数据结构与算法

Data Structures and Algorithms

第六部分 排序

数据结构考查内容

一、线性表
（一）线性表的基本概念
（二）线性表的实现
（三）线性表的应用

二、栈、队列和数组
（一）栈和队列的基本概念
（二）栈和队列的顺序存储结构
（三）栈和队列的链式存储结构
（四）多维数组的存储
（五）特殊矩阵的压缩存储
（六）栈、队列和数组的应用

三、树与二叉树
（一）树的基本概念
（二）二叉树
（三）树、森林
（四）树与二叉树的应用

四、图
（一）图的基本概念
（二）图的存储及基本操作
（三）图的遍历
（四）图的基本应用

数据结构

五、查找
（一）查找的基本概念
（二）顺序查找法
（三）分块查找法
（四）折半查找法
（五）B树及其基本操作，B+树的基本概念
（六）散列（Hash）表
（七）字符串模式匹配
（八）查找算法分析及应用

六、排序
（一）排序的基本概念
（二）插入排序
（三）起泡排序
（四）简单选择排序
（五）希尔排序
（六）快速排序
（七）堆排序
（八）二路归并排序
（九）基数排序
（十）外部排序

算 法

教学要求

- 掌握排序的基本概念和常用术语；
- 熟练掌握插入排序、快速排序、选择排序、堆排序、归并排序和基数排序的基本思想、算法原理、和算法实现；
- 掌握各种排序算法的性能及其分析方法，以及各种排序方法的比较和选择等。

主要内容

【分类(Sorting)】也叫排序(Ordering)，是将一组数据按照规定顺序进行排列，其目的是为了方便查询和处理。

对于给定数组A，经过分类处理之后，满足关系：

$$A[1].key \le A[2].key \le \cdots\cdots \le A[n].key$$

如果在分类之前存在关系

$$A[i].key = A[j].key \quad (1 \le i < j \le n)$$

经分类后，A[i]和A[j]分别被移至$A[i_1]$和$A[j_1]$，并且i_1和j_1满足关系

$$1 \le i_1 < j_1 \le n$$

我们称这种分类是稳定的，否则称其为不稳定的分类。

分类的种类：

- 按分类时分类对象存放的设备，分成内部分类(Internal Sorting)和外部分类(External Sorting)。
- 分类过程中数据对象全部在内存中的分类，叫内部分类。
- 分类过程数据对象并非完全在内存中的分类，叫外部分类。

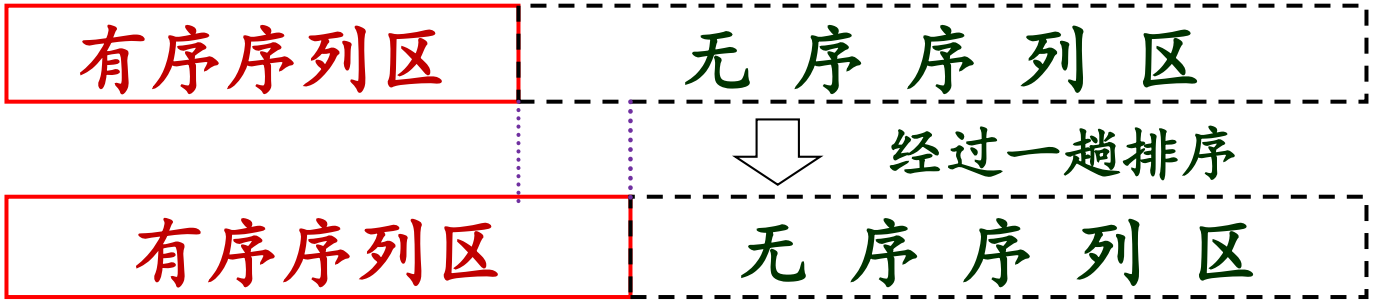
分类表的存储结构：

```
struct  records {
        keytype   key ;
        fields   other ;
};
typedef  records  LIST[maxsize] ;
```

影响分类性能的因素：

- 比较关键字的次数—当关键字是字符串时，是主要因素。
- 交换记录位置和移动记录的次数—当记录很大时，是主要因素。

内部排序的过程是一个逐步扩大记录的有序序列长度的过程。



基于“扩大” 有序序列长度的不同方法，内部排序大致可分为：

插入类：将无序子序列中的一个或几个记录“插入”到有序序列中，从而增加记录的有序子序列的长度。

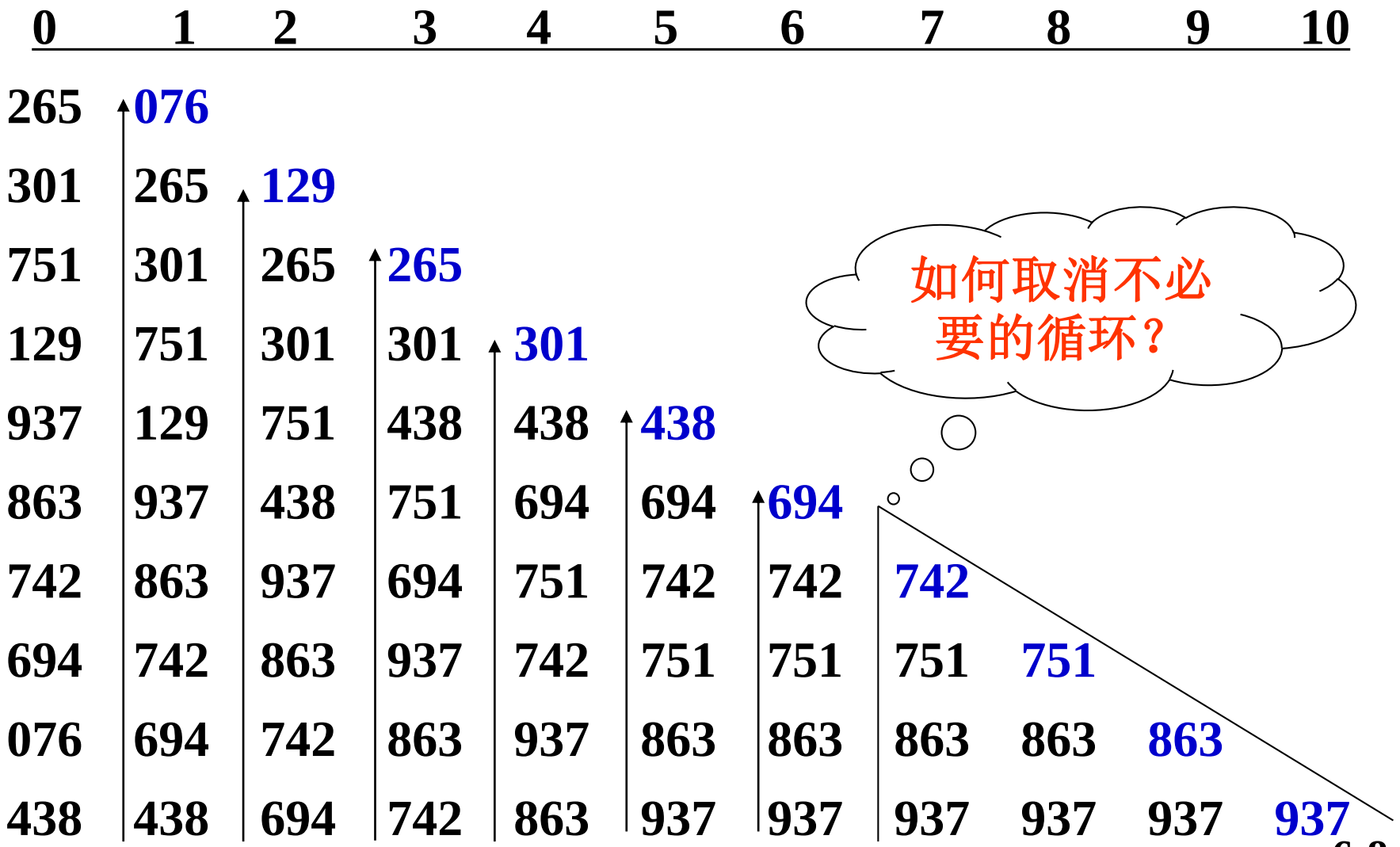
交换类：通过“交换”无序序列中的记录得到其中最小或最大的记录，并将它加入到有序子序列中，以此增加有序子序列的长度。

选择类：从无序子序列中“选择”关键字最小或最大的记录，并将它加入到有序子序列中，以此增加有序子序列的长度。

归并类：通过“归并”两个或两个以上的记录有序子序列，逐步增加记录有序序列的长度。

6.1　简单的分类算法

6.1.1 冒泡分类



冒泡分类算法：

```
Void  BubbleSort ( int  n , LIST  &A )
{   int  x, y ;
    for ( i =1; i <= n-1; i++ )
        for ( j =n; j >= i+1; j-- )
            if ( A[j].key < A[j-1].key )
                swap ( A[j], A[j-1] )
}
```

```
Void  swap(x, y)
records &x, records  &y
{    records   w ;
      w = x ;
      x = y ;
      y = w ;
}
```

算法分析：

$$f(n)=C_3n+\sum_{i=1}^{n-1}C_2.(n-i)$$

$$=1/2\cdot C_2n^2+(C_3-1/2\cdot C_2)\cdot n$$

$$\leq(C_2/2+C_3)n^2 \quad 当n^2\geq 1时$$

$$=Cn^2$$

空间复杂度：常数个辅助单元，O(1)

时间复杂度：最好情况，直接有序；

最坏情况，比较 n(n-1)/2；

移动次数，每次比较都有3次交换：

3 * n(n-1)/2

稳定性：i>j时, A[i]>A[j], 稳定的排序方法。

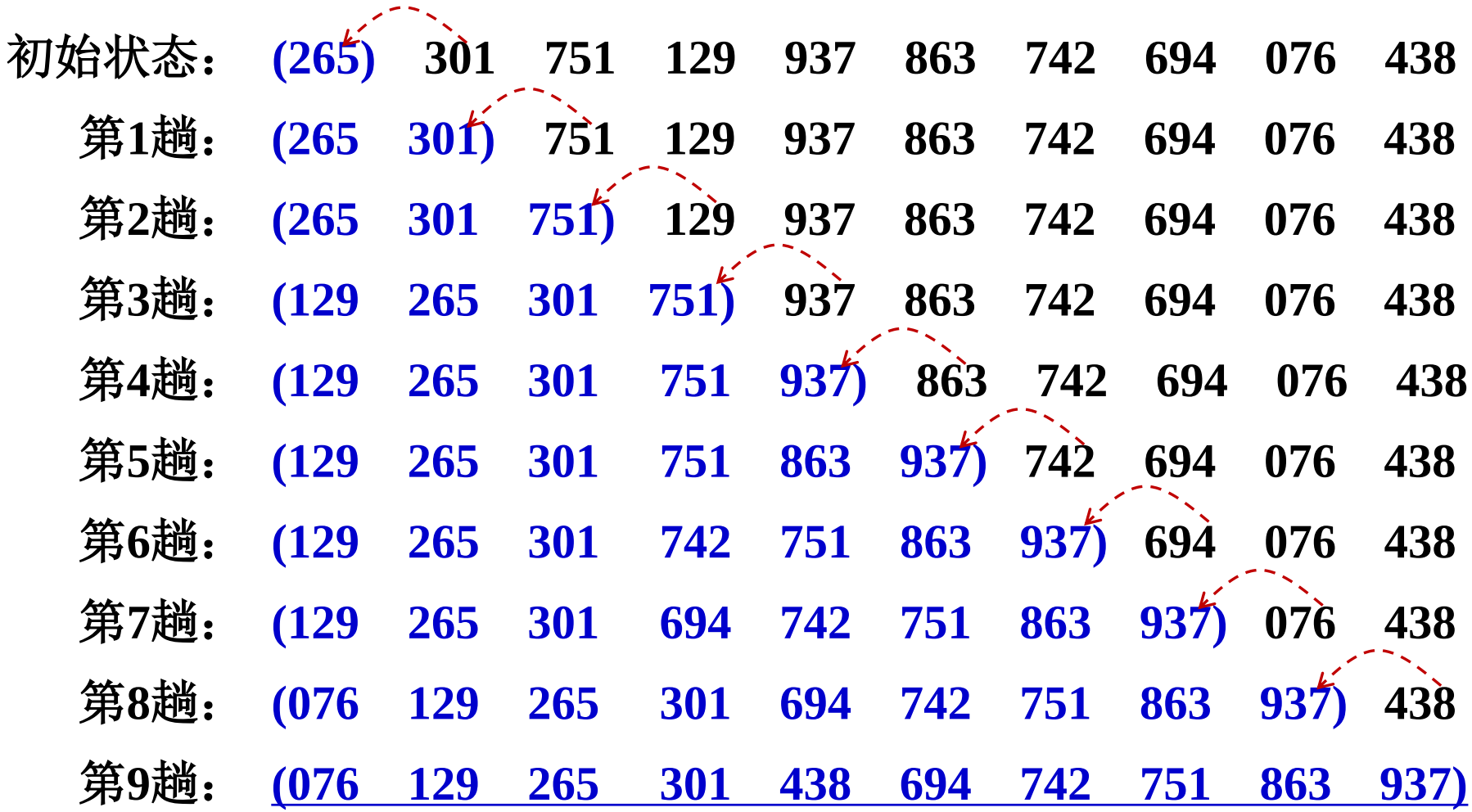
$T(n) = O(f(n)) = O(C\cdot n^2) = O(n^2)$

【例6-1】双向气泡排序算法，在排序过程中交替改变扫描方向。

```
Void DoubleBubbleSort( LIST &A, int  n)
{  int i=1,flag=1;
   while( flag )
    {   flag = 0;
        for( j=n-i+1; j>=i+1; j--)     //较小元素A[j]
           if( A[j].key < A[j-1].key ) {   flag=1;
                                           swap(A[j],A[j-1]);   }
        for( j=i+1; j<=n-i+1; j++)   //较大元素A[n-i+1]
           if( A[j].key > A[j+1].key ) {   flag=1;
                                           swap(A[j],A[j+1]);  }
        i++;
    }
}
```

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

6.1.2 插入分类　　---直接插入排序



插入分类算法：

```
void  InsertSort ( n, A )
int  n; LIST  A ;
{    int  i, j ;
     A[0].key = -∞ ;
     for(i=1; i<=n; i++)
     { j=i;
       while(A[j].key<A[j-1].key)
       {  swap(A[j],A[j-1]) ;
          j=j-1;
       }
     }
}
```

```
j=i;   temp=A[j];
while (temp.key<A[j-1].key)
  {  A[j]=A[j-1];
     j=j-1;          }
A[j]=temp;
```

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

有序序列L[1,2,…,i-1]	L[i]	无序序列L[i+1, …, N]

空间复杂度：只使用了常数个辅助单元，复杂度为O(1)。

时间复杂度：总的插入次数为n-1趟；每趟都要比较和移动元素。

比较和移动取决于有序序列的状态：

最好的情况(有序)是只需比较依次，且不需要移动元素，O(n)；

最坏的情况(逆序)是比较n(n+1)/2；移动n(2+(n+1))/2。

平均情况下总的比较和移动次数大于为 $n^2/4$.

直接插入排序算法的时间复杂度是$O(n^2)$。

稳定性：每次都是从后向前比较并移动元素，不会出现相同数据元素相对位置的变化问题，所以是一个稳定的排序方法。

折半插入排序思想

(1)在直接插入排序中，L[1..i-1] 是一个按关键字有序的有序序列;

(2)可以利用折半查找实现“在L[1..i-1]中查找L[i]的插入位置”;

(3)称这种排序为折半插入排序。

```
void BiInsertionSort ( SqList &L )
{
   for ( i=2; i<=L.length; ++i ) {
      L[0] = L[i];                    // 将 L[i] 暂存到 L[0]
      在 L[1..i-1]中折半查找插入位置;
      for ( j=i-1;  j>=high+1;  --j ){ //插入位置为high+1
         L[j+1] = L[j];
      } //for 记录后移
      L[high+1] = L[0];  // 插入
}// for
}                                     // BInsertSort
```

```
low = 1;   high = i-1;
while (low<=high) {
   m = (low+high)/2;
   if (L[0].key < L[m].key)
      high = m-1;
   else  low = m+1;
}
```

6.1.3 选择分类　　---简单选择排序

初始状态：	265	301	751	129	937	863	742	694	076	438
第1趟：	076	301	751	129	937	863	742	694	265	438
第2趟：	076	129	751	301	937	863	742	694	265	438
第3趟：	076	129	265	301	937	863	742	694	751	438
第4趟：	076	129	265	301	937	863	742	694	751	438
第5趟：	076	129	265	301	438	863	742	694	751	937
第6趟：	076	129	265	301	438	694	742	863	751	937
第7趟：	076	129	265	301	438	694	742	863	751	937
第8趟：	076	129	265	301	438	694	742	751	863	937
第9趟：	076	129	265	301	438	694	742	751	863	937

简单选择分类算法

```
void  SelectSort ( n, A )
int  n; LIST  A ;
{    int  i, j，  lowindex ;
     for(i=1; i<n; i++)
      {  lowindex = i ;
        for ( j = i+1; j<=n; j++){
           if ( A[j].key < A[lowindex].key )
              lowindex = j ;
        }// for
        if (lowindex != i )
           swap(A[i], A[lowindex]) ;
       }// for
}
```

```
for(i=1; i<=n-1; i++)      对比冒泡
    for(j=n; j>=i+1; j--)
        if(A[j].key<A[j-1].key)
            swap(A[j], A[j-1]) ;
```

空间复杂度：O(1)

时间复杂度：

最坏情况，比较 n(n-1)/2；

移动次数，最多 3 * (n-1)；

稳定性：不稳定的方法。

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

【例6-2】设计算法，实现奇偶转换排序。

基本思想：

第一趟对所有奇数的i，将A[i]和A[i+1]进行比较；

第二趟对所有偶数的i，将A[i]和A[i+1]进行比较；

```
if(A[i]>A[i+1])   swap(A[i],A[i+1]);
```

重复上述二趟交换过程交换进行，直至整个数组有序。

那么，

(1)结束条件是什么？

(2)实现算法。

(1)结束条件为不产生交换

(2)奇偶转换排序算法

```
Void  OESort(LIST &A , int n)
{   int  i , flag ;
    do{  flag = 0 ;
         for( i=0; i<n ; i+=2)
              if( A[i] > A[i+1] )  {  flag = 1;
                                      swap(A[i],A[i+1]); }
         for( i=1; i<n ; i+=2)
              if( A[i] > A[i+1] )  {  flag = 1;
                                      swap(A[i],A[i+1]); }
     }while(flag);
}
```

时间复杂度：$O(n^2)$

6.2 Shell 分类

希尔（Shell）排序又称缩小增量法，基本思想为：

先取定一个整数$d_1<n$，把全部关键字分成d_1个组，所有距离为d_1倍数的记录放在一组中，形成$L[i, i+d_1, i+2*d_1, \cdots, i+kd_1]$的特殊子表，并在各组内进行直接插入排序；然后取$d_2<d_1$，重复上述分组和排序工作，直至取$d_i=1$，即所有记录放在一个组中排序为止。

特点：
每次以不同的增量分组，组内进行直接插入排序，在最后一次作组内直接插入排序时，所有记录“几乎”有序了。

“逆序”结点作跳跃移动，提高了排序速度。

【例6-3】希尔排序过程

初始状态： 265 301 751 129 937 863 742 694 076 438

第1趟分组 $d_1=5$:

排序结果： 265 301 694 076 438 863 742 751 129 937

第2趟分组 $d_2=d_1/2=2$:

排序结果： 129 076 265 301 438 751 694 863 742 937

第3趟分组 $d_3=d_2/2=1$:

排序结果： 076 129 265 301 438 694 742 751 863 937

希尔排序算法：

```
Void  Shellsort ( LIST  &A , int n )
{      for( k = n/2;  k >= 1; k /= 2 )
       {  for( i=k+1; i<=n; i++ )
              {  x = A[i]; j = i-k;
                 while( (j>0) && (x.key<A[j].key) )
                    {   A[j+k] = A[j];
                        j -= k;                }
                 A[j+k] = x;
              }        //组内排序，将x直接插入到组内合适的位置
       }
}
```

■ 算法的性能分析

① 步长由大到小：希尔排序开始时增量（步长）较大，每个子序列中的记录个数较少，从而排序速度较快；当增量（步长）较小时，虽然每个子序列中记录个数较多，但整个序列已基本有序，排序速度也较快。

② 步长的选择是希尔排序的重要部分。只要最终步长为1，任何步长序列都可以工作（当步长为1时，算法变为直接插入排序，这就保证了数据一定会被排序）。

③ 希尔排序算法的时间性能是所取增量（步长）的函数，而到目前为止尚未有人求得一种最好的增量序列。已知的最好步长序列是由Sedgewick提出的(1，5，19，41，109,...)

④ 希尔排序的时间性能在$O(n^2)$和$O(n\log_2 n)$之间。当n在某个特定范围内，希尔排序所需的比较次数和记录的移动次数约为$O(n^{1.3})$到$O(n^{1.5})$。

Shell 增量序列

Shell 增量序列的递推公式为： $h_t = \lfloor \frac{N}{2} \rfloor, h_k = \lfloor \frac{h_{k+1}}{2} \rfloor$

Hibbard 增量序列

Hibbard 增量序列的通项公式为： $h_i = 2^i - 1$ $h_1 = 1, h_i = 2 * h_{i-1} + 1$

Knuth 增量序列

Knuth 增量序列的通项公式为： $h_i = \frac{1}{2}(3^i - 1)$ $h_1 = 1, h_i = 3 * h_{i-1} + 1$

Gonnet 增量序列

Gonnet 增量序列的递推公式为： $h_t = \lfloor \frac{N}{2.2} \rfloor, h_k = \lfloor \frac{h_{k+1}}{2.2} \rfloor (若h_2 = 2则h_1 = 1)$

Sedgewick 增量序列

Sedgewick 增量序列的通项公式为： $h_i = \max(9 * 4^j - 9 * 2^j + 1, 4^k - 3 * 2^k + 1)$

6.3 快速分类—划分交换分类

是C.A.R. Hoare 1962年提出的一种划分交换排序。采用的是分治策略(一般与递归技术结合使用)，以减少分类过程之中的比较次数。

- 分解: 将原问题分解为若干个与原问题相似的子问题，又称划分
- 求解: 递归地求解子问题，若子问题的规模足够小，则直接求解
- 组合: 将每个子问题的解组合成原问题的解。

基本思想：

对于： A[1].key, A[2].key, ……, A[n].key

选定中间值v(基准)，使之对某个k有：

A[1].key, A[2].key, ……, A[k-1].key＜A[k].key

而： A[k+1].key, A[k+2].key, ……, A[n].key≥A[k].key

然后分别对A[1],…,A[k-1] 和 A[k+1],…,A[n] 作同样的处理。

算法要点：

（1）中间值：中间值 v 的选择，其位置确定在 A[k]

设函数findpivot(i,j), 求A[i].key,…,A[j].key的中间值v。

i: v = (A[i].key， A[(i+j)/2].key, A[j].key的中值)

ii: v = 从A[i].key 到 A[j].key 最先找到的两个不同关键字中的极大值。

（2）划分：A[i].key,…,A[j].key 分割成

A[i],…,A[k-1] ； A[k],A[k+1],…,A[j] 两部分。

扫描：令游标 L 从左（L=i）向右扫描，越过key小于 v的记录，直到A[L].key ≥v 为止；同时,令游标 R 从右（R=j）开始向左扫描，越过key大于等于 v 的记录,直到A[R].key＜v 的记录A[R]为止；

测试：若L>R（L=R+1），成功划分，L是右边子序列的起始下表；

交换：若L<R, 则swap(A[L], A[R])；

重复上述操作，直至过程进行到L>R（L=R+1）为止。

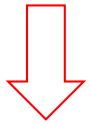
找中间值算法：

```
int   FindPivot( int  i, int  j )
{   keytype   firstkey ;   int  k ;
    firstkey = A[i].key ;
    for ( k=i ;  k<=j; k++ )
        if ( A[k].key > firstkey )
            return k ;
        else  if ( A[k].key <firstkey )
            return  i ;
    return  0 ;
}
```

分割算法：

```
int  Partion ( i, j, pivot )
int i, j;  keytype  pivot ;
{   int   L, R ;
    L = i ;   R = j ;
    do {
            swap ( A[L], A[R] ) ;
            while ( A[L].key < pivot )
                    L = L +1 ;
            while ( A[R].key >= pivot )
                    R = R -1 ;
    } while ( L <= R );
    return L ;
}
```

分割：[i, … …, j]



[i, … k-1, k, k+1, … j]

参考算法-1：划分算法

```
int  Partion ( Sqlist &L, int low, int high )
// 交换顺序表L中子表 [low…high]的记录，枢轴记录到位，
//并返回所在位置，此时在它之前(后)记录均不大(小)于它
{  keytype pivotkey;
   pivotkey = L[low].key;
   while (low<high) {
          while ((low<high)&& (L[high].key>=pivotkey))
                 --high;
          L[low] ←→ L[high];
          while ((low<high)&& (L[low].key<=pivotkey))
                 ++low;
          L[low] ←→ L[high];
   }
   return low;
}
```

v
49

初始关键字　49　49*　65　97　17　27　50

low　　high

pivotkey=L[low].key;
while ((low<high)&& (L[high].key>=pivotkey))
--high;
L[low] ←→ L[high]; --high;

一趟快排　27　49*　65　97　17　49　50

low　　high

while ((low<high)&& (L[low].key<pivotkey))
++low;
L[low] ←→ L[high]; ++low;

第二趟快排　27　17　65　97　49*　49　50

low　　high

参考算法-2：划分算法（教材）

```
int  Partion ( Sqlist &L, int low, int high )
// 返回枢轴所在位置，它之前(后)记录均不大(小)于它
{  KeyType pivotkey;
   pivotkey = L[low].key;  L[0] = L[low];//第一个值为枢轴
   while (low<high) {
      while ((low<high)&& (L[high].key>=pivotkey))
            --high;
      L[low] = L[high];//将比枢轴大的元素移到左端
      while ((low<high)&& (L[low].key<=pivotkey))
            ++low;
      L[high] = L[low];//将比枢轴小的元素移到右端
   }
   L[low] = L[0];//枢轴元素存放到最终位置
   return low;     //返回枢轴位置
}
```

参考算法-3：划分方法

```
int Partition(int i，int j，int  pivot)
{  //对A[low..high]做划分，并返回基准记录的位置
    while(i<j)            //从区间两端交替向中间扫描，直至i=j为止
   {  while(i<j&&A[j].key>=pivot)          //pivot相当于在位置i上
       j--;                //从右向左扫描，查找第1个关键字小于pivot的记录A[j]
     if(i<j)               //表示找到的A[j]的关键字<pivot
       A[i++]=A[j];  //相当于交换A[i]和A[j]，交换后i指针加1
     while(i<j&&A[i].key<=pivot)          //pivot相当于在位置j上
       i++;                //从左向右扫描，查找第1个关键字大于pivot的记录A[i]
     if(i<j)               //表示找到了A[i]，使R[i].key>pivot
       A[j--]=A[i];     //相当于交换A[i]和A[j]，交换后j指针减1
    } //endwhile
   A[i]=pivot;                                  //基准记录已被最后定位
   return i;
  } //partition
```

快速分类算法　QuickSort(1, n)调用

```
Void   QuickSort ( int i, int  j )
{   keytype  pivot ; int  piovtindex , k ;
    pivotindex = FindPivot( i, j ) ;//下标
    if ( pivotindex )
       {   pivot = A[pivotindex].key ;
           k = Partion ( i, j, pivot ) ;
           QuickSort( i, k-1);
           QuickSort( k, j );
       }
}
```

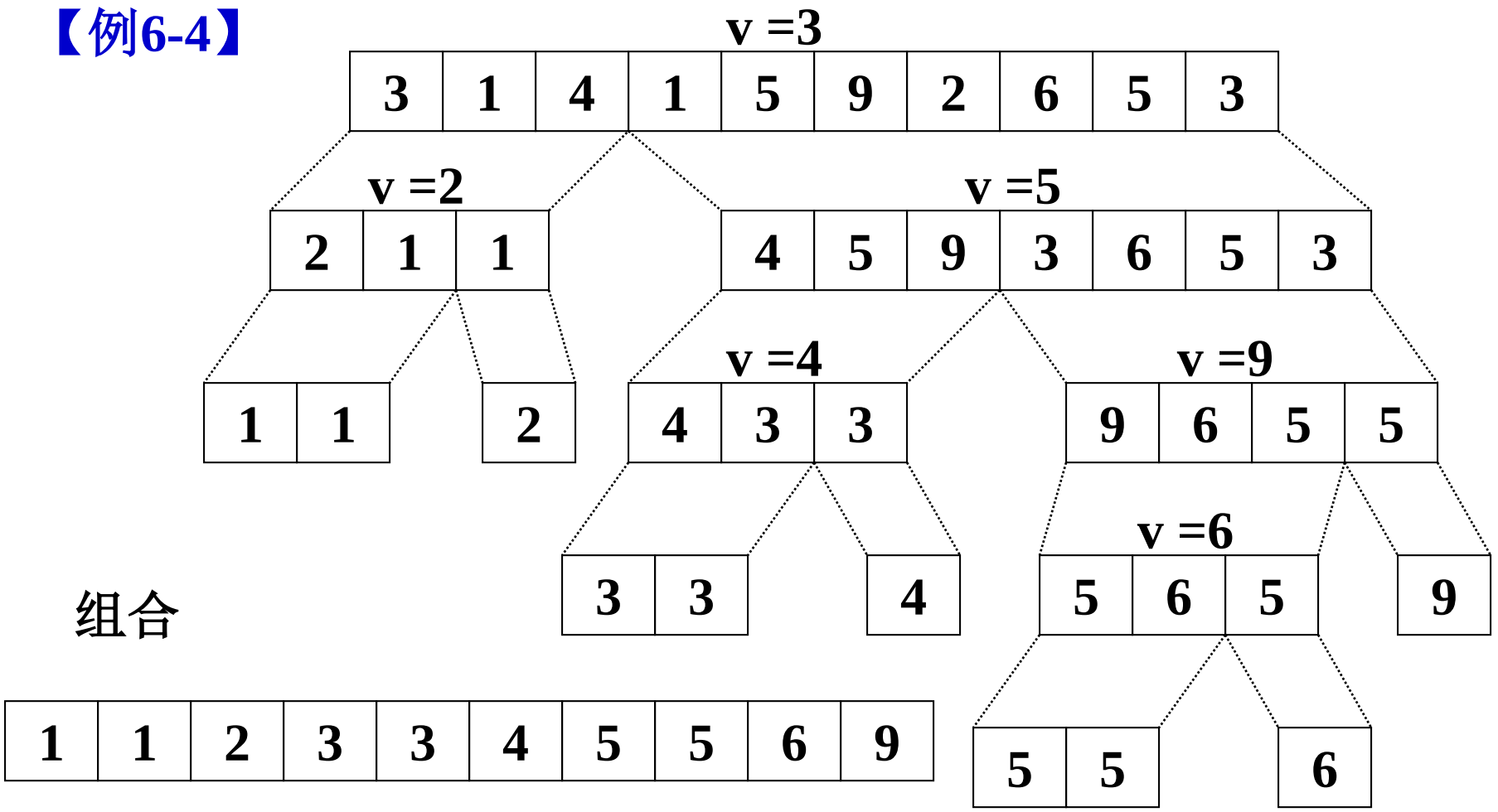
快速排序是所有内部排序算法中平均性能最优的排序算法！！！

时间、空间复杂性和稳定性：

时间复杂度：$T(n) = O(n\log_2 n)$，稳定性：是不稳定的分类

空间复杂度：最好情况$O(\log_2 n)$，最坏情况$O(n)$，平均情况$O(\log_2 n)$

【例6-4】

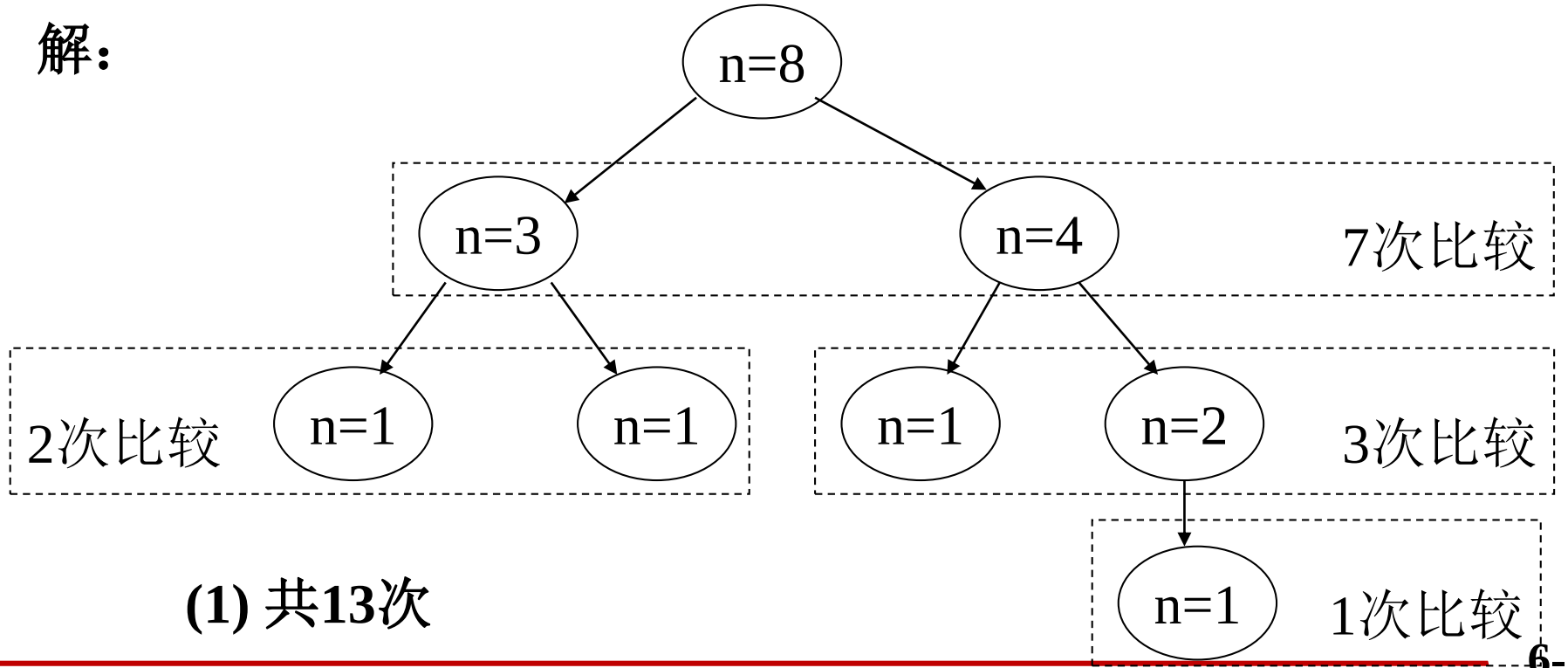

v =3
3 1 4 1 5 9 2 6 5 3
v =2
2 1 1
v =5
4 5 9 3 6 5 3
1 1
2
v =4
4 3 3
v =9
9 6 5 5
3 3
4
v =6
5 6 5
9
5 5
6
组合
1 1 2 3 3 4 5 5 6 9

【例6-5】对长度为n的记录序列进行快速排序时，所需要的比较次数依赖于这n个元素的初始序列。

问：(1) 当n=8时，在最好情况下需要进行多少次比较？

(2) 给出n=8时的最好情况的初始排序实例。

解：



(1) 共13次

(2) 如： 23 13 17 21 60 30 18 28

一次划分后：{18 13 17 21} 23 {30 60 28}

二次划分后：{17 13} 18 {21}

三次划分后：{13} 17

13

21

四次划分后： {28} 30 {60}

28 60

结果： 13 17 18 21 23 28 30 60

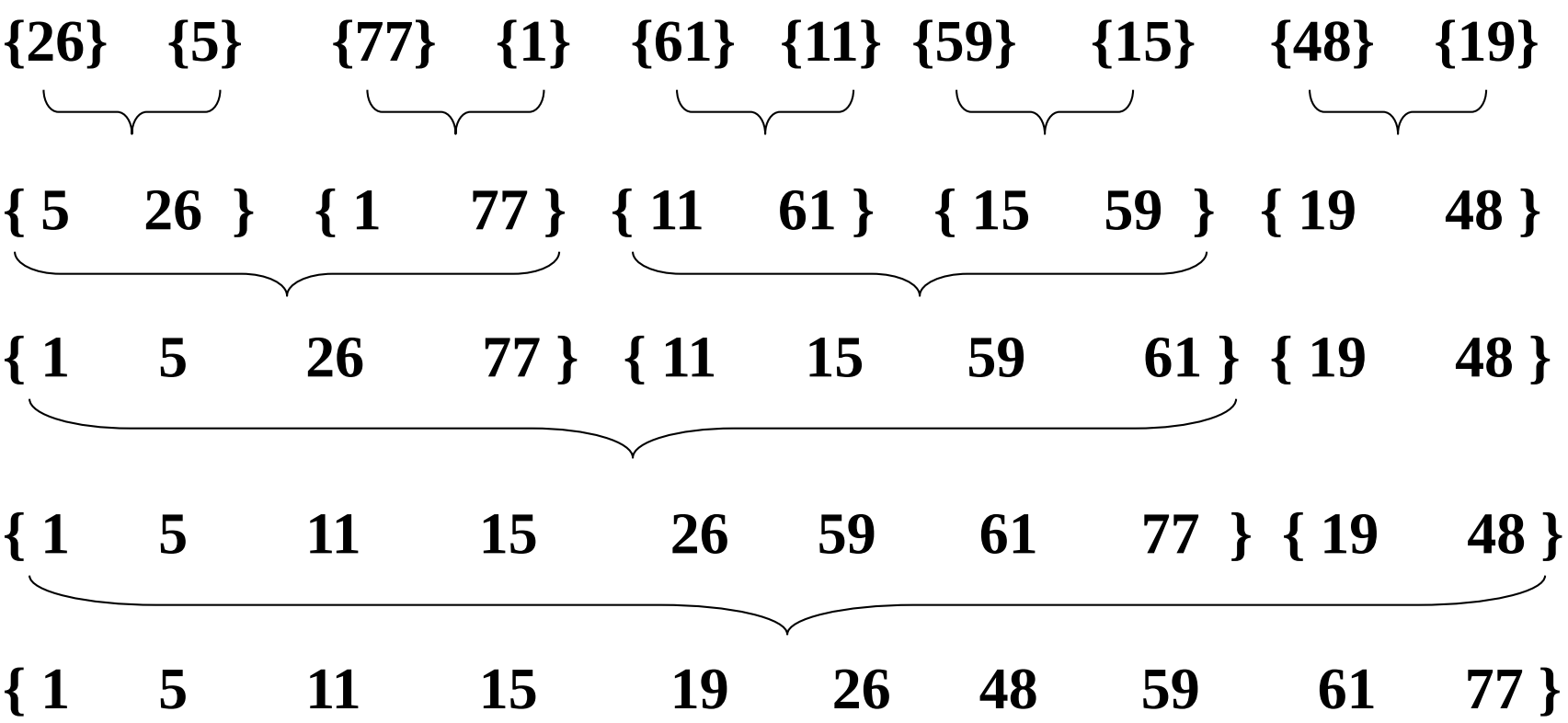
6.4　归并分类

合并两个有序序列

合并A[ℓ],…,A[m] 和 A[m+1],…,A[n] 形成新的分类序列 B[ℓ],…,…,B[n]

```
Void   Merge ( ℓ, m, n, A, B )
Int ℓ, m, n ;  LIST A, &B ;
{  int  i, j, k, t ;
   i = ℓ ; k = ℓ ; j = m+1 ;
   while ( ( i <= m ) && ( j <= n ) )
      {   if ( A[i].key <= A[j].key )
             B[k++] = A [i++] ;
          else
             B[k++] = A[j++] ;
       }
   if ( i > m )   for ( t = j ; t <= n ;  t++ )    B[k+t-j] = A[t] ;
   else           for ( t = i ; t <= m ; t++ )    B[k+t-i] = A[t] ;
}
```

算法时间复杂度 $T(n) = O(n-ℓ+1)$

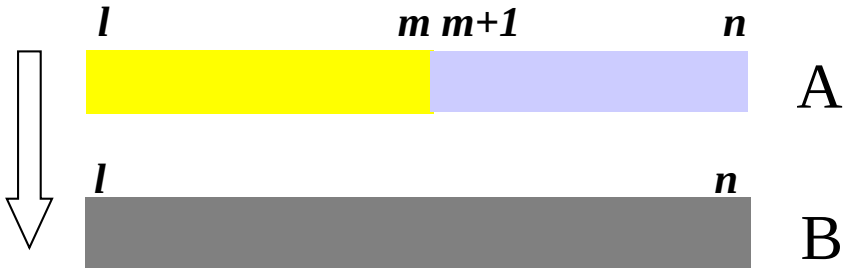


归并次数：*1，2，…，2^{i-1}*，若长度为 n，则共需归并 $log_2 n$ 总的时间复杂性为：$T(n) = O(n \cdot log_2 n)$

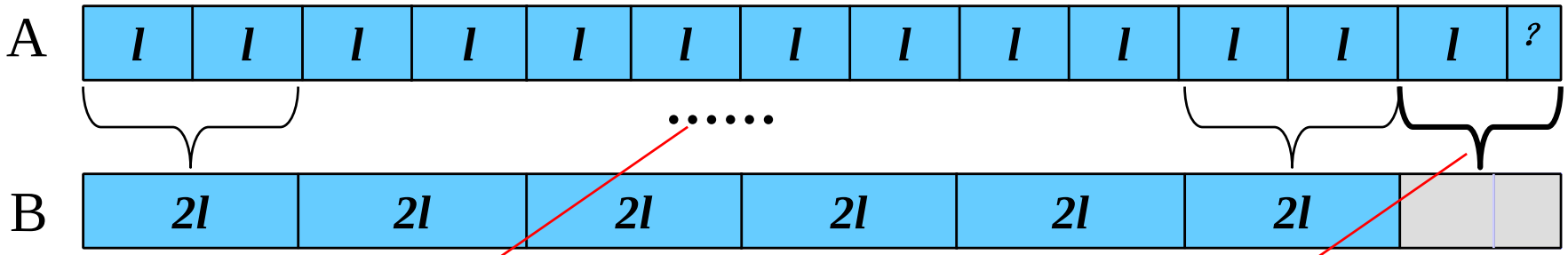
```
Void   Mpass ( n, ℓ, A , B )
int  n, ℓ ;  LIST A, &B ;
{  int  i , t ;
   i = 1 ;
   while ( i <= (n-2*ℓ+1) )
      {   Merge ( i, i+ℓ-1, i+2*ℓ-1, A, B )
          i = i + 2*ℓ ;  }
    if (( i+ℓ-1) < n )
        Merge ( i , i+ℓ-1, n, A, B );
    else
        for ( t = i ; t <= n ; t++ )   B[t] = A[t];
}
```

```
Void  T_W_SORT(n , A )
Int n ;  LIST  &A ;
{   int ℓ ;
    LIST  B ;
    ℓ = 1 ;
    while( ℓ < n )
    {   Mpass( n, ℓ, A, B ) ;
        ℓ = 2* ℓ ;
        Mpass( n, ℓ, B, A ) ;
        ℓ = 2*ℓ ;
     }
}
```

Void Merge (*l* , m, n, A, B)



Void Mpass (n, *l*, A , B)

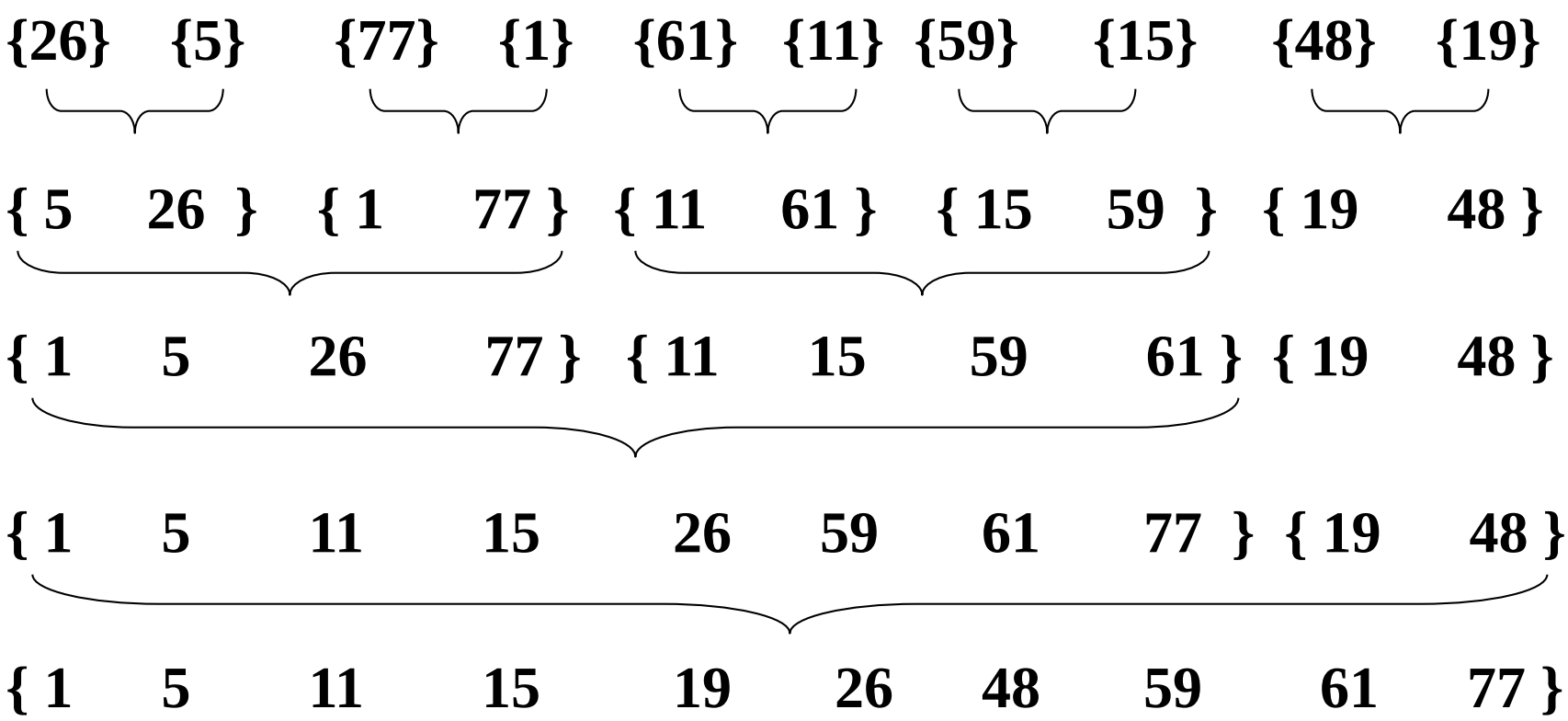



```
i = 1 ;
while ( i <= (n-2*l+1) )
{   Merge ( i, i+l-1, i+2*l-1, A, B )
    i = i + 2*l ;  }
```

```
if (( i+l-1) < n )
   Merge ( i , i+l-1, n, A, B );
else
   for ( t = i ; t <= n ; t++ )   B[t] = A[t];
```

>l

<l



归并次数：*1，2，…，2^{i-1}*，若长度为 n，则共需归并 $log_2 n$ 总的时间复杂性为：$T(n) = O(n \cdot log_2 n)$

归并排序算法性能分析：

■时间性能：

① 一趟归并操作是将A[1]~A[n]中相邻的长度为h的有序序列进行两两归并，并把结果存放B[1]~B[n]中，这需要$O(n)$时间。整个归并排序需要进行$\lceil \log_2 n \rceil$趟，因此，总的时间代价是$O(n\log_2 n)$。

■空间性能：

① 算法在执行时，需要占用与原始记录序列同样数量的存储空间，因此，空间复杂度为$O(n)$。

6.5　堆分类

【定义】把具有如下性质的数组A表示的二元树称为堆（Heap）：

(1) 若2*i≤n，则A[i].key ≤ A[2*i].key ;

(2) 若2*i+1≤n，则A[i].key≤A[2*i+1].key ;　　小顶堆

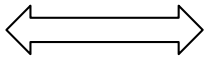
或：　把具有如下性质的数组A表示的二元树称为堆（Heap）：

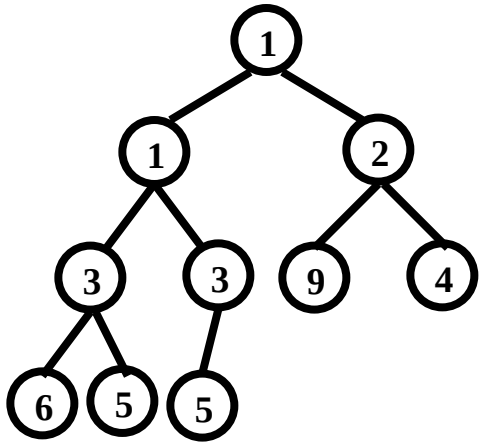
(1) 若2*i≤n，则A[i].key ≥ A[2*i].key ;

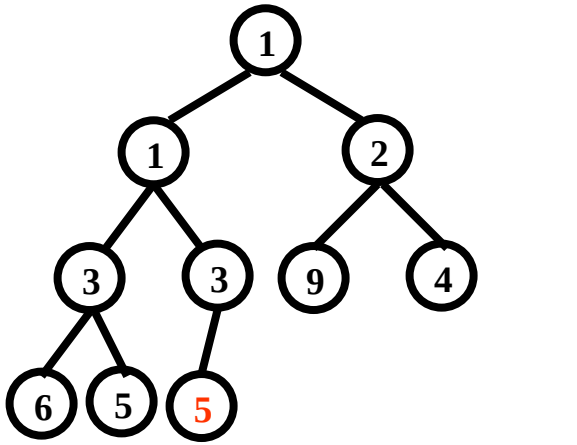
(2) 若2*i+1≤n，则A[i].key≥ A[2*i+1].key ;　　大顶堆

【例6-6】

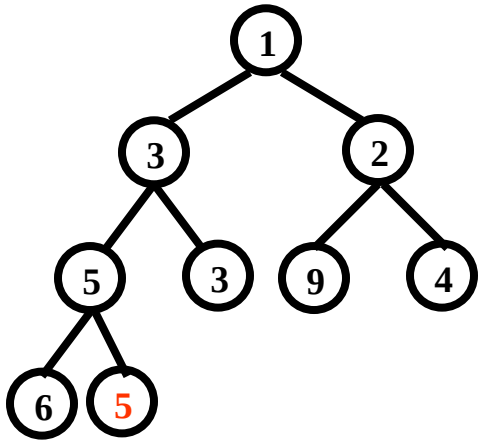
1	1	2	3	3	9	4	6	5	5



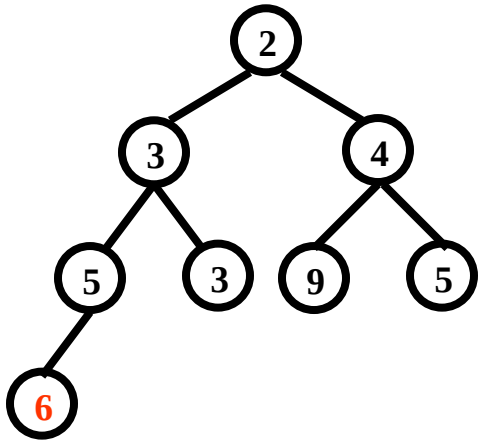




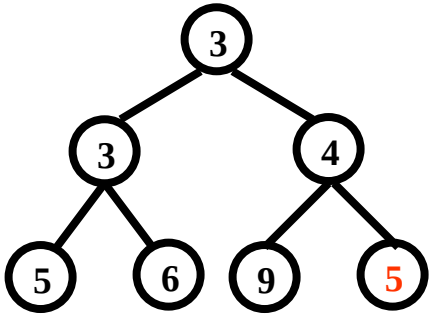
1	1	2	3	3	9	4	6	5	5



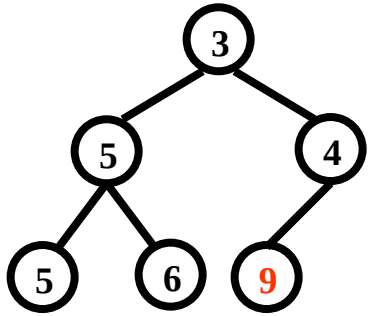
1	3	2	5	3	9	4	6	5	1



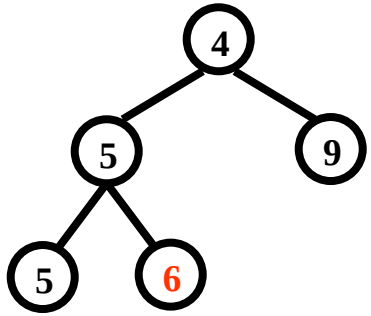
2	3	4	5	3	9	5	6	1	1



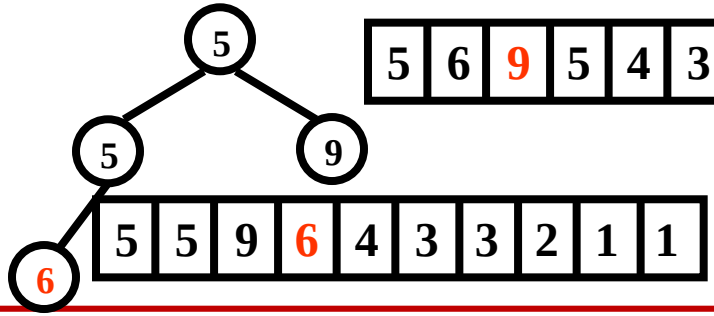
3	3	4	5	6	9	5	2	1	1



3	5	4	5	6	9	3	2	1	1

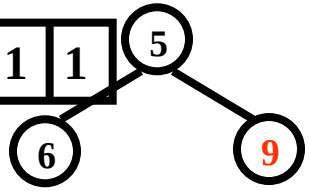


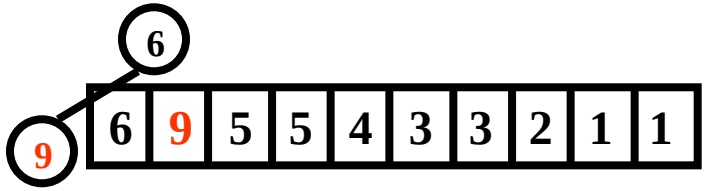
4	5	9	5	6	3	3	2	1	1



5	5	9	6	4	3	3	2	1	1

5	6	9	5	4	3	3	2	1	1





堆的性质：

① 对于任意一个非叶结点的关键字，都不大于其左、右儿子结点的关键字。

即A[i/2].key≤A[i].key　　1 ≤ i/2 < i ≤n。

② 在堆中，以任意结点为根的子树仍然是堆。特别地，每个叶也点也可视为堆。每个结点都代表(是)一个堆。

- 以堆（的数量）不断扩大的方式进行初始建堆。

③ 在堆中（包括各子树对应的堆），其根结点的关键字是最小的。去掉堆中编号最大的叶结点后，仍然是堆。

- 以堆的规模逐渐缩小的方式进行堆排序。

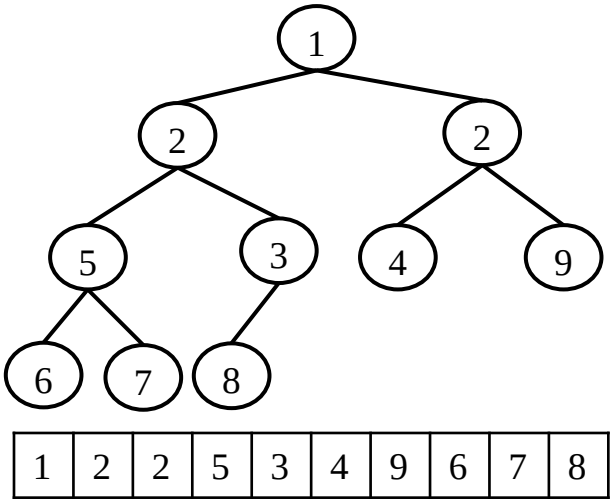
整理堆的算法

```
Void  PushDown( first, last )
int  first , last ;
{   int r ;
    r = first ;
    while ( r <= last/2 )
      if ( ( r == last/2 ) && ( last%2 == 0 ) ) //有度为1的节点，只能是r
        {  if ( A[r].key > A[2*r].key )
              swap ( A[r], A[2*r] ) ;
           r = last ;  /*结束循环*/       }
      else  if ( ( A[r].key > A[2*r].key ) && ( A[2*r].key <= A[2*r+1].key ) )
        {  swap ( A[r], A[2*r] ) ;          /*左孩子比右孩子小，与左孩子交换*/
           r = 2*r ;                        }
      else  if ( ( A[r].key > A[2*r+1].key ) && ( A[2*r+1].key < A[2*r].key ) )
        {  swap ( A[r], A[2*r+1] ) ;     /*右孩子比左孩子小，与右孩子交换*/
           r = 2*r+1 ;                      }
      else
          r = last ;
}
```

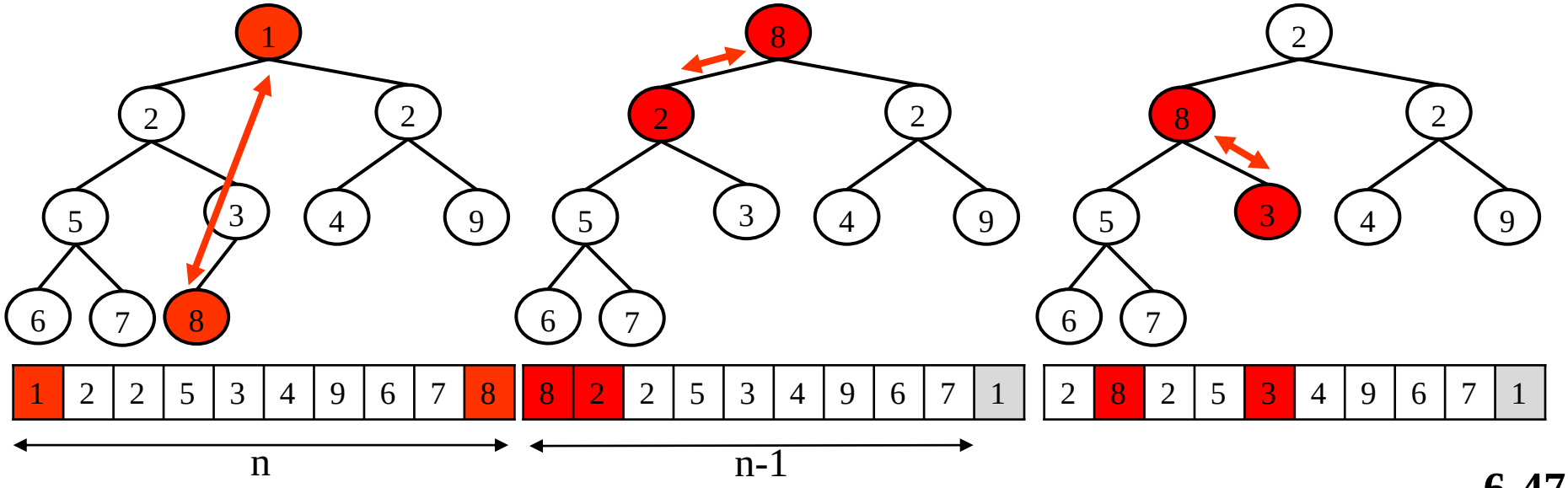
last/2?

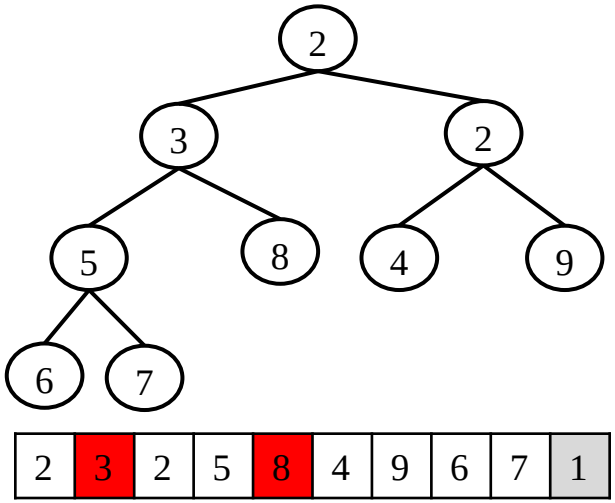
O(last/first)

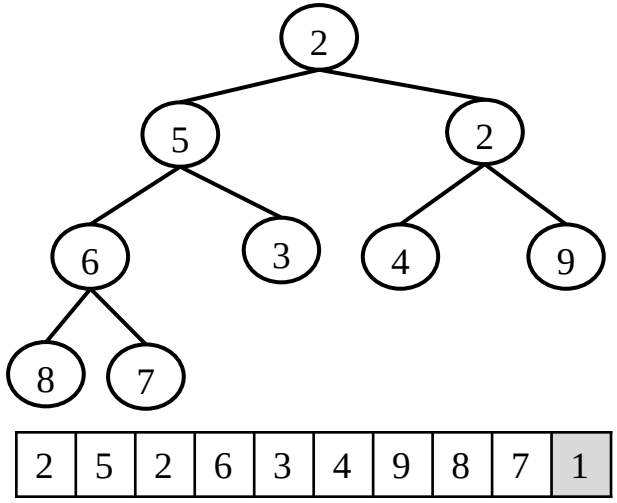
1次Pushdown




```
while ( r <= last/2 )
    if ( ( r == last/2 ) && ( last%2 == 0 ) ) //有度为1的节点，只能是r
      {  if ( A[r].key > A[2*r].key )
            swap ( A[r], A[2*r] ) ;
         r = last ;  /*结束循环*/      }
    else  if ( ( A[r].key > A[2*r].key ) && ( A[2*r].key <= A[2*r+1].key ) )
      {  swap ( A[r], A[2*r] ) ;        /*左孩子比右孩子小，与左孩子交换*/
         r = 2*r ;                      }
    else  if ( ( A[r].key > A[2*r+1].key ) && ( A[2*r+1].key < A[2*r].key ) )
       {  swap ( A[r], A[2*r+1] ) ;     /*右孩子比左孩子小，与右孩子交换*/
          r = 2*r+1 ;                   }
    else    r = last ;
```




2
3
2
5
8
4
9
6
7
2 3 2 5 8 4 9 6 7 1


2
5
2
6
3
4
9
8
7
2 5 2 6 3 4 9 8 7 1

堆分类算法：

```
Void   Sort ( n,  A )
 int   n ;  LIST   A ;
{    int  i ;
     for ( i = n/2 ; i >= 1 ; i-- )
          PushDown( i , n ) ;
     for ( i = n ; i >= 2 ; i-- )
          {    swap ( A[1], A[i] ) ;
               PushDown ( 1,  i -1 ) ;  }
}
```

整理堆➔建堆

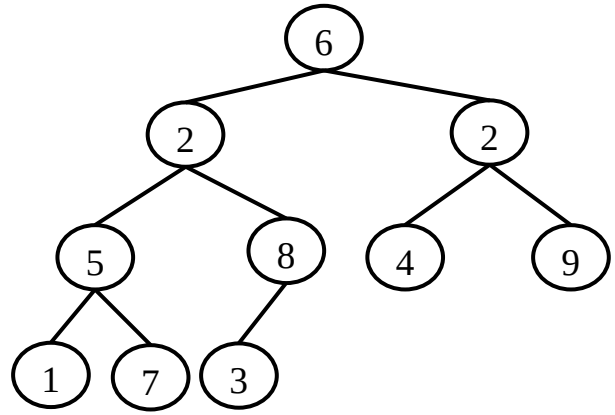
?

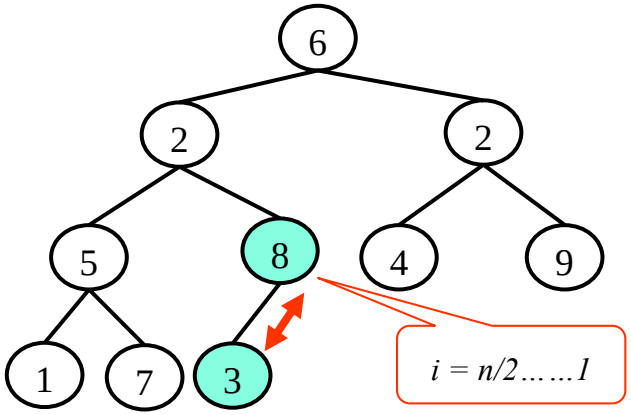
堆排序

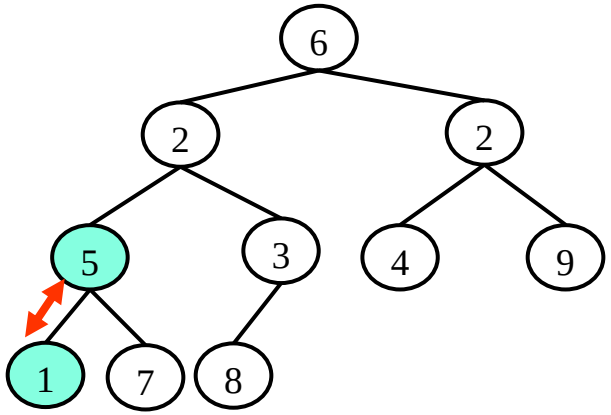
first　?
last　?

$T(n) = O(n \cdot \log_2 n)$

n/2次pushdown





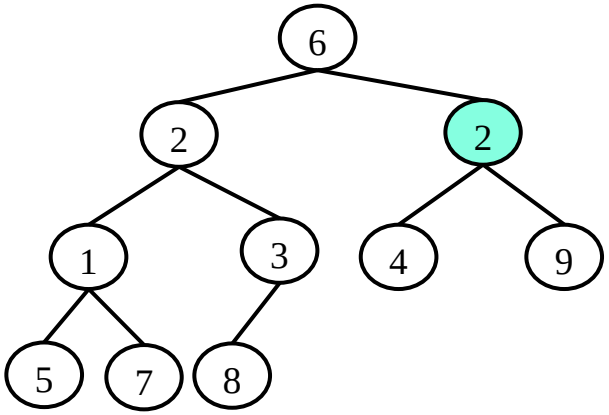


6	2	2	5	8	4	9	1	7	3

6	2	2	5	8	4	9	1	7	3

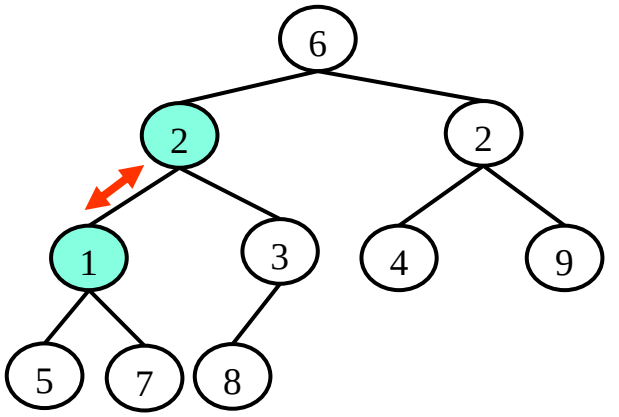
6	2	2	5	3	4	9	1	7	8

```
while ( r <= last/2 )
    if ( ( r == last/2 ) && ( last%2 == 0 ) ) //有度为1的节点，只能是r
      {  if ( A[r].key > A[2*r].key )
            swap ( A[r], A[2*r] ) ;
         r = last ;   /*结束循环*/       }
    else  if ( ( A[r].key > A[2*r].key ) && ( A[2*r].key <= A[2*r+1].key ) )
      {  swap ( A[r], A[2*r] ) ;          /*左孩子比右孩子小，与左孩子交换*/
         r = 2*r ;                        }
    else  if ( ( A[r].key > A[2*r+1].key ) && ( A[2*r+1].key < A[2*r].key ) )
      {  swap ( A[r], A[2*r+1] ) ;     /*右孩子比左孩子小，与右孩子交换*/
         r = 2*r+1 ;                      }
    else
       r = last ;
```

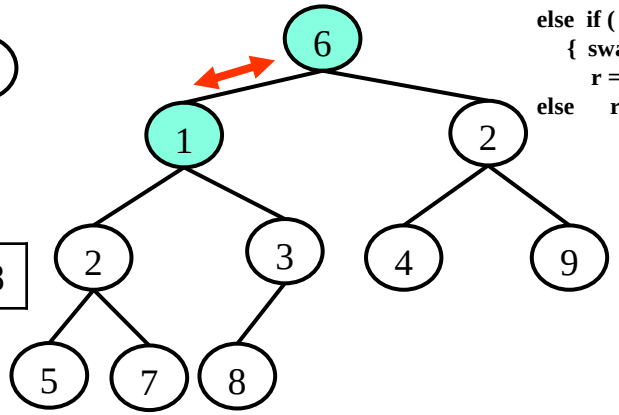


6	2	2	1	3	4	9	5	7	8

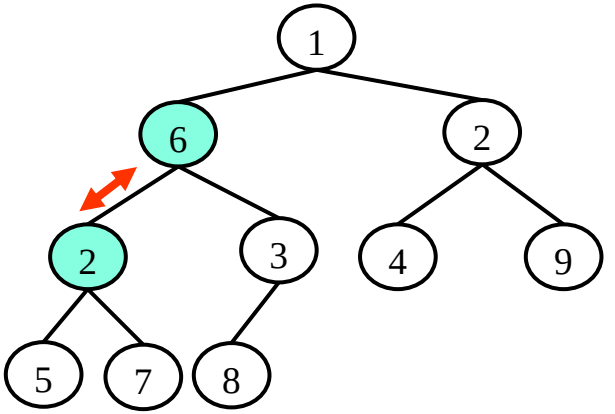
```
while ( r <= last/2 )
    if ( ( r == last/2 ) && ( last%2 == 0 ) ) //有度为1的节点，只能是r
      {  if ( A[r].key > A[2*r].key )
            swap ( A[r], A[2*r] ) ;
         r = last ;  /*结束循环*/      }
    else  if ( ( A[r].key > A[2*r].key ) && ( A[2*r].key <= A[2*r+1].key ) )
      {  swap ( A[r], A[2*r] ) ;        /*左孩子比右孩子小，与左孩子交换*/
         r = 2*r ;                      }
    else  if ( ( A[r].key > A[2*r+1].key ) && ( A[2*r+1].key < A[2*r].key ) )
      {  swap ( A[r], A[2*r+1] ) ;     /*右孩子比左孩子小，与右孩子交换*/
         r = 2*r+1 ;                    }
    else     r = last ;
```



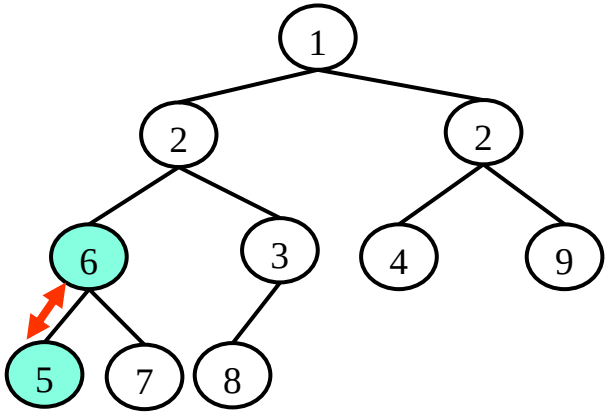
6	2	2	1	3	4	9	5	7	8



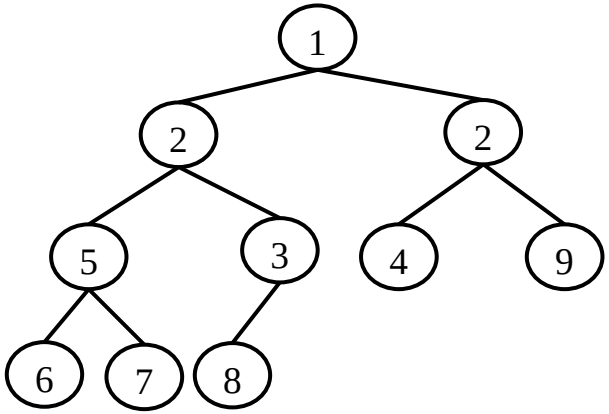
6	1	2	2	3	4	9	5	7	8



1	6	2	2	3	4	9	5	7	8



1	2	2	6	3	4	9	5	7	8



1	2	2	5	3	4	9	6	7	8

■ 算法性能分析

① PutDown函数中，执行一次while循环的时间是一个常数。因为r 每次至少为原来的两倍，假设while循环执行次数为 i，则当r 从first 变为first*2^i 时循环结束。此时r=first*2^i >last/2， 即 i > $\log_2$(last/first)-1。所以while循环体最多执行$\log_2$(last/first)次，即PushDown时间复杂度O(log(last/first))=O($\log_2$n)。

② 所以，HeapSort时间复杂度： O(n $\log_2$n)。

③ 这是堆排序的最好、最坏和平均的时间代价。

④ HeapSort空间复杂度： O(1)。

优先级队列

【定义1】优先队列是一种用来维护一组元素构成的集合 S的数据结构，其中每个元素都有一个关键字key，元素之间的比较都是通过 key 来比较的。优先队列包括最大优先队列和最小优先队列。

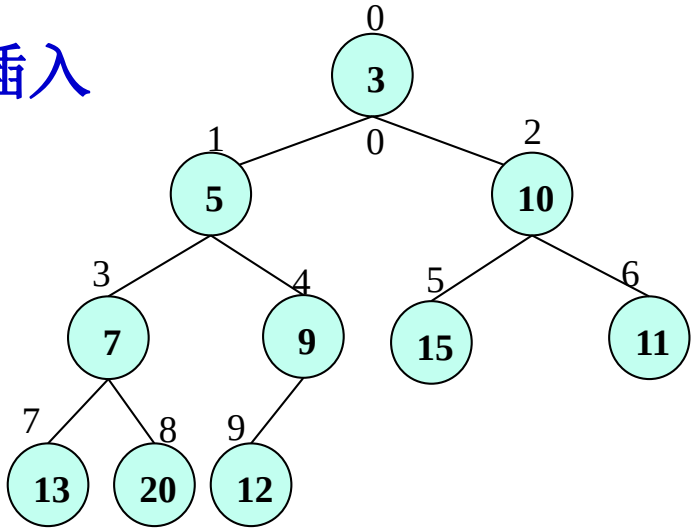
【定义2】首先它是一个队列，但是它强调了“优先”二字， 所以，已经不能算是一般意义上的队列了，它的“优先”意指取队首元素时，有一定的选择性，即根据元素的属性选择某一项值最优的出队~

【定义3】是不同于先进先出队列的另一种队列。每次从队列中取出的是具有最高优先权的元素

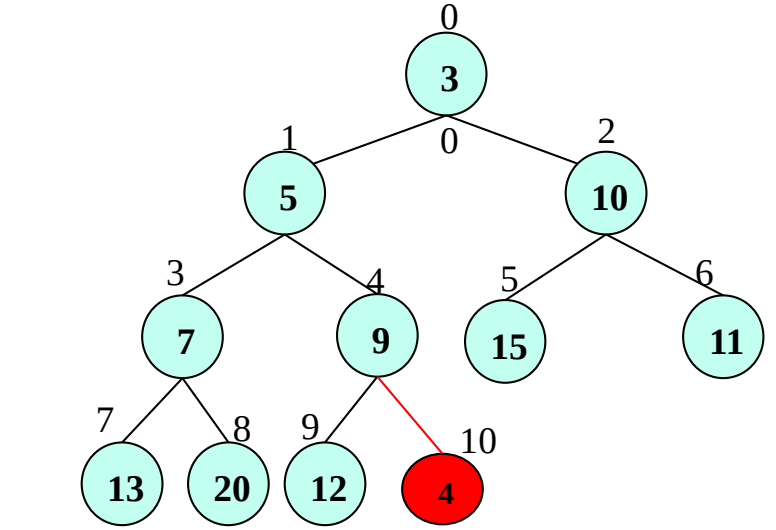
优先级队列应用：优先队列的应用比较广泛，比如作业系统中的调度程序，当一个作业完成后，需要在所有等待调度的作业中选择一个优先级最高的作业来执行，并且也可以添加一个新的作业到作业的优先队列中。

基本操作：插入和删除

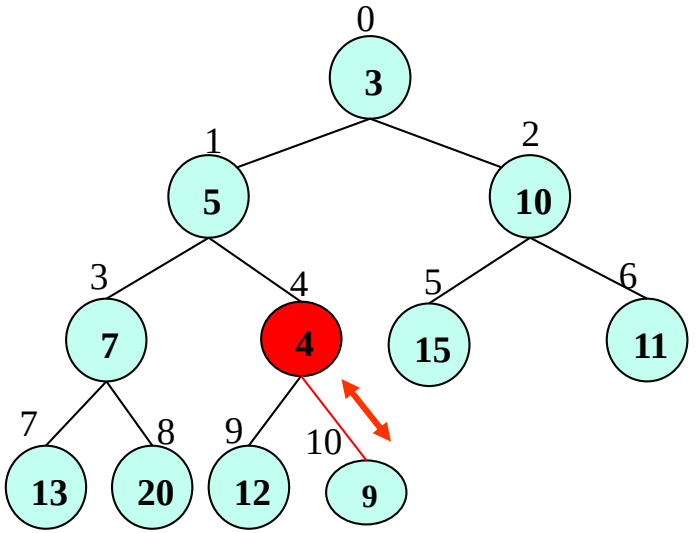
插入



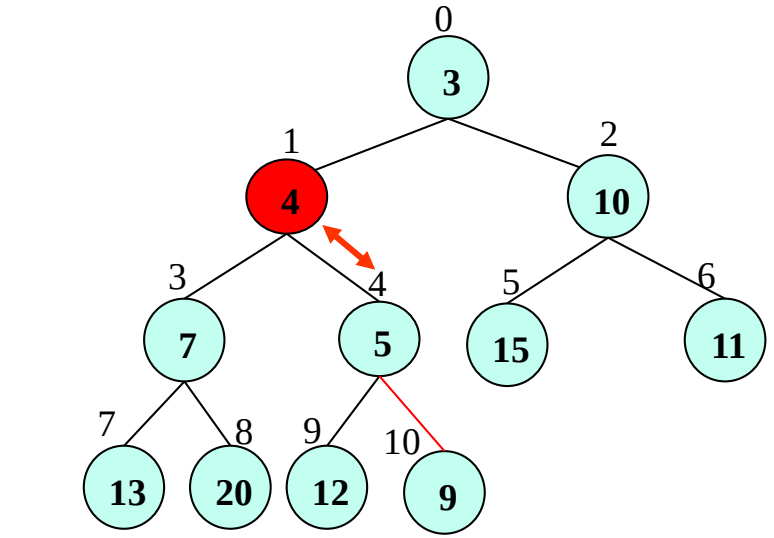
0	1	2	3	4	5	6	7	8	9	10	11
3	5	10	7	9	15	11	13	20	12	…	



0	1	2	3	4	5	6	7	8	9	10	11
3	5	10	7	9	15	11	13	20	12	4	…

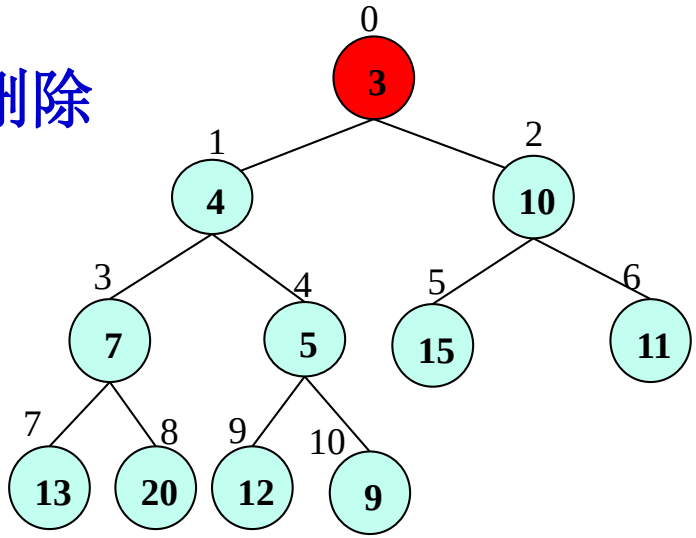


0	1	2	3	4	5	6	7	8	9	10	11
3	5	10	7	4	15	11	13	20	12	9	…

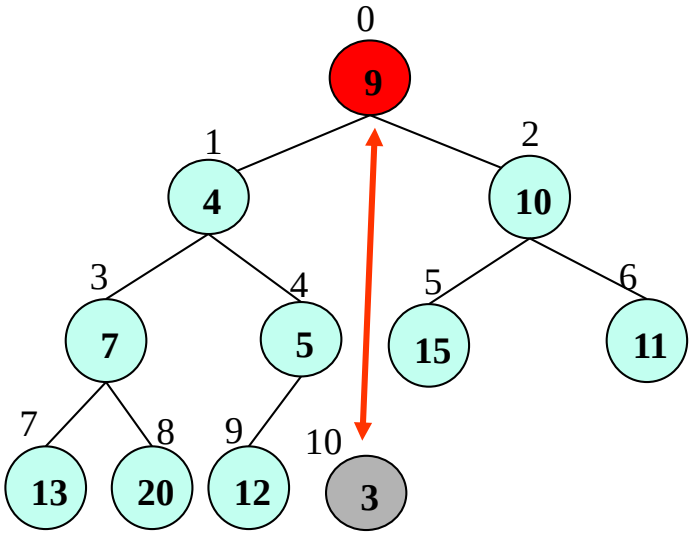


0	1	2	3	4	5	6	7	8	9	10	11
3	4	10	7	5	15	11	13	20	12	9	…

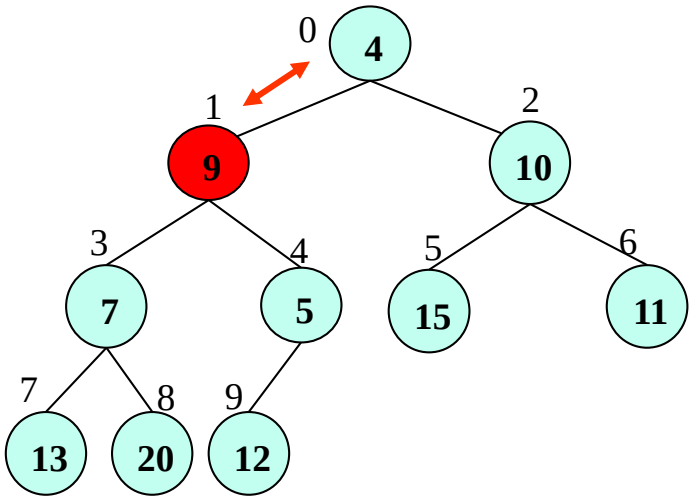
删除



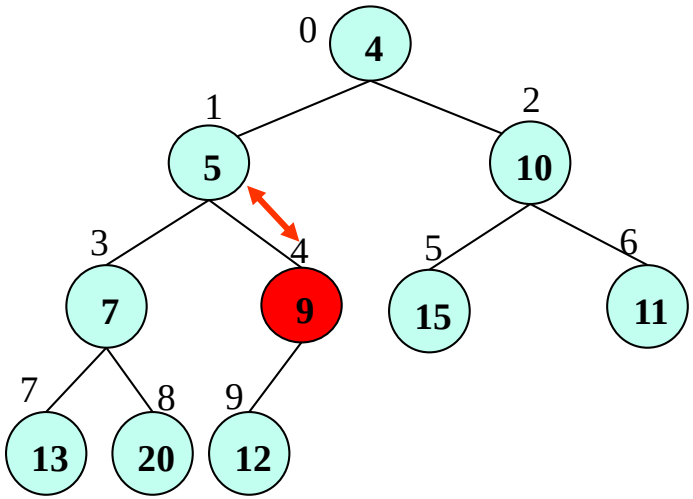
3	4	10	7	5	15	11	13	20	12	9	…	



9	4	10	7	5	15	11	13	20	12	3	…	



4	9	10	7	5	15	11	13	20	12	3	…	

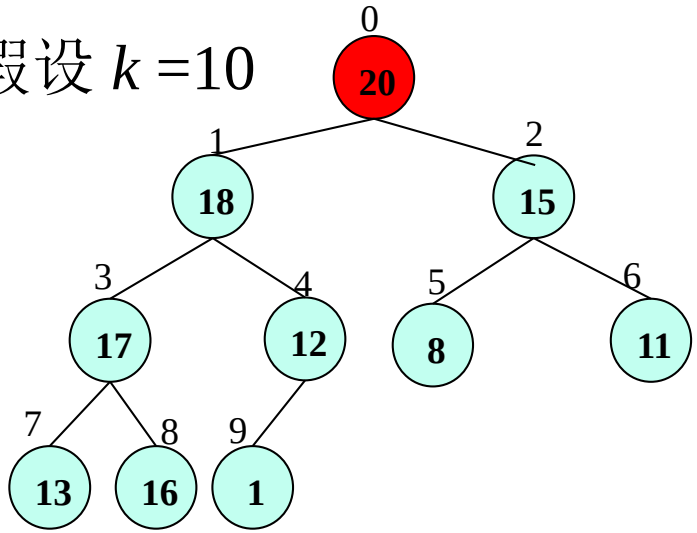


4	5	10	7	9	15	11	13	20	12	3	…	

堆应用：求某一个很大的数据集合中的 k 个最小的元素。

10，21，34，5，6，78，
90，1012，234，56，876，
90，100，20，45，789，
45，67，34，9，8，01，
34，54，32，654，
321，……

假设 k =10



20	18	15	17	12	8	11	13	16	1	…	

将每一个小于堆顶的元素替换堆顶元素，然后重新整理堆。

如果是求k个最大的元素呢？

6.6　基数分类——多关键字分类　♥ ♠ ♦ ♣

321 986 123 432 543 018 765 678 987 789 098 890 109 901 210 012

Q[0]:890 210
Q[1]:321 901
Q[2]:432 012
Q[3]:123 543
Q[4]:
Q[5]:765
Q[6]:986
Q[7]:987
Q[8]:018 678 098
Q[9]:789 109

Q[0]:901 109
Q[1]:210 012 018
Q[2]:321 123
Q[3]:432
Q[4]:543
Q[5]:
Q[6]:765
Q[7]:678
Q[8]:986 987 789
Q[9]:890 098

Q[0]:012 018 098
Q[1]:109 123
Q[2]:210
Q[3]:321
Q[4]:432
Q[5]:543
Q[6]:678
Q[7]:765 789
Q[8]:890
Q[9]: 901 986 987

890 210 321 901 432 012 123 543 765 986 987 018 678 098 789 109

901 109 210 012 018 321 123 432 543 765 678 986 987 789 890 098

012 018 098 109 123 310 321 432 543 678 765 789 890 901 986 987

```
Void   RadixSort ( figure , A )
int  figure ; QUEUE  &A ;
{    QUEUE   Q[10] ;
     records  data ;
     int  pass, r, i ;
     for ( pass=1 ; pass <=figure ; pass+ )
          {   for ( i=0 ; i<=9 ; i++ )
                 MakeNull( Q[i] ) ;
              while ( !Empty( A ) )
                 {    data = FRONT ( A ) ;
                      DeQueue ( A ) ;
                      r = Radix( data. key , pass ) ;
                      EnQueue( data , Q[r] ) ;          }
               for ( i=0 ; i <=9 ; i++ )
                    while ( !Empty( Q[i] ) )
                          {   data = Front ( Q[i] ) ;
                              DeQueue( Q[i] ) ;
                              EnQueue( data , A ) ;     }
           }
}
```

```
Int   Radix( k, p )
Int  k, p ;
{    int  power , i ;
     power = 1 ;
     for ( i=1; i<=p-1 ; i++ )
          power = power * 10 ;
     return (( k%(power*10))/power) ;
 }
```

■ 算法性能分析

① n----记录数，d----关键字（分量）个数，r----基数

② **时间复杂度：** 分配操作：O（n），收集操作O(r)，需进行d趟分配和收集。时间复杂度：O(d(n+r))

③ **空间复杂度：** 所需辅助空间为队首和队尾指针2r个，此外还有为每个记录增加的链域空间n个。空间复杂度O(n+r)

■ 算法的推广

① 若被排序的数据关键字由若干域组成，可以把每个域看成一个分量按照每个域进行基数排序。

② 若关键字各分量不是整数，则把各分量所有可以取值与一组自然数对应。

■ 举例

① 如何在O(n)时间内，对0到n^2-1之间的n个整数进行排序

各种排序方法的比较

序号	排序方法	平均时间	最好情况	最坏情况	辅助空间	稳定性
1	简单选择排序	$O(n^2)$	O（n）	$O(n^2)$	O(1)	不稳定
2	直接插入排序	$O(n^2)$	O(n)	$O(n^2)$	O(1)	稳定
3	折半插入排序	$O(n^2)$	$O(n\cdot\log_2 n)$	$O(n^2)$	O(1)	不稳定
4	冒泡排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	稳定
5	希尔排序	$O(n^{1.3})$	O（n）	$O(n^2)$	O(1)	不稳定
6	堆排序	$O(n\cdot\log_2 n)$	$O(n\cdot\log_2 n)$	$O(n\cdot\log_2 n)$	O(1)	不稳定
7	归并排序	$O(n\cdot\log_2 n)$	$O(n\cdot\log_2 n)$	$O(n\cdot\log_2 n)$	O(n)	稳定
8	快速排序	$O(n\cdot\log_2 n)$	$O(n\cdot\log_2 n)$	$O(n^2)$	$O(\log_2 n)$	不稳定
9	基数排序	$O(d\cdot(n+r))$ $O(d\cdot(n+rd))$	$O(d\cdot(n+r\cdot d))$	$O(d\cdot(n+r))$ $O(d\cdot(n+rd))$	$O(n+r\cdot d)$ $O(r\cdot d)$	稳定

对100万个数据排序统计结果(单位：毫秒)

序号	排序方法	平均情况	最坏情况（逆序）	最好情况（正序）
1	冒泡排序	549432.000	1534035.000	366936.000
	选择排序	478694.000	587240.000	367658.000
	插入排序	253115.000	515621.000	0.897
2	希尔排序/增量 3	61.000	203.000	35.000
3	堆排序	79.000	126.000	74.800
4	归并排序	70.000	140.000	61.000
5	快速排序	39.000	93.000	30.000
6	基数排序/进制 100	117.000	118.000	116.000
	基数排序/进制 1000	89.000	90.000	88.000

部分算法的时间效率比较 （单位： 毫秒）

序号	10	100	1K	10K	100K	1M
冒泡排序	0.000276	0.005643	0.545	61.000	8174.000	549432
选择排序	0.000237	0.006438	0.488	47.000	4717.000	478694
插入排序	0.000258	0.008619	0.764	56.000	5145.000	515621
希尔排序/增量 3	0.000522	0.003372	0.036	0.518	4.152	61
堆排序	0.000450	0.002991	0.041	0,531	6.506	79
归并排序	0.000723	0.006225	0.066	0.561	5.480	70
快速排序	0.000291	0.003051	0.030	0.311	3.634	39
基数排序/进制100	0.005181	0.021000	0.165	1.650	11.428	117
基数排序/进制1000	0.016134	0.026000	0.139	1.264	8.394	89

*来自于学生测试数据

对排序算法应该从以下几个方面综合考虑：

① 时间复杂度；
② 空间复杂度；
③ 稳定性；
④ 算法简单性；
⑤ 待排序记录个数n的大小；
⑥ 记录本身信息量的大小；
⑦ 关键字值的分布情况。

（讨论）不同条件下，排序方法的选择：

(1)若n较小(如n≤50)，可采用直接插入或直接选择排序。

当记录规模较小时，直接插入排序较好；否则因为直接选择移动的记录数少于直接插人，应选直接选择排序为宜。

(2)若文件初始状态基本有序(指正序)，则应选用直接插人、冒泡或随机的快速排序为宜；且以直接插入排序最佳。

(3)若n较大，则应采用时间复杂度为O(nlgn)的排序方法：

快速排序、堆排序或归并排序。

- **快速排序是目前基于比较的内部排序中被认为是最好的方法，当待排序的关键字是随机分布时，快速排序的平均时间最短；**
- **堆排序所需的辅助空间少于快速排序，并且不会出现快速排序可能出现的最坏况。这两种排序都是不稳定的。**
- **基数排序适用于 n 值很大而关键字较小的序列。**
- **若要求排序稳定，则可选用归并排序，基数排序稳定性最佳。**