



# 七

## 算法与数据结构 —— 第七章 排序

# CONTENTS

## 目录

1

选择排序

2

插入排序

3

交换排序

4

归并排序

5

基数排序

## 7.1 排序的概念与算法性能

**排序**就是把一组无序的记录（元素）调整为有序序列的过程。

学号	姓名	年龄	性别
99001	王晓佳	18	男
99002	林一鹏	19	男
99003	谢宁	17	女
99004	张丽娟	18	女
99005	周涛	20	男
99006	李小燕	16	女

排序（Sorting）是数据处理中一种很重要的运算，同时也是很常用的运算，一般数据处理工作25%的时间都在进行排序。

## 1. 排序算法的选择

【例】有1亿个随机给出的浮点数，请找出其中最大的1万个数。

方法1：简单选择法 (1万亿)

- 总比较次数为 $N-1+(N-2)+\dots+(N-10000)$ 次。当N为1亿时，大约为1万亿次。

方法2：“分而治之” (20亿)

- 将所有数据分为100块（每块100万个元素），分别进行排序。每块排序后，各取前1万个数组成一个新块。最后对新块排序取最大1万个元素。
- 如果采用快速排序，当N为100万时， $O(N\log N)$ 是2000万，所以求解101块百万（100块待排序数据+1块需2次排序）数据的排序问题，时间大约是20亿次运算。

## 方法3:堆选择 (2.4亿)

- 先读出1百万个数，建立**初始堆**（100万运算量）找出最大的1万个数。然后对剩下的**近1亿**个数进行**过滤**：每次读入剩余的下一个数，如果该数小于等于这1万个数的**最小值**，则继续读下一个数；否则，用该数**替代1万个数里的最小值**，**调整堆**。
- 100万（1百万建堆）+ 1亿（顺序过滤）+  $14 * 1$ 千万，共约**2.4亿**。

N=1亿,  $\log 10000=14$   
过滤所有数据需要  
**14亿?**

No

随机数将使得绝大多数  
很快被排除！通常有  
大约**1/10**需要替换。

排序算法的**效率**直接影响最终问题的求解



## 2. 排序的稳定性

设  $K_i$ 、 $K_j$  ( $1 \leq i \leq n, 1 \leq j \leq n, i \neq j$ ) 分别为记录  $R_i$ 、 $R_j$  的关键字, 且  $K_i = K_j$ , 在排序前的序列中  $R_i$  领先于  $R_j$  (即  $i < j$ )。若在排序后的序列中  $R_i$  仍领先于  $R_j$ , 则称所用的**排序方法是稳定的**; 反之, 则称所用的**排序方法是不稳定的**。

例:

设排序前的关键字序列为: 52, 49, 80, 36, 14, 58, 36, 23

若排序后的关键字序列为: 14, 23, 36, 36, 49, 52, 58, 80, 则**排序方法是稳定的**。

若排序后的关键字序列为: 14, 23, 36, 36, 49, 52, 58, 80, 则**排序方法是不稳定的**。



### 3.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**。



## 7.2 简单选择排序

- 首先通过  $n-1$  次关键字比较，从  $n$  个记录未排序的记录中找出关键字最小的记录，将它与第一个记录交换。
- 再通过  $n-2$  次比较，从剩余的  $n-1$  个未排序的记录中找出关键字次小的记录，将它与第二个记录交换。
- 重复上述操作，共进行  $n-1$  趟排序后，排序结束。



例:

$i=1$	初始:	[	13	38	65	97	76	49	27	]	$n-1$	} 比较 次数	
								$k$ ↓	$j$ ↑				
$i=2$	一趟:		13	[38	65	97	76	49	27	]	$n-2$		
$i=3$	二趟:		13		27	[65	97	76	49	38			
$i=4$	三趟:		13		27		38	[97	76	49	65		.....
$i=5$	四趟:		13		27		38		49	[76	97		65
$i=6$	五趟:		13		27		38		49	65	[97	76	$n-6$
排序结束: 六趟:			13		27		38		49	65	76	97	

```
void SelectSort (ElementType &A[],int N){
    for (i = 0; i < N-1; ++ i) {
        k=i;
        for(j=i+1;j<N;j++)
            if (A[j]<A[k]) k=j;
        swap(&A[i], &A[k]);
    }
}
```

### 选择排序过程:

- ① 循环找到从 $i+1$ 到 $N$ 中的最小元素
- ② 最小元素与 $a[i]$ 互换

## 算法分析：

比较次数：  $n*(n-1)/2$  时间复杂度：  $O(n^2)$

移动次数：正序：最小值为 0；最大值为  $3(n-1)$ 。

例：前  $n-1$  个为正序，第  $n$  个记录的关键字最小。

空间复杂度：  $O(1)$

简单选择排序是不稳定的

除了最后一个元素外，每个元素都要经过3步交换位置

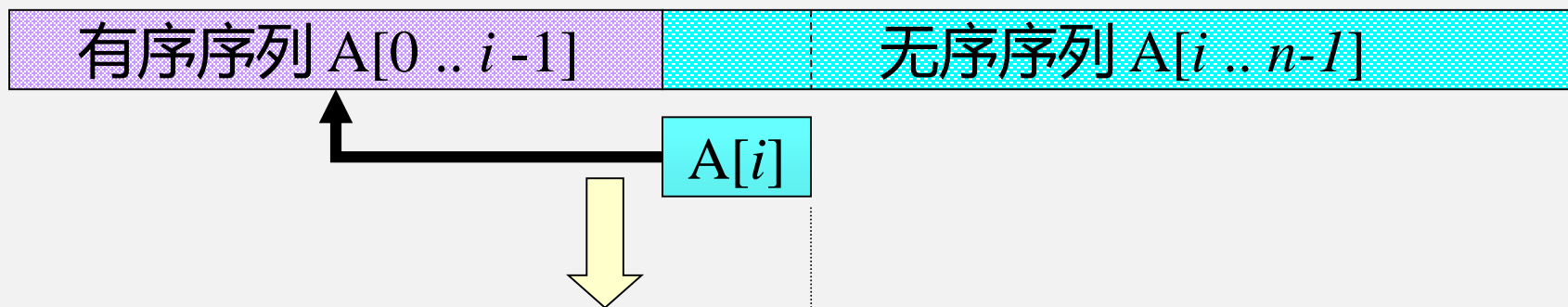


## 7.3.1 直接插入排序

- **基本思想：**
  - 将待排序的一组序列分为**已排序的**和**未排序的**两个部分；
  - 初始状态已排序序列仅包含第一个元素，未排序序列包括第2~N个元素；
  - 然后依次将未排序序列中的元素插入到已排序序列中，直到未排序序列中元素个数为0。

排序过程：先将序列中第 1 个记录看成是一个有序子序列，然后从第 2 个记录开始，逐个进行插入，直至整个序列有序。

图示一趟直接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  $A[0 .. i-1]$  中查找  $A[i]$  的插入位置， $A[0 .. j] \leq A[i] < A[j+1 .. i-1]$ ;
2. 将  $A[j+1 .. i-1]$  中的所有记录均后移一个位置；
3. 将  $A[i]$  插入（复制）到  $A[j+1]$  的位置上。

**【例】 设有关键字序列为： 7, 4, -2, 19, 13, 6, 实现直接插入排序。**

初始记录的关键字： [7] 4 -2 19 13 6

第一趟排序： [4 7] -2 19 13 6

第二趟排序： [-2 4 7] 19 13 6

第三趟排序： [-2 4 7 19] 13 6

第四趟排序： [-2 4 7 13 19] 6

第五趟排序： [-2 4 6 7 13 19]

```
void InsertSort (DataType A[ ]) {  
    // 对顺序表 A 作直接插入排序  
    DataType temp;  
    int i, j;  
    for ( i = 1; i <= N-1; ++ i )  
        if (A[i] < A[i - 1]) {  
            temp = A[i];  
            for (j=i-1; j>=0&&temp<A[j]; --j)  
                A[j+1]=A[j];           // 记录后移  
            A[j + 1] = temp;           // 插入到正确位置  
        }  
} // InsertSort
```

# 算法评价

**最好的情况：待排序记录按关键字从小到大排列（正序）**

**比较次数：** $\sum_{i=2}^n 1 = n - 1$       **移动次数：**0

**最坏的情况：待排序记录按关键字从大到小排列（逆序）**

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

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

**一般情况：待排序记录是随机的，取平均值。**

**比较次数和移动次数均约为：** $\frac{n^2}{4}$

直接插入排序是稳定排序

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

## 7.3.2 希尔排序（缩小增量排序）

**基本思想：** 将待排序的一组元素按一定间隔分为若干个序列，分别进行插入排序；然后逐轮缩小间隔，直到间隔=1。

**排序过程：**

- 先取一个正整数  $gap_1 < n$ ，把所有相隔  $gap_1$  的记录放在一组内，组内进行直接插入排序；
- 然后取  $gap_2 < gap_1$ ，重复上述分组和排序操作；
- 直至  $gap_i = 1$ ，即所有记录放进一个组中排序为止。其中  $gap_i$  称为**增量**。

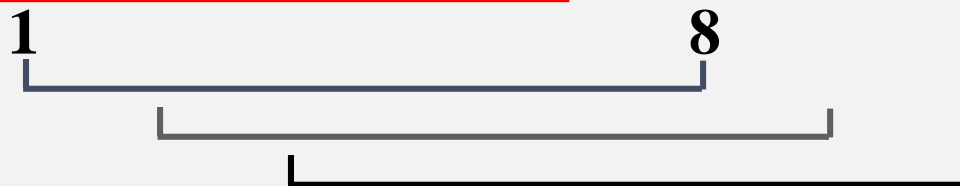


例： 设有10个待排序的记录，关键字分别为9, 13, 8, 2, 5, 13, 7, 1, 15, 11，增量序列是5, 3, 1，希尔排序的过程。

初始关键字序列: 9   13   8   2   5   13   7   1   15   11




第一趟排序过程:



第一趟排序后:

9   7   1   2   5   13   13   8   15   11



第二趟排序后:

2   5   1   9   7   13   11   8   15   13

第三趟排序后:

1   2   5   7   8   9   11   13   13   15

```
void ShellSort( ElementType A[ ], int N)
{ /* 希尔排序*/
    int Si,D ,P, i;
    ElementType tmp;
    int Sedgewick[]={5,3,1};
    for ( Si = 0; Sedgewick[Si]>=N; Si++ )
        ;
    for ( D = Sedgewick[Si]; D>0;D= Sedgewick[++Si])
        for ( P = D; P < N; P++ ) {
            tmp = A[P];
            for ( i = P; i>=D&&A[i-D]>tmp; i-=D)
                A[i] = A[i-D];
            A[k] = tmp;
        }
}
```

- ① Sedgewick: 增量
- ② Si: 增量的个数
- ③ D, P, i: 循环变量

## 希尔排序特点：不稳定排序

### ● 增量序列取法

- 希尔最早提出的选法是  $gap_1 = \lfloor n/2 \rfloor$ ,  $gap_{i+1} = \lfloor d_i/2 \rfloor$ 。
- 克努特 (Knuth) 提出的选法是  $gap_{i+1} = \lfloor (gap_i - 1) / 3 \rfloor$ 。
- .....

### ● 希尔排序时间复杂度与增量序列有关

- 1) 如果增量序列为  $\lfloor n/2 \rfloor, \lfloor n/2^2 \rfloor, \dots, 1$  , 最差时间复杂度  $O(N^2)$
- 2) 如果增量序列为  $2^k - 1, \dots, 7, 3, 1$  , 最差时间复杂度  $\theta(N^{3/2})$

**经验表明，希尔排序对规模万计的序列效果较好**



## 7.4.1 冒泡排序

- 比较第一个记录与第二个记录，若关键字为逆序则交换；然后比较第二个记录与第三个记录；依次类推，直至第  $n-1$  个记录和第  $n$  个记录比较为止——**第一趟冒泡排序**，结果关键字最大的记录被安置在最后一个记录上。
- 对前  $n-1$  个记录进行第二趟冒泡排序，结果使关键字次大的记录被安置在第  $n-1$  个记录位置。
- 重复直到 “**在一趟排序过程中没有进行过交换记录的操作**” 或 “**仅第一二个交换过**” 为止。

## 例：冒泡排序示例

初始关键字序列: 23   38   22   45   23   67   31   15   41

第一趟排序后: 23   22   38   23   45   31   15   41   67

第二趟排序后: 22   23   23   38   31   15   41   45   67

第三趟排序后: 22   23   23   31   15   38   41   45   67

第四趟排序后: 22   23   23   15   31   38   41   45   67

第五趟排序后: 22   23   15   23   31   38   41   45   67

第六趟排序后: 22   15   23   23   31   38   41   45   67

第七趟排序后: 15   22   23   23   31   38   41   45   67

# 冒泡排序算法

```
void BubbleSort(dataType A[]) {  
    int i, j;  
    bool flag;  
    for(i=N-1;i>0;i--) {  
        flag = false; //标志置为false,假定未交换  
        for (j = 0; j <= i; j++) //一趟冒泡  
            if (A[j]>A[j+1]) { //找出最大元素  
                Swap (&A[j],&A[j+1]); //交换  
                flag = true;  
            }  
        if (flag==false) break;  
    }  
}
```

**Flag标志位:**  
检查一趟扫描中是否有元素需要交换

## ❖ 算法评价

### ● 时间复杂度:

➤ 最好情况 (正序)  $T(n) = O(n)$

比较次数:  $n - 1$

移动次数: 0

➤ 最坏情况 (逆序)  $T(n) = O(n^2)$

比较次数:  $\sum_{i=1}^{n-1} (i-1) = \frac{1}{2}(n^2 - n)$  移动次数:  $3 \sum_{i=1}^{n-1} (i-1) = \frac{3}{2}(n^2 - n)$

● 空间复杂度:  $S(n) = O(1)$

● 稳定性: 稳定排序



## 7.4.2 快速排序

- ✓ 通过一趟排序，将待排序记录分割成独立的两部分，其中一部分记录的关键字均比另一部分记录的关键字小；
- ✓ 再利用递归方法分别对这两部分记录进行下一趟排序，以达到整个序列有序。





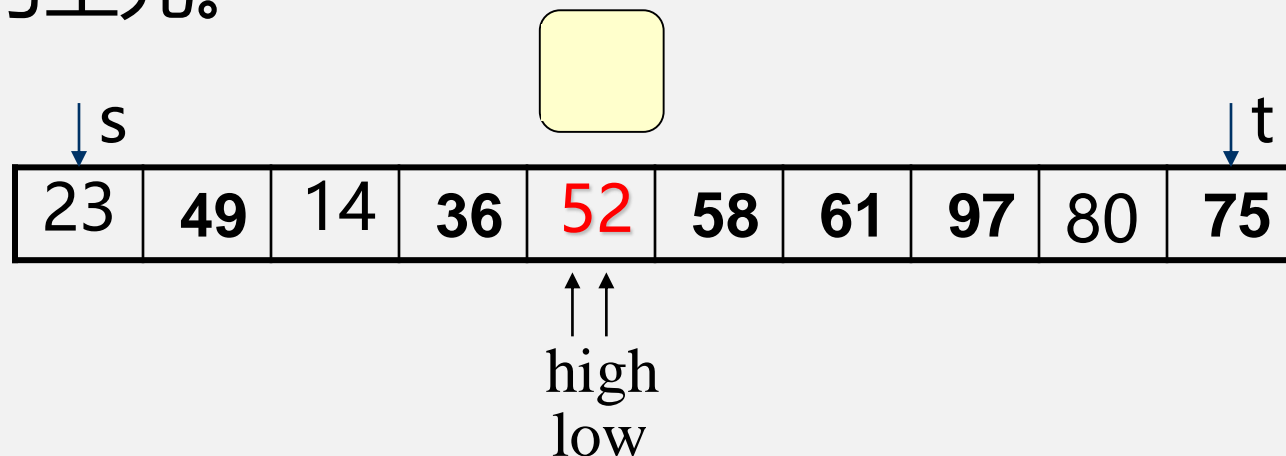


## (1) 一趟快速排序（一次划分）

一般取第一个记录

基本思想：**任选**一个记录，以它的关键字作为“主元”（基准元素），凡关键字小于主元的记录均移至基准元素之前，凡关键字大于主元的记录均移至基准元素之后。

例：设  $R[s]=52$  为主元。



## 方法一

```
while (low < right) {  
    while (low < right && A[right] > pivot)    right--;  
                                                //将大于pivot的元素置于高端区域  
  
    A[low] = A[right];                          //比基准记录小的记录移动到低端  
  
    while (low < right && A[low] <= pivot)      low++;  
  
    A[right] = A[low];                          //比基准记录大的记录移动到高端  
}  
A[low] = pivot;                                //基准记录到位
```

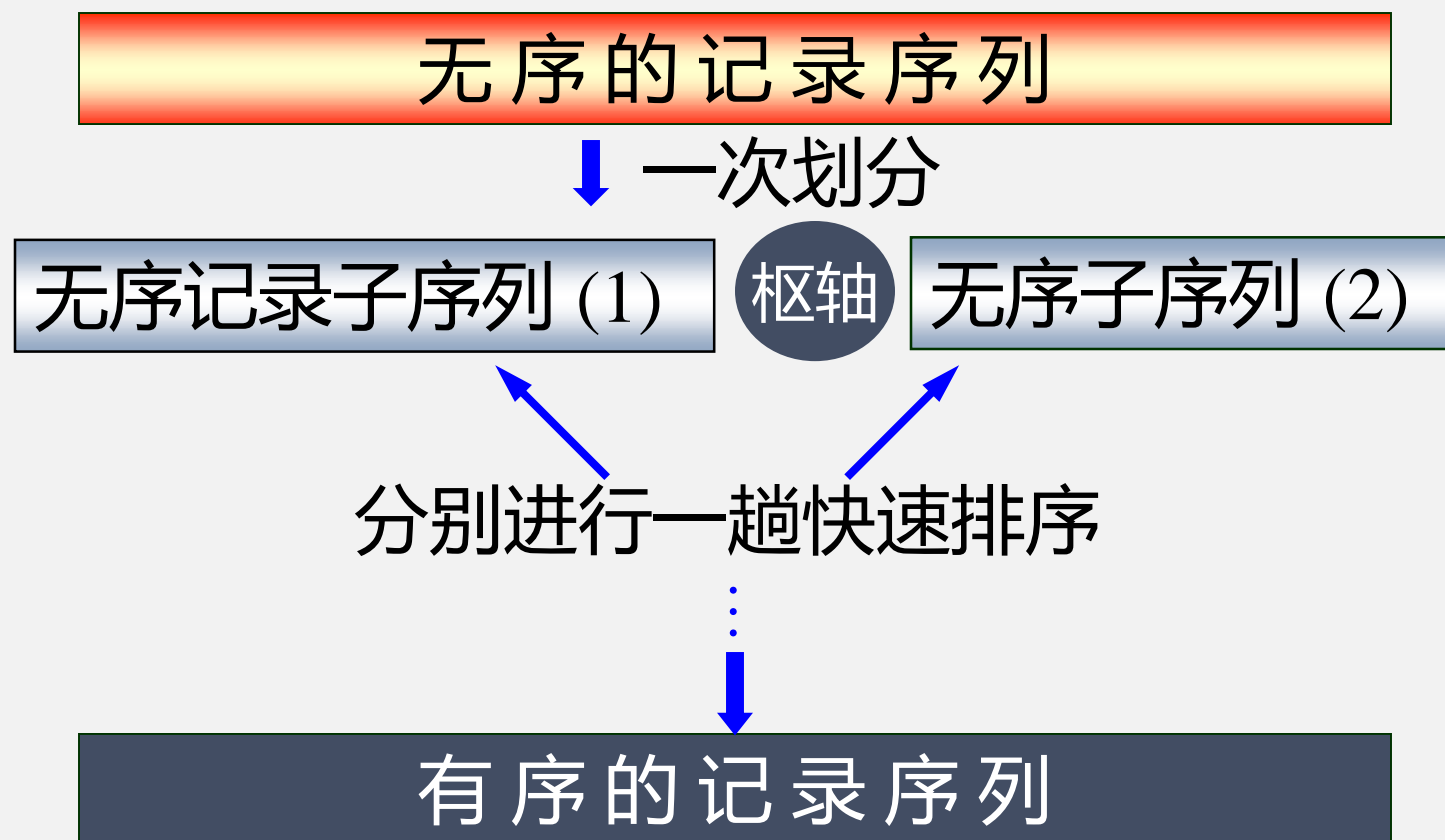
## 方法二

```
void QSort(ElementType A[ ], int Left, int Right)
{ .....
    if (Cutoff <= Right - Left) {
        low = Left, high = Right - 1 ;
        Pivot = median3 (A, Left, Right) ;    /*Pivot作为临时单元和主元 */
        while (1)
        { //大于主元游标左移
            while (Pivot < A[--high])
            while (A[++low] < Pivot)
            if (low < high) swap (&A[Low], &A[high]) ;
            else break ;
        }
        swap (&A[Low], &A[Right - 1]) ;
        .....
    }
```

为避免出现蜕化情况，需在进行一次划分之前，进行“预处理”，即：先对A(Left), A(Right) 和  $A[\lfloor (left+right)/2 \rfloor]$ ，进行相互比较，然后取关键字的大小为中间的记录为基准记录（主元）。

## ■ (2) 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”进行一趟快速排序。



## ❖ 算法评价

到目前为止快速排序是**平均速度最大**的一种排序方法。

时间复杂度为  $O(n \log n)$ 。

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  $O(n^2)$ 。所以快速排序适用于**原始记录排列杂乱无章**的情况。

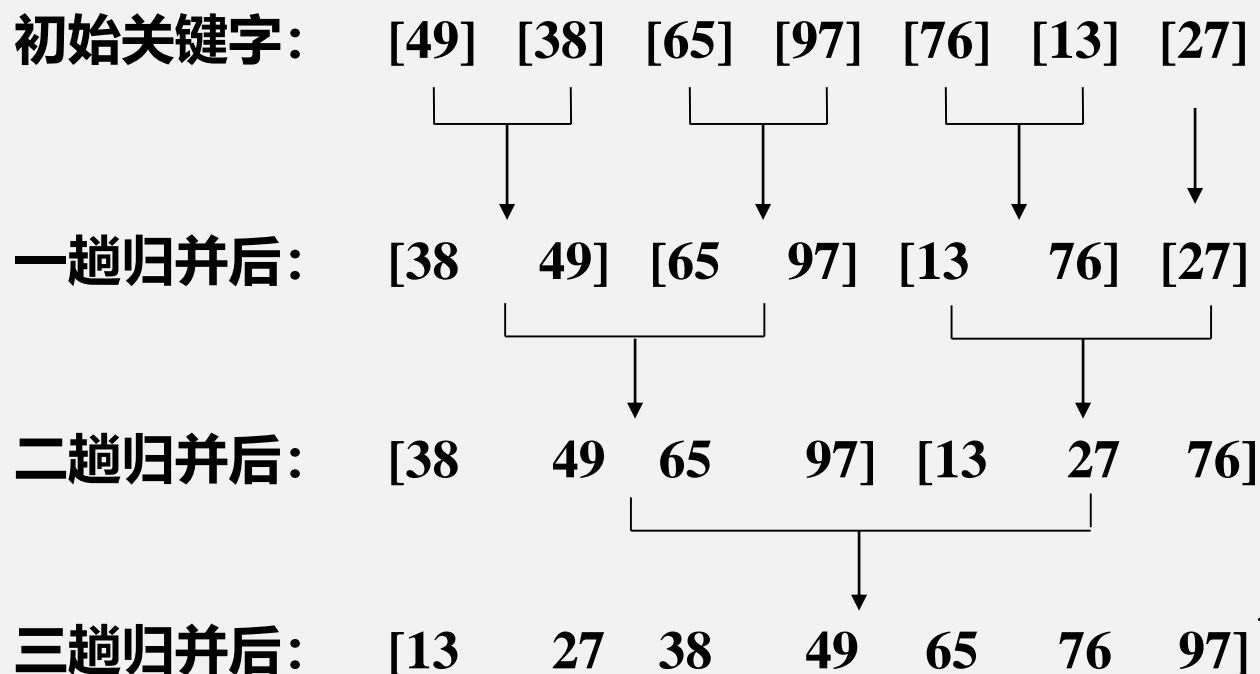
快速排序是一种**不稳定**的排序，在递归调用时需要占据一定的存储空间用来保存每一层递归调用时的必要信息。



## 7.5 归并排序

- **归并排序**是建立在**归并操作**基础上的一种排序方法。
- **归并操作**：将两个或两个以上的有序表组合成一个新的有序表。
- 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**归并**为一个记录有序的序列。

## 例：归并排序示例



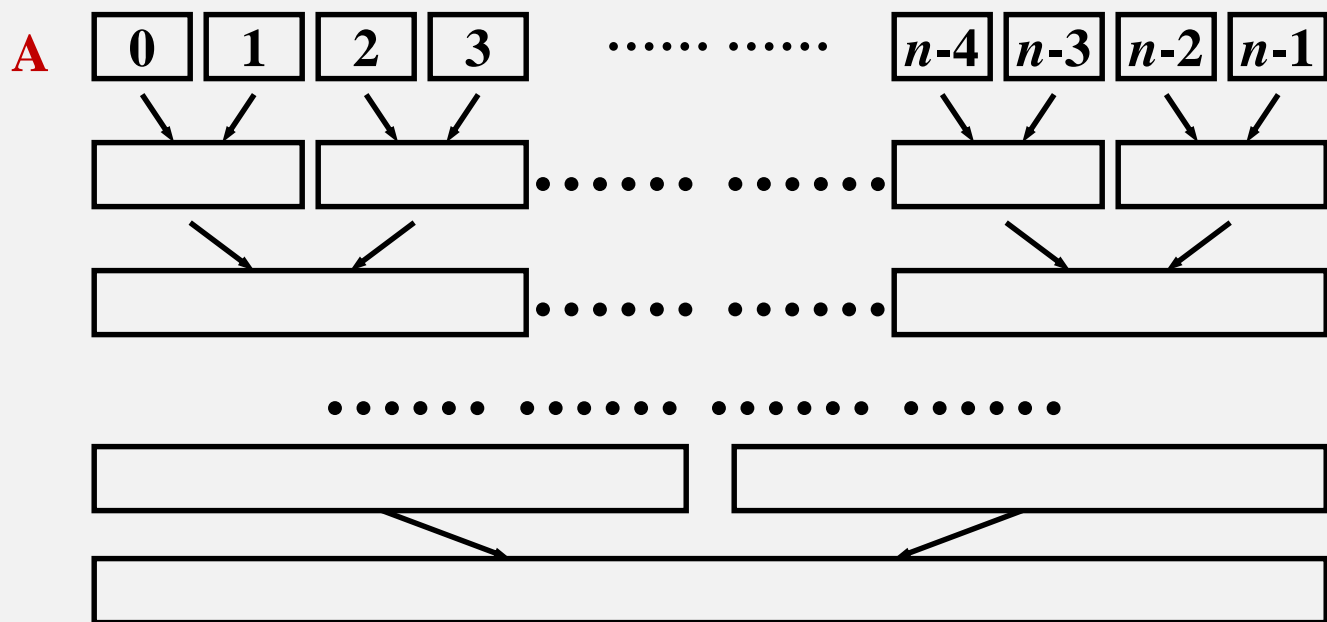
看成是  $n$  个有序的子序列（长度为 1），然后两两归并。

得到  $\lceil n/2 \rceil$  个长度为 2 或 1 的有序子序列。

每趟归并的时间复杂度为  $O(n)$ ，共需进行  $\lceil \log_2 n \rceil$  趟。

空间复杂度为:  $O(n)$ 。 时间复杂度为:  $O(n \log_2 n)$ 。 稳定。

## ➤ 非递归的算法思想:



- 将大小为 $N$ 的序列看成 $N$ 个长度为1的子序列，接下来将相邻子序列两两进行归并操作，形成 $N/2 (+1)$  个长度为2（或1）的有序子序列，然后再继续归并操作，直到剩下1个长度为 $N$ 的序列为止。



## ➤ 归并操作的过程：

- ① 首先申请额外的空间用于存放两个子序列归并后的结果；
- ② 设置两个指针分别指向两个已排序子序列的第一个位置；
- ③ 然后比较两个指针指向的元素，将较小的元素放置到已经申请的额外空间内，并将当前位置向后移动一位；
- ④ 重复以上过程，直到某一个子序列的指针指向该序列的结尾；
- ⑤ 最后将另一指针所指向的剩余元素全部放置到额外空间内，归并操作结束。

```

void Merge( ElementType A[], ElementType TmpA[],
int Left, int Mid, int RightEnd )
{ /* 将有序的A[Left]~A[Mid-1]和A[Mid]~A[Right]归并成一个有序序列 */
    int Tp, LeftEnd, i;
    Tp = Left; /* 有序序列的起始位置 */
    LeftEnd = Mid - 1; /* 左边序列终止的位置 */
    //归并过程
    while ( ( Left<=LeftEnd) && (Mid<=RightEnd) )
        if ( A[Left] <= A[Mid] )
            TmpA[Tp++] = A[Left++]; /* 将左边元素复制到TmpA */
        else
            TmpA[Tp++] = A[Mid++]; /* 将右边元素复制到TmpA */
    //剩余元素复制
    while ( Left <= LeftEnd ) /* 如果左边有元素剩下而右边已 */
        TmpA[Tp++] = A[Left++]; /* 将左边剩余元素复制到T */
    while ( Mid <= RightEnd ) /* 如果右边有元素剩下而左 */
        TmpA[Tp++] = A[Mid++]; /* 将右边剩余元素复制到T */
    //将有序的TmpA[]复制回A[]
    for ( i=RightEnd-Left; i>=0; i--, RightEnd-- )
        A[RightEnd] = TmpA[RightEnd];
}

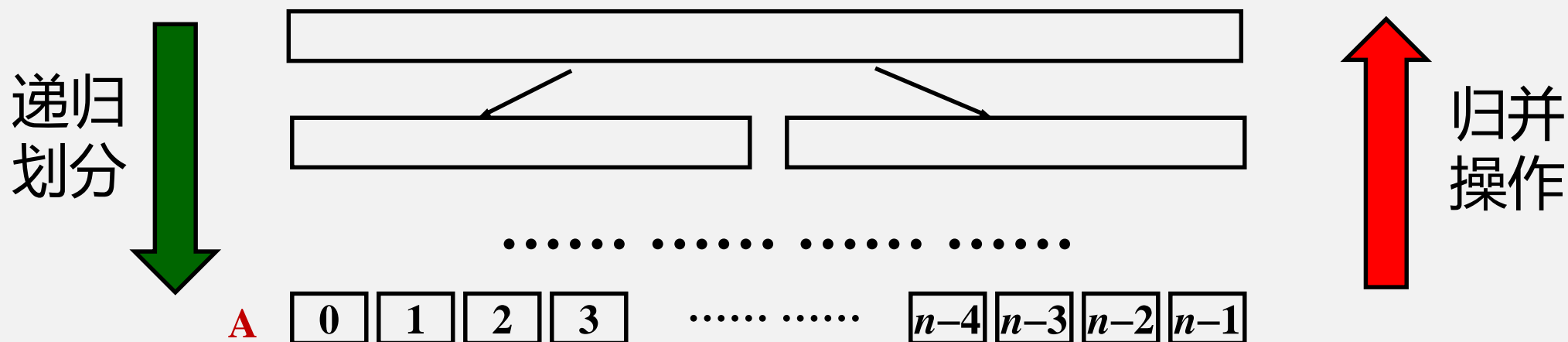
```

## 归并操作的过程：

- ① 将两个序列中的元素按顺序两两比较合并为一个序列；
- ② 其中一个序列结束后将剩余一个序列的元素复制；
- ③ 将有序序列复制回原数组；

## ➤ 递归的算法思想:

- 理解为用分治法的思想，将原序列划分为两个等长子序列，然后递归地排序这两个子序列，最后再调用归并操作合并成一个完整的有序序列。



```

void MSort(ElementType A[], ElementType TmpA[], int Left,
Right ) //核心递归排序算法
{
    int Center; /* 递归地将A[Left]~A[Right]排序 */
    if (Left < Right) { /* 如果还有元素要排序 */
        Center = (Left + Right) / 2;
        MSort(A, TmpA, Left, Center); /* 递归排左半边 T(N/2) */
        MSort(A, TmpA, Center + 1, Right); /* 递归排右半边 T(N/2) */
        Merge(A, TmpA, Left, Center + 1, Right); /* 归并, O(N) */
    }
}

```

```

void Mergesort( ElementType A[ ], int N ) //归并算法
{
    ElementType *TmpA; /* 需要 O(N) 辅助空间 */
    TmpA = malloc( N * sizeof(ElementType) );
    MSort(A, TmpA, 0, N - 1);
    free( TmpA );
}

```

## 归并算法:

- ① 计算中间位置center;
- ② 递归地将原问题分解为两个子序列的归并问题,直到子序列中只有一个元素为止;
- ③ 执行归并操作。

如果 TmpA 定义成 Merge 的局部变量, 则每次调用 Merge 都需要开辟不同的

**注意:** 归并排序需要线性的额外存储, 并且经常复制临时数组导致效率降低。它很少用于内部排序, 然而在外部分排序中非常常用。



## 7.6 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。

特点：每个记录最终的位置由两个关键字  $k^1 k^2$  决定。

我们将它称之为**复合关键字**，即**多关键字排序是按照复合关键字的大小排序**。

第一关键字  $K^1$

第二关键字  $K^2$

学号		数学		英语		其它
101	105	B	A	C	A	...
102	102	A	A	B	B	...
103	104	C	B	D	B	...
104	101	B	B	B	C	...
105	108	A	C	A	B	...
106	103	D	C	B	D	...
107	106	E	D	A	B	...
108	107	C	E	B	A	...

例：扑克牌中 52 张牌，可按**花色**和**面值**分成两个“关键字”，其

大小关系为：花色：♣ < ♦ < ♥ < ♠

花色为最主位关键字，  
面值为最次位关键字

面值：2 < 3 < 4 < 5 < 6 < 7 < 8 < 9 < 10 < J < Q < K < A

若对扑克牌按花色、面值进行升序排序，得到如下序列：

♣2, ♣3, ..., ♣A, ♦2, ♦3, ..., ♦A, ♥2, ♥3, ..., ♥A, ♠2, ♠3, ..., ♠A

即两张牌，若花色不同，不论面值怎样，花色低的那张牌小于花色高的，只有在同花色情况下，大小关系才由面值的大小确定。这也是按照复合关键字的大小排序，即：**多关键字排序**。

## 1、多关键字排序的方法

$n$  个记录的序列  $\{R_1, R_2, \dots, R_n\}$  对关键字  $(K_i^0, K_i^1, \dots, K_i^{d-1})$

有序是指：

对于序列中任意两个记录  $R_i$  和  $R_j$  ( $1 \leq i < j \leq n$ ) 都满足下列

(词典) 有序关系：

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

其中： $K^0$  被称为 **最主位关键字**， $K^{d-1}$  被称为**最次位关键字**。

## 多关键字排序的类型

- ✓ 排序的顺序是从最高位开始所有的子序列依次联接成一个有序的记录序列，该方法称为**主位优先法/最高位优先**(Most Significant Digit first)
- ✓ 另一种方法正好相反，排序的顺序是从最低位开始，称为**次位优先法/最低位优先**(Least Significant Digit first)。

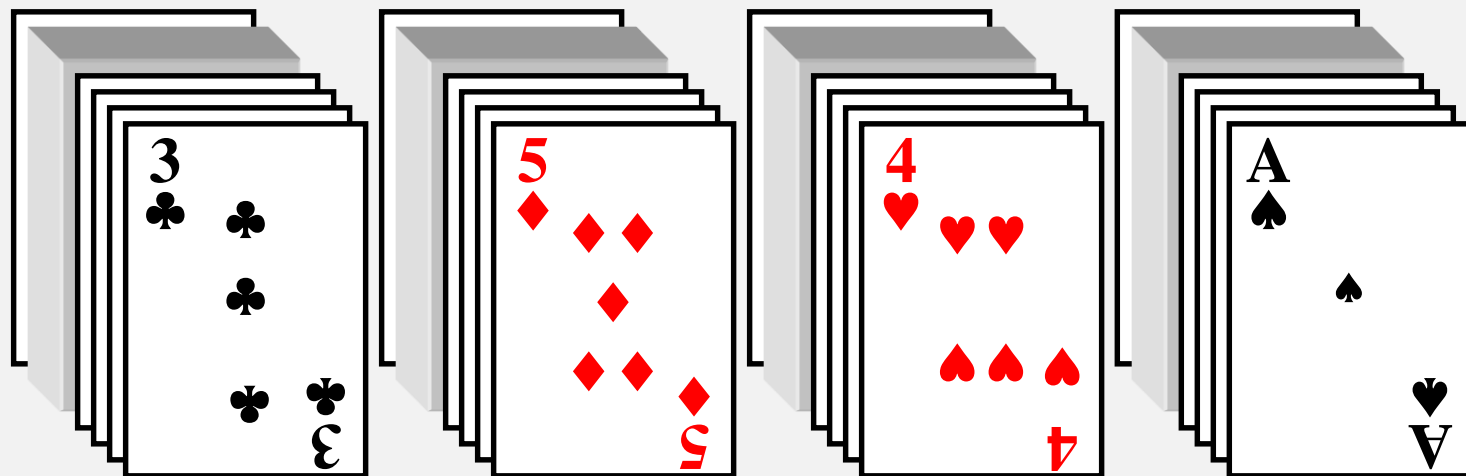


## 主位优先法

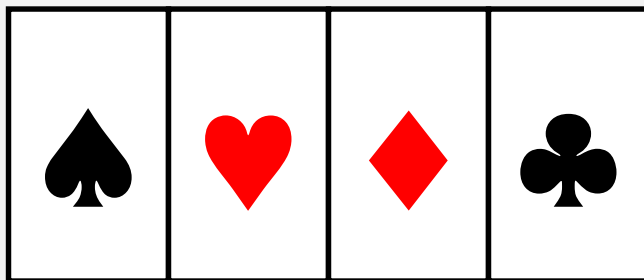
**主位优先法，简称 MSD 法：**

- ① 先按  $k^0$  排序分组；
- ② 同一组中记录的关键字  $k^0$  相等，接着对各组按  $k^1$  排序分成子组，重复这个过程，对后面的关键字依次排序分组；
- ③ 直到最后按最次位关键字  $k^d$  对各子组排序；
- ④ 然后将各组连接起来，便得到一个有序序列。

① **先排花色:** 比如, 建立4个花色的“桶”



② **再排面值:** 每个桶分别独立排序(可以采用以前介绍的任何排序方法)



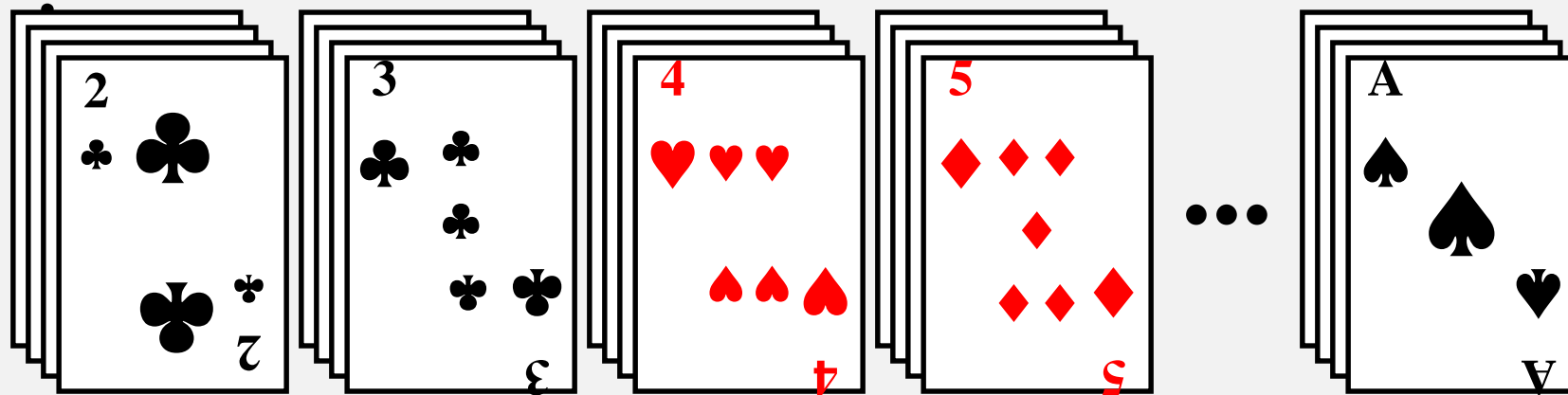
## 次位优先法排序

**次位优先法，简称 LSD 法：**

- ① 先从最低位  $k^{d-1}$  开始排序，**
- ② 接着对  $k^{d-2}$  进行排序，**
- ③ 依次重复，直到对  $k^0$  排序后便得到一个有序序列。**



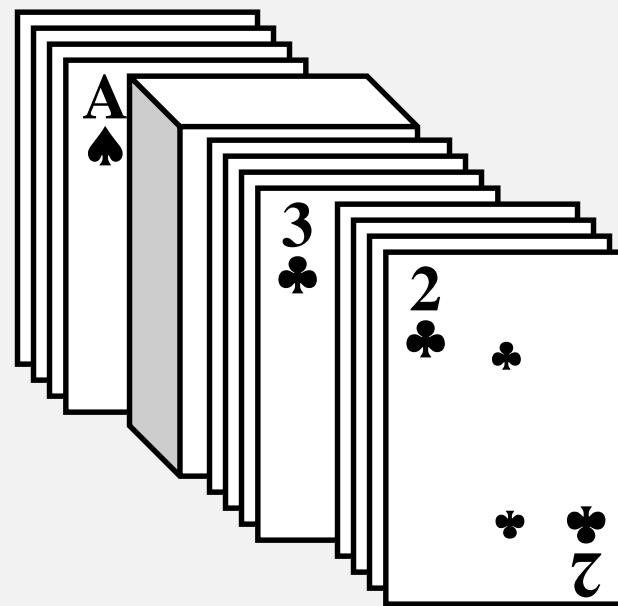
① **先排面值**: 建立13个面值的“桶”，把牌放入相应的桶中（这叫“分配”）



② 依次“收集”它们成为一叠牌；

③ **再排花色**: 建立4个桶进行再“分配”；

④ 再次“收集”它们成为一叠牌即告完成。





## 小结

### MSD 与 LSD 的不同特点

MSD

分治法：将序列逐层**分割**成若干子序列，然后对各子序列分别排序。

LSD

通过若干次**分配与收集**实现排序；不必分成子序列，对每个关键字都是整个序列参加排序。

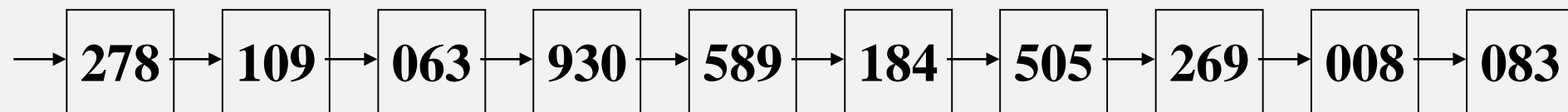
## 2、单关键字的基数排序（链式基数排序）

- 对于单个关键字的排序问题，可以将单个整数关键字按某种基数分解为多个关键字，再进行排序。
- 例如：将三位整数521，分解为5，2，1三个关键字，其中5为主位关键字，1为最次位关键字。

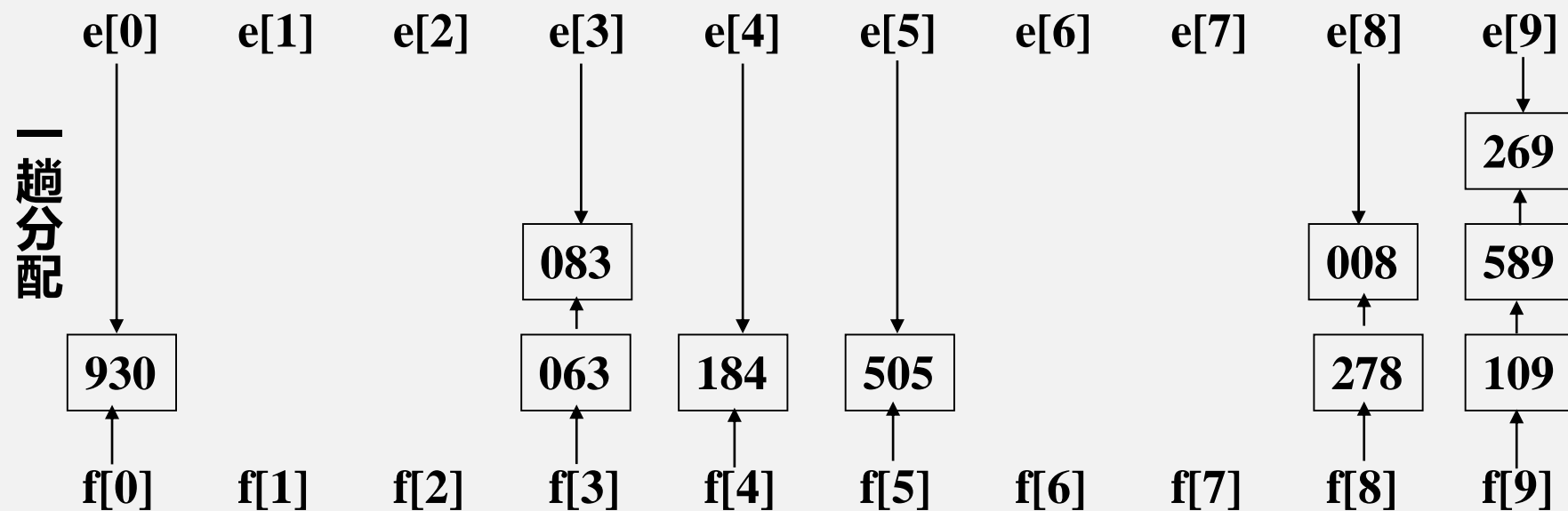
**单关键字的基数排序算法中，为减少所需辅助存储空间，可采用链表作存储结构，方法为：**

- ① 以静态链表存储待排记录，并令表头指针指向第一个记录；
- 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。

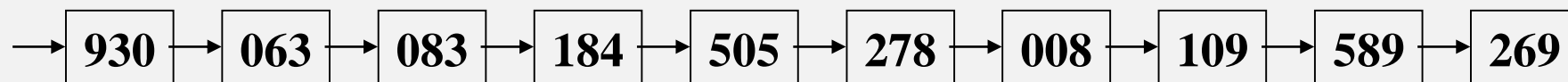
**例：以静态链表存储待排记录，并令表头指针指向第一个记录。**



**“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。**

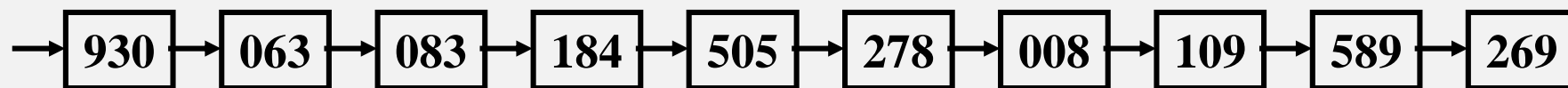


**一趟收集：按当前关键字位取值从小到大将各队列首尾相链成一个链表；**

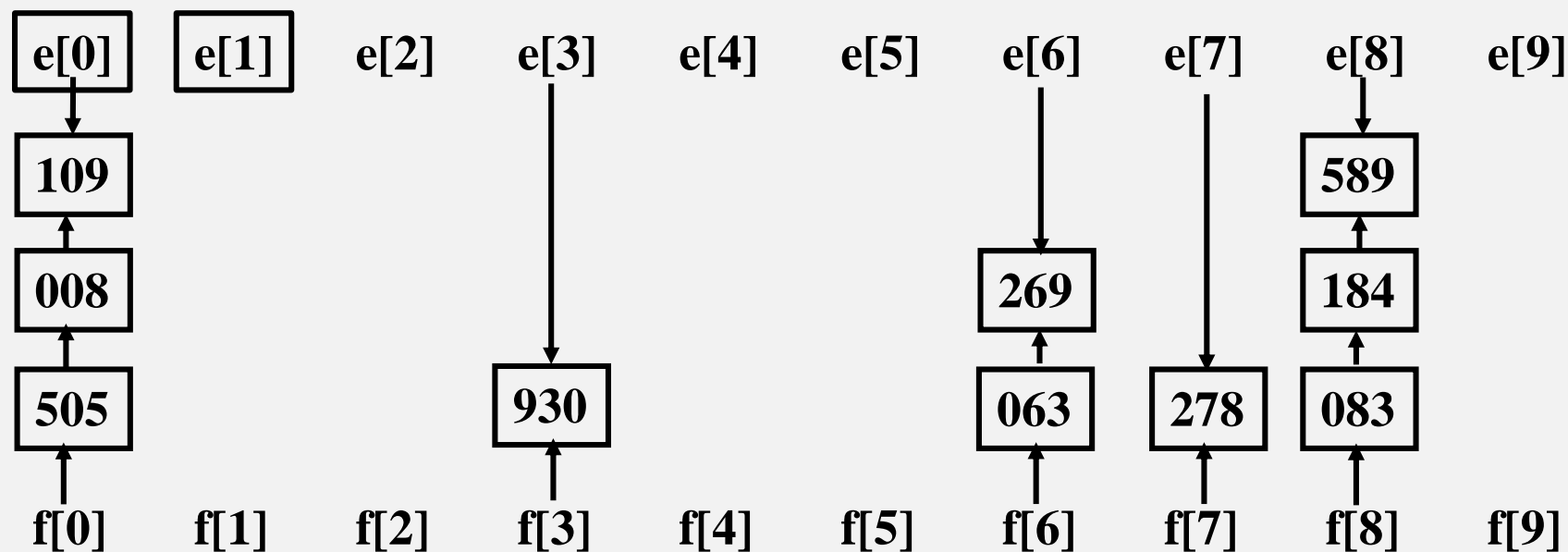




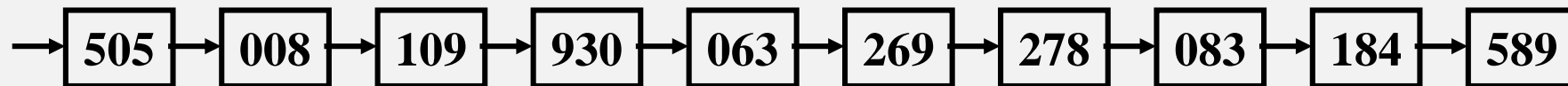
## 一趟收集:



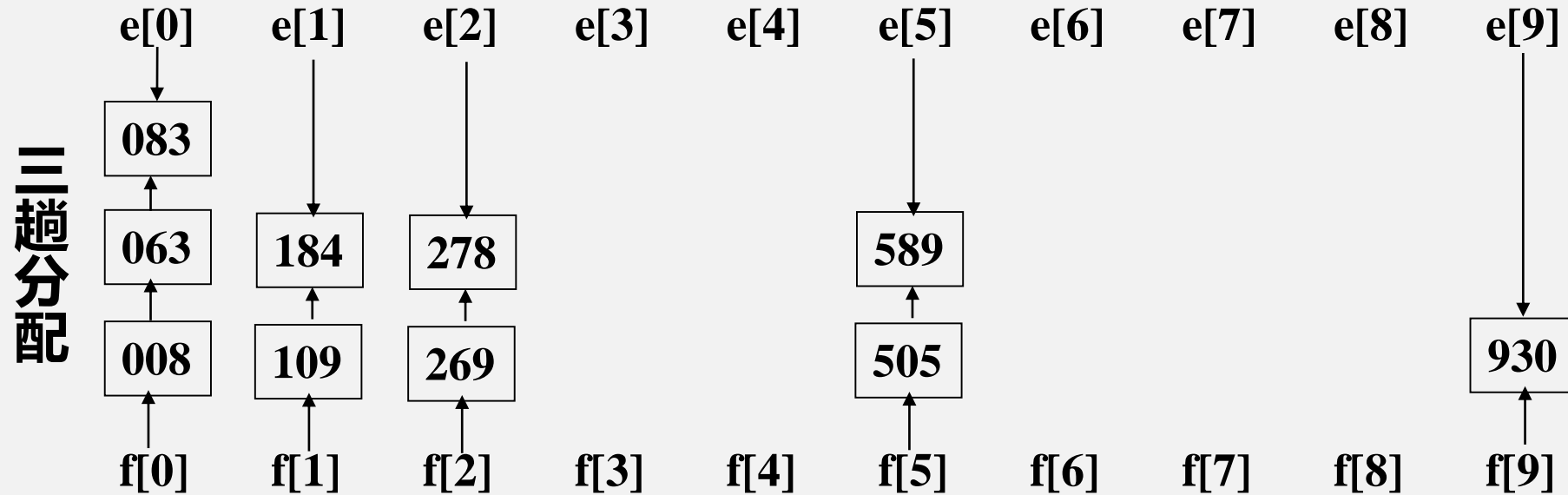
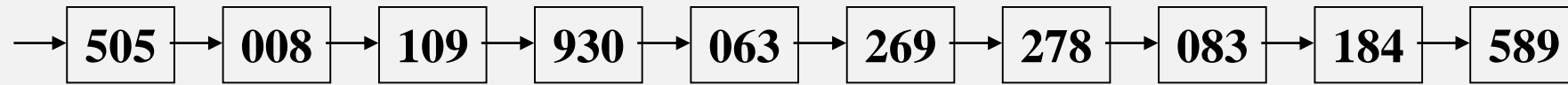
## 二趟分配



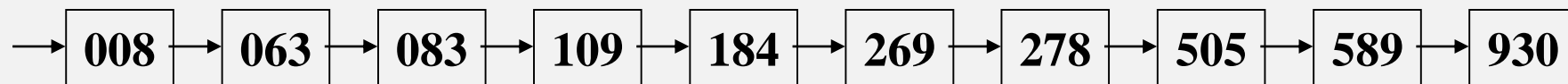
## 二趟收集:



## 二趟收集:



## 三趟收集:



为实现基数排序，用两个指针数组来分别管理所有的缓存(桶)，同时对待排序记录的数据类型进行改造，相应的数据类型定义如下：

```
#define BIT_key  8      /* 指定关键字的位数d  */
#define RADIX  10      /* 指定关键字基数r  */
typedef struct RecType
{
    char  key[BIT_key] ;      /* 关键字域  */
    infoType  otheritems ;
    struct RecType  *next ;
} SRecord, *f[RADIX] , *e[RADIX] ;
/*      桶的头尾指针数组      */
```

```

void Radix_sort(SRecord *head )
{int j, k, m ;
  SRecord *p, *q, *f[RADIX],
  *e[RADIX] ;
  for (j=BIT_key-1; j>=0; j--)
    /* 关键字的每位一趟排序 */
    { for (k=0; k<RADIX; k++)
      f[k]=e[k]=NULL ; /*初始化*/
      p=head ;

      while (p!=NULL) /*一趟分配*/
      { m=ord(p->key[j]) ;
        /* 取关键字的第j位 */
        if (f[m]==NULL) f[m]=p ;
        else e[m]->next=p ;
        p=p->next ;
      }
    }
}

```

```

head=NULL ; /*以head作为头指针收集*/
q=head ; /*q作为收集后的尾指针*/

/* 完成一趟排序的收集 */
for (k=0; k<RADIX; k++)
{ if (f[k]!=NULL)
  /*第k个队列不空则收集*/
  {if(head!=NULL q->next=f[k] ;
    else head=f[k] ;
    q=e[k] ;
  }
}

q->next=NULL ; /*修改收集链尾指针*/
} //for
}

```

# 算法评价：

时间复杂度：

假设：  $n$  —— 记录数

$d$  —— 关键字数

$rd$  —— 关键字取值范围

(如十进制为10)

分配（每趟）：  $T(n)=O(n)$

收集（每趟）：  $T(n)=O(rd)$

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

空间复杂度：  $S(n)=2rd$  个队列指针 +  $n$  个指针域空间



## 7.7 各种排序方法的综合比较

排序方法	平均时间复杂度	最坏情况下时间复杂度	额外空间复杂度	稳定性
简单选择排序	$O(N^2)$	$O(N^2)$	$O(1)$	不稳定
直接插入排序	$O(N^2)$	$O(N^2)$	$O(1)$	稳定
冒泡排序	$O(N^2)$	$O(N^2)$	$O(1)$	稳定
希尔排序	$O(N^d)(1 < d < 1.5)$	$O(N^2)$	$O(1)$	不稳定
堆排序	$O(N \log_2 N)$	$O(N \log_2 N)$	$O(1)$	不稳定
快速排序	$O(N \log_2 N)$	$O(N^2)$	$O(\log_2 N)$	不稳定
归并排序	$O(N \log_2 N)$	$O(N \log_2 N)$	$O(N)$	稳定
基数排序	$O(D(N+R))$	$O(D(N+R))$	$O(N+R)$	稳定

# 一、时间性能

## 1. 按平均时间性能来分，有三类排序方法：

时间复杂度为  $O(n\log n)$ ：快速排序、堆排序和归并排序，其中以**快速排序**为最好。

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，其中以**直接插入**为最好，特别是对那些对关键字基本有序的记录序列尤为如此。

时间复杂度为  $O(n)$ ：基数排序。

2. 当待排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、冒泡和简单选择)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**

- 1. 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。**
- 2. 对于不稳定的排序方法，只要能举出一个实例说明即可。**
- 3. 快速排序、堆排序和希尔排序是不稳定的排序方法。**