第六章　内部排序

6.1 排序的基本概念

6.2 插入排序

6.3交换排序

6.4 归并排序

6.5基数排序

6.6各种排序算法比较

6.1 排序的基本概念

排序：将一组杂乱无章的数据按一定的规律顺次排列起来。这一过程叫做排序。

排序算法的稳定性：如果在待排记录中有两 个记录$r[i]$和$r[j]$，它们的关键字 $k[i] == k[j]$，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对象全部存放在内存的排序；外排序是指在排序期间全部对象个数太多，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。

6.1 排序的基本概念

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。排序时间开销可用算法执行中的数据比较次数与数据移动次数来衡量。

算法运行时间代价的大略估算一般按平均情况进行估算。

对于那些受待排记录（对象）的关键字初始序列记录（对象）的个数影响较大的算法，需要按最好情况和最坏情况估算。

算法执行时所需的附加存储（空间开销）：

评价算法好坏的另一标准。

Selection sort

In each pass of selection sort, select the smallest element in the unsorted list and place the smallest element in the front of unsorted list.

Assuming a list of N records. In first pass of selection sort, select the smallest element (suppose R[k]) in R[1..N], and swap R[k] with R[1]. and then, the second pass selects the smallest element in R[2..N], and swap with R[2]. and so on.

In general, the ith pass selects the smallest element in R[i..N], and swap with R[i].

The list has been sorted after N-1 passes selection sort.

Selection sort (example)

Selection sort (algorithm)

```
static void Selsort(Elem rec[], int n ){
    Elem it;
    for ( i = 0; i < n; i++ ) {  //ith pass
        int k = i;
        for ( j = i + 1; j < n; j++ ) //select smallest in ren[k]
            if ( rec[j].key() < rec[k].key() ) k = j;
        if ( k != i){  //swap k and i
            it = rec[k];  rec[k] = rec[i];  rec[i] = it;
        }
    }
}
```

Selection sort (Cost)

Assuming a list of N elements.

The comparisons of worst case, best case and the average case are:

$$\sum_{j=1}^{N-1}[N-j]=\frac{1}{2}N(N-1)$$

The moves of worst case is N

The moves of best case is *0*.

The additional space cost is O(1).

The insertion sort is Not stable.

第6章 内部排序

排序方法分为以下四类：

- 插入排序
- 交换排序
- 归并排序
- 基数排序

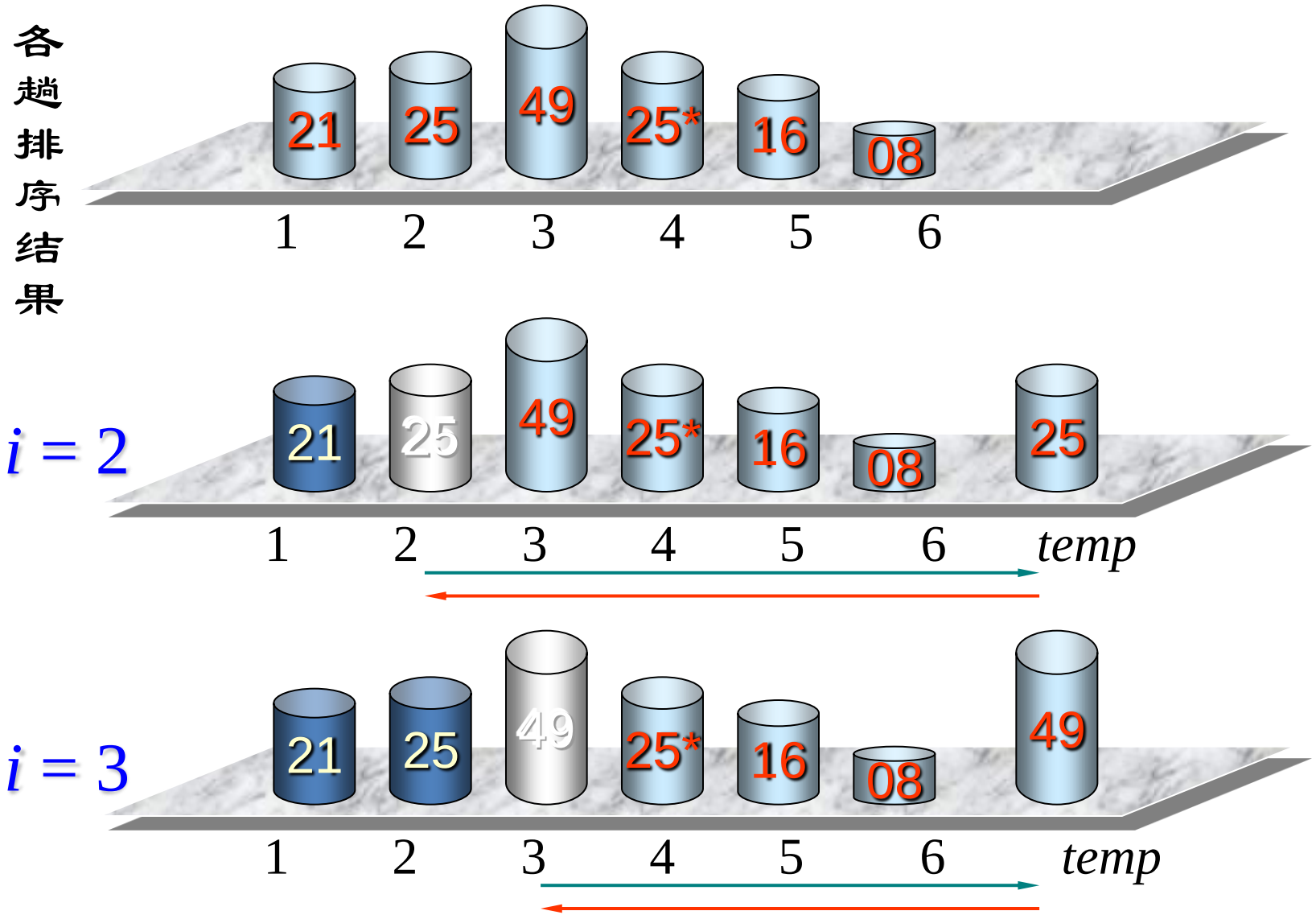
6.2 插入排序 (Insert Sorting)

- 基本方法是：每步将一个待排序的对象，按其关键字大小，插入到前面已经排好序的一组对象的适当位置上，直到对象全部插入为止。

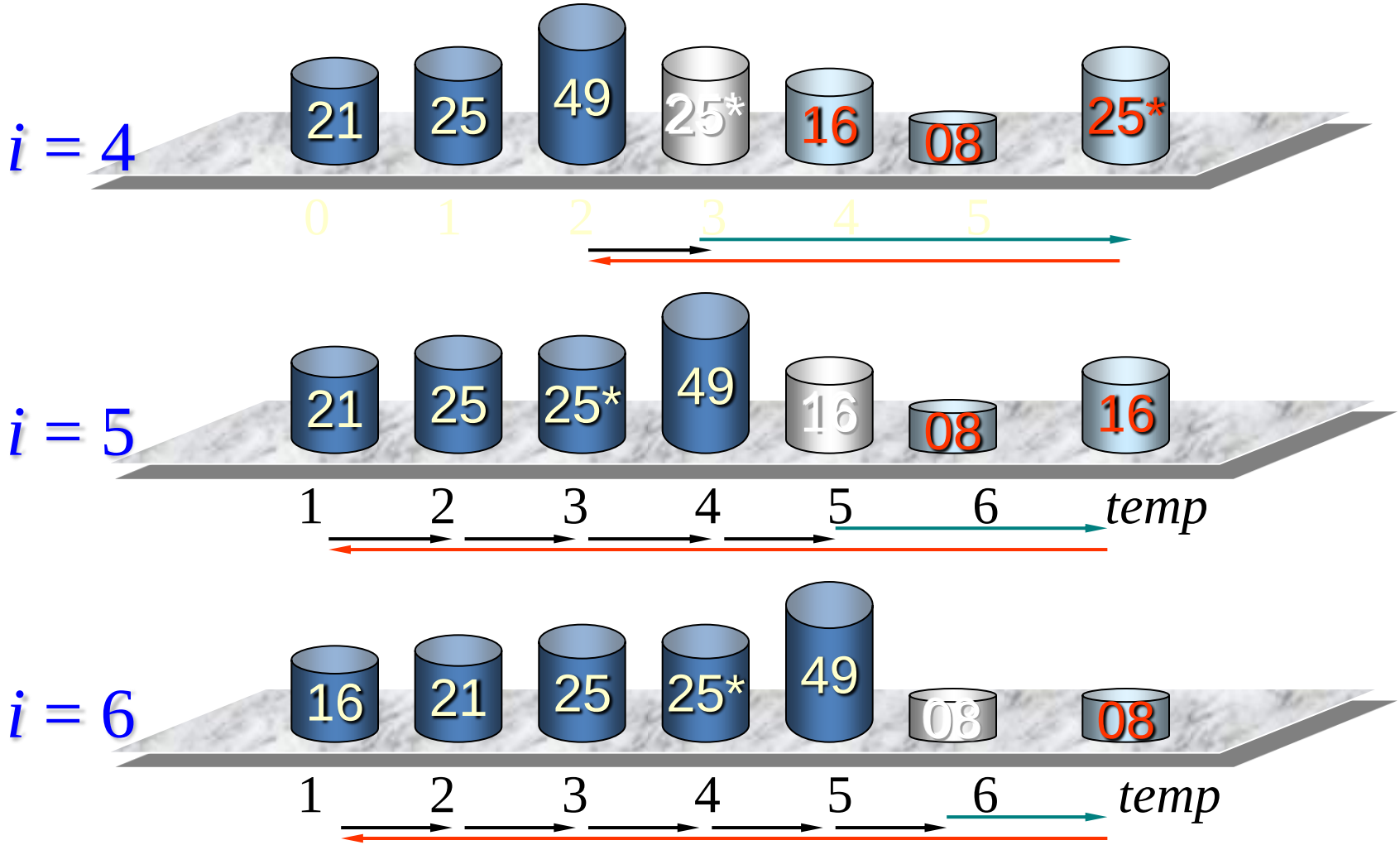
(1)直接插入排序 (Insert Sort)

- 基本思想：当插入第i ($i \geq 1$) 个对象时，前面的V[1], …., V[i−1]已经排好序。 V[0]存放待排元素；这时，用V[i]的关键字与V[i−1], V[i−2], … V[0]的关键字顺序进行比较，找到第一个小于等于元素为止，其后为插入位置，即将V[i]插入其后，原来位置上的对象向后顺移。使得前i个元素有序。

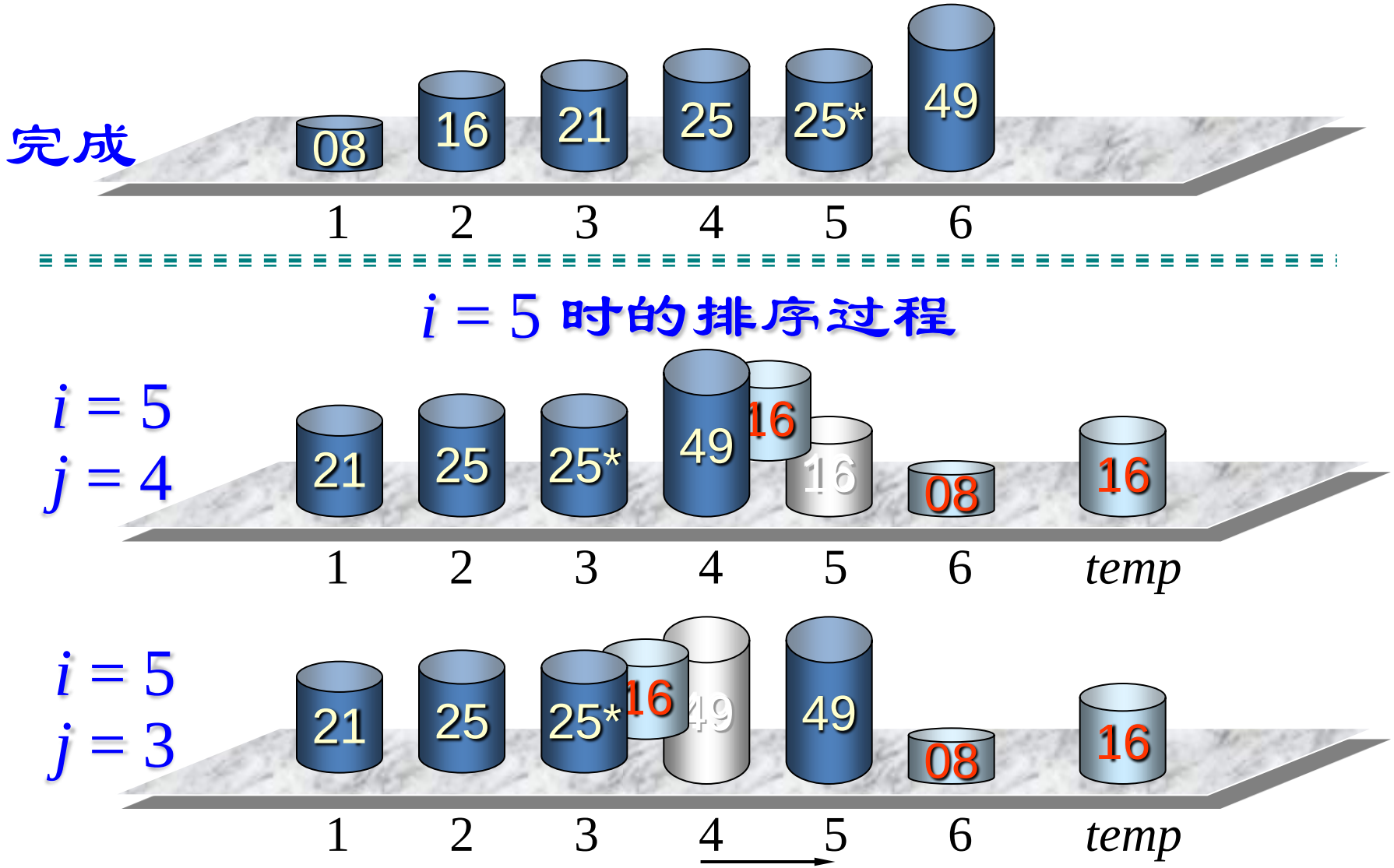
6.2 插入排序 (Insert Sorting)



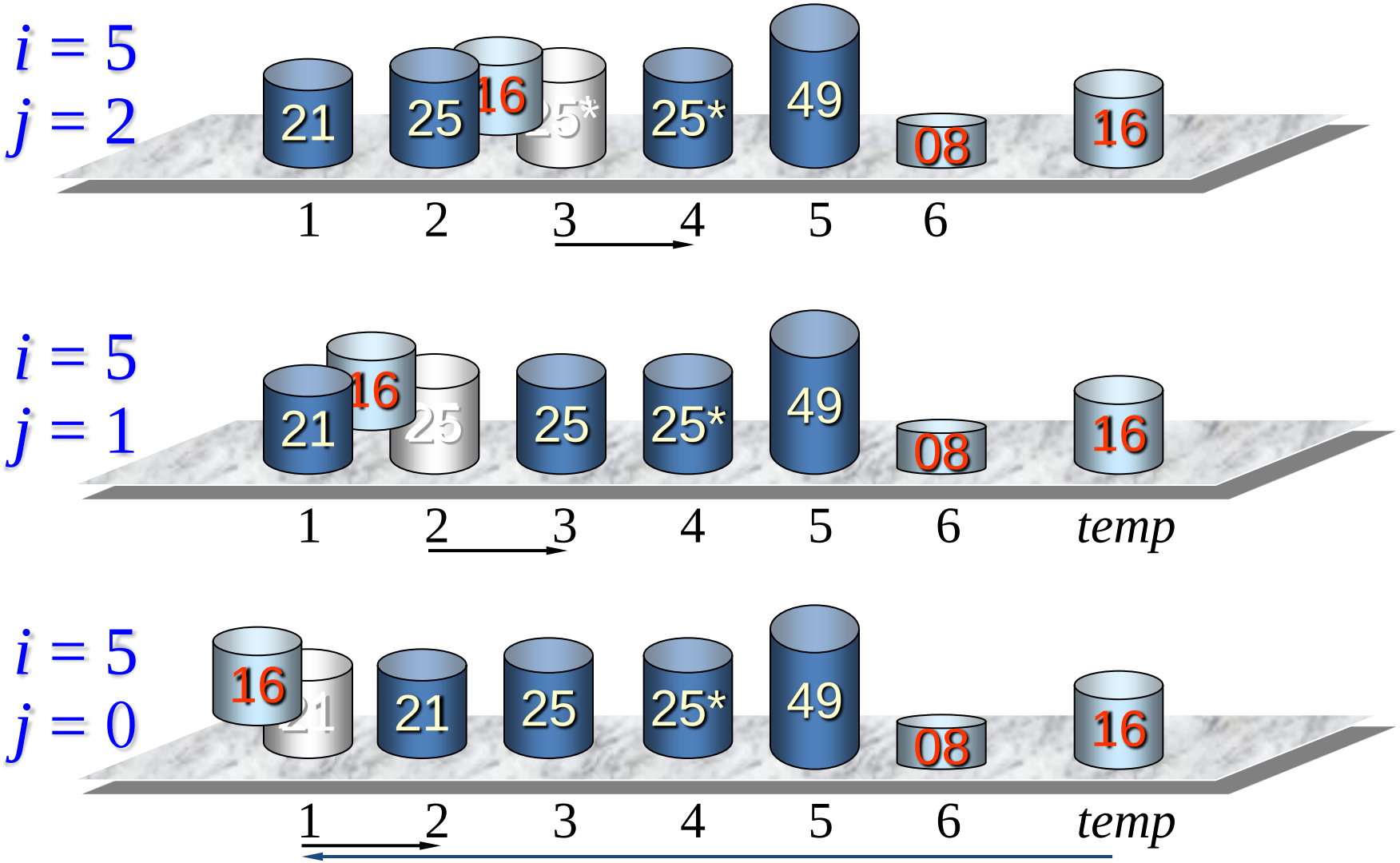
6.2 插入排序 (Insert Sorting)



6.2 插入排序 (Insert Sorting)



6.2 插入排序 (Insert Sorting)



6.2 插入排序 (Insert Sorting)

- **算法6.3：**
  - 第i个元素和前i-1个元素比较时，从后向前比较。
  - 算法是稳定的，因为总是找第一个小于等于元素。

```
void inssort(Elem rec[], int n ){
    Elem it;
    for ( i = 1; i < n; i++ ) {
        it = rec[i];
        for ( j = i; j > 0; j-- )
            if ( it.key() < rec[j-1].key() ) rec[j] = rec[j-1];
            else break;
        if  ( i != j ) rec[j] = it;
    }
}
```

6.2 插入排序 (Insert Sorting)

– 效率

- 设待排序对象个数为 n，则该算法的主程序执行n-1趟。
- 关键字比较次数和对象移动次数与关键字初始排列有关。
- 最好情况下:总的排序码比较次数为 $n-1$，对象移动次数为 0。
- 最坏情况下:第 i趟时第 i个对象必须与前面 i个对象都做关键字比较，并且每做1次比较就要做1次数据移动。则总关键字比较次数KCN和对象移动次数RMN分别为:

$$KCN = \sum_{i=2}^{n} i = (n+2)(n-1)/2 \approx n^2/2,$$

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

- 在平均情况下的关键字比较次数和对象移动次数约为 $n^2/4$。因此，直接插入排序的时间复杂度为 o(n^2)。
- 直接插入排序是一种稳定的排序方法。

6.2 插入排序 (Insert Sorting)

(2)折半插入（Binary Insertsort）

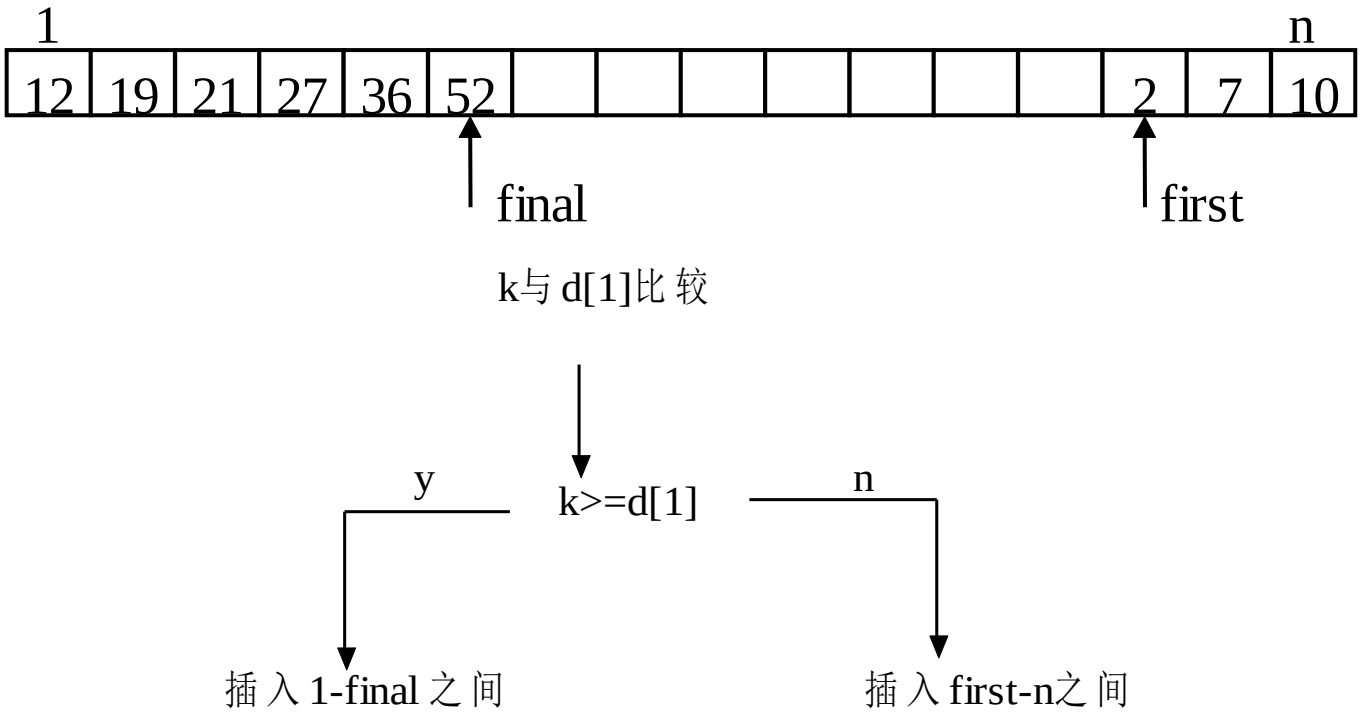
- 插入排序的基本操作是在一个有序表中，进行查找和插入，在有序表中的查找过程可以利用折半查找，即折半插入排序方法。
- 它是直接插入方法的改进，主要为了减少比较次数

- 算法6.4
  - 减少了比较次数，但没有减少移动次数。
  - 折半插入排序是一个稳定的排序方法
  - 折半搜索比顺序搜索查找快, 所以折半插入排序就平均性能来说比直接插入排序要快

6.2 插入排序 (Insert Sorting)

(3) 2-路插入排序

- 它是折半插入排序的改进，主要为了减少移动次数。
- 代价：n个数据空间(额外开销)
  - 将空间分成两个部分：前一部分存放的数据都大于第一个元素；后一部分存放的数据都小于第一个元素；两部分元素，从左至右有序。



6.2 插入排序 (Insert Sorting)

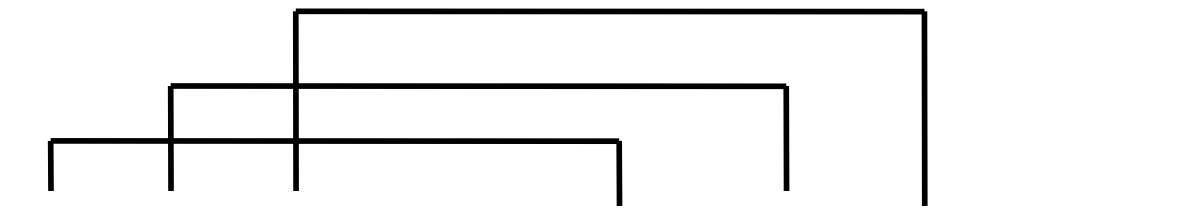
(4) 希尔排序

- 时间效率上改进较大的插入排序。基于以下两点：
  - 直接插入排序：如果待排序列基本满足排序条件时，元素的比较次数会大为减少。
  - 直接插入排序：在n较少时，效率较高。
- 希尔排序方法又称为缩小增量排序，或叫变步长子序列排序。
- 基本思想：设待排序对象序列有 n 个对象，首先取一个整数 $d_k < n$ 作为间隔，将全部对象分为 d_k 个子序列，所有距离为 d_k 的对象放在同一个子序列中，在每一个子序列中分别施行直接插入排序。然后缩小间隔 d_k，例如取 $d_{k+1} = \lceil d_k/2 \rceil$，重复上述的子序列划分和排序工作。使待排序列基本有序。
- 直到最后取 $d_k == 1$，将所有基本有序的对象放在同一个序列中排序为止。
- 给定一个d_k进行一次希尔排序，称为一趟希尔排序。
- d_k一般取素数，以避免各趟中子序列之间重叠。

6.2 插入排序 (Insert Sorting)

例如：原序列如下所示。 d_k=5、3、1

d_k=5



13 49

27 38

49 65

55 97

04 76

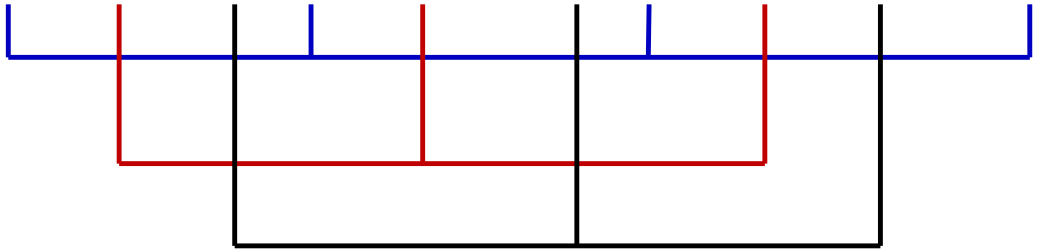
13 27 49 55 04

49 38 65 97 76

6.2 插入排序 (Insert Sorting)

d_k=3

13 27 49 55 04 49 38 65 97 76



13 38 55 76

04 27 65

49 49 97

13 04 49 38 27 49 55 65 97 76

d_k=1

04 13 27 38 49 49 55 65 76 97

6.2 插入排序 (Insert Sorting)

算法6.5 希尔排序

```
// Modify the algorithm of insertion sort:
static void inssort2(Elem rec[],  int n,  int start, int incr ){
    for (i = start + incr;  i  < n; i = i + incr ) {   // 控制插入元素
          Elem it = rec[i];
          for (  j = i; j >= incr; j = j - incr)   //找到合适位置
               if ( it < rec[j - incr])  rec[j] = rec[j - incr];
               else break;
          if  ( i != j ) rec[j] = it;
    }
}
static void Shellshort(Elem rec[], int n){
    for ( i = n / 2;  i > 2; i = i / 2 )
          for ( j = 0; j < i; j++)  inssort2(rec, n, j, i);  //控制变步长各子序列
     insort2( rec, n, 0, 1);
}
```

6.2 插入排序 (Insert Sorting)

D_k的选取： 没有其它公因子(除了1以外)，

最后一个必须是1。

Shell本人建议：
$$\begin{cases} h_k = n/2 \\ h_{k-1} = h_k / 2 \quad \text{for } i \geq 1 \end{cases}$$

时间复杂度： 执行时间是d_k增量序列的函数。

给出特殊步长序列下： $O(N^{3/2})$

如选择h为：
$$\begin{cases} h_1 = 1 \\ h_{i+1} = 3h_i + 1 \quad \text{for } i \geq 1 \end{cases}$$

则其复杂度为： $O(N^{5/4})$

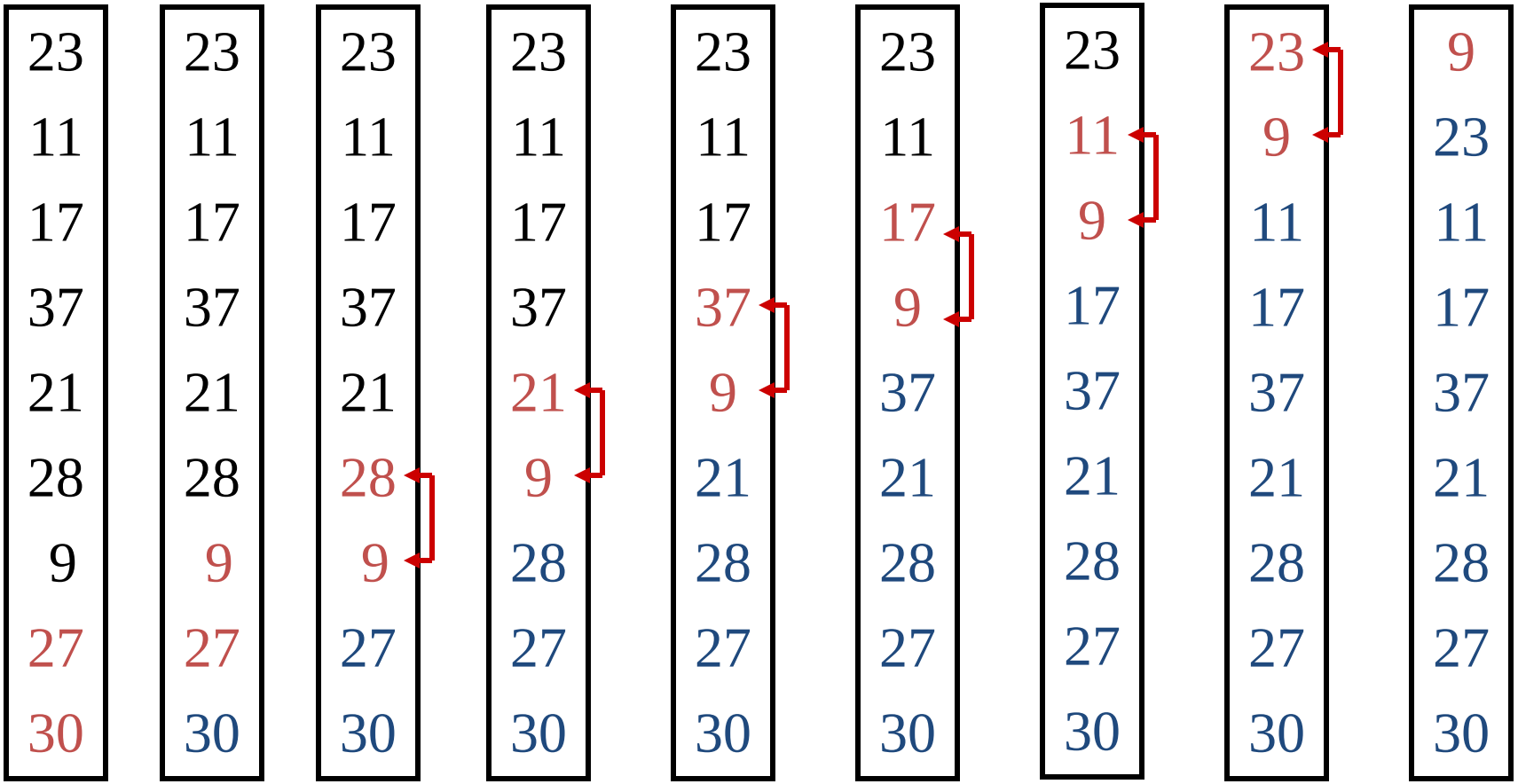
稳定性： Shell排序法是不稳定的。

6.3交换排序 (Exchange Sort)

- **基本思想：**两两比较待排序对象的关键字, 如果发现逆序(即排列顺序与要求次序正好相反)，则交换之。直到所有对象都排好序为止（**某趟没有发生元素交换**）。
- **(1) 冒泡排序 (Bubble Sort)**
  - **基本方法：**设待排序对象序列中的对象个数为 n。最多作 $n-1$ 趟，$i = 1, 2, \ldots, n-1$ 。在第 i 趟中从前向后，$j = 2, \ldots, n-i+1$，顺次两两比较V[$j-1$].*key*和V[j].*key*。如果发生逆序，则交换V[$j-1$]和V[j]。
  - 如果某趟排序过程中，没有交换操作，则说明序列已经有序，不需要进一步的处理。

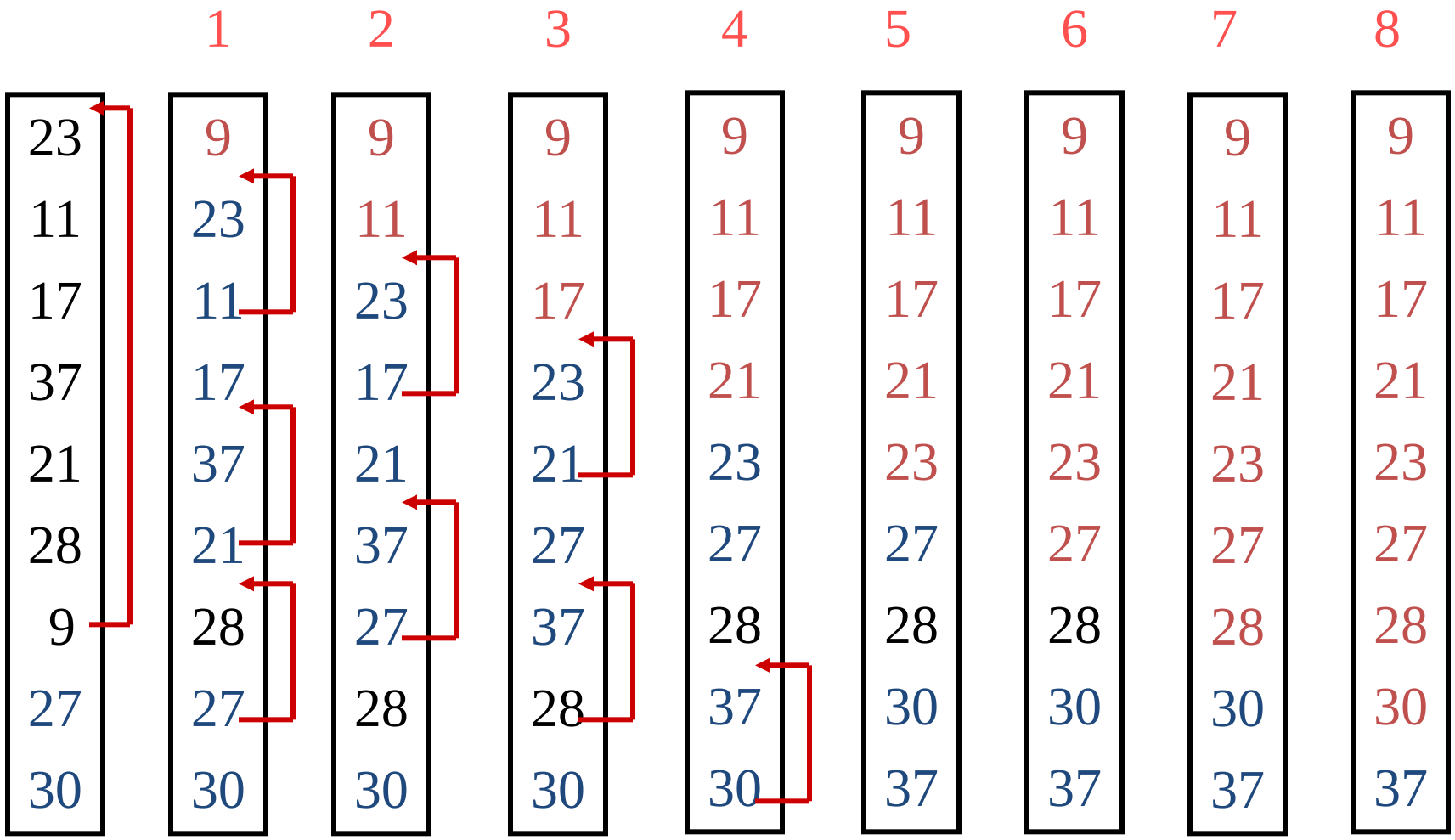
6.3交换排序 (Exchange Sort)

一趟冒泡排序过程:



6.3交换排序 (Exchange Sort)

完整冒泡排序过程:



6.3交换排序 (Exchange Sort)

算法6.6 一般冒泡排序算法

```
void bubsort(Elem rec[], int n ){
    Elem it;
    for (  i = 0; i < n - 1; i++ )
        for ( j = n - 1; j  > i; j-- )
            if ( rec[j].key() < rec[j-1].key() ){
                it = rec[j];
                rec[j] = rec[j-1];
                rec[j-1] = it;
            }
}
```

6.3交换排序 (Exchange Sort)

算法6.7 改进的冒泡排序算法

```
void bubsort(Elem rec[], int n ){
    Elem it;
    for (  i = 0; i < n - 1; i++ )
        for ( j = n - 1; j  > i; j-- )
            if ( rec[j].key() < rec[j-1].key() ){
                it = rec[j];
                rec[j] = rec[j-1];
                rec[j-1] = it;
            }
}
```

6.3交换排序 (Exchange Sort)

- 最好的情形是顺序：在对象的初始排列已经按关键字从小到大排好序时,此算法只执行一趟起泡,做n-1次关键字比较,不移动对象。
- 最坏的情形是逆序，算法执行n-1趟起泡。第i趟 $(1 \le i < n)$ 做 $n-i$ 次关键字比较, 执行 $n-i$ 次对象交换。这样在最坏情形下总的关键字比较次数KCN和对象移动次数RMN为：

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

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

- 起泡排序只需一个附加对象以实现对象值的对换: O(1)
- 起泡排序的时间复杂度：$O(n^2)$
- 起泡排序是一个稳定的排序方法。（相等时不交换）

6.3交换排序 (Exchange Sort)

(2) 快速排序 (Quick Sort)

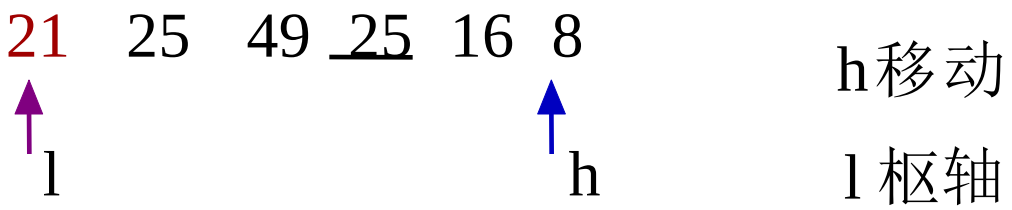
- 基本思想：任取待排序对象序列中的某个对象 (例如取第一个对象) 作为基准（枢轴），按照该对象的关键字大小，将整个对象序列划分为左右两个子序列：
  - 左侧子序列中所有对象的关键字都小于或等于基准对象的关键字
  - 右侧子序列中所有对象的关键字都大于基准对象的关键字。
- 每一趟排序，确定基准（枢轴）记录的位置。
- 对左侧子序列和右侧子序列分别做快速排序。

6.3交换排序 (Exchange Sort)

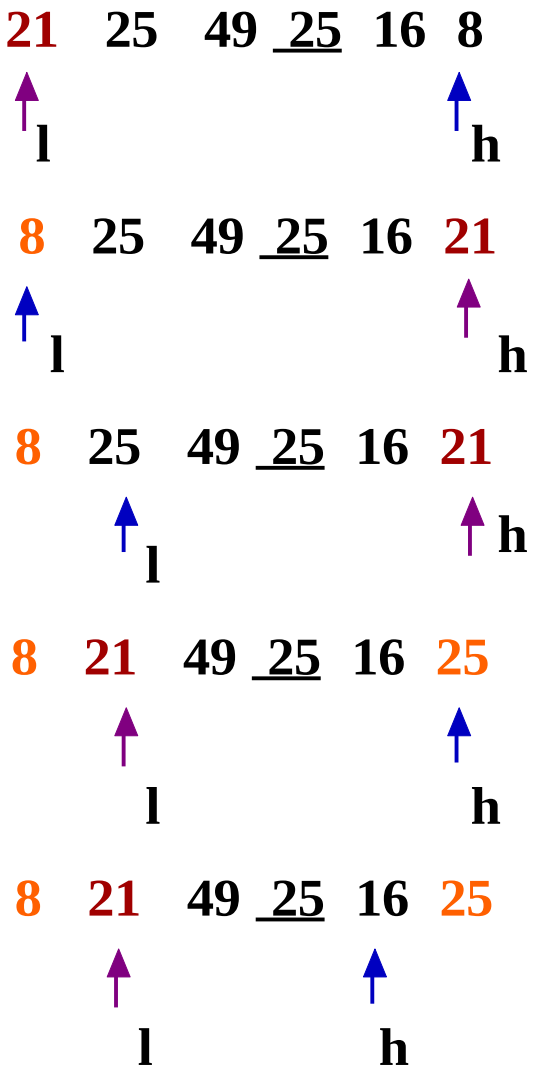
快速排序的算法：

利用两个指针，一个指针指向枢轴记录，另一个指针指示当前处理的记录，此指针向枢轴记录移动。在移动时，要与枢轴记录的关键字进行比较，保证处理过的记录满足：如果记录的关键字大于枢轴记录的关键字，此记录应该在枢轴记录的右边，否则在枢轴记录的左边。如果不满足条件，交换记录。

图示：l,h两个指针，交替充当指向枢轴的指针。



6.3交换排序（Exchange Sort）



h移动

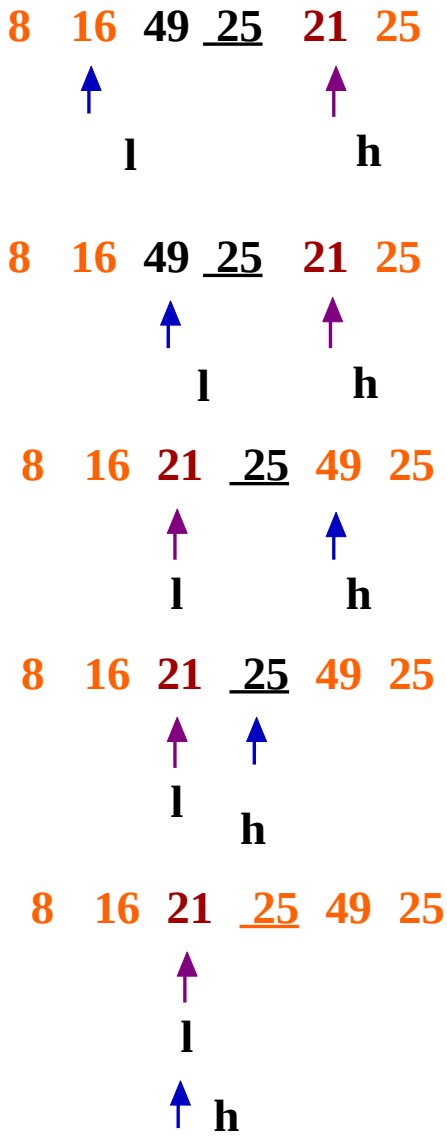
8<21, l与h交换记录，同时交换角色

l移动

25>21, l与h交换记录，同时交换角色

h移动

6.3交换排序（Exchange Sort）

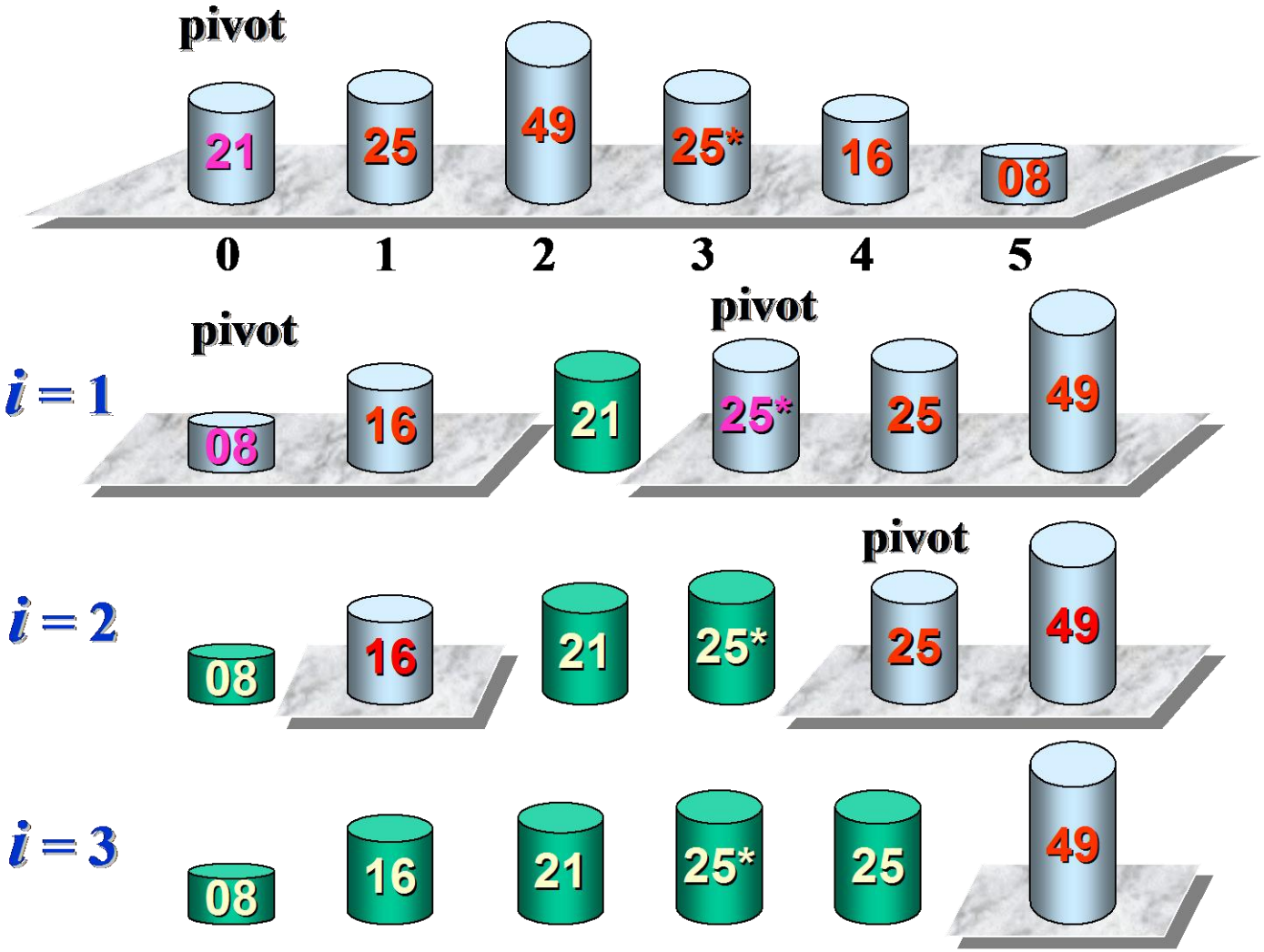


16<21, l与h交换记录，同时交换角色

l移动

49>21, l与h交换记录，同时交换角色

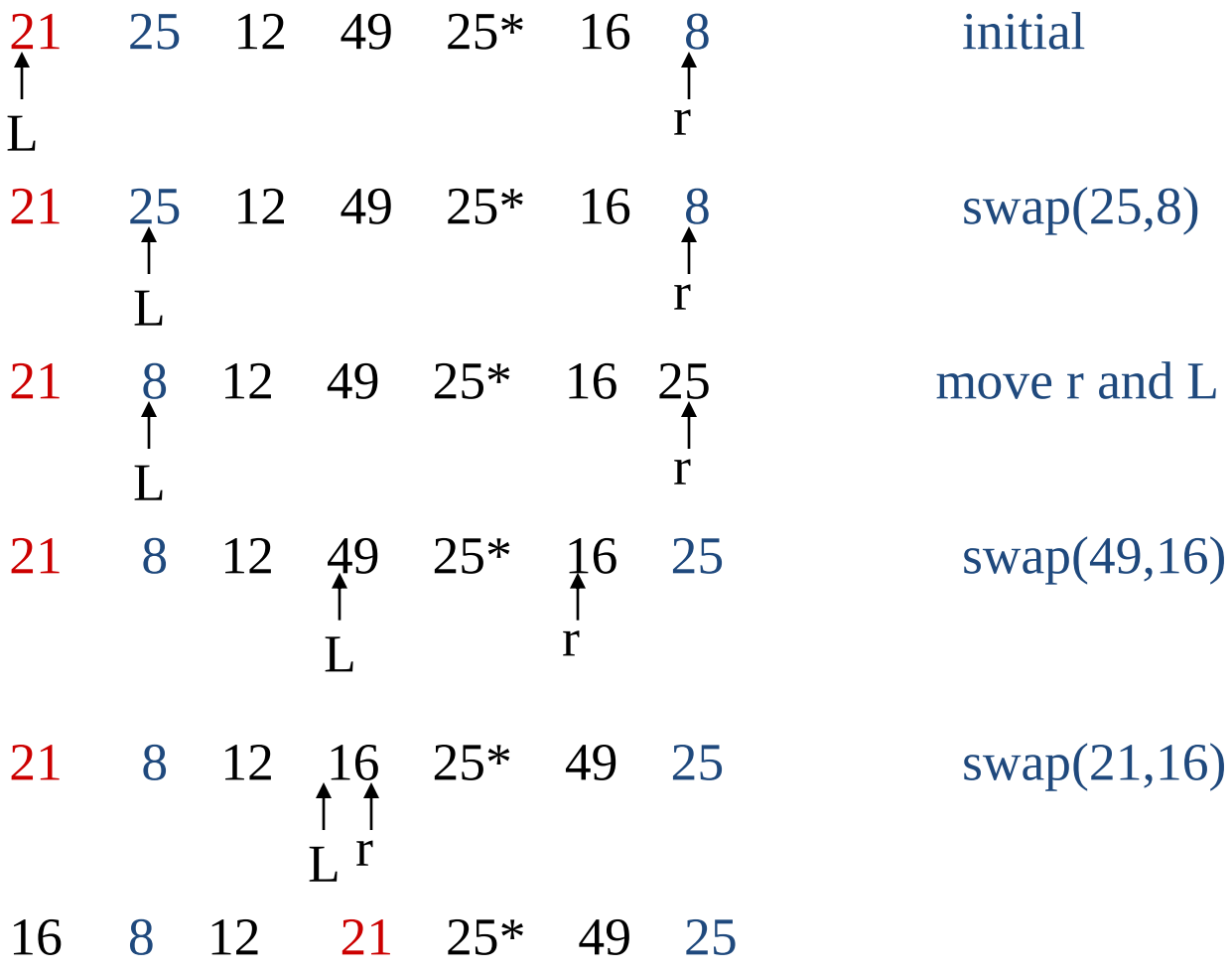
6.3交换排序 (Exchange Sort)



6.3交换排序 (Exchange Sort)

```
static int partition(Elem rec[], int l, int r){
    int k=l;
    while ( l<r ) {
        while ((l<r)&&(rec[r] >= rec[k])) r--;  //find smaller
        while ((l<r)&&(rec[l] <= rec[k])) l++; //find greater
         swap( rec[l], rec[r] );
    }
    swap( rec[k], rec[l]);   //swap pivot to mid
    return l;
}
```

6.3交换排序 (Exchange Sort)



6.3交换排序 (Exchange Sort)

```
static int qsort(Elem rec[], int i, int j){
    if  ( i < j ){
          k = partition(rec, i, j );
          qsort( rec, i, k - 1);
          qsort( rec, k + 1, j);
}
```

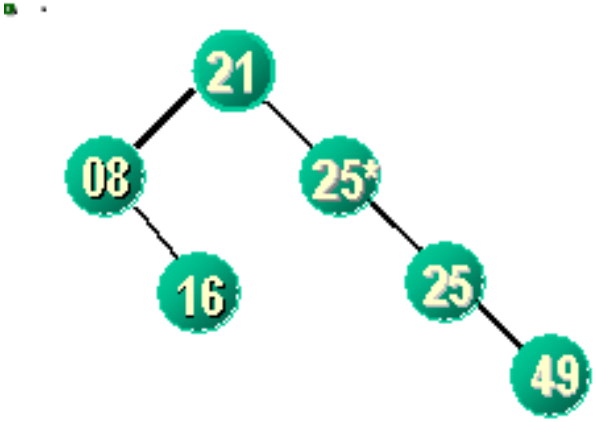
主程序调用：qsort(rec, 0, n-1).

So, on average, the given implementation of QuickSort is O(N log N) in both swaps and comparisons.

6.3交换排序 (Exchange Sort)

算法是递归的算法。

算法的时间复杂度为：**O(nlogn)**



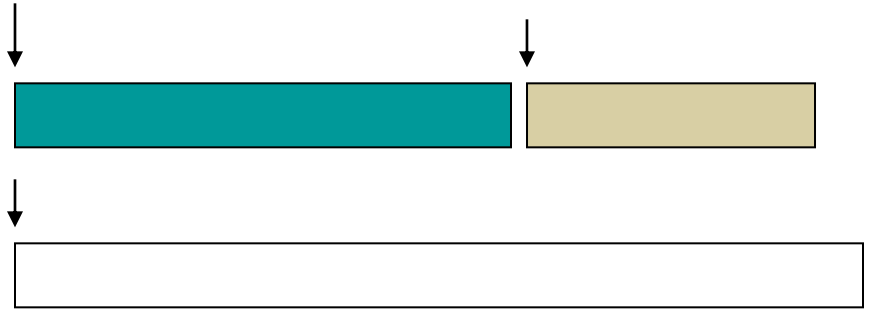
注意：

枢轴结点形成二叉树形状；

每一层就是一趟排序结果。

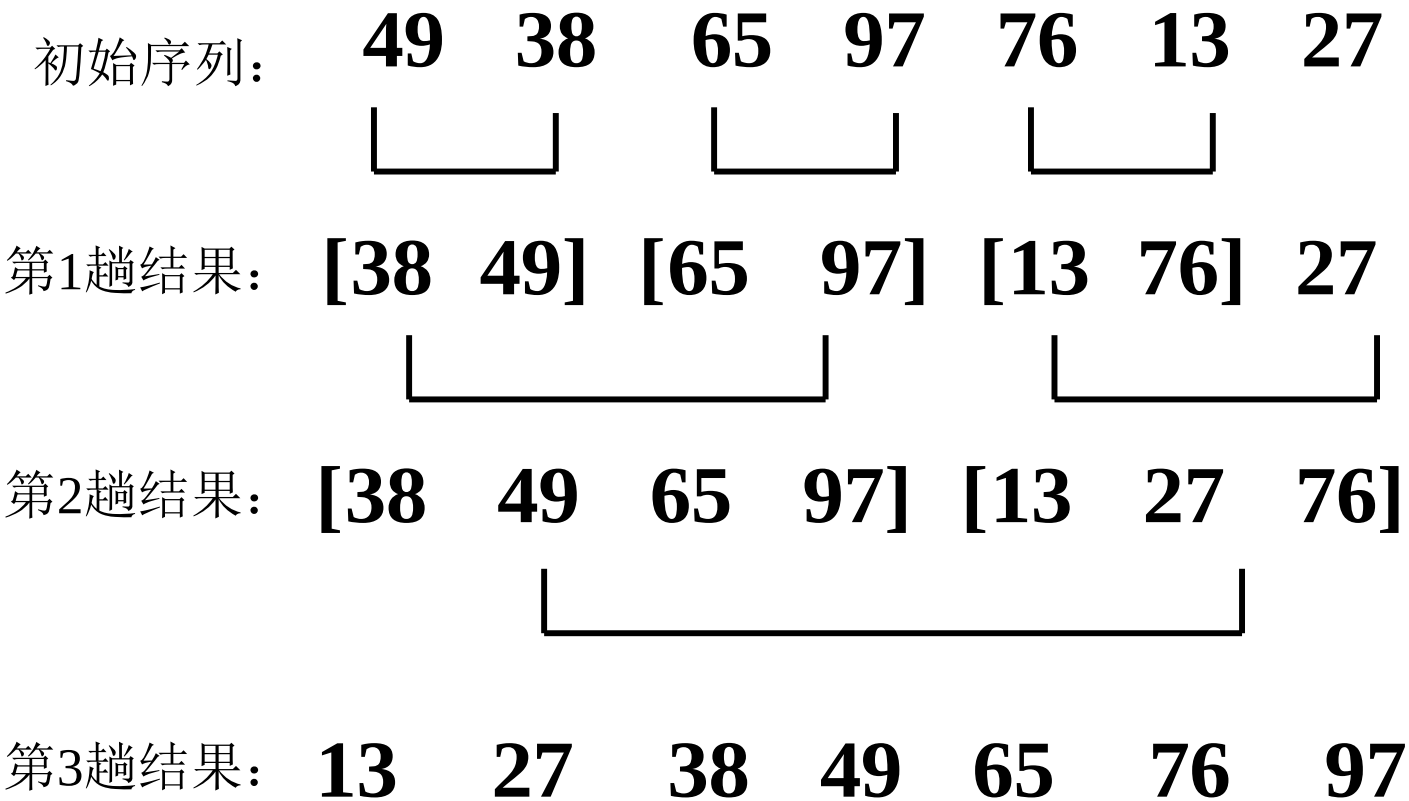
6.4 归并排序 (Merge Sort)

- 归并，是将两个或两个以上的有序表合并成一个新的有序表
- 将两个有序表合并成一个有序表时，利用两个辅助指针，依次扫描两个表，得到一个有序表，时间复杂度为O(n+m)，但是需要O(n+m)空间开销。



- 2-路归并
  - 相邻子序列两两归并。
  - 起始时，序列可以看成n个长度为1的子序列。

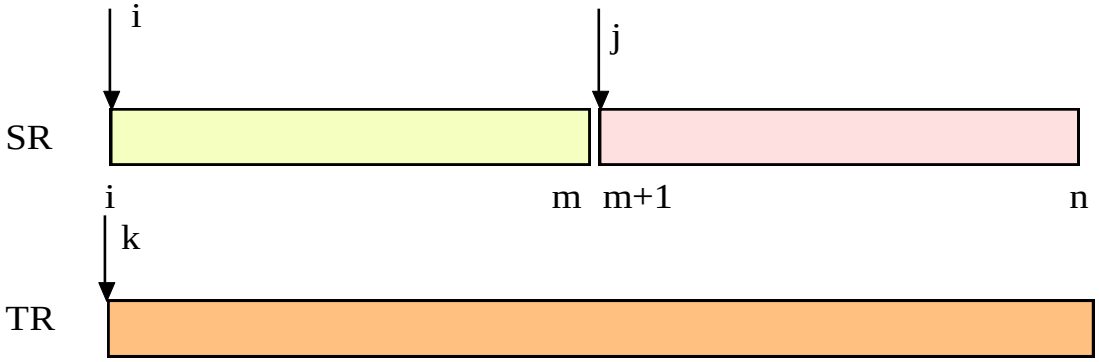
6.4 归并排序 (Merge Sort)



6.4 归并排序 (Merge Sort)

算法6.10 Merge——前后相邻的两个有序表合并成一个有序表。

将a[i..m]和a[m+1..n]归并成tmp[i..n]

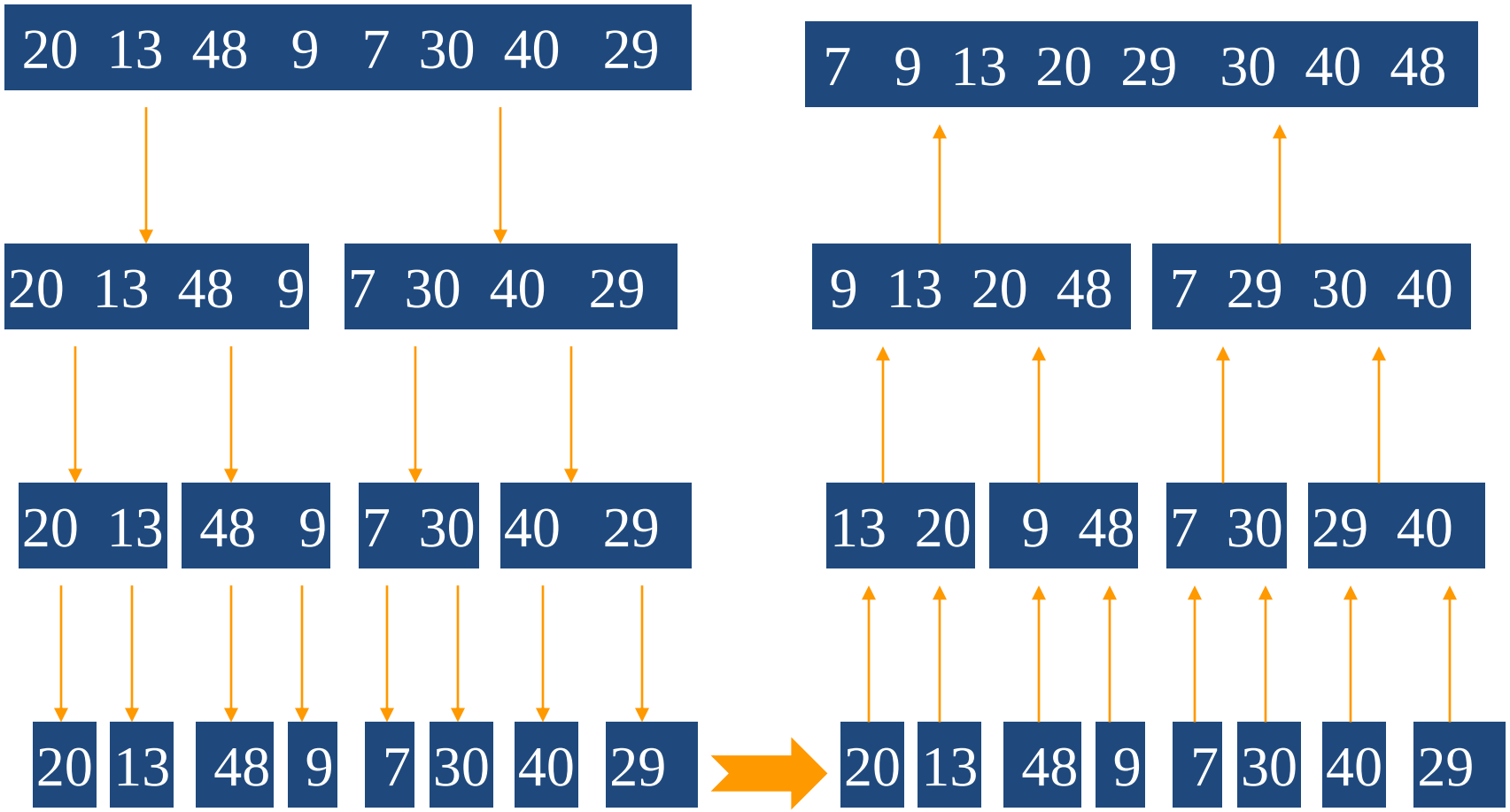


算法6.11 MergeSort——将某段序列归并排序(递归分解，逐级合并）

思路：由于将两个有序序列归并成一个有序序列，所以要先得到两个有序序列。将原有序列一分为二，每部分序列分别归并排序成有序序列，而后再归并成一个序列。各部分上的归并排序与整个序列上的相同。

将a[s..t] 归并排序成a[s..t]，需要长为t-s+1的辅助空间tmp。先在辅助空间tmp上，分别将a[s..m]和a[m+1..t]归并成有序序列。而后，将两段有序序列归并成一个有序序列。

Merge sort (example)



6.4 归并排序 (Merge Sort)

两段子序列归并成一个序列算法：

```
void merge(elem rec1[], elem rec2[], int low, int mid, int high){
    int i = low,  j=mid+1, k=low;
    while ( (i<=mid) && (j<=high) )
        if (rec1[i]  <= rec1[j] )  rec2[k++] = rec1[i++];
        else rec2[k++] = rec1[j++];
    while (i <= mid)    rec2[k++] = rec1[i++];
    while (j <= high)   rec2[k++] = rec1[j++];
}
```

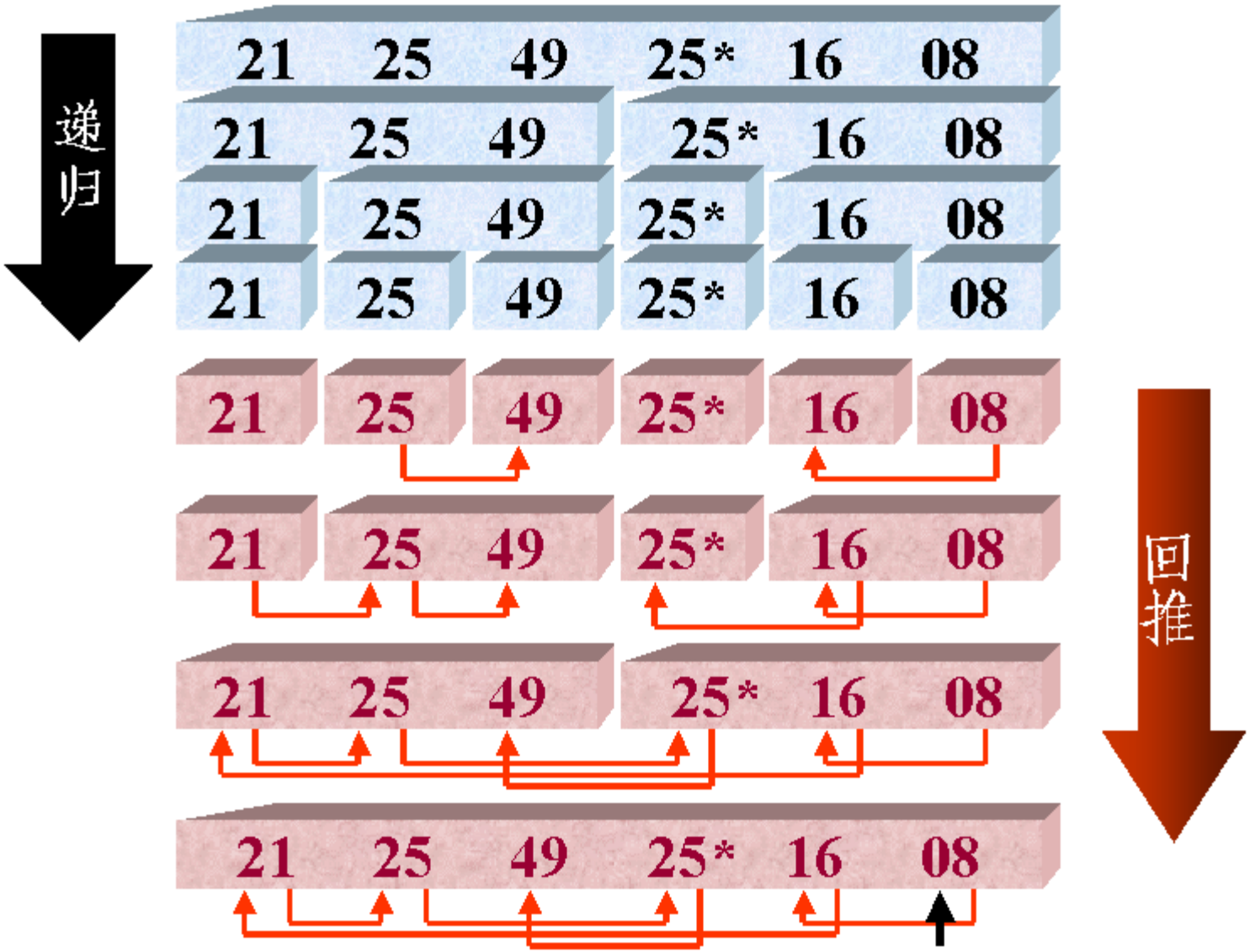
6.4 归并排序 (Merge Sort)

归并排序主程序：

```
void mergesort(elem rec[], elem temp[], int low, int high){
    int mid = ( low+high)/2;
    if (low == high) return;  //only one record is a sorted list
    mergesort(rec, temp, low, mid);
    mergesort(rec, temp, mid+1, high);
    for (i = low; i <= high; i++)   // rec[]=>temp[]
        temp[i] = rec[i];
    merge(temp, rec, low, mid, high);  //result in rec[]
}
```

主程序调用：mergesort(rec,temp 0, n-1).

6.4 归并排序 (Merge Sort)



6.4 归并排序 (Merge Sort)

递归分解的深度是： log N

每一趟归并过程是：N

归并排序的时间复杂度：O(Nlog N)

采用辅助存储，故空间复杂度：O(N)

归并排序算法是稳定的。

6.5 基数排序

- 前面的排序方法主要通过关键字值之间的比较和移动，而基数排序不需要关键字之间的比较，利用多关键字排序的思想进行排序。
- 以扑克牌排序为例。每张扑克牌有两个“关键字”：花色和面值。其有序关系为：(52张)
  - 花色：♣ < ♦ < ♥ < ♠
  - 面值：2 < 3 < 4 < 5 < 6 < 7 < 8 < 9 < 10 < J < Q < K < A

  可以把所有扑克牌排成以下次序：

  ♣ 2, …, ♣ A, ♦ 2, …, ♦ A, ♥ 2, …, ♥ A, ♠ 2, …, ♠ A 花色相同的情况下，比较面值。

6.5 基数排序

花色在排序中起的作用比面值大，可以看作主位关键字，面值是次位关键字。

- 多关键字排序
  - 一般情况下，假定有一个 n 个对象的序列 $\{V_0, V_1, \ldots, V_{n-1}\}$，且每个对象 V_i 中含有 d 个关键字。

$$\left(K_i^1, K_i^2, \cdots, K_i^d\right)$$

如果对于序列中任意两个对象 V_i 和 V_j $(0 \le i < j \le n-1)$ 都满足：

$$\left(K_i^1, K_i^2, \cdots, K_i^d\right) < \left(K_j^1, K_j^2, \cdots, K_j^d\right)$$

  - 则称序列对关键字 $(K^1, K^2, \ldots, K^d)$ 有序。其中，K^1 称为最高位排序码，K^d 称为最低位排序码。
- 如果关键字是由多个数据项组成的数据项组，则依据它进行排序时就需要利用多关键字排序

6.5 基数排序

- 实现多关键字排序有两种常用的方法
  - 最高位优先MSD (Most Significant Digit first)
  - 最低位优先LSD (Least Significant Digit first)

  十进制数“比较”可以看作是一个多关键字排序。
- 最高位优先法通常是一个递归的过程：
  - 先根据最高位关键字K^1排序, 得到若干对象组, 对象组中各对象都有相同排序码K^1。
  - 再分别对每组中对象根据关键字K^2 进行排序, 按 K^2 值的不同, 再分成若干个更小的子组, 每个子组中的对象具有相同的 K^1和 K^2值。
  - 依此重复, 直到对排序码K^d完成排序为止。
  - 最后, 把所有子组中的对象依次连接起来，就得到一个有序的对象序列。

6.5 基数排序

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

按百位排序		按十 位排序
008,063,083	⇨	008 063 083
109,184	⇨	109 184
269,278	⇨	269 278
589		589
930		930

6.5 基数排序

最低位优先法首先依据最低位排序码K^d对所有对象进行一趟排序，再依据次低位排序码K^{d-1}对上一趟排序的结果再排序，依次重复，直到依据排序码K^1最后一趟排序完成，就可以得到一个有序的序列。使用这种排序方法对每一个排序码进行排序时，不需要再分组，而是整个对象组都参加排序。

278，109，063，930，184，589，269，008，083

按个位排序

930，063，083，184，278，008，109，589，269

按十位排序

008，109，930，063，269，278，083，184，589

按百位排序

008，063，083，109，184，269，278，589，930，

6.5 基数排序

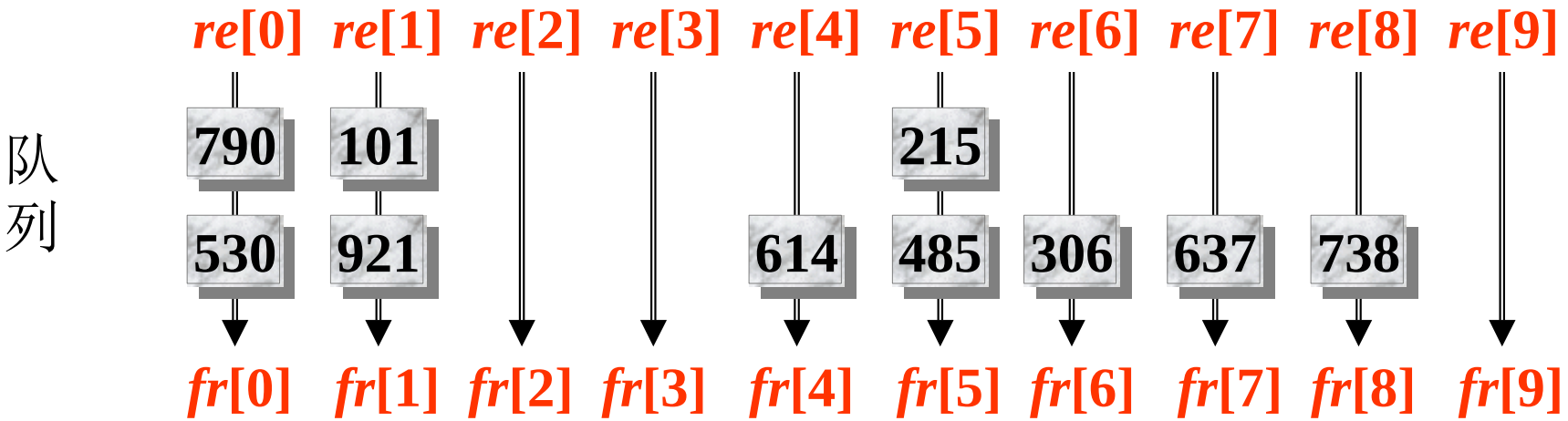
- 链式基数排序
  - 利用“分配”和“收集”对关键字进行排序
  - 先决条件：
    - 知道各级关键字的主次关系（收集）
    - 知道各级关键字的取值范围（分配）
  - 过程
    - 首先对低位关键字排序，各个记录按照此位关键字的值‘分配’到相应的序列里。关键字有多少种取值，就有多少个序列。
    - 按照序列对应的值的大小顺序，依次从各个序列中将记录‘收集’，收集后的序列按照此位关键字有序。
    - 在此基础上，对前一位关键字进行排序。

6.5 基数排序

基数排序的“分配”与“收集”过程 第一趟

614 → 738 → 921 → 485 → 637 → 101 → 215 → 530 → 790 → 306

第一趟分配（按最低位 $i = 3$）



第一趟收集

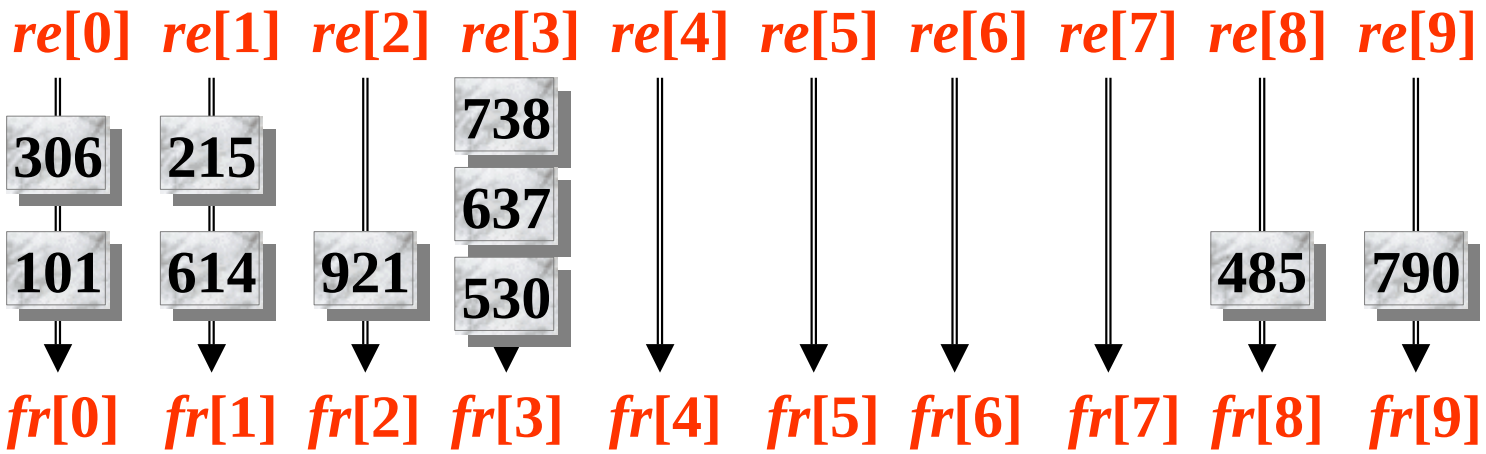
530 → 790 → 921 → 101 → 614 → 485 → 215 → 306 → 637 → 738

6.5 基数排序

基数排序的“分配”与“收集”过程 第二趟

530 → 790 → 921 → 101 → 614 → 485 → 215 → 306 → 637 → 738

第二趟分配（按次低位 $i = 2$）



第二趟收集

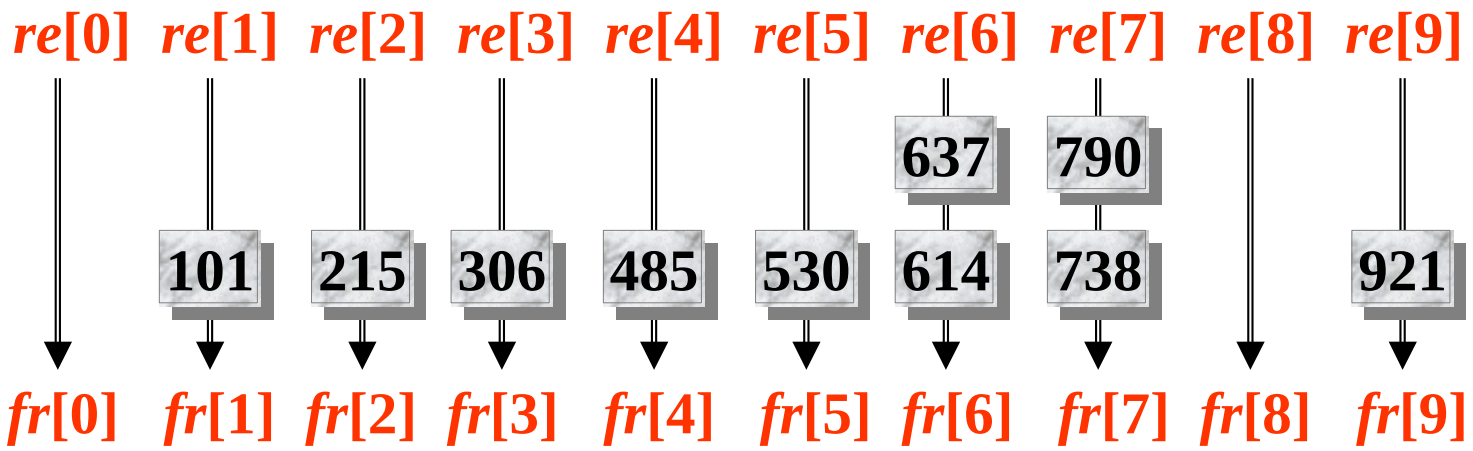
101 → 306 → 614 → 215 → 921 → 530 → 637 → 738 → 485 → 790

6.5 基数排序

基数排序的“分配”与“收集”过程 第三趟

101 → 306 → 614 → 215 → 921 → 530 → 637 → 738 → 485 → 790

第三趟分配（按最高位 $i = 1$）



第三趟收集

101 → 215 → 306 → 485 → 530 → 614 → 637 → 738 → 790 → 921

6.5 基数排序

分配与收集算法见书上算法6.12和算法6.13

算法的数据结构：

- 记录——多位关键字；在分配到的序列中，记录之间有序列关系；在收集后的序列中，记录之间有序列关系。
- 对于分配时每个序列，只要给定头、尾，就可以确定特定的序列。
- 收集的时候，只是序列依次头尾相连。
- 分配时，记录之间的序列关系变化。
- 每次收集和分配时，序列的个数相同，但每次序列中的记录不同。

让记录自己保存序列关系。即存储它在某个时刻时，在序列中的后继记录（Next)。

用头、尾指针表示一个序列。

6.5 基数排序

278,109,063,930,580,184,505,269,008,083

- 若每个关键字有 d 位, 需要重复执行d 趟“分配”与“收集”。每趟对 n 个对象进行“分配”，对radix个队列进行“收集”。总时间复杂度为O (d (n+radix))。
- 若基数radix相同，对于对象个数较多而关键字位数较少的情况, 使用链式基数排序较好。
- 基数排序需要增加n+2radix个附加链接指针。
- 基数排序是稳定的排序方法。

6.6 各种排序方法的比较

各种排序算法的复杂度比较总结见书上表6.6。

排序方法	比较次数		移动次数		稳定性	附加存储	
	最好	最差	最好	最差		最好	最差
直接插入排序	n	n^2	0	n^2	√	1	
折半插入排序	$n\log_2 n$		0	n^2	√	1	
起泡排序	n	n^2	0	n^2	√	1	
快速排序	$n\log_2 n$	n^2	$n\log_2 n$	n^2	×	$\log_2 n$	n
归并排序	$n\log_2 n$		$n\log_2 n$		√	n	

基数排序： 时间复杂度O(d(n+rd)),空间复杂度O(rd)， 稳定排序。

Thank you