

# 数据结构与算法

## 第八章 排序(一)

王 昭

北京大学信息科学技术学院

wangzhao@pku.edu.cn



# 排序

- **排序**是数据处理中经常使用的一种重要运算，如何进行排序，特别是高效率排序，是计算机领域的重要课题之一。
- 研究问题：
  - 如何进行**排序**
  - 如何进行**高效率排序**



# 内容提要

- 排序的基本概念
- 选择排序
- 交换排序
- 排序函数



# 排序的基本概念

- 假设条件：
  - 假定排序的对象是由一组**记录**组成的文件（**序列**），记录由若干字段组成，以**排序码**为依据排序。
- **排序码**-记录中的一个(或多个)字段
  - 关键码-此时按关键码排序
  - 不是关键码-则可能有多个记录具有相同的排序码，排序的结果不唯一
- 排序码的类型可以是整数类型，也可以是字符类型等
- 本章假设排序码为整型。

学号	姓名	性别	英语	数学
20	李泉	男	100	81
22	王怡	女	90	100
29	张三	男	87	81
41	马丁	男	68	99
78	赵明	男	89	90

# 排序

- **排序**: 设 $\{R_0, R_1, \dots, R_{n-1}\}$ 是由 $n$ 个记录组成的文件（序列）， $\{K_0, K_1, \dots, K_{n-1}\}$ 是排序码集合，排序是将记录按排序码**非递增（或非递减）**的次序排列。
- 设排序后的有序序列为 $\{R'_0, R'_1, \dots, R'_{n-1}\}$   
对应的排序码序列为  $\{K'_0, K'_1, \dots, K'_{n-1}\}$
- 排序码的顺序
  - 其中 $K'_0 \leq K'_1 \leq \dots \leq K'_{n-1}$ ，称为**非递减序**
  - 或 $K'_0 \geq K'_1 \geq \dots \geq K'_{n-1}$ ，称为**非递增序**

学号	姓名	性别	英语	数学
20	李泉	男	100	81
22	王怡	女	90	100
29	张三	男	87	81
41	马丁	男	68	99
78	赵明	男	89	90

学号	姓名	性别	英语	数学
20	李泉	男	100	81
29	张三	男	87	81
78	赵明	男	89	90
41	马丁	男	68	99
22	王怡	女	90	100

学号	姓名	性别	英语	数学
20	李泉	男	100	81
22	王怡	女	90	100
78	赵明	男	89	90
29	张三	男	87	81
41	马丁	男	68	99



# 正序 vs. 逆序

不一定说是递增还是递减

- “正序”序列：待排序序列正好符合排序要求。
- “逆序”序列：把待排序序列逆转过来，正好符合排序要求。
- 例如，要求非递减序列

○ 06      10      12      26      36

正序！

○ 36      26      12      10      06

逆序！



# 排序的分类

- 按**稳定性**：稳定排序和不稳定排序
- 按数据存储介质：内部排序和外部排序
- 按辅助空间：原地排序和非原地排序
- 按主要操作：比较排序和基数排序



稳定：  
相同的编码的相对次序不变

# 排序的稳定性

- 排序的稳定与不稳定：在待排序的文件中，如果存在多个排序码相同的记录，
  - 经过排序后记录的相对次序保持不变，则这种排序方法称为是“稳定的”

● 16    12    16'    07    31    直接插入排序

● 07    12    16    16'    31

- 否则是“不稳定的”（只需列举出一组关键字说明不稳定即可）

● 16    12    16'    07    31    直接选择排序

● 07    12    16'    16    31





先用数学成绩排序，然后用总分排序，并且需要具有稳定性

这样，总分相同的，按照数学成绩来排序

# 排序的稳定性

- 排序的稳定性只对**结构(struct)类型**数据排序有意义
- 例如：已知n个学生的信息（姓名、学号、语文、英语、数学、总分）  
要求：1.编程统计每个人的**总分**，并按从高到低排序  
2.总分相同的情况下，**数学**成绩高的排在前面

A	B	C	D	E	F
姓名	学号	语文	英语	数学	总分
李 玫	21009	96	95	97	288
张 虹	21005	78	80	90	248
孙宏宇	21008	90	92	90	272
李 霞	21004	86	79	86	251
张 鹏	21001	80	90	81	251
刘 薇	21003	72	79	80	231
王 旭	21007	89	86	80	255
丁一凡	21002	90	74	75	239

A	B	C	D	E	F
姓名	学号	语文	英语	数学	总分
李 玫	21009	96	95	97	288
孙宏宇	21008	90	92	90	272
王 旭	21007	89	86	80	255
李 霞	21004	86	79	86	251
张 鹏	21001	80	90	81	251
张 虹	21005	78	80	90	248
丁一凡	21002	90	74	75	239
刘 薇	21003	72	79	80	231

# 内排序vs外排序

- 按数据存储介质：

- 内部排序： 数据量不大，数据在内存  
是把待排数据元素全部调入内存中进行的排序。

- 外部排序： 数据量较大，数据在外存  
因数量太大，把数据元素分批导入内存，排好序后再分批导出到磁盘和磁带外存介质上的排序方法。

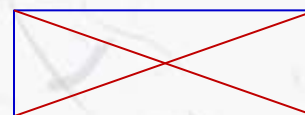
# 原地VS非原地排序

- 按辅助空间可分为

- 原地排序：辅助空间用量为 $O(1)$ 的排序方法



原始数据存储空间



辅助空间

- 非原地排序：辅助空间用量超过 $O(1)$ 的排序方法  
需要申请辅助空间

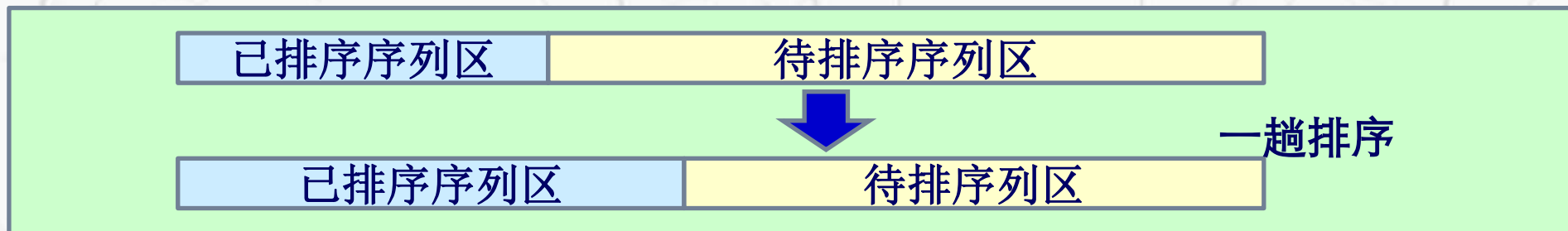
# 按主要操作分类

● 基本思想 → 排序过程 → 算法实现 → 时间和空间性能的分析  
→ 各种排序方法的比较和选择。

● 比较排序方法：用比较元素大小的方法

□ 选择类：每趟从待排序记录序列中“选择”关键字最小或最大的记录加入到已排序序列中

□ 交换类：两两比较待排序记录的排序码，“交换”不满足顺序要求的偶对，从而得到其中排序码最大或最小的记录，把它加入到已排序序列中



一趟排序，增加记录已排序（有序）序列的长度



# 排序算法的评价

- 评价排序算法好坏的标准
  - **时间复杂度**: 它主要是分析记录关键字的比较次数和记录的移动次数
  - **空间复杂度**: 算法中使用的内存辅助空间的多少
  - **稳定性**
  - **算法本身的复杂程度**也是考虑的一个因素
- 注: **排序的时间开销**是算法好坏的最重要的标志
- 给出各算法的主要参数, 一般分**最坏**、**最好**或**平均**三种情况估算, 在应用时要根据情况计算实际的开销, 选择合适的算法
- 执行排序算法所需的附加空间一般不大, 一般只给出结论。





# 比较排序的基本操作

- 比较两个关键字的大小(必须)
- 将一个记录从一个位置移动到另一个位置(不必须，可通过存储方式来避免)



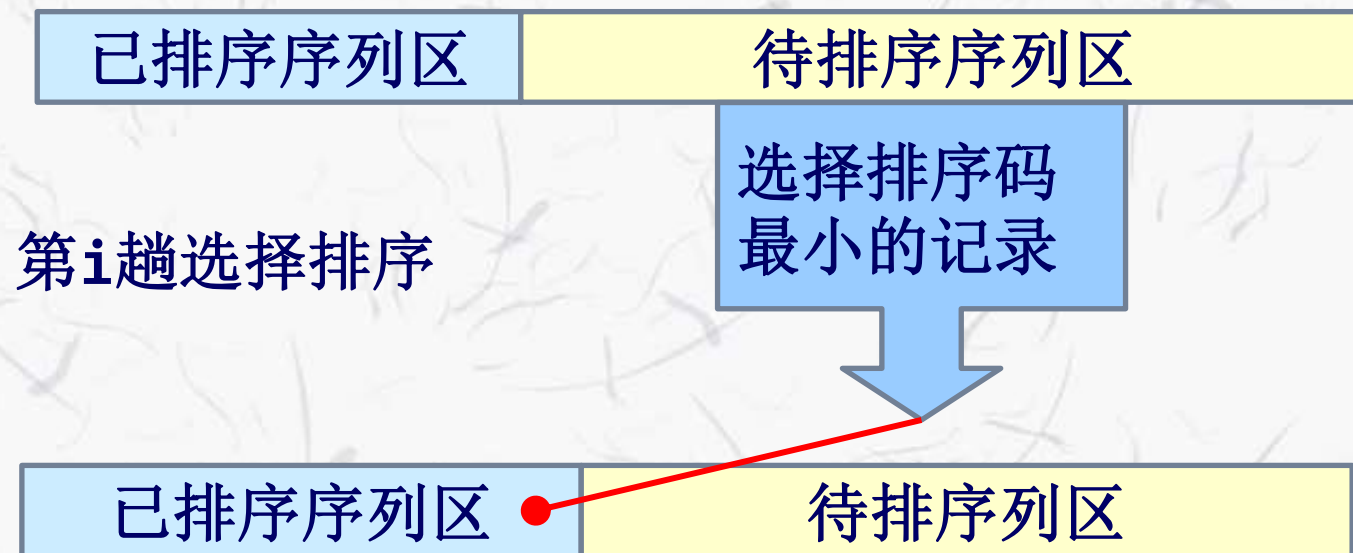
# 内容提要

- 排序的基本概念
- 选择排序 ←
- 交换排序
- 排序函数



# 选择排序

- **思想**：每趟从待排序的记录序列中“选择关键字最小的记录”放置到已排序表的最前位置，直到全部排完。
- **关键问题**：在剩余的待排序记录序列中找到最小关键码记录。
- **方法**：直接选择排序、堆排序



# 直接选择排序之各趟排序

	0	1	2	3	4	5
初始数据	21	25	49	25*	16	08
i=0	21	25	49	25*	16	08
i=1	08	25	49	25*	16	21
i=2	08	16	49	25*	25	21
i=3	08	16	21	25*	25	49
i=4	08	16	21	25*	25	49
结果	08	16	21	25*	25	49

最小者08，交换21，08

最小者16，交换25，16

最小者21，交换49，21

最小者25\*，无交换

最小者25，无交换



# 直接选择排序

- 方法是：
  - 首先在所有记录中选出排序码最小的记录，与第一个记录交换
  - 然后在其余的记录中再选出排序码最小的记录与第二个记录交换
  - 以此类推，直到所有记录排好序
- 动画演示
- 21, 25, 49, 25, 16, 08





# □ 直接选择排序算法

```
void selectSort( int * a, int n )    // 直接选择排序，求递增序列
{
    int i, j, k; int temp;
    for( i = 0; i < n-1; i++ ) /* 做n-1趟选择排序,i控制选择的遍数*/
                                /*i也是无序区的最前面的位置*/
    {
        k=i;    //k记录每趟待排序序列区最小元素的下标
        for(j=i+1; j<n; j++) /*选择排序码最小的记录a[k], n-i次比较*/
            if(a[j]<a[k]) k=j;
        if(k!=i) /*记录a[k]与a[i] 互换，将最小元素交换到无序区最前面*/
        {
            temp=a[i];    只交换一次
            a [i]= a [k];
            a [k]=temp;
        }
        // 只要比无序区的第一个小，那么就交换，
        // 这样就只用比较n-i次
    }
}
```

	0	1	2	3	4	5
初始数据	21	25	49	25*	16	08
i=0	21	25	49	25*	16	08
i=1	08	25	49	25*	16	21
i=2	08	16	49	25*	25	21
i=3	08	16	21	25*	25	49
i=4	08	16	21	25*	25	49
结果	08	16	21	25*	25	49

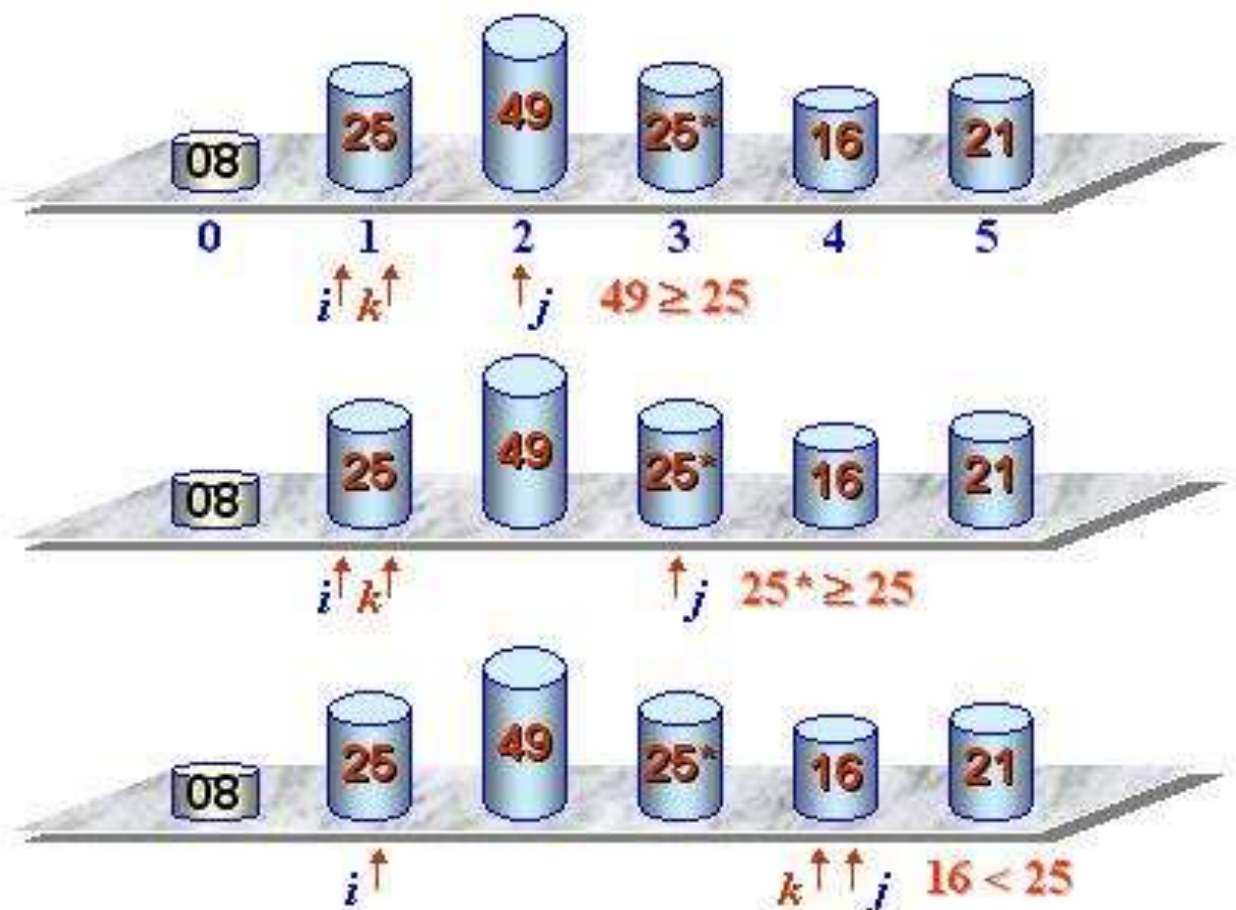
## □ 直接选择排序算法-教材

```
void selectSort(SortObject * pvector) /* 按非递减序进行直接选择排序 */
{
    int i, j, k;
    RecordNode temp;
    for( i = 0; i < pvector->n-1; i++ ) /* 做n-1趟选择排序*/
    {
        k=i;
        for(j=i+1; j<pvector->n; j++) /*在无序区内找出排序码最小的记录Rk */
            if(pvector->record[j].key<pvector->record[k].key) k=j;
        if(k!=i) /*记录Rk 与Ri 互换*/ 可以对结构体进行排序
        {
            temp=pvector->record[i];
            pvector->record [i]= pvector->record [k];
            pvector->record [k]=temp;
        }
    }
}

typedef struct
{
    KeyType key; /* 排序码字段 */
    DataType info; /* 记录的其他字段 */
}RecordNode;

typedef struct
{
    int n; /* n为文件中的记录个数
    RecordNode *record;
}SortObject;
```

### $i=1$ 时选择排序的过程

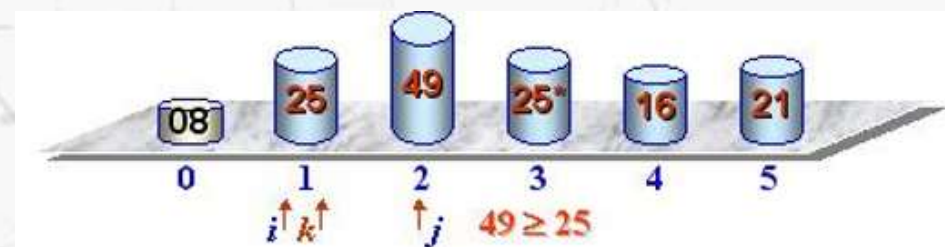


$k$  指示当前序列中最小者

# 直接选择排序性能分析

- 直接选择排序的**比较次数与记录的初始状态无关**。
- 第*i*趟排序：从**第*i*个记录开始**，顺序比较选择最小关键码记录需要 **$n-i$ 次比较**。

- 总的比较次数：
$$\sum_{i=1}^{n-1} (n-i) = \frac{n(n-1)}{2}$$



- **移动次数**： $M_{\min} = 0$ （初始为正序时）
- 最多移动次数： $M_{\max} = 3(n-1)$ （**每趟1次交换，3次移动完成**）

例如： 7   1   2   3   4   5   6

- 时间复杂度： $T(n)=O(n^2)$ ，
- 辅助空间1个记录单位：Temp，
- 稳定性：**不稳定的排序**。  
因为进行了直接交换

平均时间复杂性	辅助空间	稳定性
$O(n^2)$	$O(1)$	不稳定



# 内容提要

- 排序的基本概念
- 选择排序
- 交换排序
- 排序函数





# 交换排序

- 逆序：  $i < j$  时，  $a[i] > a[j]$
- 交换排序的基本思想：
  - 若  $i < j$  时， 而  $a[i] > a[j]$ ， 则交换  $a[i]$  与  $a[j]$ 。
  - 当对任何  $i < j$ ，  $a[i] \leq a[j]$  时， 排序完成。

$a[0]$	$a[1]$	$a[2]$	...	$a[i]$	...	$a[j]$	...	$a[n-1]$
3	4	7	8	9	11	13	15	17



# 冒泡排序

- 冒泡排序的基本方法

- 两两比较待排序相邻记录的排序码，交换不满足顺序要求的偶对，直到无逆序对为止。
- 每一趟冒泡排序得到排序码最大或最小的记录，把它加入到已排序（有序）序列中。

- 两种扫描方法：

- 下降法：自上而下地扫描，排序码最大的记录下降到底部
- 上升法：自下而上地扫描，排序码最小的记录上升到数组顶部



# 冒泡排序

- 方法

- ① 先将序列中的第一个记录 $R_0$ 与第二个记录 $R_1$ 比较，若前者大于后者，则两个记录交换位置，否则不交换
  - ② 然后对新的第二个记录 $R_1$ 与第三个记录 $R_2$ 作同样的处理
  - ③ 依次类推，直到处理完第 $n-1$ 个记录和第 $n$ 个记录
- 从 $(R_0, R_1)$ 到 $(R_{n-2}, R_{n-1})$ 的 $n-1$ 次比较和交换过程称为一趟冒泡。
  - 经过这趟冒泡， $n$ 个记录中最大者被安置在第 $n$ 个位置上

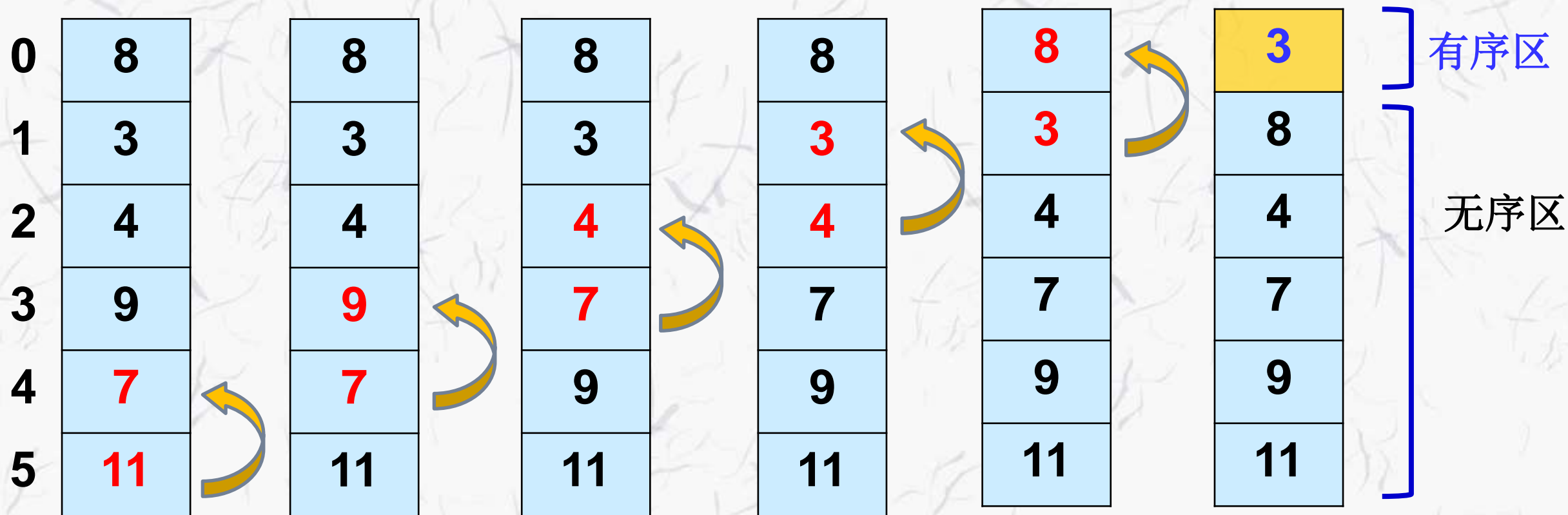


# 冒泡排序方法

- ④ 此后，再对前 $n-1$ 个记录进行同样处理，使 $n-1$ 个记录的最大者被安置在整个序列的第 $n-1$ 个位置上。
- ⑤ 然后再对前 $n-2$ 个记录重复上述过程.....，这样最多做 $n-1$ 次冒泡就能完成排序。
- 可以设置一个标志`noswap`表示本趟冒泡是否有记录交换，如果没有交换则表示整个排序过程完成。
- 冒泡排序是通过相邻记录之间的比较与交换，使值较大的记录逐步从前(上)向后(下)移，值较小的记录逐步从后(下)向前(上)移，就像水底的气泡一样向上冒，故称为冒泡排序。
- 冒泡排序过程
- 49, 38, 65, 97, 76, 13, 27, 49



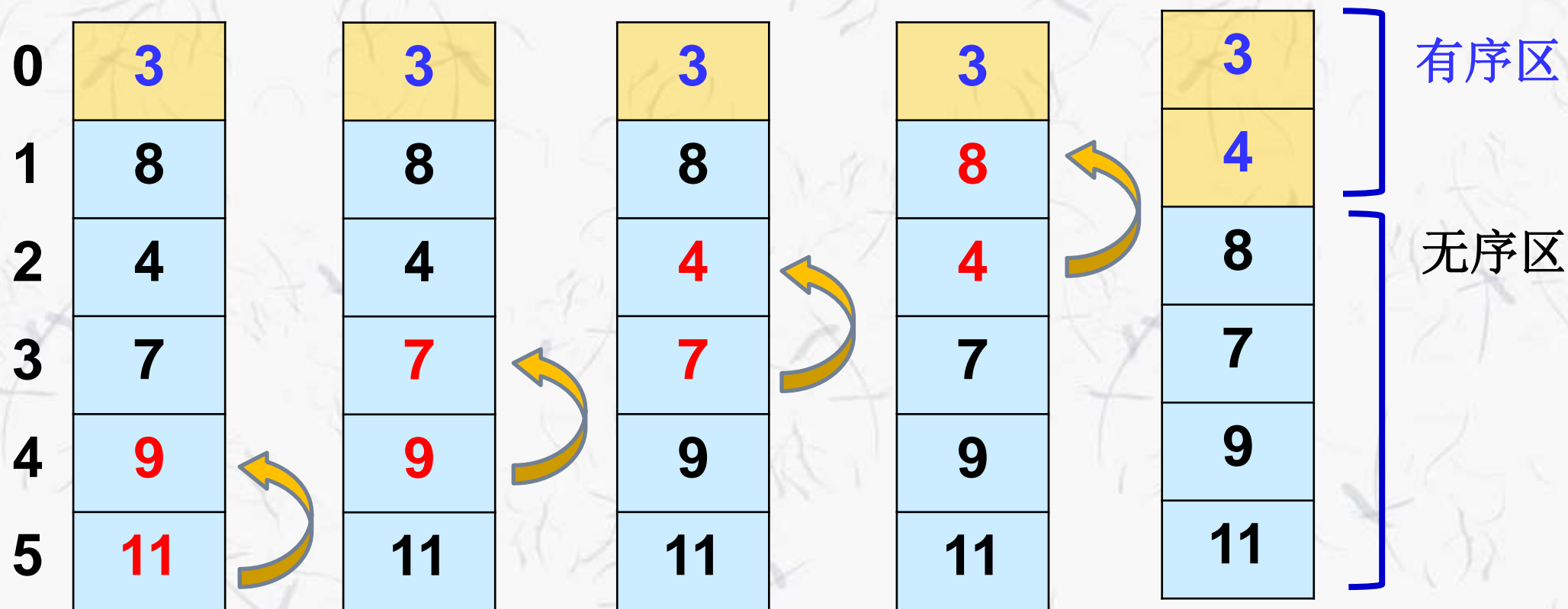
# “上升法” 冒泡排序示例1第一趟冒泡



第一趟冒泡，排序码最小的记录上升到数组顶部

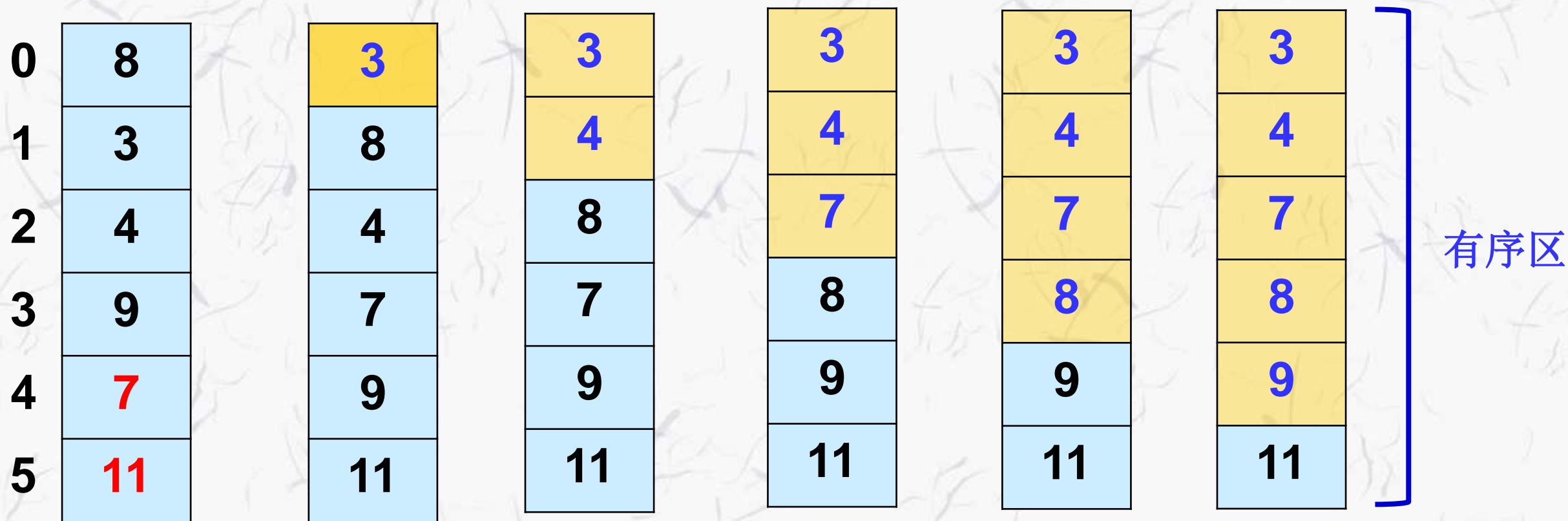


# “上升法” 冒泡排序示例1第二趟冒泡



第二趟冒泡，排序码最小的记录上升到无序段数组顶部，有序区扩大

# “上升法” 冒泡排序示例1全过程



原始数据

第一趟冒泡

第二趟冒泡

第三趟冒泡

第四趟冒泡

第五趟冒泡

```
void bubbleSort( int * a, int n )
{
    int i, j, noswap; int temp;
    for(i=0; i<n-1; i++) //做n-1次冒泡,
    {
        for(j=n-2; j>=i; j--)
            //从后向前扫描, i是本趟扫描的终点下标
            if(a[j]>a[j+1]) //发现逆序就交换
            {
                temp=a[j];
                a[j]=a[j+1];
                a[j+1]=temp;
            }
        }
    }
```

## □ 简单的冒泡排序算法 (上升法)

稳定、原地

# “上升法” 冒泡排序示例2

0	13	13	有序区
1	27	27	
2	38	38	无序区
3	49	49	
4	65	65	
5	76	76	

原始数据

第一趟冒泡



```

void bubbleSort( int * a, int n )
{
    int i, j, noswap; int temp;
    for(i=0; i<n-1; i++)    /* 做n-1次冒泡, */
    {
        noswap=1;    /*置交换标志*/
        for(j=n-2; j>=i; j--) //从后向前扫描, i是本趟扫描的终点下标
            if(a[j]>a[j+1]) //发现逆序就交换
            {
                temp=a[j];
                a[j]=a[j+1];
                a[j+1]=temp;
                noswap=0;
            }
        if(noswap) break;
        /*本趟冒泡未发生记录交换, 算法结束*/
    }
}

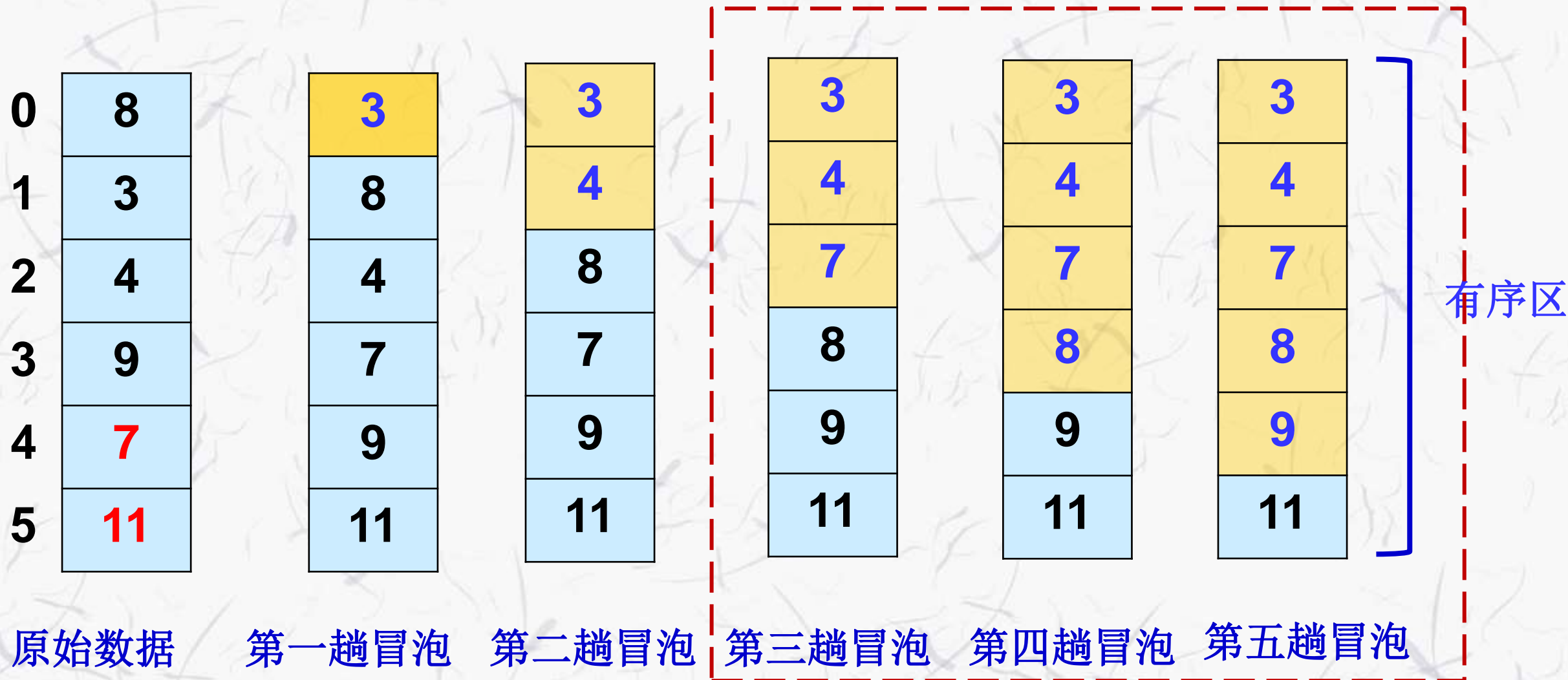
```

## □ 改进的冒泡排序算法1 (上升法)

稳定、原地



# “上升法” 冒泡排序示例全过程

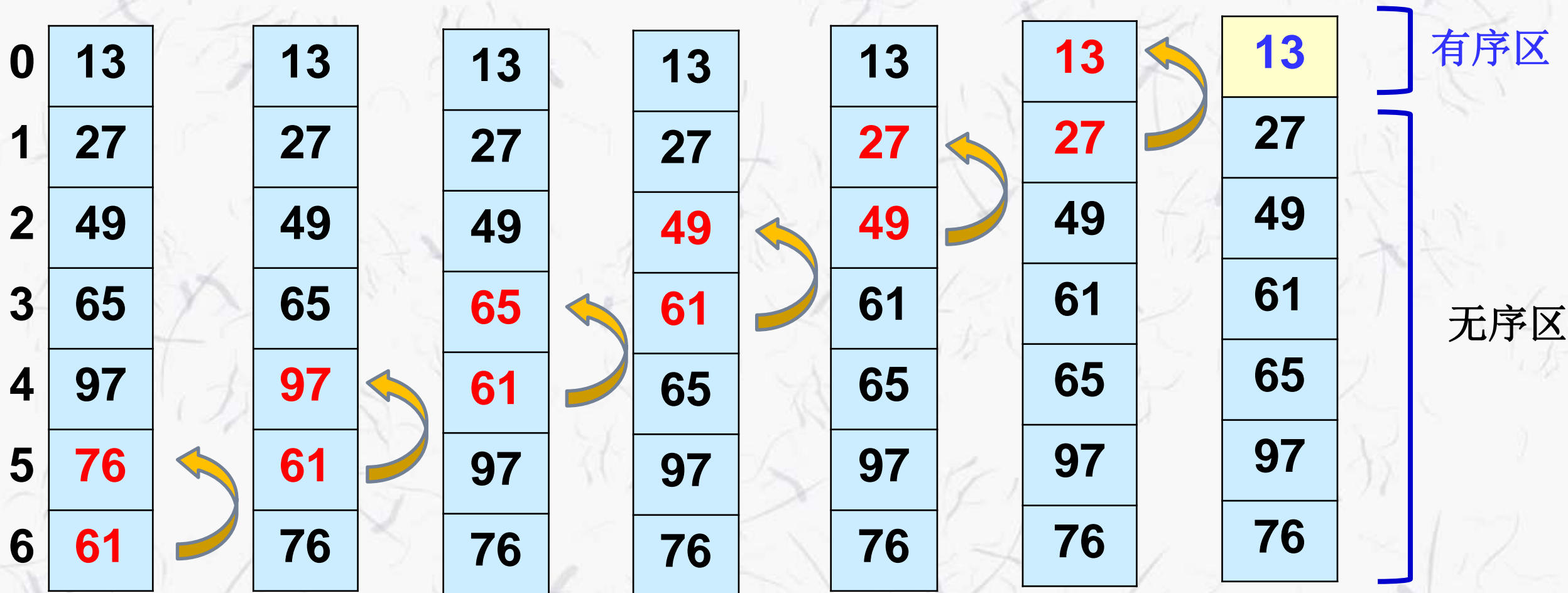


## □ 改进的冒泡排序算法2 (上升法)

```
void bubbleSort( int * a, int n )
{
    int i=0, j, noswap=1; int temp;
    while(noswap)    /* 做i次冒泡 */
    {
        noswap=0;    /* 每趟初始化交换标志, 还未发生交换 */
        for(j=n-2; j>=i; j--) /* 从后向前扫描 */
            if(a[j]>a[j+1]) /* 发现逆序就交换 */
            {
                temp=a[j];
                a[j]=a[j+1];
                a[j+1]=temp;
                noswap=1; /* 发现交换, 说明无序 */
            }
        i++; /* i是本趟扫描的终点下标 */
    }
}
```

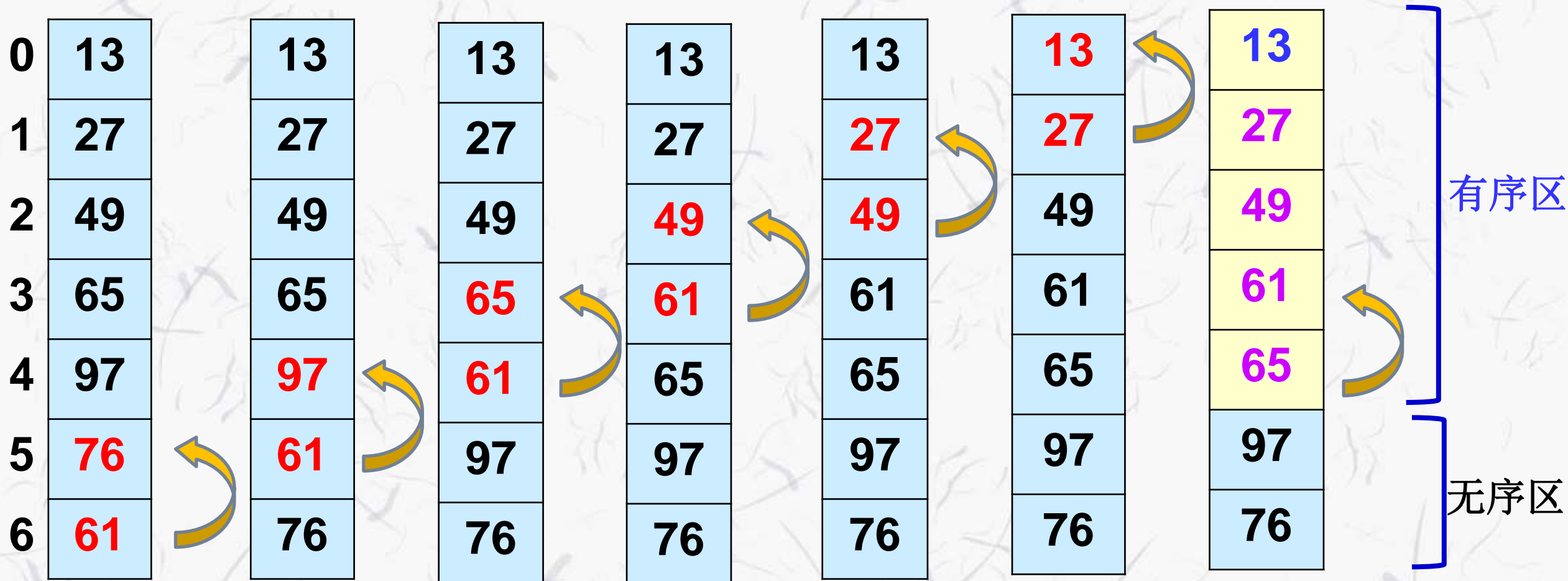
稳定、原地

# “上升法” 冒泡排序示例3之第一趟冒泡



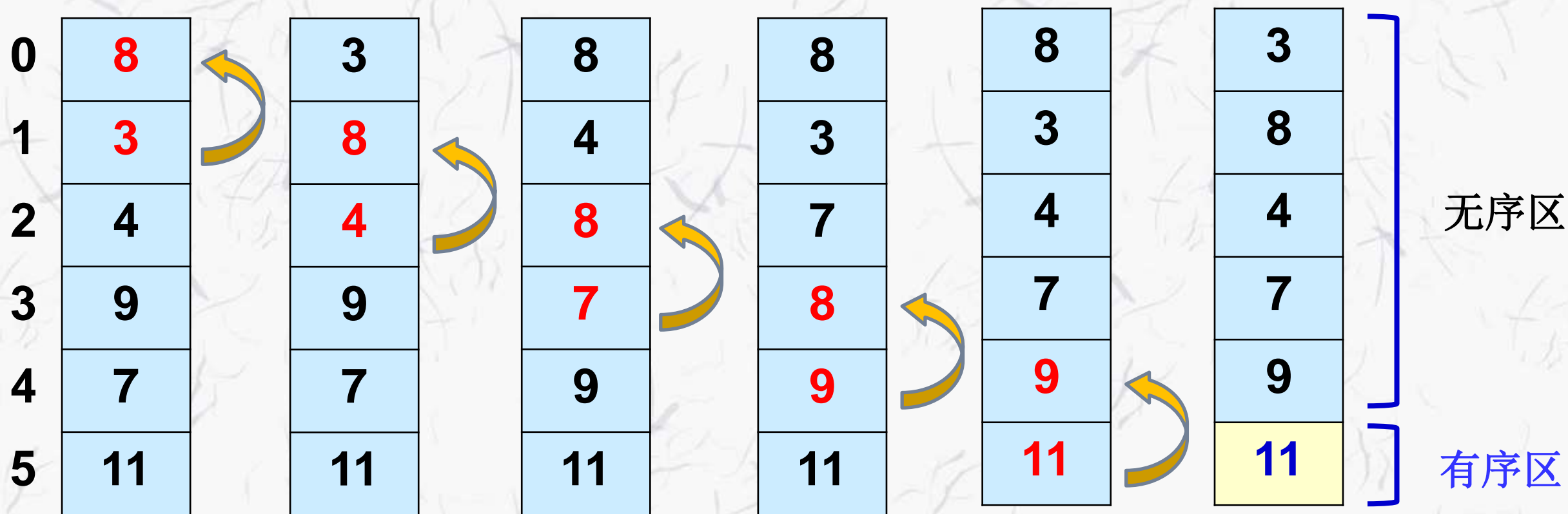
原始数据

# “上升法” 冒泡排序示例3之第一趟冒泡



原始数据

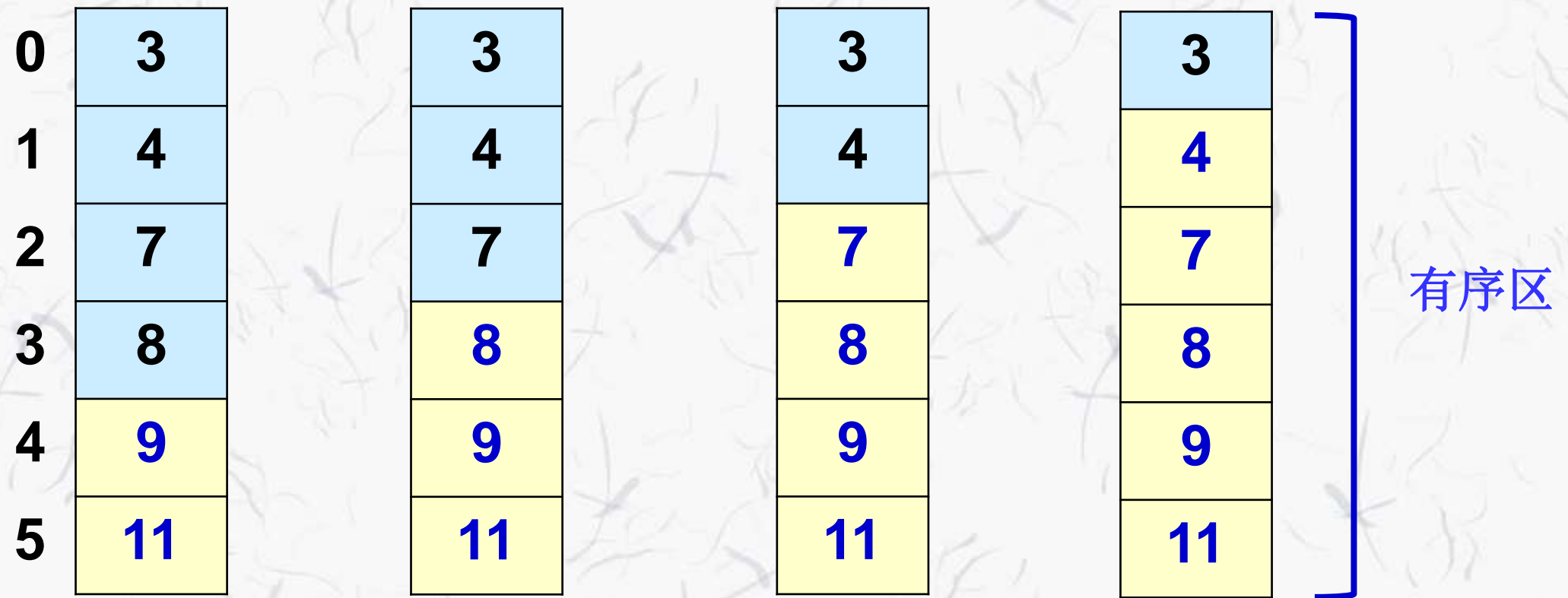
# “下降法” 冒泡排序示例之第一趟



第一趟冒泡，排序码最大的记录下降到数组底部



# “下降法” 冒泡排序示例之第2-5趟



第二趟冒泡

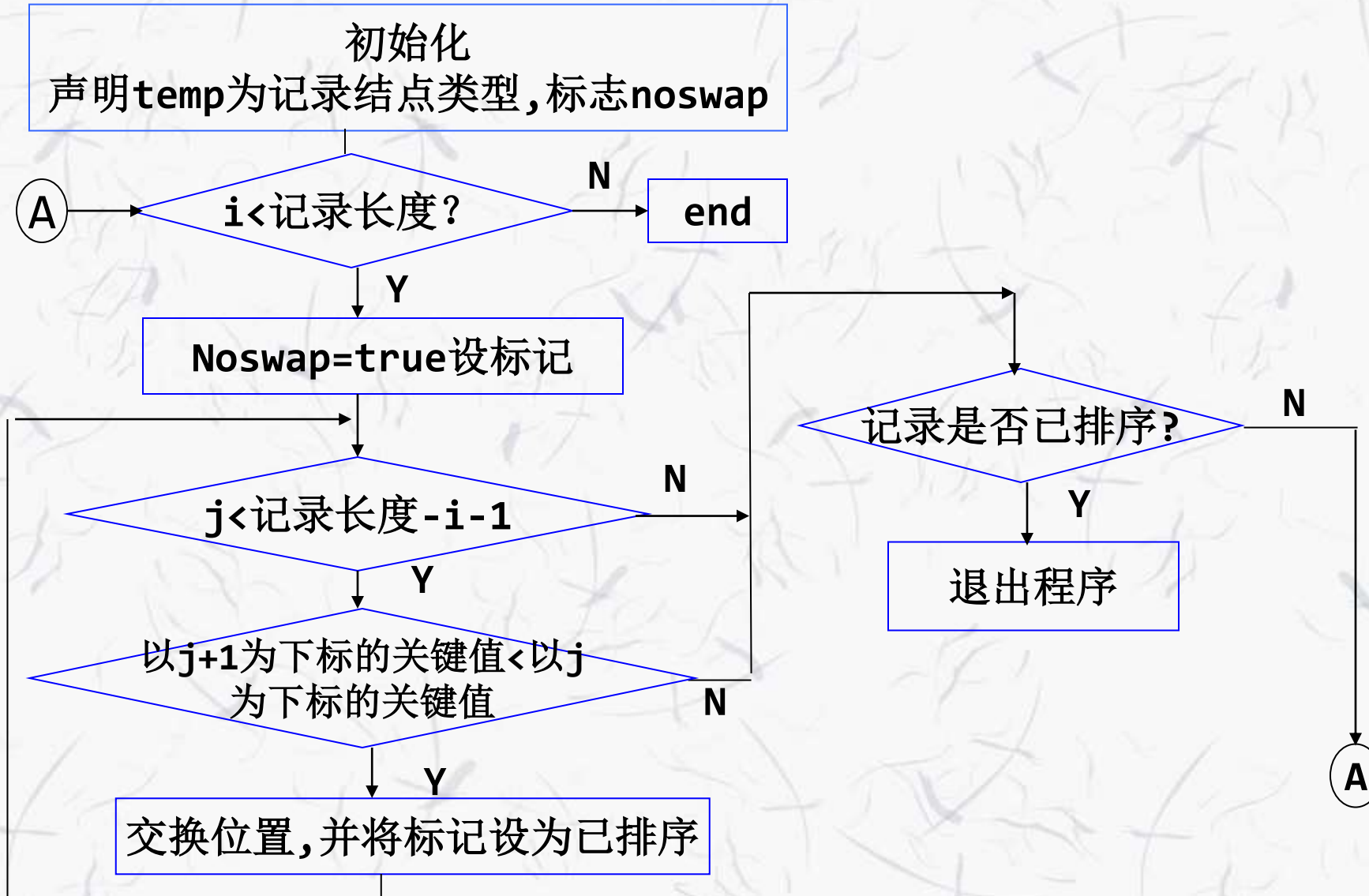
第三趟冒泡

第四趟冒泡

第五趟冒泡



# 冒泡排序算法



## □ 改进的冒泡排序算法 (下降法)

```
void bubbleSort(SortObject * pvector)
{
    int i, j, noswap;
    RecordNode temp;
    for(i=0; i<pvector->n-1; i++) /* 做n-1次起泡 */
    {
        noswap=TRUE; /* 置交换标志 */
        for(j=0; j<pvector->n-i-1; j++) /* 从前向后扫描 */
            if(pvector->record[j+1].key<pvector->record[j].key)
                //发现逆序就交换
            {
                temp=pvector->record[j];
                pvector->record[j]=pvector->record[j+1];
                pvector->record[j+1]=temp;
                noswap=FALSE;
            }
        if(noswap) break; /* 本趟起泡未发生记录交换, 算法结束 */
    }
}
```

# 冒泡排序的算法评价

- 若文件初状为**正序**，则一趟起泡就可完成排序，排序码的比较次数为 **$n-1$** ，且没有记录移动，时间复杂度是 **$O(n)$**
- 若文件初态为**逆序**，则需要 **$n-1$ 趟**起泡，每趟进行 **$n-i$ 次**排序码的比较，且每次比较都移动三次，比较和移动次数均达到最大值：

$$C_{\max} = \sum_{i=1}^{n-1} (n-i) = n(n-1)/2 = O(n^2)$$

$$M_{\max} = \sum_{i=1}^{n-1} 3(n-i) = 3n(n-1)/2 = O(n^2)$$



# 冒泡排序的算法评价(续)

- 改进的冒泡排序**最好**时间复杂度是 $O(n)$
- 冒泡排序**最坏**时间复杂度为 $O(n^2)$
- 冒泡排序**平均**时间复杂度为 $O(n^2)$
- 冒泡排序算法中增加一个辅助空间temp，辅助空间为 $S(n)=O(1)$
- 冒泡排序是**稳定的**。

平均时间复杂性	辅助空间	稳定性
$O(n^2)$	$O(1)$	<b>稳定</b>





# 简单排序算法总结

- 一个长度为 $n$ 序列平均有  $n(n-1)/4$  对逆置
- 任何一种只对相邻记录进行比较的排序算法的平均时间代价都是  $O(n^2)$



# 简单排序算法的时间代价

比较次数	改进的冒泡排序算法1	直接选择排序
最佳情况	$n-1$	$n(n-1)/2$
平均情况	$O(n^2)$	$n(n-1)/2$
最差情况	$n(n-1)/2$	$n(n-1)/2$

移动次数	改进的冒泡排序算法1	直接选择排序
最佳情况	$\theta$	$\theta$
平均情况	$O(n^2)$	$O(n)$
最差情况	$3n(n-1)/2$	$3(n-1)$

└──────────┘  
交换排序

└──────────┘  
选择排序



# 普通冒泡排序冒泡次数统计

- 题目描述

- 给定N个无序且不相同的数字，求使用冒泡排序将其排列为从小到大排列的有序数组所需要的冒泡次数（数字交换次数）。  
N不超过1000；

- 关于输入

- 第一行为一个正整数N，表示数组的大小。  
第二行包含N个数，为无序数组的数字。

- 关于输出

- 输出一个正整数，表示冒泡排序所需要的交换次数。



- 解题思路：数据量不大，一种思路可以在直接进行冒泡排序的同时统计。另一种思路是**不用排序**，只**统计有多少逆序对**（排在前面的数大于排在后面的数）。因为冒泡排序每冒泡一次就减少一个逆序对，最终减少到0，因此二者数量是相等的。

```
● #include<stdio.h>
int main(){
    int n, p[1010] = {}, res = 0;
    scanf("%d", &n);
    for(int i=0;i<n;++i) {
        scanf("%d", &p[i]);
        for(int j=0;j<i;++j) if(p[i] < p[j]) ++res;
    }
    printf("%d", res);
    return 0;
}
```

# 内容提要

- 排序的基本概念
- 插入排序
- 交换排序
- 排序函数





# qsort()

- **stdlib.h** 中的 **qsort()** 函数采用了优化实现的快速排序算法，其调用格式如下：
- **qsort(数组名, 元素个数, sizeof(数组元素的类型), 比较大小的函数);**
- 其中用于 **比较大小的函数** 由用户自行定义，该函数接受两个指向待比较元素的指针 **a** 和 **b**，返回一个整数。当要求按 **递增序排序** 时，返回正数表示 **\*a** 比 **\*b** 大，负数反之，**0** 表示相等。



# 应用qsort进行排序-1

- 对int数组排序

```
int cmp(const void *a, const void *b)
{
    return(*(int *)a-*(int *)b);
}

int main()
{
    int data[10],n,i;
    scanf("%d",&n);
    for(i=0;i<n;i++) scanf("%d",&data[i]);
    qsort(data,n,sizeof(data[0]),cmp);
    for(i=0;i<n;i++) printf("%d ",data[i]);
    return(0);
}
```



# 应用qsort进行排序-2

- 对double型数组排序

```
int cmp(const void * a, const void * b)
{
    return ((* (double*)a - * (double*)b > 0) ? 1 : -1);
}

int main()
{
    double data[10];
    int i, n;
    scanf("%d", &n);
    for(i=0; i<n; i++) scanf("%lf", &data[i]);
    qsort(data, n, sizeof(data[0]), cmp);
    for(i=0; i<n; i++) printf("%lf ", data[i]);
    return(0);
}
```





# 应用qsort进行排序-3

- 对字符数组排序

```
int cmp(const void *a, const void *b)
{
    return(*(char *)a - *(char *)b);
}

int main()
{
    char data[10];
    int i, n;
    scanf("%s", data);
    n = strlen(data);
    qsort(data, n, sizeof(data[0]), cmp);
    printf("%s", data);
    return(0);
}
```



# 应用qsort进行排序-4

- 对字符串数组排序，char s[][]型

```
int cmp(const void *a, const void *b)
{
    return(strcmp((char*)a, (char*)b));
}

int main()
{
    char data[20][10];
    int i, n;
    scanf("%d", &n);
    for(i=0; i<n; i++) scanf("%s", data[i]);
    qsort(data, n, sizeof(data[0]), cmp);
    for(i=0; i<n; i++) printf("%s\n", data[i]);
    return(0);
}
```





# 应用qsort进行排序-5

- 对字符串数组排序，char \*s[] 型

```
int cmp(const void *a, const void *b)
{
    return(strcmp(*(char**)a, *(char**)b));
}

int main()
{
    char *data[10];
    int i, n;
    scanf("%d", &n);
    for(i=0; i<n; i++)
    {
        data[i] = (char*)malloc(sizeof(char*));
        scanf("%s", data[i]);
    }
    qsort(data, n, sizeof(data[0]), cmp);
    for(i=0; i<n; i++) printf("%s\n", data[i]);
    return(0);
}
```

# 应用qsort进行排序-6

- 对结构体排序

```
#include <stdio.h>
#include <stdlib.h>
```

```
struct date
{   int month;
    int day;
} data[10];
```

```
int i,n;
```

```
int cmp(const void *a,const void *b)
{   return((((const struct date*)a)->month)-
        ((const struct date *)b)->month);
}
```

```
qsort(data,n,sizeof(data[0]),cmp);
```



# 模拟Excel排序

- Excel可以对一组记录按任意指定列排序，现编写程序实现类似功能。
  - (1) 输入说明：输入的第一行包含两个正整数 $N$  ( $N \leq 10000$ ) 和 $C$ ，其中 $N$ 是记录的条数， $C$ 是指定排序的列号。之后有 $N$ 行，每行包含一条学生记录。每条学生记录由学号（6位数字，不重复）、姓名（不超过8位且不含空格的字符串）、成绩（ $[0, 100]$ 内的整数）组成，相邻属性用1个空格隔开。
  - (2) 输出说明：在 $N$ 行中输出按要求排序后的结果，即当 $C=1$ 时，按学号递增排序，当 $C=2$ 时，按姓名的非递减字典排序，当 $C=3$ 时，按成绩的非递减排序。若干学生具有相同姓名或者成绩时，则按学号递增排序。

难点：当按姓名、成绩排序出现并列时，按“学号”排序





```
# include <stdio.h>
# include <string.h>
# include <stdlib.h>
```

```
#define MAXID 6
#define MAXNAME 8
#define MAXN 100000
```

```
struct student{
    char id[MAXID+1];
    char name[MAXNAME+1];
    int grade;
}record[MAXN];
```



- 比较学号的函数

```
int CompareID(const void *a, const void *b)
{
    /* 比较学号大小 */
    return strcmp(((const struct student *)a)->id,
                  ((const struct student *)b)->id);
}
```

- 比较姓名的函数

```
int CompareName(const void *a, const void *b)
{
    /* 按字典序比较姓名 */
    int k = strcmp(((const struct student *)a)->name,
                  ((const struct student *)b)->name);
    if (!k) /* 重名时, 按学号大小排序 */
        k = strcmp(((const struct student *)a)->id,
                  ((const struct student *)b)->id);
    return k;
}
```





- 比较成绩的函数

```
int CompareGrade(const void *a, const void *b)
{
    /*按非递增序比较成绩*/
    int k=((const struct student *)a)->grade>
        ((const struct student *)b)->grade)?-1:0;
    if(!k){
        k=((const struct student *)a)->grade<
            ((const struct student *)b)->grade)?1:0;
        if(!k) /*成绩相同时, 按学号大小排序*/
            k=strcmp(((const struct student *)a)->id,
                ((const struct student *)b)->id);
    }
    return k;
}
```

# 函数的调用方式

```
switch(C){  
    case 1:qsort(record,N,sizeof(struct student),CompareID);  
    break;  
    case 2:qsort(record,N,sizeof(struct student),CompareName);  
    break;  
    case 3:qsort(record,N,sizeof(struct student),CompareGrade);  
    break;  
}
```



# sort () 函数基本用法

- STL即Standard Template Library
- 包括五大类组件：算法、容器、迭代器、函数对象、适配器
- `#include <algorithm>` //sort算法在此头文件中定义
- `using namespace std;`
- `sort(begin, end, cmp)`
- `begin`为指向待排序的数组的**第一个元素**的指针
- `end`为指向待排序的数组的**最后一个元素的下一个位置**的指针
- `cmp`参数为排序准则，如果没有的话，默认以非降序排序。

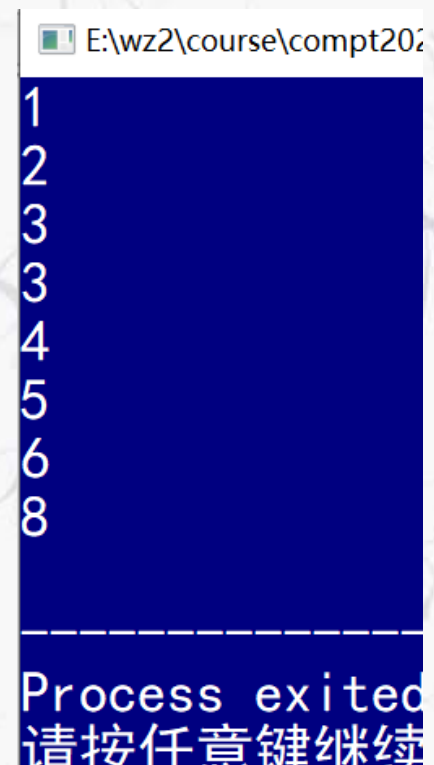


# sort () 函数基本用法

- 默认以非降序排序。

```
#include <stdio.h>
#include <algorithm>
using namespace std;
int main(){
    int n=8, p[10]={1,3,5,6,4,2,3,8};
    sort(p,p+8);
    for(int i=0;i<n;++i)
    { printf("%d\n", p[i]); }
    return 0;
}
```

```
bool cmp(double x, double y) {
    return x < y; } //以非降序排序
```



```
E:\wz2\course\compt202
1
2
3
3
4
5
6
8
-----
Process exited
请按任意键继续
```



```
#include <stdio.h>
#include <algorithm>
using namespace std;
bool cmp(double x, double y) {
    return x > y; }    //以非升序排序。
```

```
int main(){
    int n=7;
    double p[10]={1.2,3.4,5.1,6.0,4.2,2.7,3.8};
    sort(p,p+7,cmp);
    for(int i=0;i<n;++i) {
        printf("%.1f\n", p[i]);
    }
    return 0;
}
```

E:\wz2\course\compt20

6.0  
5.1  
4.2  
3.8  
3.4  
2.7  
1.2

Process exited  
请按任意键继续



```

#include<stdio>
#include<algorithm>
#include<cstring>
using namespace std;
typedef struct student{
    char name[20];
    int math;
    int english;
}Stu;
bool cmp(Stu a,Stu b);
int main(){
    int n=4; //先按math从小到大排序, math相等, 再按english从大到小排序
    Stu a[5]={{ "Alice",80,89},{ "Bob",90,62},{ "John",90,91},{ "Tony",91,99}};
    sort(a,a+n,cmp);
    for(int i=0;i<n;i++) printf("%-5s %3d %3d\n",a[i].name,a[i].math,a[i].english);
}
bool cmp(Stu a,Stu b){
    if(a.math >b.math ) return a.math >b.math ;//按math从大到小排序
    else if(a.math ==b.math ) return a.english>b.english ;
        //math相等, 按english从大到小排序
}

```

```

E:\wz2\course\compt2021a\cpp\ch
Tony    91    99
John    90    91
Bob     90    62
Alice   80    89
-----
Process exited after
请按任意键继续. . .

```

# 本讲算法小结

排序方法	平均时间复杂性	辅助空间	稳定性
直接选择排序	$O(n^2)$	$O(1)$	不稳定
改进的冒泡排序算法1	$O(n^2)$	$O(1)$	稳定

