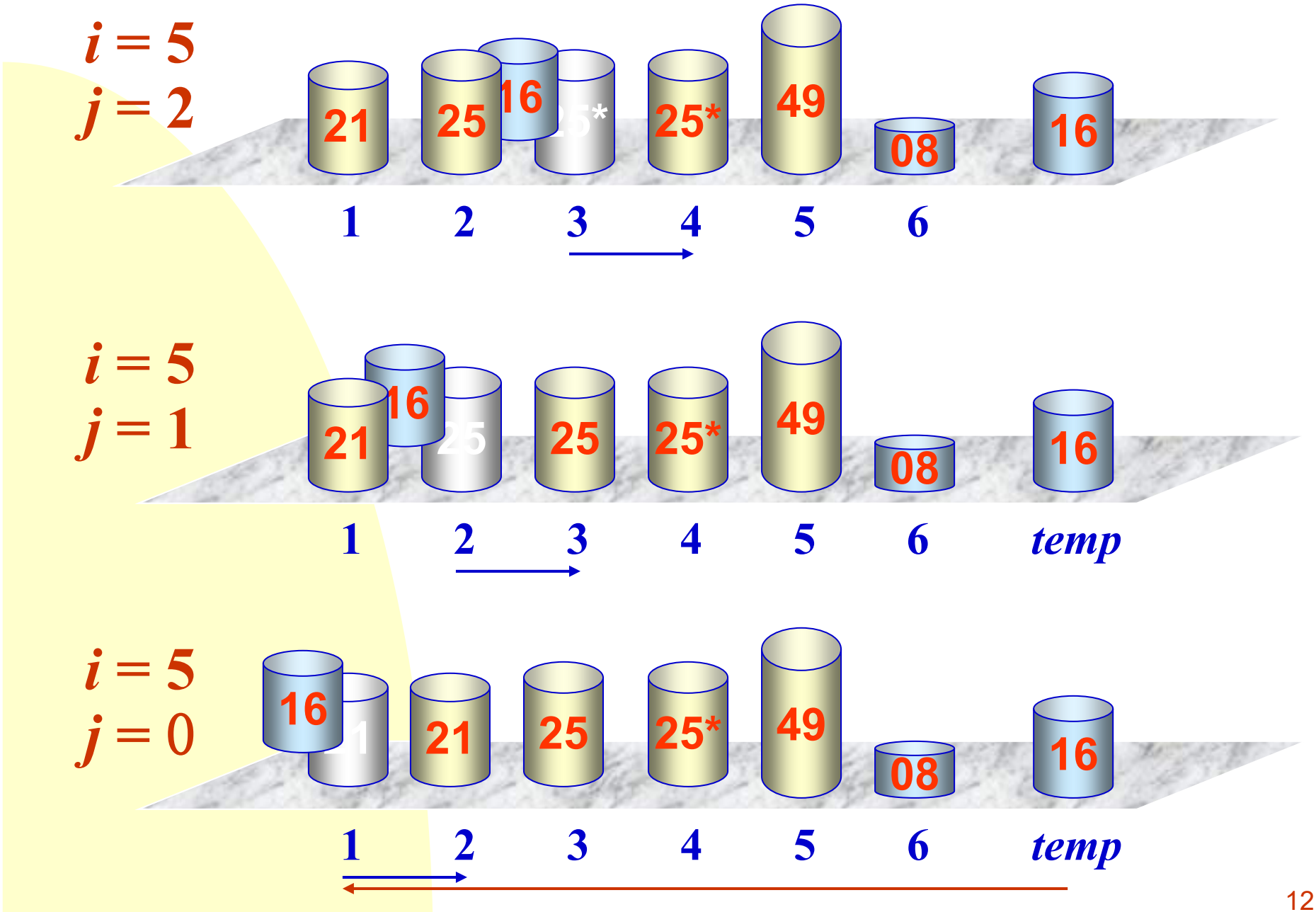
$i=5$ 时的排序过程

$i=5$
$j=4$



$i=5$
$j=3$




i = 5
j = 2
21
25
16
25*
25*
49
08
16
1
2
3
4
5
6
i = 5
j = 1
21
16
25
25
25*
49
08
16
1
2
3
4
5
6
temp
i = 5
j = 0
16
1
21
25
25*
49
08
16
1
2
3
4
5
6
temp

直接插入排序的算法：

```
void InsertSort( Record r[], int n )
{
    for( i=1; i<n; ++i )
    {
      temp=r[i];
      for(j=i-1; j>=0 && temp.key<r[j].key; --j )
          r[j+1]=r[j];    // 记录后移
      r[j+1]=temp;
    }
}
```

算法分析：

- 直接插入排序的算法简洁，容易实现。从时间来看，排序的基本操作为：比较两个记录的大小和移动记录。其中：
  - ✦ 最小比较次数：Cmin = n-1 = $O(n)$
  - ✦ 最大比较次数：Cmax =(1+2+…+n-1)=n(n-1)/2 = $O(n^2)$
  - ✦ 最小移动次数：Mmin = 2(n-1)= $O(n)$
  - ✦ 最大移动次数：Mmax = (2+1 + 3+1 + … + n+1) = $O(n^2)$

平均情况？

在平均情况下的关键字比较次数和记录移动次数约为 $n^2/4$。

因此，直接插入排序的时间复杂度为 $O(n^2)$。

- 关键字比较次数和记录移动次数与记录关键字的初始排列有关。
- 直接插入排序是一种稳定的排序方法。
- 从空间看，直接插入排序算法只需一个记录的辅助空间。

折半插入排序 (Binary Insertsort)

基本思想：

如何提高每次插入操作的效率?

(R[1], R[2], …, R[*i*-1]) R[*i*]

前i-1个记录已经排好序!

在插入 R[*i*] 时，利用折半搜索法寻找 R[*i*] 的插入位置。

折半插入排序的算法:

```
void BInsertSort( Record r[], int n) {
    for( i=1; i<n; ++i ){
        temp=r[i];   // 将L.r[i]暂存到L.r[0]
        low=0; high=i-1;
        while( low <= high ){   // 在r[low..high]中折半查找有序插入的位置
            mid = (low+high)/2;    // 折半
            if ( temp.key < r[mid].key )
                high = mid-1;  // 插入点在低半区
            else
                low = mid+1;   // 插入点在高半区
        }
        for( j=i-1; j>=low; --j ) r[j+1]=r[j];     // 记录后移
        r[low]=temp;      // 插入
    }
}
```

算法分析：

折半插入排序所需要的关键字比较次数与待排序记录序列的初始排列的关系？

无关，关键字比较次数仅依赖于记录个数。

在插入第 i 个记录时，需要经过 $\lfloor \log_2 i \rfloor + 1$ 次关键字比较，才能确定它应插入的位置。将 n 个记录用折半插入排序所进行的关键字比较次数为 $O(n\log_2 n)$.

当 n 较大时，总的关键字比较次数比直接插入排序的最坏情况要好得多，但比其最好情况要差。

折半插入排序的记录移动次数与直接插入排序相同，依赖于记录的初始排列。

折半插入排序的时间复杂度仍为$O(n^2)$

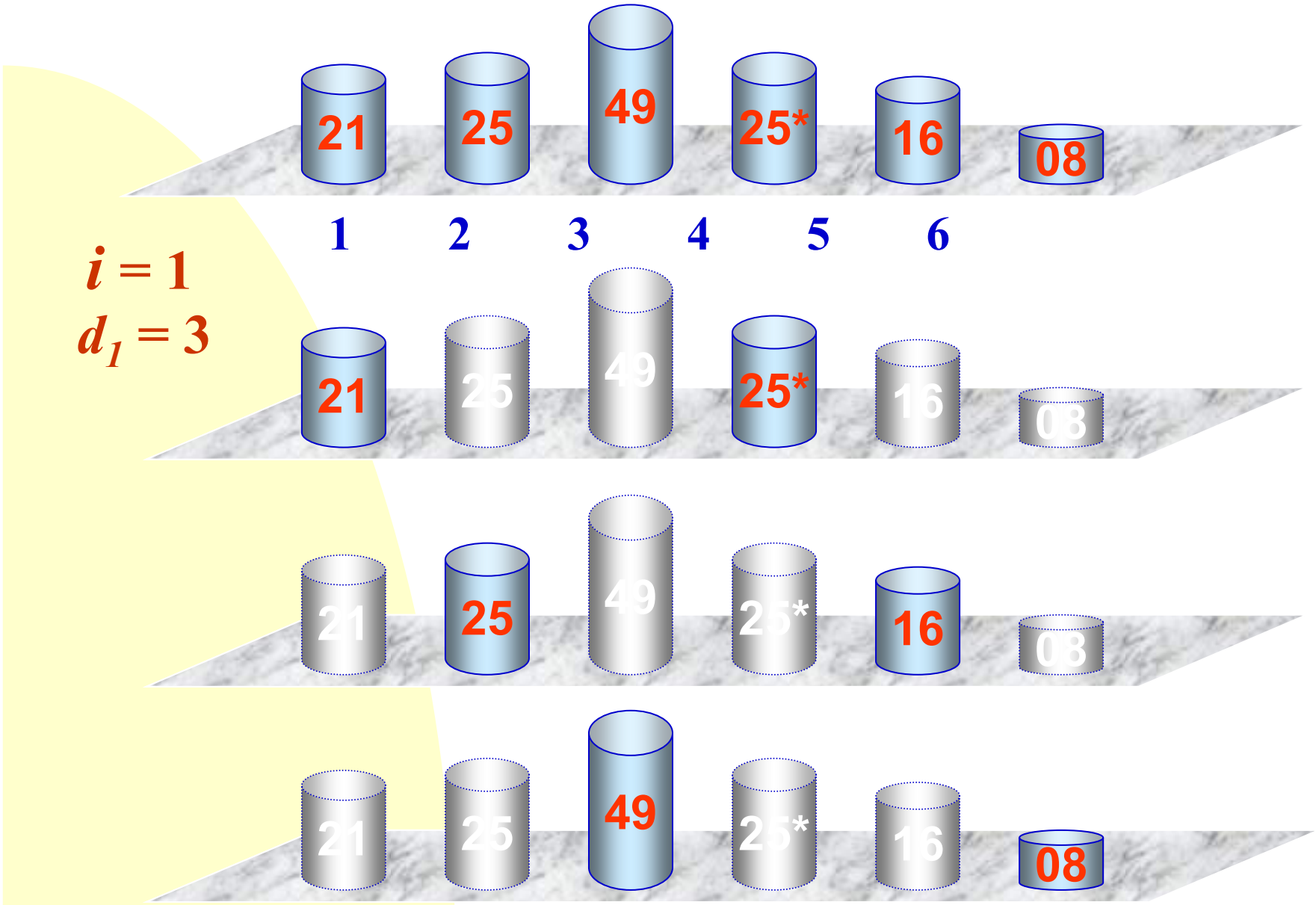
折半插入排序是一个稳定的排序方法。

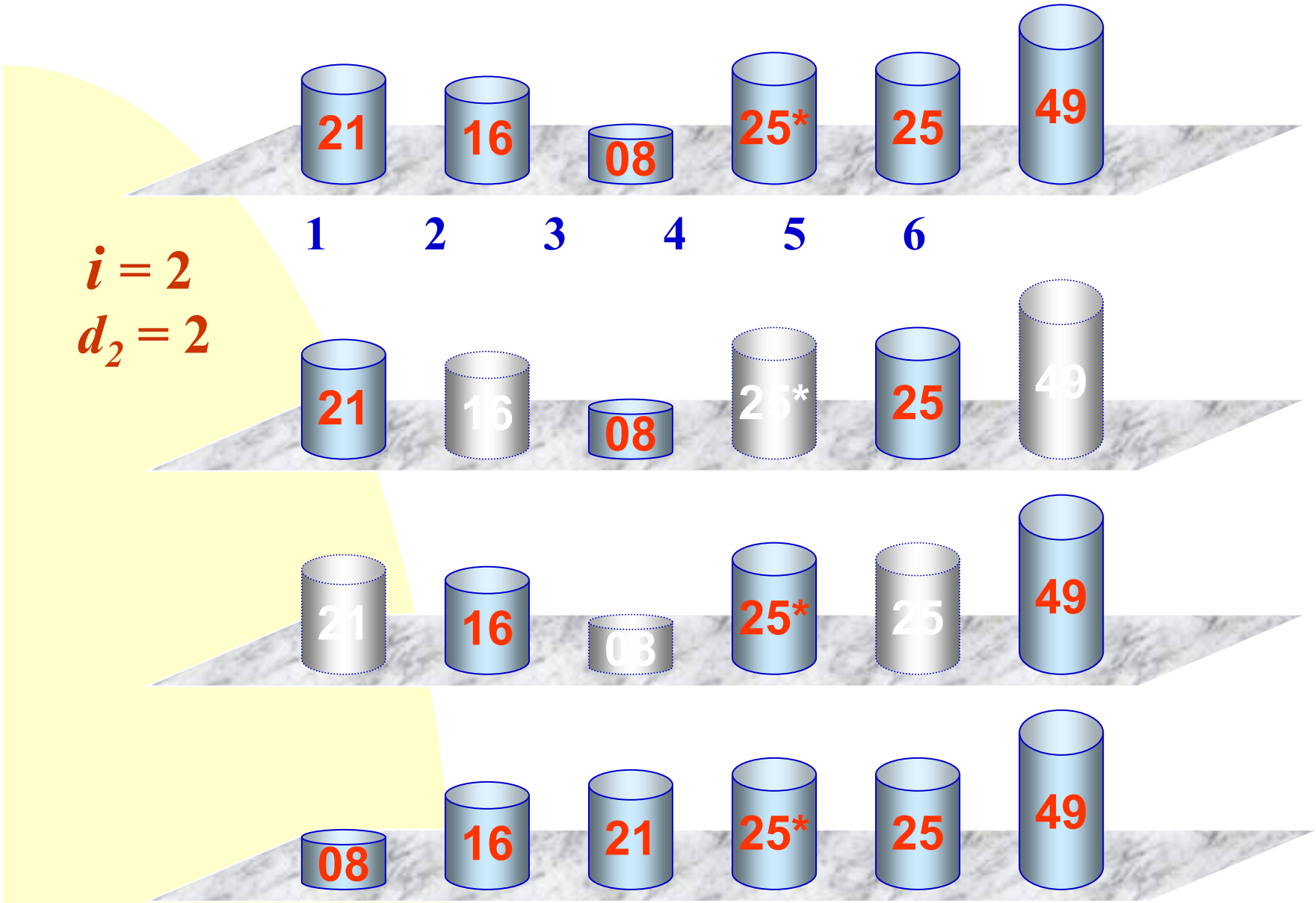
希尔排序 (Shell Sort)

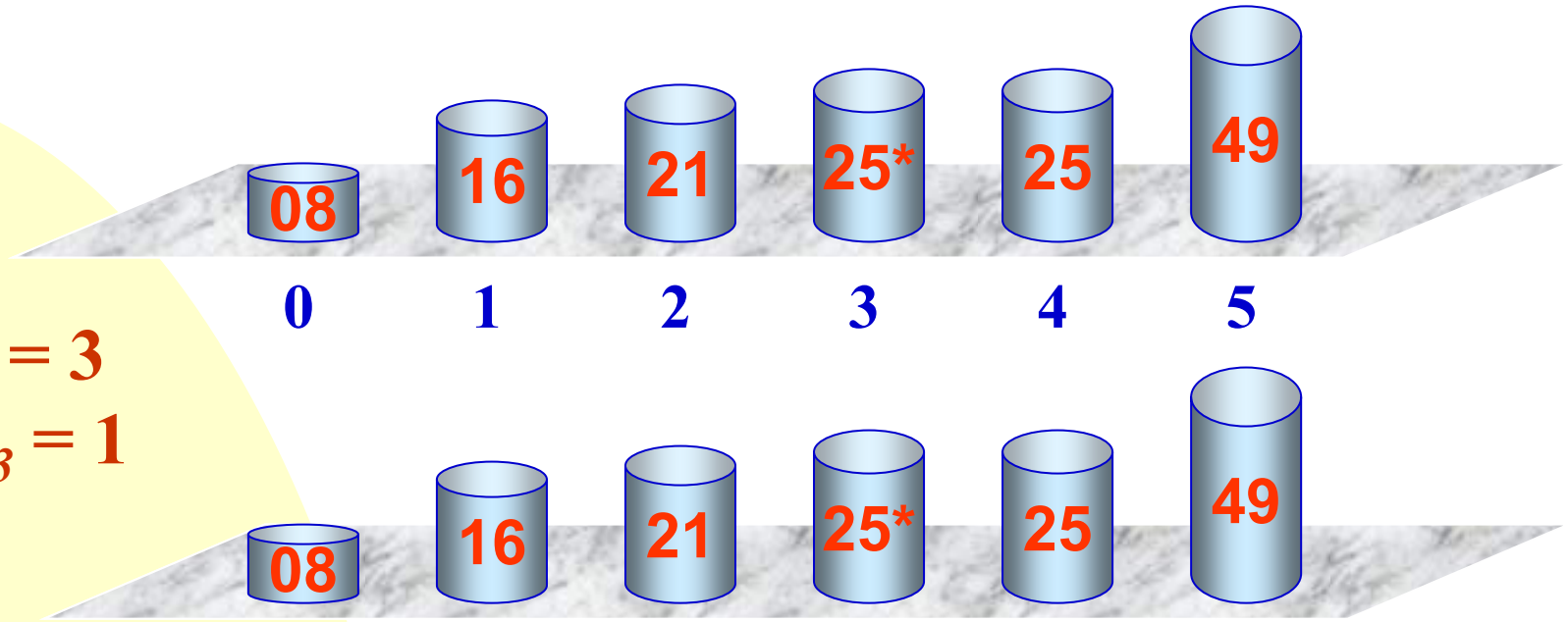
- 直接插入排序在以下两种情况下效率较高：
  - 当待排序记录处于“基本有序”状态
  - 当待排序记录个数n值较小时
- **Shell** 排序方法正是基于这两点考虑对直接插入排序加以改进而提出的一种插入排序算法。
- **Shell** 排序方法的基本思想？
  - 将待排序记录通过引入间隔(增量)分成若干个组（或称子序列），然后在每一组内进行直接插入排序；
  - 通过采用一种“逐渐缩小插入增量”的方法，逐步增长组的长度。

Shell 排序的基本做法：

- 先确定一个小于n 的整数d_1作为第一个增量，把记录序列分为d_1个组，所有距离为d_1倍数的记录放在同一个组中，在各组内进行直接插入排序；
- 然后，取第二个增量d_2（ $d_2 < d_1$ ）,重复上述分组和排序，直至所取的增量$d_t=1$（$d_t<d_{t-1}<\ldots< d_2 < d_1$ ）, 即所有记录处在同一组中进行直接插入排序为止。


21
25
49
25*
16
08
1
2
3
4
5
6
$i = 1$
$d_1 = 3$


21
16
08
25*
25
49
1
2
3
4
5
6
$i = 2$
$d_2 = 2$
21
16
08
25*
25
49
21
16
08
25*
25
49
08
16
21
25*
25
49



开始时 *增量*值较大，子序列中的记录较少，排序速度较快；

随着排序进展，增量值逐渐变小，子序列中记录个数逐渐变多，但由于前面工作的基础，大多数记录已基本有序，所以排序速度仍然很快。

希尔排序的算法：

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

void ShellInsert( Record r[], int n, int dk ){
    for( i=dk; i<n; i++ ){    //需将r[i]插入有序增量子表
          temp = r[i];
          for( j=i-dk; j>=0 && temp.key<r[j].key; j-=dk)
                r[j+dk] = r[j];          // 记录后移，查找插入位置
          r[j+dk] = temp;
    }
}
```

*增量*的取法有多种。

最初 shell 提出取 $d_1 = \lfloor n/2 \rfloor$，$d_2 = \lfloor d_1/2 \rfloor$，直到$d_t = 1$。

后来knuth 提出取$d_{i+1} = \lfloor d_i/3 \rfloor + 1$。

还有人提出都取奇数为好，也有人提出各*增量*互质为好。

算法分析：

对特定的待排序记录序列，可以准确地估算关键字的比较次数和记录移动次数。

但想要弄清关键字比较次数和记录移动次数与增量选择之间的依赖关系，并给出完整的数学分析，还没有人能够做到。

Knuth利用大量的实验统计资料得出，当 n 很大时，关键字平均比较次数和记录平均移动次数大约在 $n^{1.25}$ 到 $1.6n^{1.25}$ 的范围内。

这是在利用直接插入排序作为子序列排序方法的情况下得到的。

第十章 内部排序

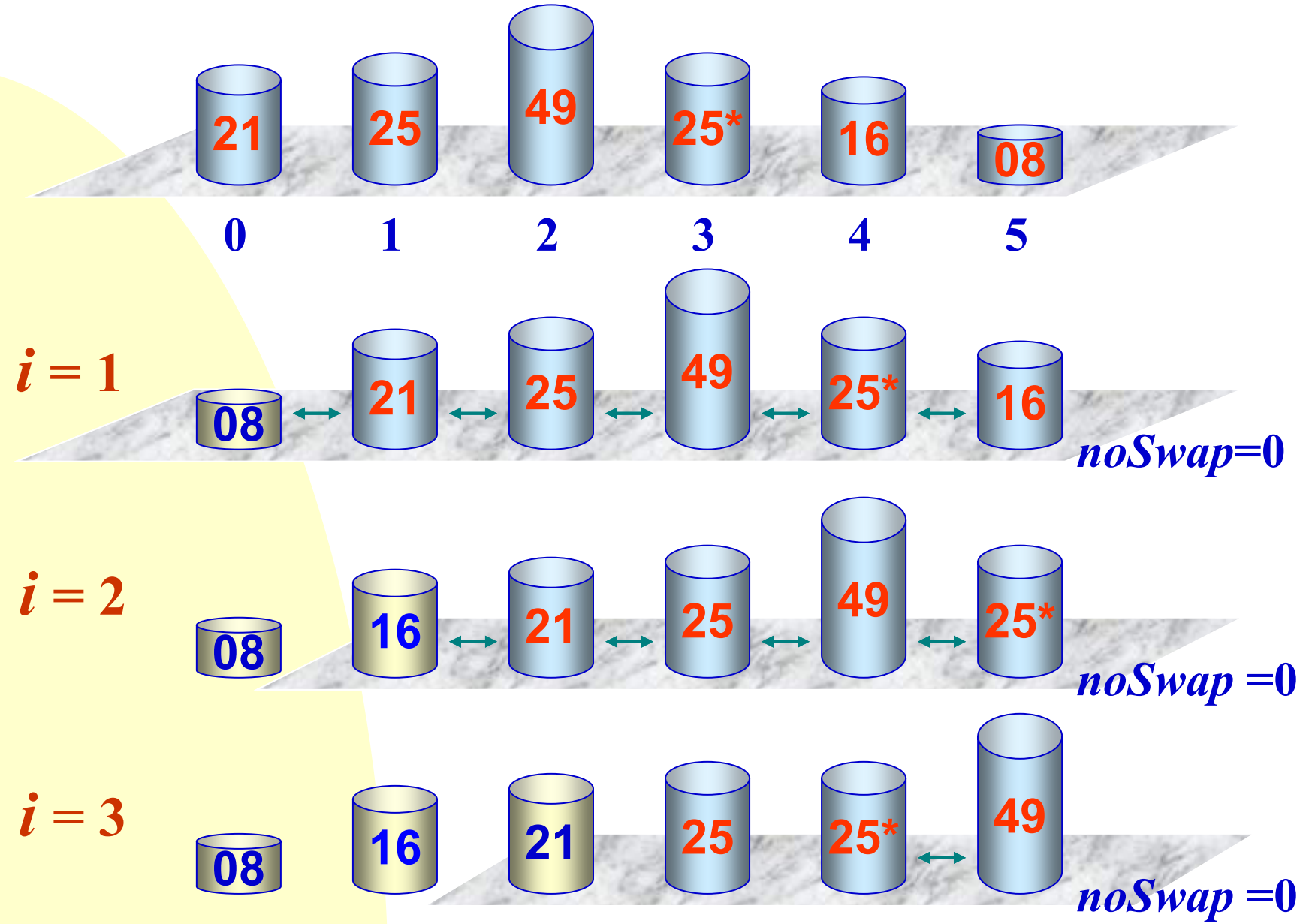
- 概述
- 插入排序
- 交换排序
- 选择排序
- 归并排序
- 小结

交换排序 (Exchange Sort)

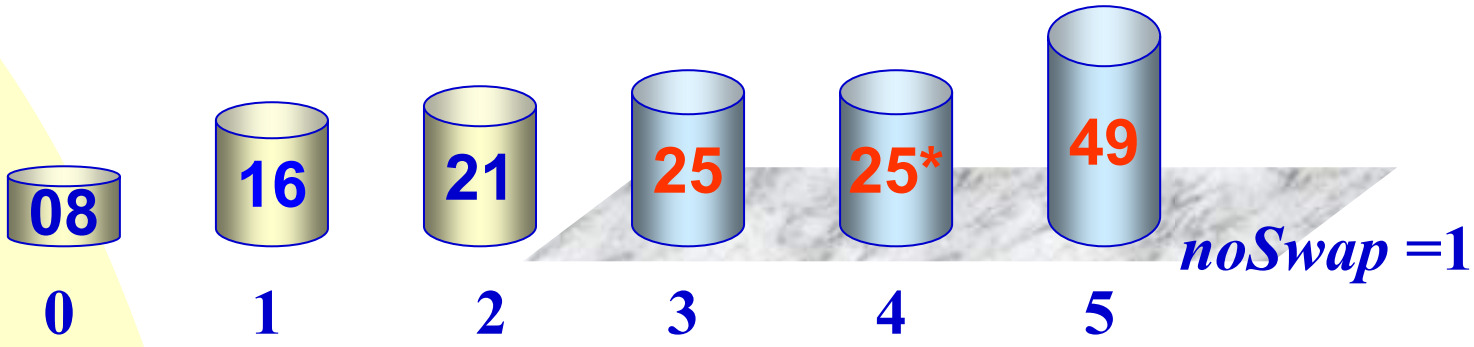
基本思想：是两两比较待排序记录的关键字，如果发生逆序(即排列顺序与排序后的次序正好相反)，则交换之，直到所有记录都排好序为止。

起泡排序 (Bubble Sort)

基本方法：比较相邻两个记录的关键字，若R[i].key>R[i+1].key，则交换之，其中i从0到n-pass-1（pass的初值为1）称之为一趟起泡排序，其结果是使最大关键字的记录被交换到n-pass的位置上，如果某一趟起泡排序过程中没有进行一次记录交换，则排序过程结束。最坏情况下需n-1趟排序。


21
25
49
25*
16
08
0
1
2
3
4
5
i = 1
08
21
25
49
25*
16
noSwap=0
i = 2
08
16
21
25
49
25*
noSwap =0
i = 3
08
16
21
25
25*
49
noSwap =0

$i = 4$



起泡排序的算法：

```
void BubbleSort( Record r[], int n ){
    // 从下往上扫描的起泡排序
    for( i=0; i<n-1; i++ ){      // 做n-1趟排序
        noSwap=true;          // 置未交换标志
        for( j=n-1; j>i; j-- )  // 从下往上扫描
            if( r[j].key < r[j-1].key){
                temp=r[j-1];  // 交换记录
                r[j-1]=r[j];
                r[j]=temp;
                noSwap=false;
            }
        if( noSwap ) break;  // 本趟扫描未发生交换，则终止算法
    }
}
```

还能做什么进一步改进吗？

算法分析：

在记录的初始状态为有序时，此算法只执行一趟起泡，做 n-1 次关键字比较，不移动记录。
这是最好的情形。

最坏的情形是算法执行了n-1趟起泡，第 i 趟 ($1 \le i < n$) 做了 $n-i$ 次关键字比较，执行了$n-i$ 次记录交换。
这样在最坏情形下总的关键字比较次数KCN和记录移动次数RMN为：

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

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

起泡排序需要一个附加记录以实现记录值的对换。

起泡排序是一个稳定的排序方法。

快速排序 (Quick Sort)

基本思想：

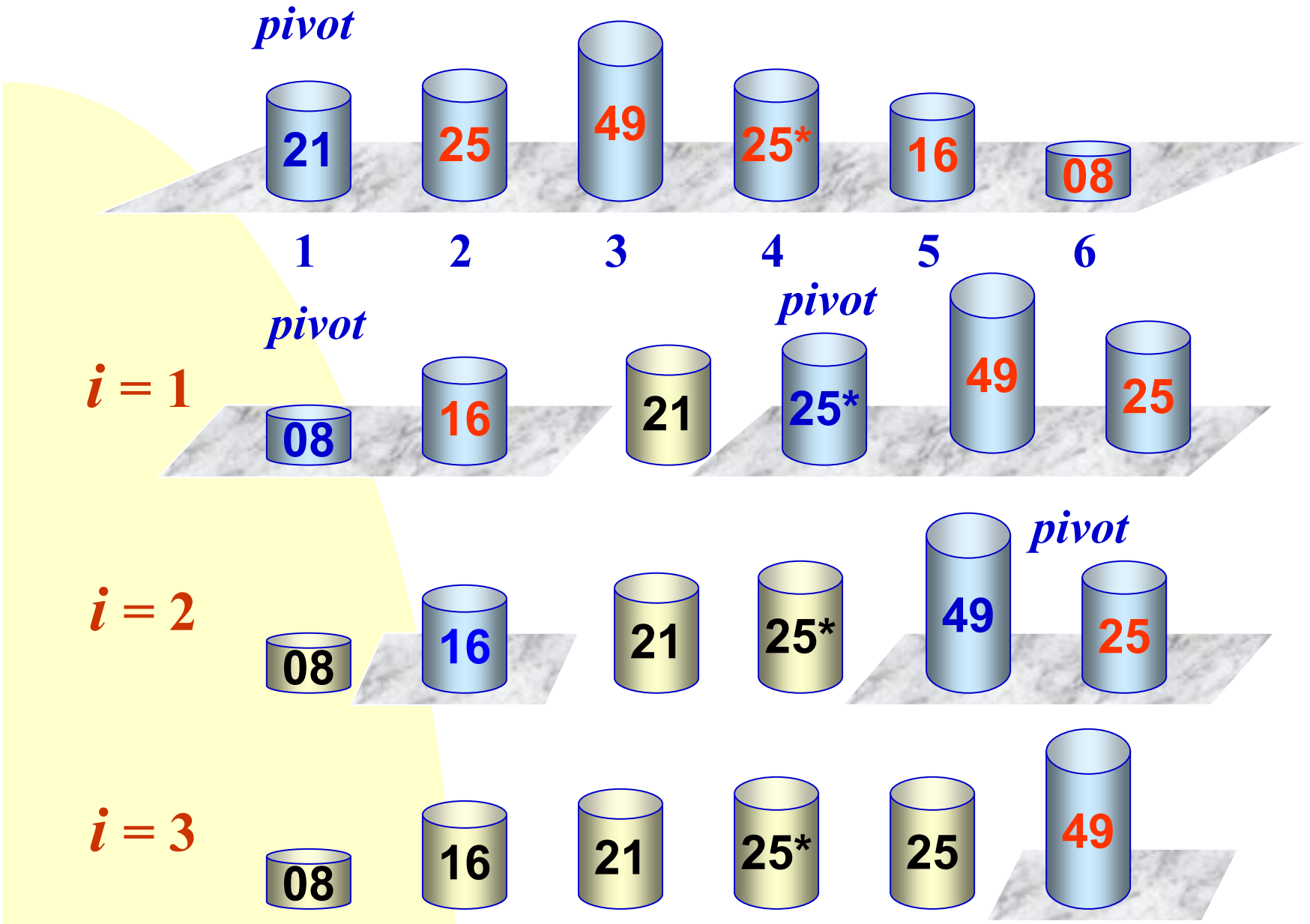
任取待排序记录序列中的某个记录（例如取第一个记录）作为基准，按照该记录的关键字大小，将整个记录序列划分为左右两个子序列：

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（或等于）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安放的位置)。

然后分别对这两个子序列重复施行上述方法，直到所有的记录都排在相应位置上为止。


pivot
21
25
49
25*
16
08
1
2
3
4
5
6
pivot
pivot
i = 1
08
16
21
25*
49
25
i = 2
pivot
08
16
21
25*
49
25
i = 3
08
16
21
25*
25
49

算法描述：

一趟划分算法：

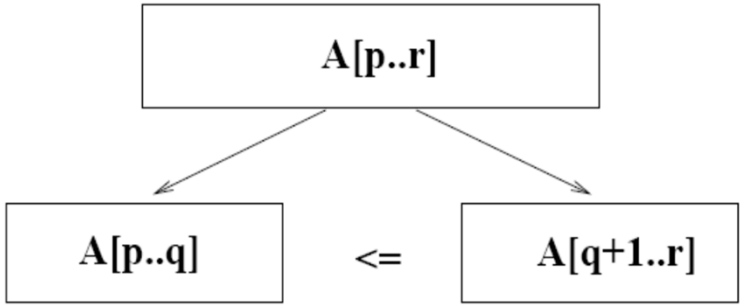
```
int Partition( Record r[], int low, int high){
    temp = r[low];      // 用子表的第一个记录作基准记录
    while( low<high ){      // 从表的两端交替地向中间扫描
        while(low<high && r[high].key>=temp.key)  high--;
        if(low<high) r[low++] = r[high];
        while(low<high && r[low].key<= temp.key)  low++;
        if(low<high) r[high--] = r[low];
    }
    r[low] = temp;              //基准记录到位
    return low;
}
```

```
void QSort( Record r[], int low, int high)
{
    if( low < high )    // 长度大于1
    {
        pivotloc=Partition(r, low, high);
        QSort(r, low, pivotloc-1);   // 对低子表递归排序
        QSort(r, pivotloc+1,high);  // 对高子表递归排序
    }
}
```

快速排序算法背后的思想?

分治法：

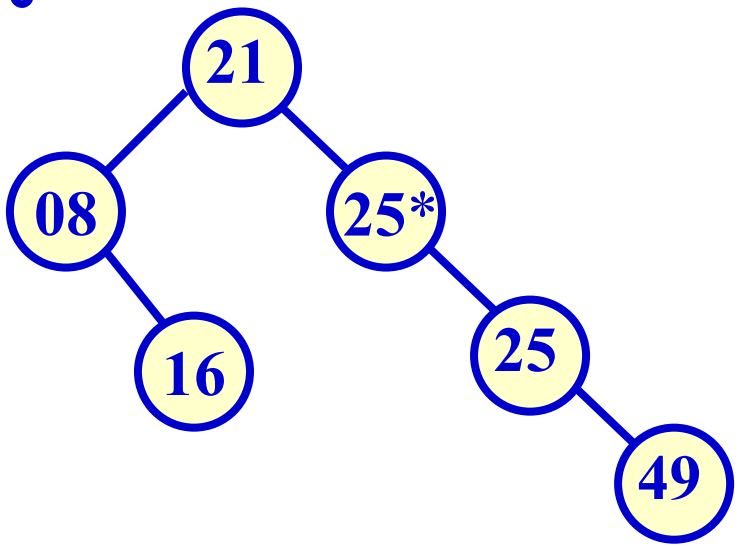
Divide：



Conquer：

Combine：

算法*quicksort*是一个递归的算法，其递归树如图所示。



✓ 算法*partition*利用序列第一个记录作为基准，将整个序列划分为左右两个子序列，将基准记录安放到位。

算法分析：

从快速排序算法的递归树可知，快速排序的趟数取决于递归树的深度。

最好的情况是?

如果每次划分对基准记录定位后，该记录的左侧子序列与右侧子序列的长度接近，则下一步将是对两个长度减半的子序列进行排序。

在n个元素的序列中，对一个记录定位所需时间为 $O(n)$。若设 $\mathrm{T}(n)$ 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：

$$T(n) \le n + 2\,T(n/2) \quad \text{// 一趟划分比较次数为n-1次}$$

$$\le n + 2\,(\,n/2 + 2T(n/4)\,) = 2n + 4T(n/4)$$

$$\le 2n + 4\,(\,n/4 + 2T(n/8)\,) = 3n + 8T(n/8)$$

$$\cdots\cdots$$

$$\le n\log_2 n + nT(1) = O(n\log_2 n)$$

可以证明，快速排序的平均计算时间也是$O(n\log_2 n)$。

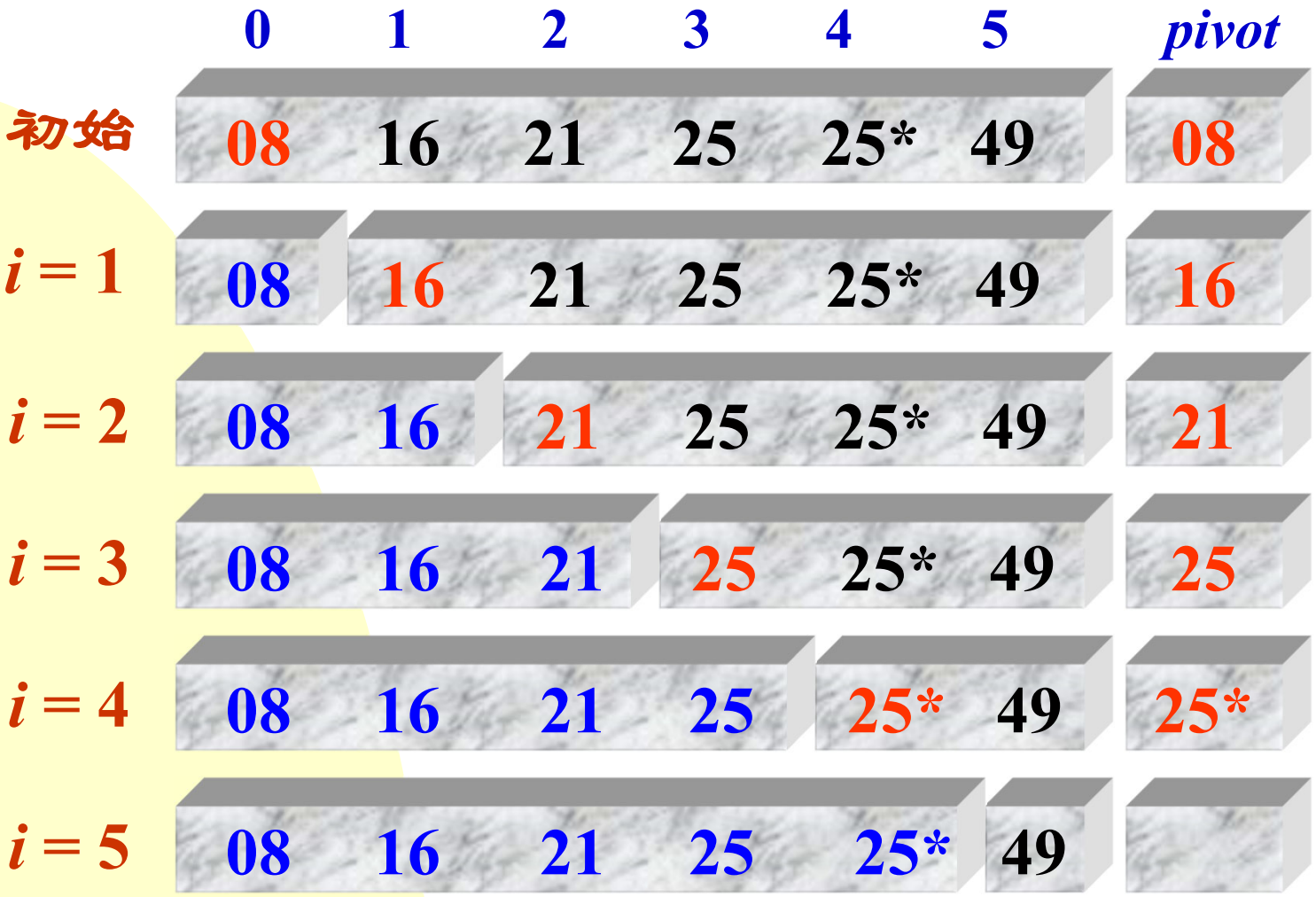
实验结果表明：就平均计算时间而言，快速排序是我们所讨论的所有内排序方法中最好的一个。

最坏的情况?

即待排序记录序列已经按其关键字从小到大排好序的情况下，其递归树成为单支树，每次划分只得到一个比上一次少一个记录的子序列。

这样，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：

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



用第一个记录作为基准记录

快速排序退化的例子

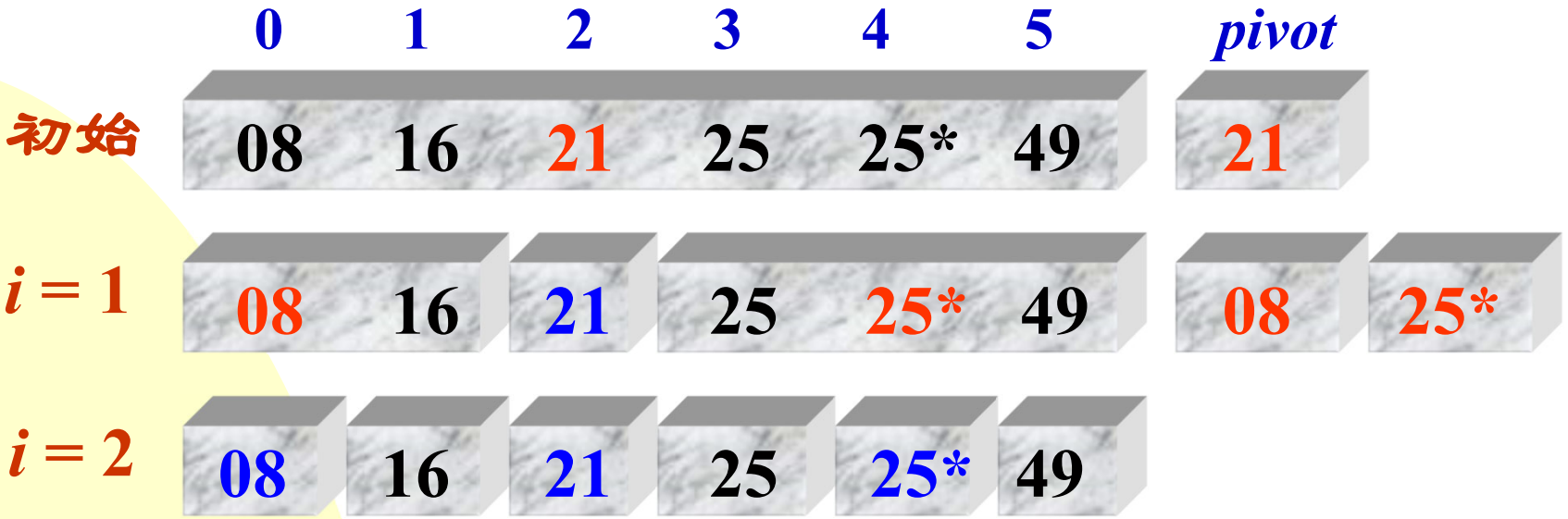
若能更合理地选择基准记录，使得每次划分所得的两个子序列中的记录个数尽可能地接近，可以加速排序速度；

但是由于记录的初始排列次序是随机的，这个要求很难办到!

有一种改进办法：

选择更好的基准记录；

取每个待排序记录序列的第一个记录、最后一个记录和位置接近正中的3个记录，取其关键字居中者作为基准记录。



用居中关键字记录作为基准记录

快速排序是一种不稳定的排序方法。

对于 n 较大的平均情况而言，快速排序是“快速”的，但是当 n 很小时，这种排序方法往往比其它简单排序方法还要慢。