第5章 排序与选择

5.0 引言

1、学习要点:

- 理解排序问题的实质
- 掌握简单排序算法的设计思想与分析方法
- 掌握快速排序算法的设计思想与分析方法
- 理解随机化思想在快速排序算法中的应用
- 理解三数取中划分算法和三划分算法的改进策略
- 掌握合并排序算法的基本思想及实现方法
- 掌握计数排序算法的设计思想与分析方法
- 掌握桶排序算法的设计思想与分析方法
- 理解线性时间排序与基于比较排序算法的差别和适用范围
- 掌握平均情况下线性时间选择算法的设计思想与分析方法
- 掌握最坏情况下线性时间选择算法的设计思想与分析方法

2、基本概念与术语：

- (1)数据对象(记录)的键值key和卫星数据
- (2)稳定的排序算法与不稳定的排序算法
- (3)就地排序与非就地排序
- (4)排序中的基本操作：比较和移动(交换)
- (5)排序算法复杂度的度量:

  { 算法步数
  { 键值比较次数

3、排序算法分类

- 基于比较的排序算法
  - 交换排序
    - 冒泡排序
    - 快速排序
  - 插入排序
    - 直接插入排序
    - 二分插入排序
    - **Shell**排序
  - 选择排序
    - 简单选择排序
    - 堆排序
  - 合并排序
- 基于数字和地址计算的排序方法
  - 计数排序
  - 桶排序
  - 基数排序

返回章目录

5.1 简单排序算法

5.1.1 冒泡排序

(1)基本思想：

- 将第一个记录的关键字与第二个记录的关键字进行比较，若为逆序(即：**a[0].key>a[1].key)**，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
- 重复上述过程，直到“在一趟排序过程中没有进行过交换记录的操作”为止

例：

初始关键字	第一趟	第二趟	第三趟	第四趟	第五趟
38	38	38	13	13	13
49	49	13	27	27	27
76	13	27	30	30	30
13	27	30	38	38	
27	30	49	49		
30	76	76			
97	97				

一趟	二趟	三趟	四趟	五趟	六趟	七趟	八趟	九趟
13								
14								
12								
10								
08								
03								
06								
11								
05								
15								
04								
16								
01								
09								
02								
07								

(2)算法描述

```
void bubble(Item a[ ]， int n)
{// 冒泡排序
  for (int i = n; i > 1; i--)
    for (int j = 0; j < i - 1; j++)
      if (a[j] > a[j+1])
        Swap(a[j]，a[j+1]);
}
```

(3)算法分析

- 时间复杂度
  - 最好情况（正序）
    - 比较次数：$\sum_{i=1}^{n-1}(n-i)=\frac{1}{2}(n^2-n)$
    - 交换次数：**0**
  - 最坏情况（逆序）
    - 比较次数：$\sum_{i=1}^{n-1}(n-i)=\frac{1}{2}(n^2-n)$
    - 交换次数：$\sum_{i=1}^{n}(n-i)=\frac{1}{2}(n^2-n)$

∴ T(n)=O(n²)

(4)改进的冒泡排序

```
void bubble_Modified(Item a[ ],int n)
{// 改进的冒泡排序
  int flag=1;
  for (int i = n; i>1 && flag; i--)
  {
    flag=0;
    for (int j = 0; j<i-1 ; j++)
      if (a[j] > a[j+1])
      {
        flag=1;
        Swap(a[j]，a[j+1]);
      }
  }
}
```

(5)改进算法分析

- 时间复杂度
  - 最好情况（正序）
    - 比较次数：**n-1**
    - 交换次数：**0**
  - 最坏情况（逆序）
    - 比较次数：$\sum_{i=1}^{n-1}(n-i)=\frac{1}{2}(n^2-n)$
    - 交换次数：$\sum_{i=1}^{n}(n-i)=\frac{1}{2}(n^2-n)$

∴ T(n)=O(n²)

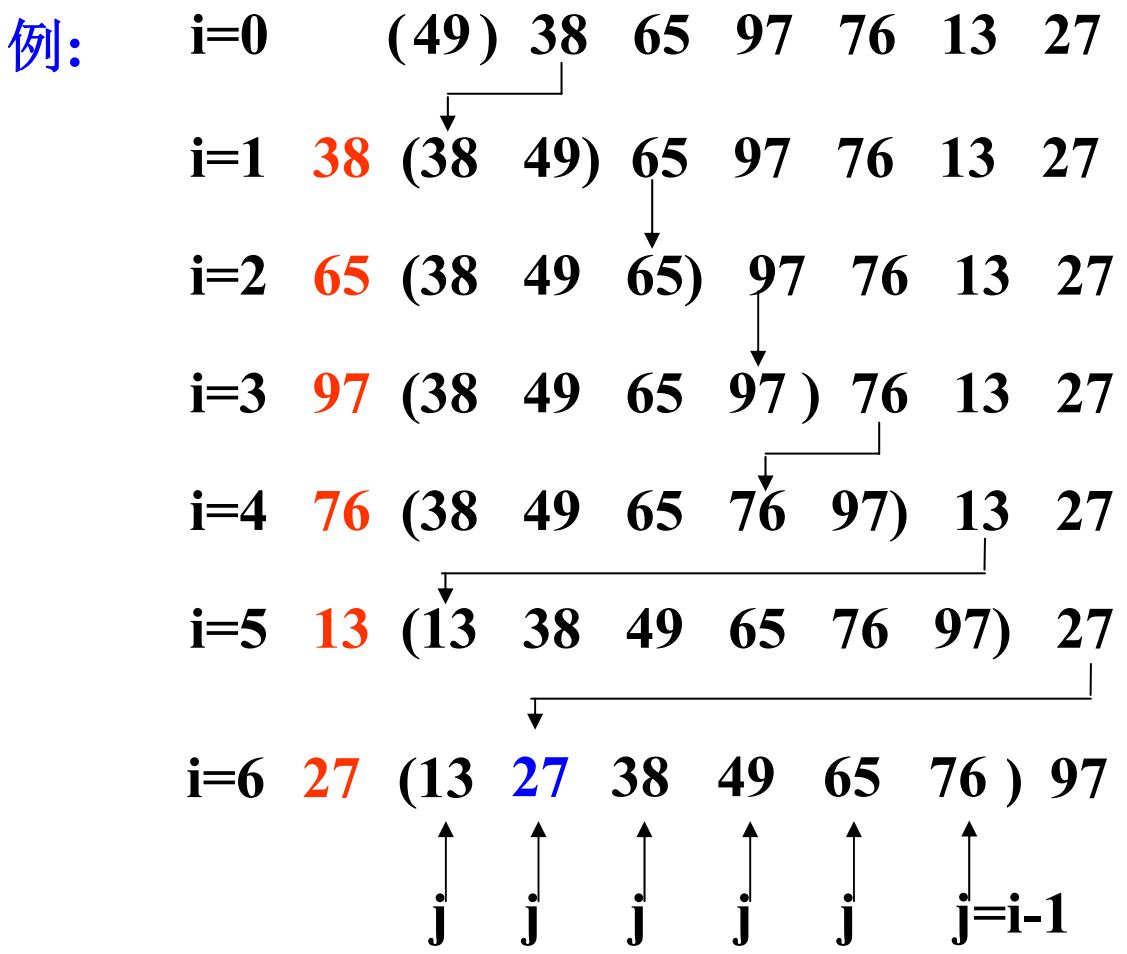
5.1 简单排序算法

5.1.2 直接插入排序

(1)基本思想：

整个排序过程为n-1趟插入，即先将序列中第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。

例：



排序结果：(13 27 38 49 65 76 97)

(2)算法描述

```
void insertion ( Item a[]， int n)
{// 插入排序算法
   int i,j;
   Item temp;
   for (i=1; i<n; i++)
   {
      temp = a[i];
      j=i-1;
      while(temp<a[j]&&j>=0)
      {
         a[j+1]=a[j];
         j--;
      }
      a[j+1]=temp;
   }
}
```

(3)算法分析

时间复杂度

若待排序记录按关键字从小到大排列(正序)

- 关键字比较次数：$\sum_{i=1}^{n-1} 1 = n-1$
- 记录交换次数：0

若待排序记录按关键字从大到小排列(逆序)

- 关键字比较次数：$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$
- 记录交换次数：$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$

∴ $T(n)=O(n^2)$

5.1.3 二分插入排序

- （1）基本思想：用二分查找方法确定插入位置的排序

例

i=1 (30) 13 70 85 39 42 6 20

i=2 13 (13 30) 70 85 39 42 6 20

⋮

i=7 6 (6 13 30 39 42 70 85) 20

i=8 20 (6 13 30 39 42 70 85) 20

s m j

i=8 20 (6 13 30 39 42 70 85) 20

s m j

i=8 20 (6 13 30 39 42 70 85) 20

s mj

i=8 20 (6 13 30 39 42 70 85) 20

j s

i=8 20 (6 13 20 30 39 42 70 85)

- （2）算法描述

```
void Binary_InsertionSort(T a[],int n)
{  int i,j,s,m,k;
   T x;
   for(i=1;i<n;i++)
   {
      x=a[i];
      s=0; j=i-1;
      while(s<=j)
      {  m=(s+j)/2;
         if(x<a[m])  j=m-1;
         else s=m+1;
      }
      for(k=i-1;k>=s;k--)
         a[k+1]=a[k];
      a[s]=x;
   }
}
```

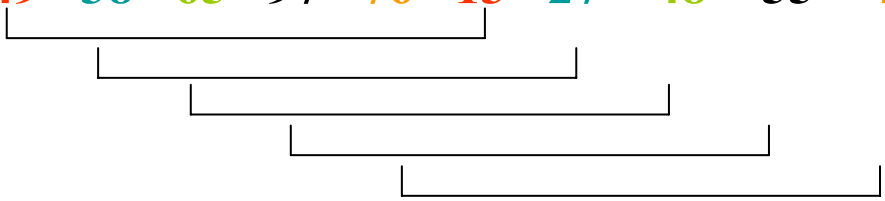
- （3）算法分析：
  - 时间复杂度：$T(n)=O(n^2)$
  - 空间复杂度：$S(n)=O(n)$

- **5.1.4** 希尔排序(缩小增量法)
  - （1）基本思想：先取一个正整数**d1<n**，把所有相隔**d1**的记录放一组，组内进行直接插入排序；然后取**d2<d1**，重复上述分组和排序操作；直至**di=1**，即所有记录放进一个组中排序为止。

例　初始：49　38　65　97　76　13　27　48　55　4

取d1=5

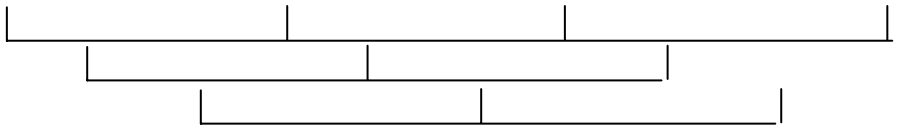
一趟分组：49　38　65　97　76　13　27　48　55　4



一趟排序：13　27　48　55　4　49　38　65　97　76

取d2=3

二趟分组：13　27　48　55　4　49　38　65　97　76



二趟排序：13　4　48　38　27　49　55　65　97　76

取d3=1

三趟分组：13　27　48　55　4　49　38　65　97　76

三趟排序：4　13　27　38　48　49　55　65　76　97

- （2）算法描述

```
void ShellSort(T a[],int n,int increment[],int incresize) //incresize表示增量个数
{ //按增量序列increment[]对长度为n的元素a[]作希尔排序
  int i,j,incr;
  T tmp;
  for(int m=0;m<incresize;m++)
  {   //按递增序列进行多遍插入排序
     incr=increment[m];    //取递增量
     for(i=incr;i<n;i++)
     {  //对多个子序列进行插入排序
        //将a[i]插入有序增量子表中
        if(a[i]<a[i-incr])
        tmp=a[i];  //用tmp来保存要插入的元素
        //在有序增量子表中找插入位置
        //从该位置开始将子表中元素依次后移
        for(j=i-incr;j>0 && tmp<a[j];j-=incr)
            a[j+incr]=a[j];
        a[j+incr]=tmp; //插入待排序元素
     }
  }
}
```

- （3）希尔排序特点
  - 子序列的构成不是简单的“逐段分割”，而是将相隔某个增量的记录组成一个子序列
  - 希尔排序可提高排序速度，因为
    - 分组后n值减小，n^2更小，而$T(n)=O(n^2)$,所以$T(n)$从总体上是减小了
    - 关键字较小的记录跳跃式前移，在进行最后一趟增量为1的插入排序时，序列已基本有序
- （4）增量序列取法
  - 无除1以外的公因子
  - 最后一个增量值必须为1
  - 比如：5　3　1

5.1 简单排序算法

5.1.5 选择排序

（1）基本思想：

- 首先通过n-1次关键字比较，从n个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


k k k
i=0 初始： [13 38 65 97 76 49 27]
j j j j j j
k k
i=1 一趟： 13 [27 65 97 76 49 38]
j j j j j
二趟： 13 27 [65 97 76 49 38]
三趟： 13 27 38 [97 76 49 65]
四趟： 13 27 38 49 [76 97 65]
五趟： 13 27 38 49 65 [97 76]
六趟： 13 27 38 49 65 76 [97]
排序结束： 13 27 38 49 65 76 97

（2）算法描述

```
void selection (Item a[]，int n)
{// 选择排序算法
   int i,j,k;
   Item temp;
   for (i=0; i<n-1; i++)
   {
      //在a[i],a[i+1],…,a[n-1]中选最小的a[k]
      for(k=i,j=i+1;j<n;j++)
         if(a[k]>a[j])
            k=j;
      if(k!=i)
      {//a[i]与a[k]交换
         temp=a[i];a[i]=a[k];a[k]=temp;
      }
   }
}
```

- **(3)算法评价**
  - 时间复杂度
    - 记录交换次数：
      - 最好情况：**0**
      - 最坏情况：**n-1**
    - 比较次数：$\sum_{i=1}^{n-1}(n-i)=\frac{1}{2}(n^2-n)$

∴ T(n)=O(n²)

- （4）改进算法描述

```
int Max(T a[ ]，int n,int& m)
{// 确定a[0:n-1]中最大元素的下标
   int pos = 0;
   for (int i = 1; i < n; i++)
      if (a[pos] < a[i])
      {
         pos = i;
         m++;
      }
   return pos;
}

void SelectionModified(T a[]，int n)
{// 选择排序算法
   int m=0;
   for (int size = n; size > 1; size--)
   {
      int j = Max(a，size);
      Swap(a[j]，a[size-1]);
      if(m==size-1)
        break;    }
}
```

- **(5)改进算法分析**
  - 时间复杂度
    - 记录交换次数：
      - 最好情况：**0**
      - 最坏情况：**n-1**
    - 比较次数：
      - 最好情况：**n-1**
      - 最坏情况：$\sum_{i=1}^{n-1}(n-i)=\frac{1}{2}(n^2-n)$

∴ T(n)=O(n²)

返回章目录

5.2 快速排序算法

5.2.0 算法的基本策略思想

——非平衡、预处理二分分治

第一步 分解： 对于给定的数组**a[p:r],p<r,**以**x=a[p]**为基准，调整**a[p:q]**，使得能够确定一个下标**q, p≤q≤r**，将**a[p:r]**分成**3**段,即：**a[p:q-1]**，**a[q]**和**a[q+1:r],** 满足**a[p:q-1]**中的任一元素都不大于**x,** 同时，**a[q+1:r]**中的任一元素都不小于**x**；这叫划分**(Partition)** 。

第二步 递归求解： 将这种策略依次分别递归地运用于**a[p:q-1]**和 **a[q+1:r],** 使得**a[p:q-1]**和**a[q+1:r]**分别从小到大排好序;从而达到数组**a[p:r]**从小到大就地排好序。

第三步 合并：由于对**a[p:q-1]**和**a[q+1:r]**的排序是就地进行的，所以在**a[p:q-1]**和**a[q+1:r]**都已排好序后不需要执行任何计算， **a[p:r]**就已排好序。

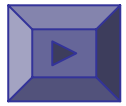
5.2 快速排序算法

5.2.1 快速排序(QuickSort)算法的实现:

```
void QuickSort (T a[ ], int p, int r)//p,r都是下标
{
  if (p<r) {
                int  q=Partition(a, p, r);//划分
                QuickSort (a, p, q-1); //对左段快速排序
                QuickSort (a, q+1, r); //对右段快速排序
               }
}
```

//对a[0:n-1]快速排序只要调用QuickSort (a,0,n-1);

5.2.2 划分(Partition) 的实现

动画演示 

```
int Partition (T a[ ], 
{
    int i = p, j = r + 1
    T x=a[p];
    while (true) {
        while (a[++i] <x
        while (a[--j] >x
        if (i >=j)  breal
        Swap(a[i],  a[j]
    }
    a[p]=a[j];
    a[j]=x;
    return j;
}
```

问题：

1、a[++i] <x能否改成a[++i] <=x???

2、是否一定要取a[p]作为基准值？

取a[r]可以吗？=〉程序该如何修改？

是否有其它基准值选择方案？

3、当n很小时，快速排序很慢。

=〉改进办法1：用简单排序算法来处理较小数组

=〉改进办法2：当快速排序的子数组小于某个长度时，什么也不要做，使用插入排序。（实验表明：当n=16时选用插入排序这种组合效率最高。）

4、递归快速排序—〉非递归快速排序？

思路：用栈模拟，如何实现？

	K_m	K_{m+1}	K_{m+2}	…	…	…	…	…	…	…	…	…	K_n	K_{n+1}
开　始	27	99	0	8	13	64	86	16	7	10	88	25	90	100

5.2.3 算法的性能分析

快速排序的运行时间与划分是否平衡密切相关。

对于输入序列a[0:n-1]，**Partition**的计算时间显然为O(n)。它的最坏情况发生在划分产生的两段分别包含n-1个元素和1个元素的时候。如果算法每一次调用**Partition**都出现这种不平衡划分，则**QuickSort (a,0,n-1)**的耗时T(n)满足

$$T(n)=\begin{cases} O(1) & n\le 1 \\ T(n-1)+O(n) & n>1 \end{cases}$$

根据大O的定义有：$\exists c>0$，使得：

$$T(n)\le T(n-1)+cn$$

同理：$T(n-1)\le T(n-2)+c(n-1),\cdots$

从而有：$T(n)\le T(n-2)+cn+c(n-1)\le\cdots$

$$\le T(n-k)+cn+c(n-1)+\cdots+c(n-k+1)\le\cdots$$

$$\le T(1)+cn(n+1)/2$$

$$=O(1)+cn(n+1)/2$$

$$\therefore\ T(n)=O(n^2)$$

算法的性能分析：

在最好情况下，每次划分所取的基准都恰好为中值，即每次划分都产生大小相当的2段，则T(n)满足

$$T(n)=\begin{cases} O(1) & n\le 1 \\ 2T(n/2)+O(n) & n>1 \end{cases}$$

实践中，快速排序算法是相当快的，所以称之为快速排序。

根据大O的定义有：$\exists c>0$，使得：

$$T(n)\le 2T(\frac{n}{2})+cn$$

同理：$T(\frac{n}{2})\le 2T(\frac{n}{2^2})+c\frac{n}{2},\cdots$

从而有：$T(n)\le 2^2T(\frac{n}{2^2})+2cn\le\cdots$

$$\le 2^kT(\frac{n}{2^k})+kcn\le\cdots$$

由$\frac{n}{2^k}=1\Rightarrow k=\log n$

$$\therefore\quad T(n)\le nT(1)+cn\log n$$

$$=O(n)+cn\log n$$

$$\therefore\quad T(n)=O(n\log n)$$

5.2 快速排序算法

5.2.4 随机快速排序算法

(1) 算法的动因

可以证明，如果在a[0:n-1]中选择的作为划分**(Partition)**的基准值是随机的，那么，快速排序算法在平均情况下的时间复杂性就是**O(nlogn)** ，即达到基于比较的排序算法类中的最好水平。

因此，人们提出:划分**(Partition)** 的基准值不固定为数组的第一个值或是最后一个值而是随机在**a[p:r]**中地挑选。其他思想不变，这就是随机快速排序算法。

5.2 快速排序算法

5.2.4 随机快速排序算法(续)

(2)随机版的划分的实现

```
int RandomizedPartition (T a[ ], int p, int r)
{
  int i = Random(p, r);
  Swap(a[i], a[p]);
  return Partition (a, p, r);
}
```

5.2 快速排序算法

5.2.4 随机快速排序算法(续)

（3）随机快速排序算法的实现

```
void RandomizedQuickSort (T a[ ],  int p,  int r)
{
  if (p<r) {
        int q=RandomizedPartition(a, p, r);
        RandomizedQuickSort (a, p, q-1); //对左段排序
        RandomizedQuickSort (a, q+1, r); //对右段排序
  }
}  // 对a[0:n-1]随机快速排序只要调用RandomizedQuickSort (a,0,n-1);
```

5.2 快速排序算法

5.2.5 小者优先递归快速排序算法

```
void quicksort(Item a[], int p, int r)
{
  int i;
  if(p<=r)  return;
  i=partition(a,p,r);
  if(i-1>r-i) {quicksort(a,i+1,r); quicksort(a,p,i-1);}
  else        {quicksort(a,p,i-1); quicksort(a,i+1,r);}
}
```

优点：在最坏情况下只耗费logn栈空间。

5.2 快速排序算法

5.2.6 三数取中划分算法

对快速排序算法的划分对称性还有可以改进的余地。

三数取中划分算法的主要思想基于划分基准的选取。对于待排序数组a[p:r]，算法选取a[p]，a[r]，a[(p+r)/2]这三个数的中位数作为划分基准，从而改进划分的对称性。

三数取中快速排序算法比原快速排序算法的效率提高20%~25%。

5.2 快速排序算法

5.2.7 三划分快速排序算法

- 当待排序数组**a[p:r]**中有大量键值相同的元素时，采用三划分快速排序算法可以明显改进算法的性能。
- 算法只需要对左右两段数组递归排序。
- 另一方面，当待排序数组**a[p:r]**中没有大量键值相同的元素时，三划分快速排序算法也不降低原快速排序算法的效率。

5.3 合并排序算法(非就地)

5.3.1 递归版的合并排序算法

(1)基本策略思想——平衡、简单二分分治：

将待排序元素叙列简单地分成大小大致相等的左右两段，接着依次分别对这两段子序列递归地进行合并排序，然后利用这两段子序列已得到的有序性，将它们有序地合并在一个工作区，最后用工作区中排好序的全序列更新原待排序的元素序列成为所要求的排好序的元素序列。

5.3 合并排序算法(非就地)

5.3.1 递归版的合并排序算法

(2)算法的实现

```
void MergeSort(T a[ ],  int left,  int right)
{ if (left<right) {//至少有2个元素
  int i=(left+right)/2;  //取中点
  MergeSort(a, left, i);
  MergeSort(a, i+1, right);
  Merge(a, b, left, i, right); //合并到数组b
  Copy(a, b, left, right); //将b复制回a
  }
} //对a[0:n-1]排序只要调用MergeSort(a,0,n-1)
```

5.3 合并排序算法(非就地)

5.3.1 递归版的合并排序算法

(2)算法的实现(续)

```
                    // 有序地合并c[l:m]和c[m+1:r]到d[l:r]
void Merge(T c[ ], T d[ ], int l, int m, int r)
{ int i = l, j = m+1, k = l;
  while ((i <= m) && (j <= r))
     if (c[i] <= c[j]) d[k++] = c[i++];
     else              d[k++] = c[j++];
  if (i > m)
       for (int q = j; q <= r; q++)
           d[k++] = c[q];// c[l:m]到位
  else for (int q = i; q <= m; q++)
           d[k++] = c[q];// c[m+1:r]到位
}
```

R.Sedgewick发明的一个优化归并排序方法：

```
void MergeSort(T * a,T * b,int left,int right)
{   int i,j,k,mid;
    if (left<right)//至少有2个元素
    {    mid=(left+right)/2;  //取中点
         MergeSort(a,b,left,mid);
         MergeSort(a,b,mid+1,right);
         for(i=mid;i>=left;i--)
             b[i]=a[i];
         //本算法巧妙之处：将第二个子数组中的元素顺序颠倒过来
         for(j=1;j<=right-mid;j++)
             b[right-j+1]=a[j+mid];
         for(i=left,j=right,k=left;k<=right;k++)
             if(b[i]<b[j])   a[k]=b[i++];
             else            a[k]=b[j--];
    }
}
```

5.3 合并排序算法(非就地)

5.3.1 递归版的合并排序算法

(3) 算法的复杂度

显然,Copy可在**O(n)**时间内完成。也容易理解，Merge可在**O(n)**时间内完成。因此合并排序算法对n个元素进行排序，在最坏情况下所需的计算时间**T(n)**满足

$$T(n)=\begin{cases} O(1) & n\le 1 \\ 2T(n/2)+O(n) & n>1 \end{cases}$$

解此递归方程可知**T(n)=O(nlogn)**。

由于基于比较的排序问题的计算时间下界为Ω**(nlogn)**，故合并排序算法是一个渐近最优算法。

但它需要加倍的空间,即需要**O(n)**的辅助空间。

5.3 合并排序算法(非就地)

5.3.2 非递归版的合并排序算法

(1)基本思想

事实上，算法MergeSort的递归过程只是将待排序数组一分为二，直至待排序数组中只有1个元素为止(它已有序)。然后不断地合并相邻的2个已有序的数组段。按此机制，可以首先将数组a中相邻元素两两配对。用Merge算法将它们合并，得到n/2个长度为2的排好序的数组段。然后再将它们Merge成长度为4的排好序的数组段，...，如此继续下去，直至整个数组排好序。


开始
25
57
48
37
12
92
86
33
重新演示

5.3 合并排序算法(非就地)

5.3.2 非递归版的合并排序算法

(2) 算法的实现

```
void MergeSort(T a[ ], int n)
{   T *b = new T [n];
    int s = 1;
    while (s < n) {
        MergePass(a, b, s, n);
        //合并到数组b
        s += s;
        MergePass(b, a, s, n);
        //合并到数组a
        s += s; }
}
```

5.3 合并排序算法(非就地)

5.3.2 非递归版的合并排序算法

(3) 需要的函数

```
                    // 合并x[ ]中大小为s的相邻子数组到y[ ]中
void MergePass(T x[ ], T y[ ], int s, int n)
{ int i = 0;
  while (i <= n - 2 * s) {
     Merge(x, y, i, i+s-1, i+2*s-1);
     i = i + 2 * s;          }
     // 剩下的元素个数少于2s
  if (i + s < n)  Merge(x, y, i, i+s-1, n-1);
  else      for (int j = i; j <= n-1; j++)
            y[j] = x[j];
}
```

5.3 合并排序算法(非就地)

5.3.2 非递归版的合并排序算法

(4)算法的复杂度分析

容易理解**Merge(c，d，*l*，m，r)**只需**O(r-*l*+1)**的时间,因而**MergePass(x，y，s，n)**只需

O(n/(2s)*(2s))= O(n)

的时间。从而算法**MergeSort(a，n)**只需要**O(nlogn)**的时间，因为其中的主循环**while**的循环体只需**O(n)**的时间，而循环次数只有**O(logn)**次。算法需要的空间为**O(n)**。

5.3 合并排序算法(非就地)

5.3.3 自然合并排序算法

(1)基本思想

事实上，任意给定的数组a都是由不多于n段自然有序的子数组拼接起来的，如{4，8，3，7，1，5，6，2}就是由自然排好序的子数组{4，8}，{3，7}，{1，5，6}和{2}拼接起来的。而且可以在O(n)的时间里把这些子数组的界限找出来。因此我们不妨按照这种自然的有序段，通过调用

Merge (c，d，*l*，m，r)反复地合并其相邻的两段，最后也可达到将**a** 排序的目的。例如由{4，8，3，7，1，5，6，2}⇒{4，8}，{3，7}，{1，5，6}和{2}⇒{3，4，7，8}和{1，2，5，6}⇒{1，2，3，4，5，6，7，8}。

返回章目录

5.4 特殊有序集的线性时间排序算法

5.4.0 引言

到目前为止，所讨论的排序算法有2个共同的特点：

(1)它们要排序的集合的全集的基数没有限制。

(2)它们用于确定排序结果的主要运算是输入元素间的比较。这类排序算法称为基于比较的排序算法。基于比较的排序算法的计算时间下界是$\Omega(n\log n)$。

本节讨论要排序的集合的全集的基数m有限制的情形。对这种情形的排序，有比基于比较更有效的算法——线性时间排序算法，但是，它们都要以花费更多的空间为代价。

5.4 特殊有序集的线性时间排序算法

5.4.1 计数排序算法

(1)算法的基本思想：

对于要排序的a[0:n-1]中的每一个元素，设法计算出它在最终排序结果序列中的序号，然后对号入座。

设全集S的基数为m。由于S的有序性，我们可以按照它的序，给S的元素设立一个计数器数组c[0:m]。

其中的c[i]用来统计S的第i个元素出现在a[0:n-1]中的次数，i=1,2,3,...,m。由c[0:m]便可计算出a[0:n-1]中每一个元素在排序结果序列中的序号，从而解决a[0:n-1]的排序问题。

5.4 特殊有序集的线性时间排序算法

5.4.1 计数排序算法

(2)算法的实现：

```
void CountingSort(int m,  int a[ ], int n, int b[ ])
{ int c[m+1];
  for (int i = 0; i <= m; i++) c[i] = 0;
  for (int i = 0; i < n; i++) c[a[i]] += 1;
  //c[i]中存放的是a中键值等于i的元素个数
  for (int i = 1; i < =m; i++) c[i] += c[i-1];
  //c[i]中存放的是a中键值小于或等于i的元素个数
  for (int i = n; i > 0; i--) {
     b[c[a[i-1]]-1]=a[i-1];  c[a[i-1]] - = 1; //具有排序的稳定性
     }
 }
```

5.4 特殊有序集的线性时间排序算法

5.4.1 计数排序算法

(3)算法的复杂度分析：

对数组**c**初始化需要**O(m)**的时间。

统计出现在a[0:n-1]中的元素的出现次数需要**O(n)**的时间。

对所有**i**统计出现在a[0:n-1]中的元素值小于或等于i **(1≦i≦m)** 的元素个数需要**O(m)**的时间。

最后，让a[0:n-1]中的所有元素到达排序结果数组**b**中正确位置需要**O(n)**的时间。这样，整个算法所需的计算间为**O(m+n)**。当**m=O(n)**时算法的计算时间复杂性为**O(n)**。

算法需要的空间显然为O(m)。

5.4 特殊有序集的线性时间排序算法

5.4.2 桶排序算法

(1) 算法的基本思想：

给全集的每一个元素键值设一个相应的桶，并将要排序的元素按键值对号入桶，让同一个桶内的元素保有它们被输入时的相对顺序；然后利用全集的有序性，按键值的序收集各相应的桶中的元素。

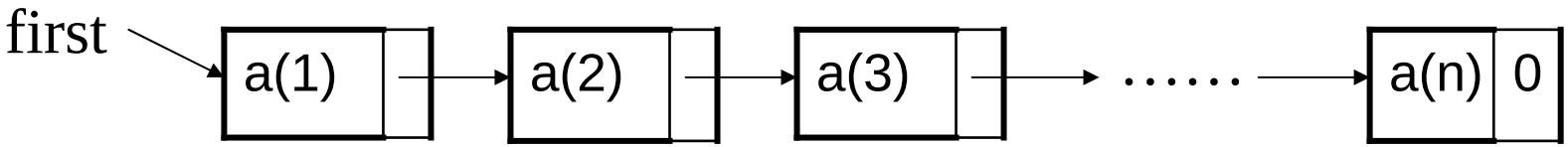
这样，所得到的元素序列就是所要求的排好序的序列。

5.4 特殊有序集的线性时间排序算法

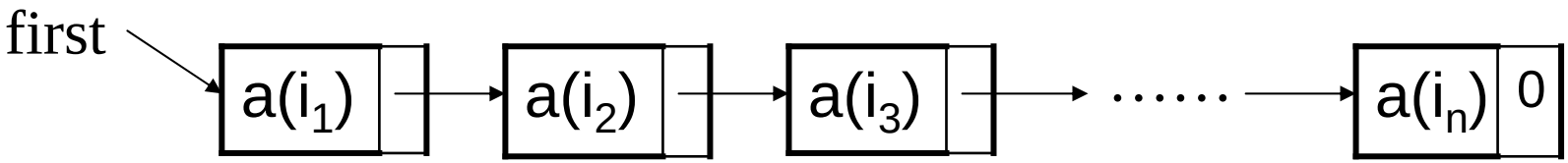
5.4.2 桶排序算法

(2) 实现算法的准备—数据结构的选择

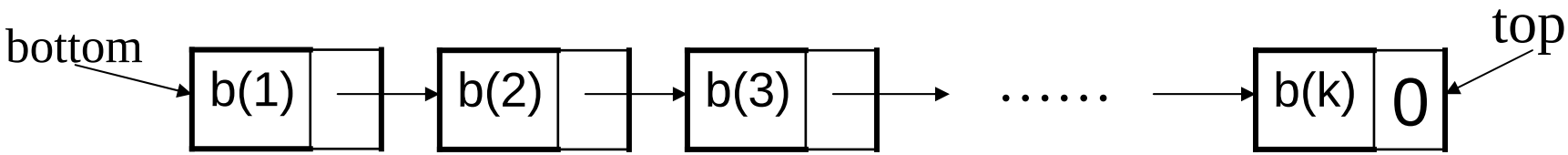
- 待排序的元素序列：用单链实现的表List<T>



- 排好序的元素序列：用单链实现的表List<T>



- 同一个桶中的元素序列：用单链实现的队列。它以分别指向队首和队尾结点的两个指针为标识。

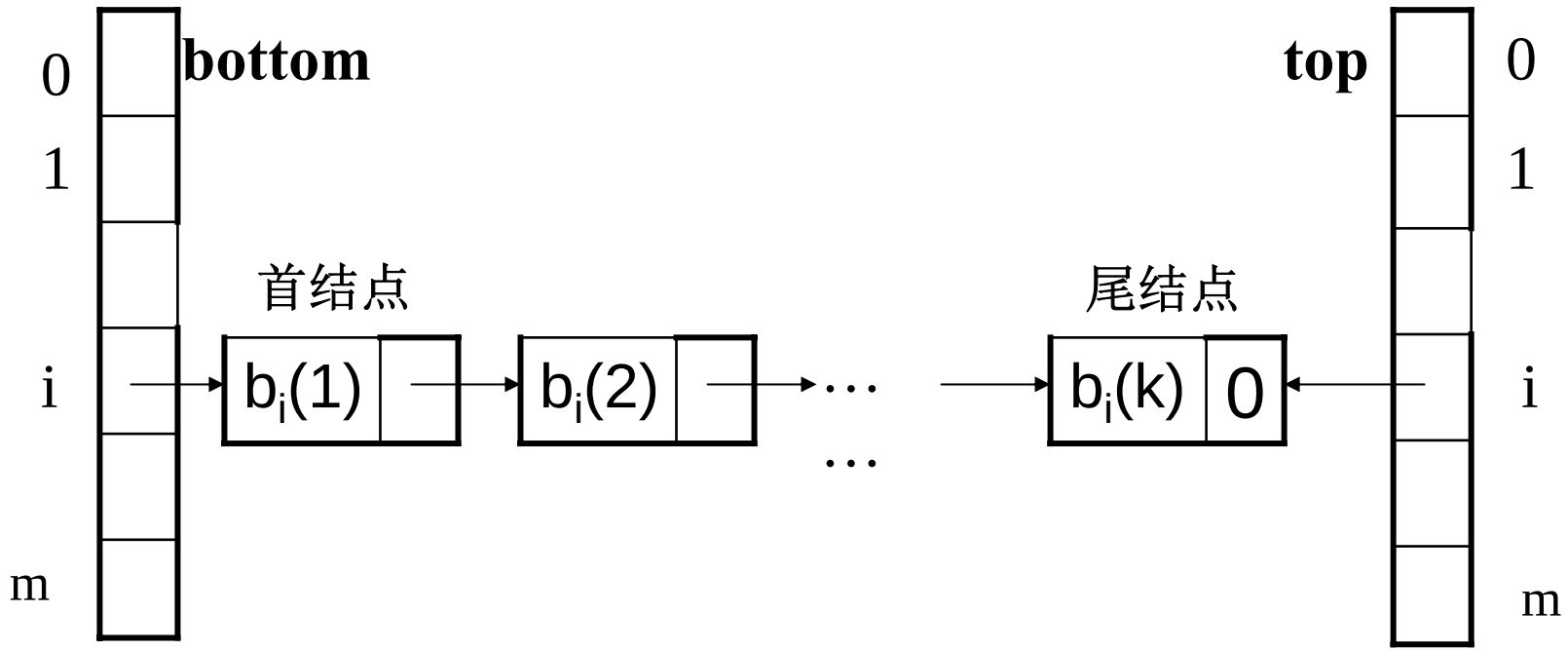


5.4 特殊有序集的线性时间排序算法

5.4.2 桶排序算法

(2)实现算法的准备—数据结构的选择(续)

- 为全集有序元素键值(在1与m之间)序列设置的桶序列：由两个指针数组**bottom**和**top**来表达。**Bottom[i]**和**top[i]**分别指向第i桶中元素队列首结点和尾结点。



5.4 特殊有序集的线性时间排序算法

5.4.2 桶排序算法

(3)算法的复杂度分析：

桶排序算法与计数排序算法大致相同，它们都需要O(m+n)计算时间。初始化空桶需要O(m)时间。将所有待排序元素对号装入桶中共需O(n)时间。将桶中元素依序连接共需O(m)时间。于是，整个桶排序算法共要O(m+n)时间。与计数排序算法类似，如果m=O(n)，则桶排序算法只需要O(n)计算时间。

桶排序算法也需要O(m)的空间。

5.4.3 基数排序

- 多关键字排序
  - 定义：

例 对52张扑克牌按以下次序排序：

♣2<♣3<……<♣A<♦2<♦3<……<♦A<
♥2<♥3<……<♥A<♠2<♠3<……<♠A

两个关键字：花色（♣<♦<♥<♠）
面值（2<3<……<A）
并且“花色”地位高于“面值”

5.4.3 基数排序

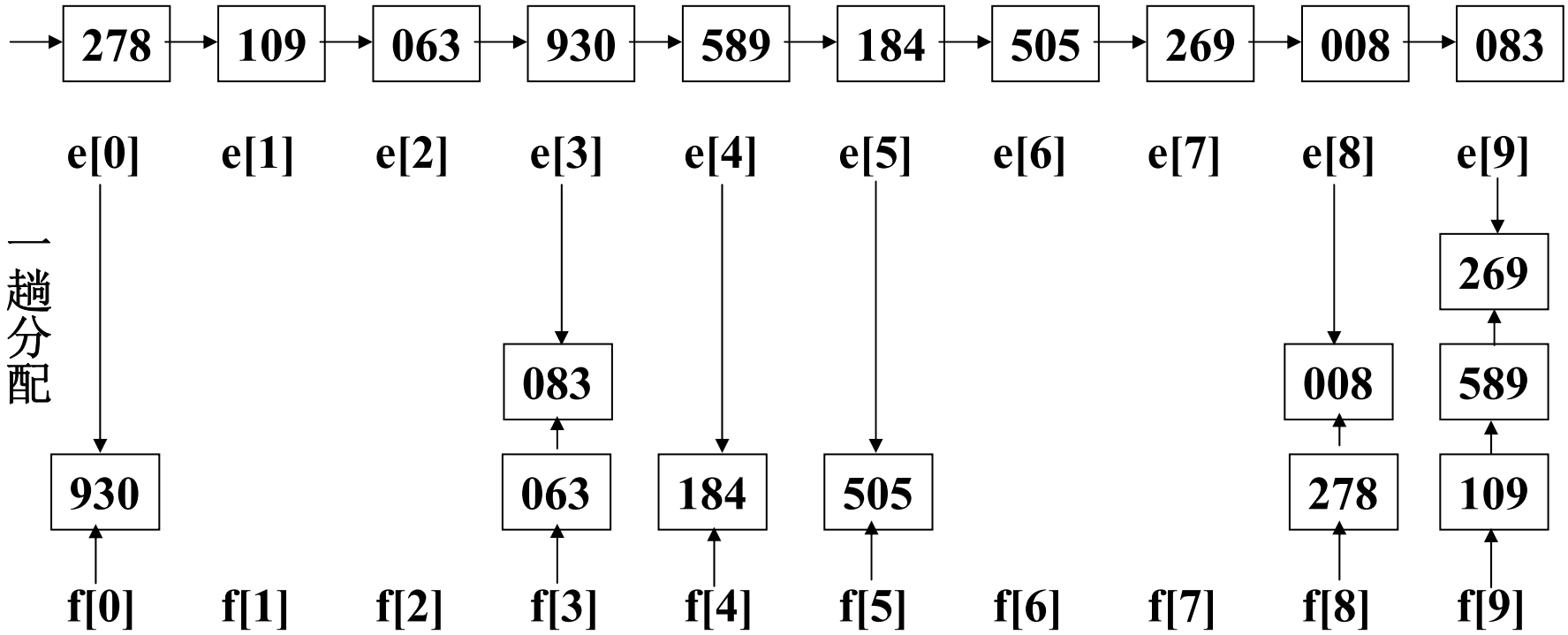
- 多关键字排序方法
  - 最高位优先法（**MSD)**：先对最高位关键字**k1**（如花色）排序，将序列分成若干子序列，每个子序列有相同的**k1**值；然后让每个子序列对次关键字**k2**（如面值）排序，又分成若干更小的子序列；依次重复，直至就每个子序列对最低位关键字**kd**排序；最后将所有子序列依次连接在一起成为一个有序序列
  - 最低位优先法**(LSD)**：从最低位关键字**kd**起进行排序，然后再对高一位的关键字排序，……依次重复，直至对最高位关键字**k1**排序后，便成为一个有序序列

- **MSD与LSD不同特点**
  - 按**MSD**排序，必须将序列逐层分割成若干子序列，然后对各子序列分别排序
  - 按**LSD**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；并且可不通过关键字比较，而通过若干次分配与收集实现排序

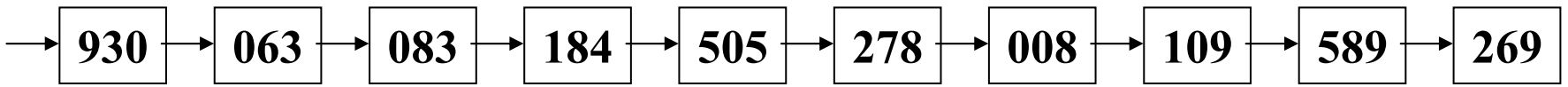
- 链式基数排序
  - 基数排序：借助“分配”和“收集”对单逻辑关键字进行排序的一种方法
  - 链式基数排序：用链表作存储结构的基数排序

- 链式基数排序步骤
  - 设置**10**个队列，**f[i]**和**e[i]**分别为第**i**个队列的头指针和尾指针；
  - 第一趟分配对最低位关键字（个位）进行，改变记录的指针值，将链表中记录分配至**10**个链队列中，每个队列记录的关键字的个位相同；
  - 第一趟收集是改变所有非空队列的队尾记录的指针域，令其指向下一个非空队列的队头记录，重新将**10**个队列链成一个链表；
  - 重复上述两步，进行第二趟、第三趟分配和收集，分别对十位、百位进行，最后得到一个有序序列。

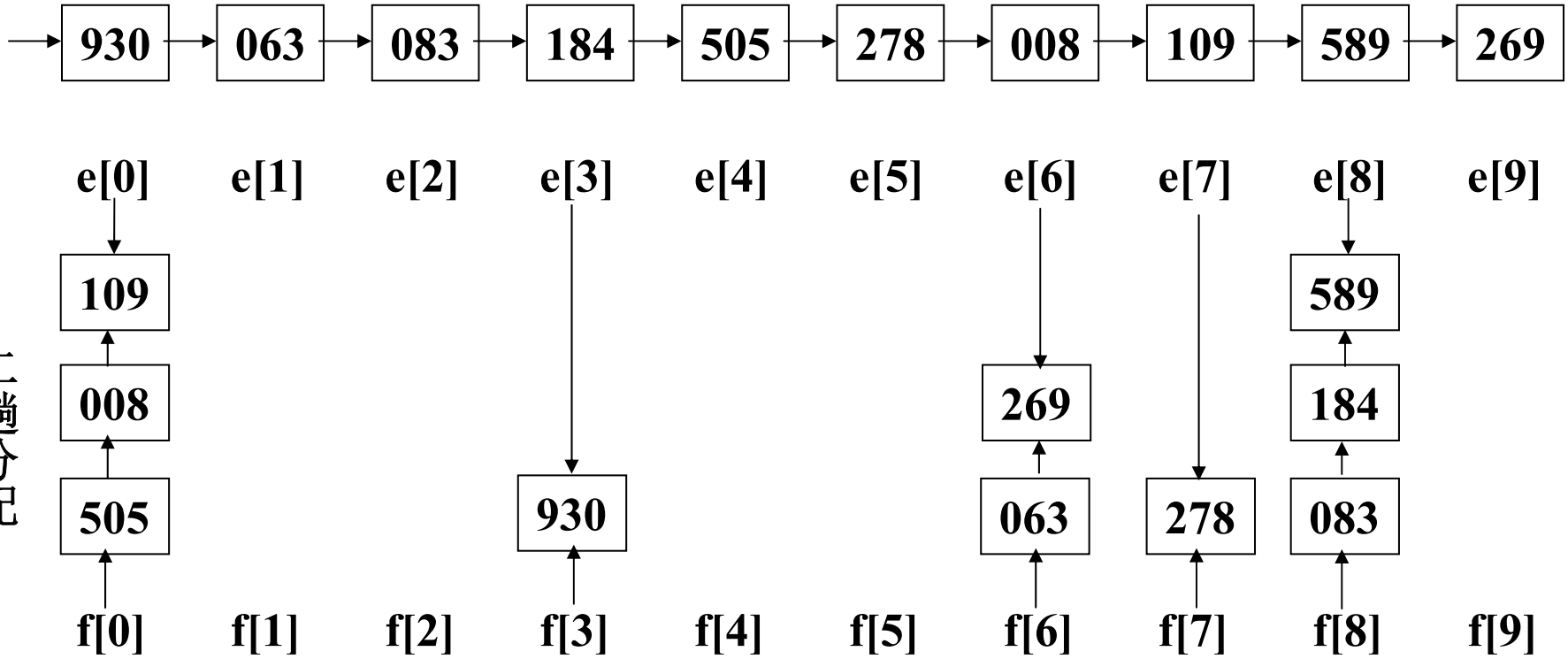
例　初始状态：


278
109
063
930
589
184
505
269
008
083
e[0]
e[1]
e[2]
e[3]
e[4]
e[5]
e[6]
e[7]
e[8]
e[9]
一趟分配
930
083
063
184
505
269
008
589
278
109
f[0]
f[1]
f[2]
f[3]
f[4]
f[5]
f[6]
f[7]
f[8]
f[9]

一趟收集：

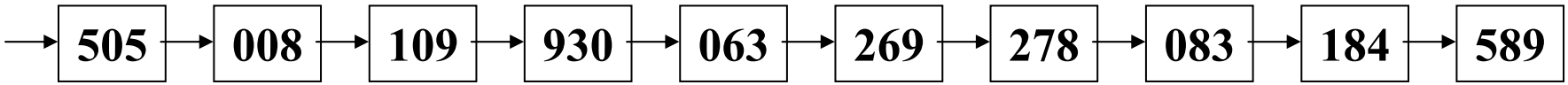

930
063
083
184
505
278
008
109
589
269

一趟收集：

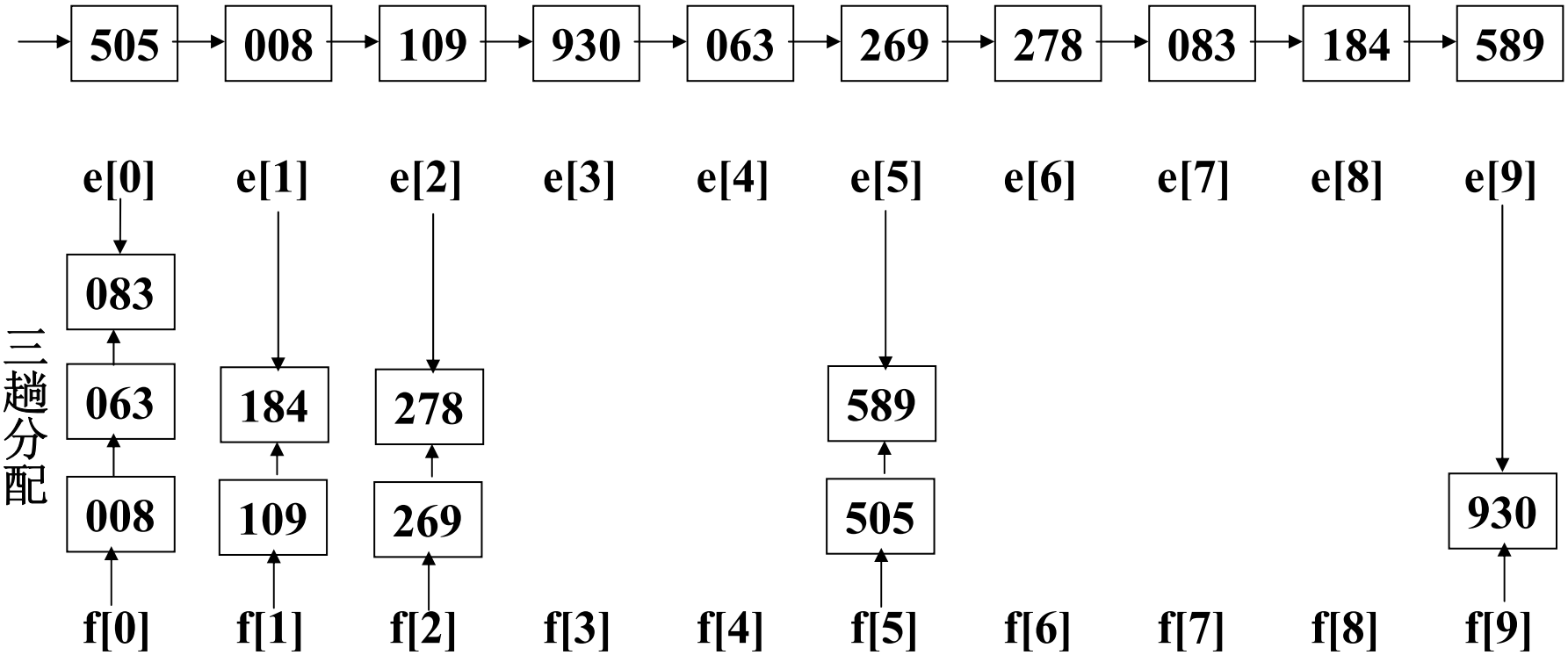


二趟分配

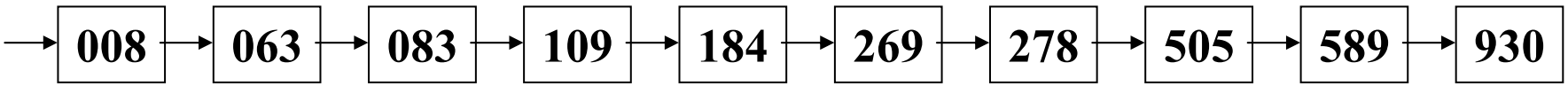
二趟收集：



二趟收集：


505
008
109
930
063
269
278
083
184
589
e[0]
e[1]
e[2]
e[3]
e[4]
e[5]
e[6]
e[7]
e[8]
e[9]
三趟分配
083
063
008
184
109
278
269
589
505
930
f[0]
f[1]
f[2]
f[3]
f[4]
f[5]
f[6]
f[7]
f[8]
f[9]

三趟收集：


008
063
083
109
184
269
278
505
589
930

- 算法描述

```
#define D 3
typedef struct node
{ int key;
  float info;
  int link;
}JD;

int radixsort(JD r[], int n)
{ int i,j,k,t,p,rd,rg,f[10],e[10];
  for(i=1;i<n;i++)  r[i].link=i+1;
  r[n].link=0;
  p=1;
  rd=1;
  rg=10;
  for(i=1;i<=D;i++)
 {  for(j=0;j<10;j++)
    {  f[j]=0;
       e[j]=0;
    }
```

- 算法描述（续）

```
    do{
        k=(r[p].key%rg)/rd;
        if(f[k]==0)
            f[k]=p;
        else
            r[e[k]].link=p;
        e[k]=p;
        p=r[p].link;
      }while(p>0);
    j=0;
    while(f[j]==0) j++;
    p=f[j];
    t=e[j];
    for(k=j+1;k<10;k++)
       if(f[k]>0)
       {  r[t].link=f[k];
          t=e[k];
       }
    r[t].link=0;
    rg*=10;
    rd*=10;
 }
  return(p);
}
```

- 算法分析
  - 时间复杂度：
    - 分配：**T(n)=O(n)**
    - 收集：**T(n)=O(rd)**

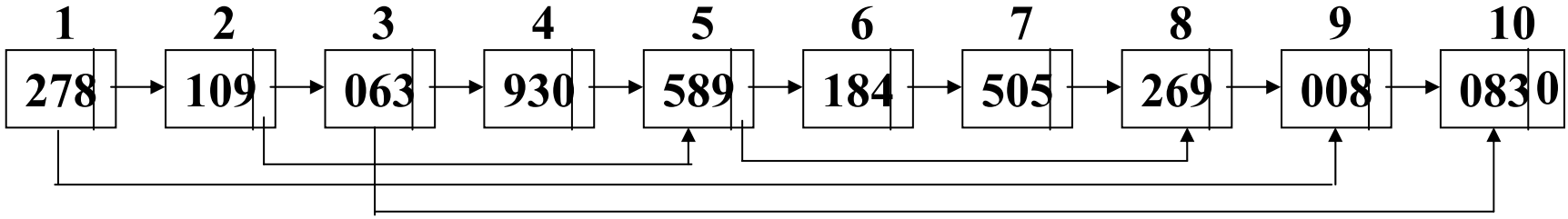
      T(n)=O(d(n+rd))

      其中：　**n**——记录数

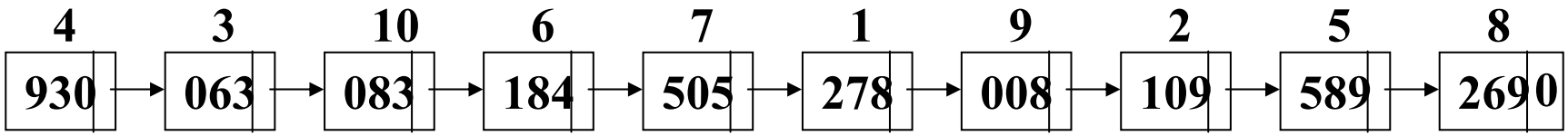
      d——关键字数

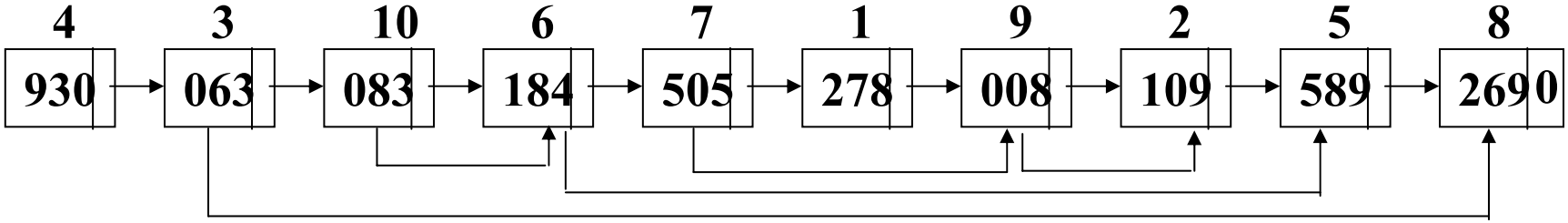
      rd——关键字取值范围
  - 空间复杂度：**S(n)=2rd**个队列指针**+n**个指针域空间

返回章目录



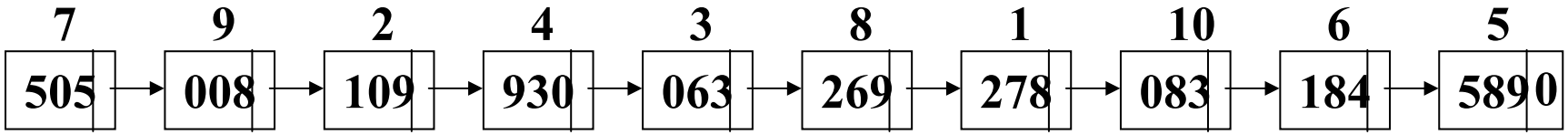
	f[0]= 4	e[0]= 4
	f[1]=0	e[1]=0
	f[2]=0	e[2]=0
	f[3]= 3	e[3]= 10
第一趟收集：	f[4]= 6	e[4]= 6
	f[5]= 7	e[5]= 7
	f[6]=0	e[6]=0
	f[7]=0	e[7]=0
	f[8]= 1	e[8]= 9
	f[9]= 2	e[9]= 8

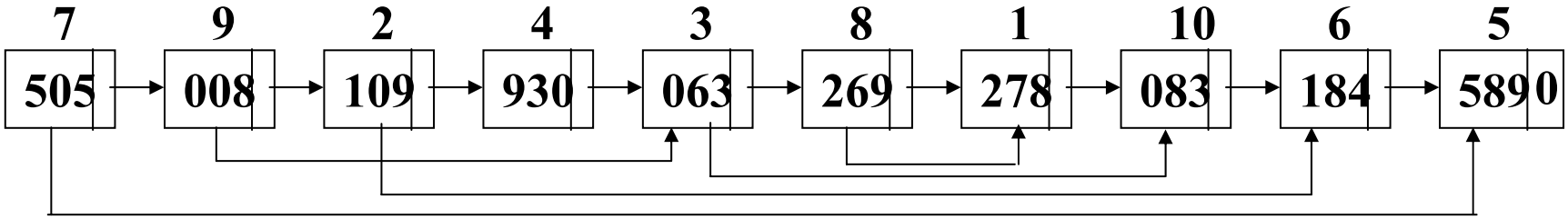




第二趟收集：

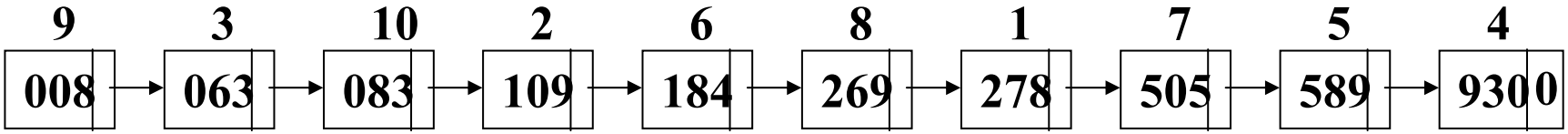
f[0]= 7	e[0]= 2
f[1]=0	e[1]=0
f[2]=0	e[2]=0
f[3]= 4	e[3]= 4
f[4]=0	e[4]=0
f[5]=0	e[5]=0
f[6]= 3	e[6]= 8
f[7]= 1	e[7]= 1
f[8]= 10	e[8]= 5
f[9]=0	e[9]=0





第三趟收集：

f[0]= 9	e[0]= 10
f[1]= 2	e[1]= 6
f[2]= 8	e[2]= 1
f[3]=0	e[3]=0
f[4]=0	e[4]=0
f[5]= 7	e[5]= 5
f[6]=0	e[6]=0
f[7]=0	e[7]=0
f[8]=0	e[8]=0
f[9]= 4	e[9]= 4



5.5 中位数与第k小元素

5.5.0 引言

(1) 概念与术语

- 中位数
- 第k小元素

(2) 问题的提法：

给定线性序集中n个元素和一个整数k，1≤k≤n,要求不经排序而找出这n个元素中第k小(大)的元素。当k=1时，就是要找最小(大)元素;当k=n时，就是要找最大(小)元素;当k=(n+1)/2时，就是要找中位元素。

(3)在某些特殊情况下，容易设计解选择问题的线性时间算法。例如，找最大或最小元素显然可以在O(n)，如果k≤n/logn，或者k≥n-n/logn，通过堆排序算法可以在O(n+klogn)=O(n)时间内找出第k个元素。

建堆时间　　排序时间

5.5 中位数与第k小元素

5.5.1 平均情况下的线性时间选择算法

(1)基本思想——不平衡、预处理的二分

分治与二分查找类似。但二分查找的预处理只做一次，而且二分是平衡二分。

这里借助随机划分**RandomizedPartition**做预处理，然后根据k值决定在分出的两段中哪一段递归地查找。

5.5 中位数与第k小元素

5.5.1 平均情况下的线性时间选择算法

(2) 算法的实现

```
Item randomselect(Item a[],int l,int r,int k)
{
  int i,j,p;
  if(r<=l)  return a[r];
  i=randompartition(a,l,r);
  j=i-l+1;
  if(j==k) return a[i];
  if(j>k)   return randomselect(a,l,i-1,k);
  else      return randomselect(a,i+1,r,k-j);
}
```

5.5 中位数与第k小元素

5.5.1 平均情况下的线性时间选择算法

(3)算法的复杂度分析

容易看出,在最坏情况下,算法RandomizedSelect需要$O(n^2)$的计算时间。例如在找最大元素时，总是在最小元素处划分。但该算法的平均性能很好。

由于随机划分函数**RandomizedPartition**使用了一个随机数产生器**Random**，它能随机地产生p和r之间的一个随机整数，因此，**RandomizedPartition**产生的划分基准是随机的。

在这个条件下，可以证明：算法**RandomizedSelect**可以在平均时间O(n)内找出n个输入元素中的第k小元素。

5.5 中位数与第k小元素

5.5.2 最坏情况下的线性时间选择算法

（1）**Select**算法的基本思想

在线性时间内找一个划分基准，使得按这个基准将输入的数组划分成的两个子数组的长度，即使在最坏的情况下，也不会严重失衡，以便采用类似于**RandomizedSelect**的分治策略，在最坏情况下用**O(n)**的时间完成选择任务。

5.5 中位数与第k小元素

5.5.2 最坏情况下的线性时间选择算法

(2) **Select**算法找划分基准的一种构思

①不计n个输入元素的最后(n mod 5)个，分成⌊n/5⌋个组，每组5个元素。用任意一种排序算法，将每组中的元素排好序。然后，取出每组的第3小元素，共⌊n/5⌋个。

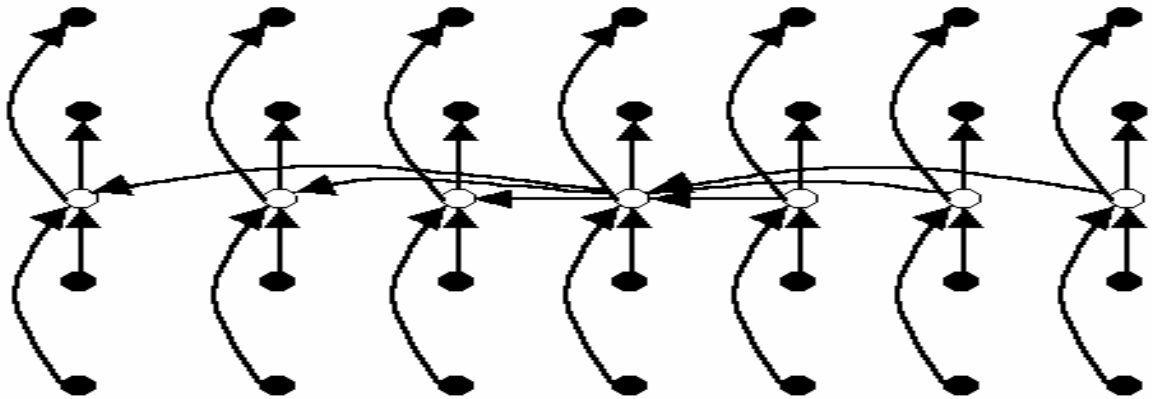
②递归调用**Select**来找出这⌊**n/5**⌋个元素的中位元素，即第⌊**(n+5)/10**⌋小的元素。然后以这个元素作为所输入的**n**个元素的数组的划分基准元素，对其进行划分。

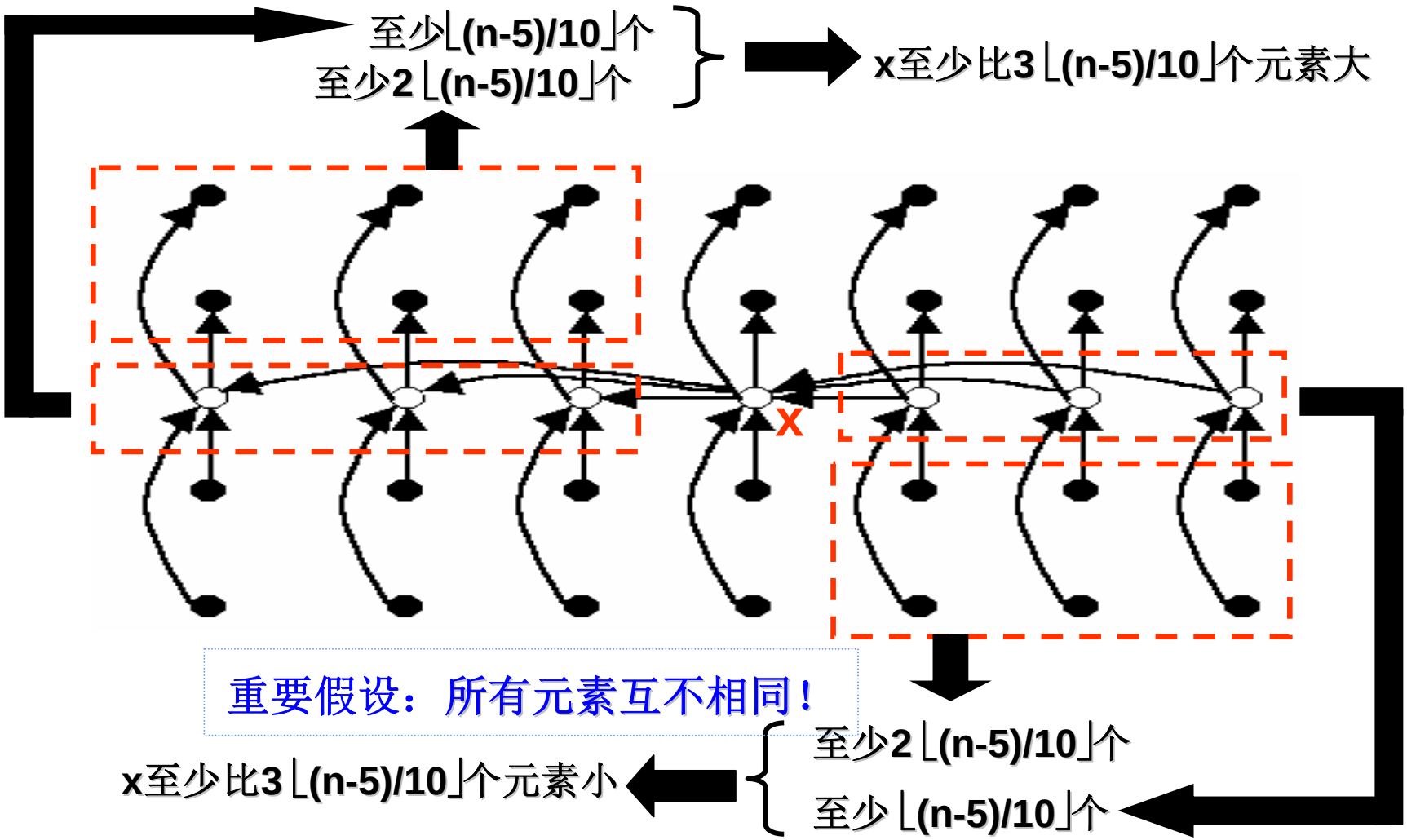
5.5 中位数与第k小元素

5.5.2 最坏情况下的线性时间选择算法

(3) **Select**算法找划分基准的图解及分析

按上述构想，所选出的基准**x**如下图所示。其中n个元素用实心小圆点表示，每组元素的中位元素用空心小圆点表示。为了图的清晰，已将空心小圆点带着各组排好序，并指示出它们的中位元素x。箭头指示较大元素指向较小元素。





当n≥75时，3⌊(n-5)/10⌋ ≥ n/4 =〉划分后的两个子数组长度都至少缩短1/4
这一点是该算法只需O(n)时间的关键！

5.5 中位数与第k小元素

5.5.2 最坏情况下的线性时间选择算法

(4) 在所有元素互不相同的情况下， Select算法的描述

```
T Select(T a[ ],int p,int r,int k) //要求p≤r,1≤k≤r-p+1
{if (r-p<75) {用某个简单排序算法对数组a[p:r]排序;
              return a[p+k-1]; }
 for (int i = 0;i<=(r-p-4)/5; i++ )
 //找a[p+5*i]至a[p+5*i+4]的第3小元素，并与a[p+i]交换位置;
 //下面找所有第3小元素的中位元素即第(n+5)/10小元素。
 T x = Select(a, p, p+(r-p-4)/5, (r-p-4)/10);
 int i=Partition(a, p, r, x ), j=i-p+1;
 if (k<=j)  return  Select(a,p,i,k);
 else       return  Select(a,i+1,r,k-j);
}
```

5.5 中位数与第k小元素

5.5.2 最坏情况下的线性时间选择算法

(5) **Select**算法的复杂度分析

设n为输入数组的长度。又设对n个元素的数组调用**Select**需要T(n)时间。

①算法的递归调用只在n>=75时才执行。因此，当n＜75时算法**Select**所用的计算时间不超过一个常数。

②算法**Select**的for循环体共执行n/5次，每一次只需O(1)时间。因此，执行for循环共需时间O(n)。

③找n/5个第**3**小元素的中位数x需要T(n/5)的时间。

④找到x后调用**Partition (a，l，r， x)**只需要**O(n)**的时间。

5.5 中位数与第k小元素

5.5.2 最坏情况下的线性时间选择算法

(5) **Select**算法的复杂度分析(续)

⑤已经证明，按照算法所选的基准x进行划分所得到的两个子数组分别至多有3n/4个元素。所以无论对哪一个子数组递归调用**Select**都只需要T(3n/4)的时间。于是，可以得到关于T(n)的递归方程：

$$T(n)\le\begin{cases}C_1 & n<75\\ C_2n+T(n/5)+T(3n/4) & n\ge 75\end{cases}$$

解此递归方程可得T(n)=0(n)。

5.5 中位数与第k小元素

5.5.2 最坏情况下的线性时间选择算法

(6)补充关于T(n)的递归方程的求解

右边两个参变量的系数**1/5**与**3/4** 之和**19/20**小于**1**,是算法只需O(n)时间的关键。

用代入法。首先猜测T(n)≤c*n, 其中c为某个常数, 然后用数学归纳法证明之。

- 如果取c≥c_1, 则当n≤74时, T(n)≤c*n。
- 现归纳假设: 当74≤k<n时T(k)≤c*k成立。
- 然后证明: 当k=n时T(k)≤c*k仍成立。
- 事实上，由所得到的递归关系式和归纳假设，有
- $T(n)\leqslant c_2*n+c*n/5+3*c*n/4\leqslant c_2*n+19/20*c*n$
- 只要取c=max(c1, 20*c2)，便可得:
- $T(n)\leqslant 1/20*c*n+19/20*c*n=c*n$
- 即当k=n时T(k)≤c*k仍成立。证毕。

5.5 中位数与第k小元素

5.5.2 最坏情况下的线性时间选择算法

(7)处理退化的情形即输入的元素中相同的情形

这时**Select**应删去划分之后的3条语句，并替之以：

①让所有与基准x相等的元素整成连续段，然后更新i为该段左端的下标；

②计算**j=i-l+1**；

③设与基准x相等的元素的个数为m(≥1)。那么，当j≤k≤j+m-1时，不必再递归调用**Select**，只要返回a[i]即可；

④最后是条件语句：当1≤k＜j时，返回**Select(a, l, i-1，k)**; 否则返回**Select(a, i+m，r，k-j-m+1)**。

5.6 输油管道问题

- 某石油公司计划建造一条由东向西的主输油管道。
- 该管道要穿过一个有*n*口油井的油田。
- 从每口油井都要有一条输油管道沿南北向最短路经与主管道相连。
- 如果给定*n*口油井的地理位置，即它们的x坐标（东西向）和y坐标（南北向），应如何确定主管道的最优位置，使得各油井到主管道之间的输油管道的长度总和最小？
- 设计一个线性时间算法计算各油井到主管道之间的输油管道最小长度总和。

THE END