第6章 内部排序

学习目标

- 掌握排序的基本概念和常用术语
- 熟练掌握插入排序、希尔排序；气泡排序、快速排序；选择排序、堆排序；归并排序；基数排序的基本思想、算法原理、排序过程和算法实现。
- 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。

本章主要内容

- 6.1 基本概念
- 6.2 气泡排序
- 6.3 快速排序
- 6.4 直接选择排序 /锦标赛排序
- 6.5 堆排序
- 6.6（直接）插入排序
- 6.7 希尔排序
- 6.8 （二路）归并排序
- 6.9 基数排序
- 本章小结

一个元素，自然有序
局部有序，到全局有序

练习：排序算法

- 排序算法是数据处理的最基本和最重要的操作。其目的是将一组“无序”的记录序列调整为“有序”的记录序列。
- 内容：

实现经典的排序算法，通过实验数据的设计，考察不同规模和分布的数据对排序算法运行时间影响的规律，验证理论分析结果的正确性。

- 要求： 实现以下四组排序算法中的任意三组：
  - 冒泡排序和快速排序；
  - 直接选择排序和堆排序。
  - 插入排序和希尔排序；
  - （二路）归并排序, 基数排序
  - 产生不同规模和分布的数据，以“图或表”的方式给出输入规模和分布对排序方法运行时间变化趋势的影响（画出T(n)的曲线）。并与理论分析结果比较。
  - 将上述“图或表”采用图片等形式贴在报告中，并作适当分析或说明。

（单位：毫秒）

部分算法的时间效率比较

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

6.1 基本概念

- 排序（sorting）也称分类：
  - 假设给定一组n个记录序列$\{r_1, r_2, \ldots\ldots, r_n\}$，其相应的关键字序列为$\{k_1, k_2, \ldots\ldots, k_n\}$。排序是将这些记录排列成顺序为$\{r_{s1}, r_{s2}, \ldots\ldots, r_{sn}\}$的一个序列，使得相应的关键字的值满足$k_{s1} \le k_{s2} \le \ldots\ldots \le k_{sn}$（称为升序）或$k_{s1} \ge k_{s2} \ge \ldots\ldots \ge k_{sn}$（称为降序）。
- 排序的目的：方便查询和处理。
- 排序算法的稳定性：
  - 假定在待排序的记录集中，存在多个具有相同关键字值的记录，若经过排序，这些记录的相对次序仍然保持不变。
  - 即在原序列中，$k_i=k_j$且r_i在r_j之前，而在排序后的序列中，r_i仍在r_j之前，则称这种排序算法是稳定的；否则称为不稳定的。

6.1 基本概念（cont.）

- 排序分类：
  - 按照排序时排序对象存放的设备，分为，
    - 内部排序:排序过程中数据对象全部在内存中的排序。
    - 外部排序:排序过程数据对象并非完全在内存中的排序
  - 按照排序是的基本操作是否基于关键字的比较？分为，
    - 基于比较：基本操作——关键字的比较和记录的移动，其最好时间下限已经被证明为$\Omega(n\log_2 n)$。
      - 交换排序（气泡、快速排序）；
      - 选择排序（直接选择、堆排序）；
      - 插入排序（直接插入、折半插入、希尔排序）、
      - 归并排序（二路归并排序）。
    - 不基于比较：根据根据组成关键字的分量及其分布特征，如基数排序。

6.1 基本概念（cont.）

- 排序算法的性能：
  - 基本操作 内排序在排序过程中的基本操作：
    - 比较：关键码之间的比较；
    - 移动：记录从一个位置移动到另一个位置。
  - 辅助存储空间。
    - 辅助存储空间是指在数据规模一定的条件下，除了存放待排序记录占用的存储空间之外，执行算法所需要的其他额外存储空间。
  - 算法本身的复杂度
- 排序算法及其存储结构：

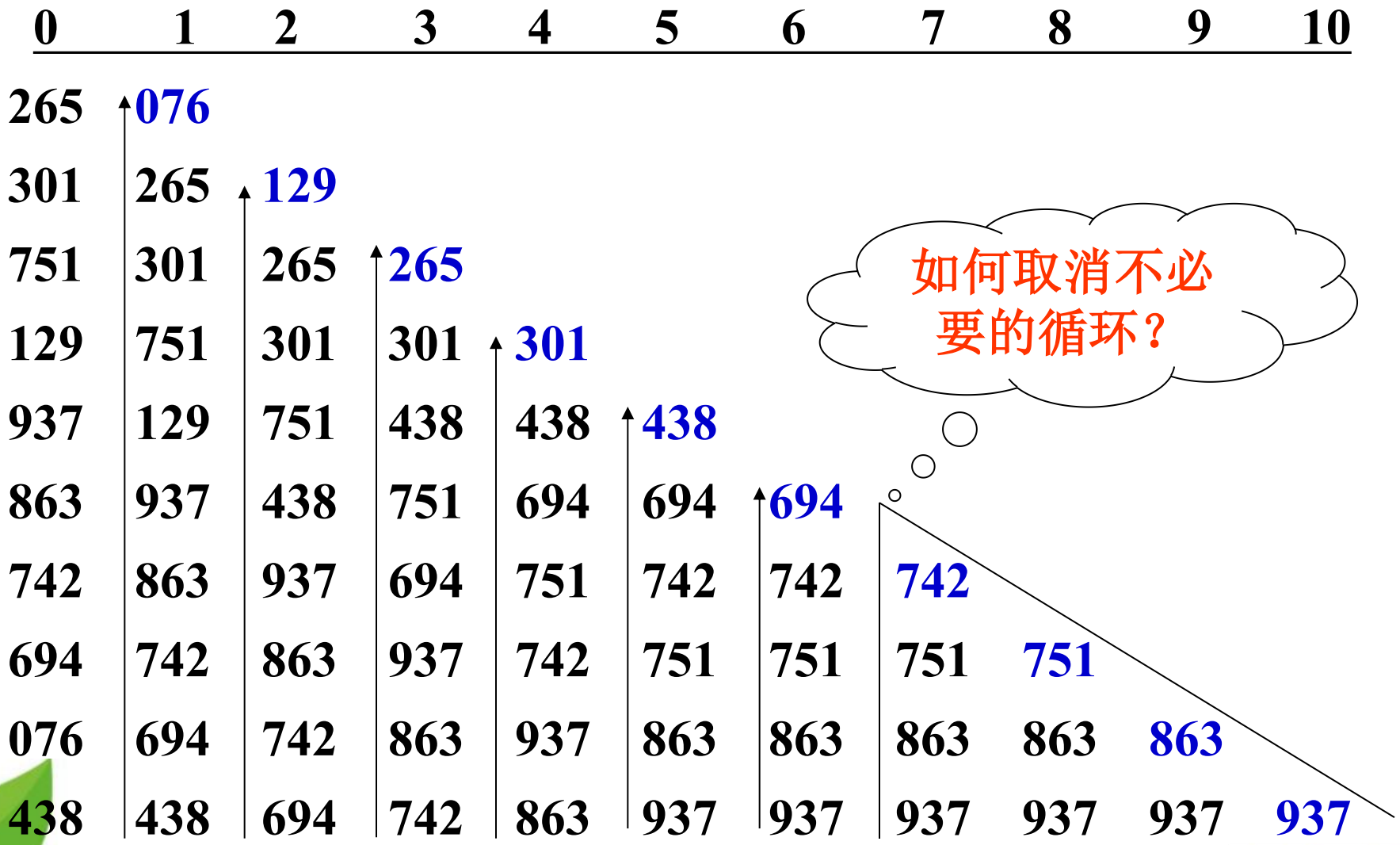
```
struct   records {
        keytype   key ;
        fields   other ;
};
typedef   records  LIST[maxsize] ;
```

  - Sort（ int n , LIST &A）:对n个记录的数组按照关键字不减的顺序进行排序。

6.2 气泡排序

- 算法的基本思想：
  - 将待排序的记录看作是竖着排列的“气泡”，关键字较小的记录比较轻，从而要往上浮。
  - 对这个“气泡”序列进行*n*-1遍（趟）处理。所谓一遍（趟）处理，就是自底向上检查一遍这个序列，并注意两个相比较的关键字的顺序是否正确。如果发现顺序不对，即“轻”的记录在下面，就交换它们的位置。
  - 显然，处理1遍之后，“最轻”的记录就浮到了最高位置；处理2遍之后，“次轻”的记录就浮到了次高位置。在作第二遍处理时，由于最高位置上的记录已是“最轻”的，所以不必检查。一般地，第i遍处理时，不必检查第i高位置以上的记录的关键字，因为经过前面*i*-1遍的处理，它们已正确地排好序。

6.2 气泡排序


0 1 2 3 4 5 6 7 8 9 10
265 076
301 265 129
751 301 265 265
129 751 301 301 301
937 129 751 438 438 438
863 937 438 751 694 694 694
742 863 937 694 751 742 742 742
694 742 863 937 742 751 751 751 751
076 694 742 863 937 863 863 863 863 863
438 438 694 742 863 937 937 937 937 937 937
如何取消不必要的循环？

6.2冒泡排序(cont.)

- 算法的实现：

```
void  BubbleSort ( int  n , LIST  &A )//内外层循环优化
{   for ( int i =1; i <= n-1; i++ ){ //一共要排序n-1趟
        int sp = 0; //每趟排序标志位都要先置为0，判断内层循环是否发生了交换
        for ( int j =n; j >i; j--) {//选出该趟排序的最小值前移；内层循环已优化
            if ( A[j].key < A[j-1].key ){
                swap (A[j], A[j-1]);
                sp = 1; //只要有发生交换，sp就置为1
            }
        }
        if (sp ==0){ //若标志位为0，说明所有元素已经有序，就直接返回
            return;
        }
    }
}
```

```
void  swap(records &x,  records &y)
{   records   t ;
    t = x ;     x = y ;     y = t ;
}
```

6.2冒泡排序(cont.)

- 算法（时间）性能分析：
  - 最好情况（正序）：
    - 比较次数：n-1
    - 移动次数：0
    - 时间复杂度：$O(n)$;
  - 最坏情况（反序）：
    - 比较次数：$\sum_{i=1}^{n-1}(n-i) = \frac{n(n-1)}{2}$
    - 移动次数：$\sum_{i=1}^{n-1}3(n-i) = \frac{3n(n-1)}{2}$
    - 时间复杂度：$O(n^2)$;
  - 平均情况：时间复杂度为$O(n^2)$。

空间复杂度：$O(1)$。
稳定性：稳定

6.3 快速排序

- 快速算法是对气泡排序的改进，改进的着眼点：
  - 在气泡排序中，记录的比较和移动是在相邻单元中进行的，记录每次交换只能上移或下移一个单元，因而总的比较次数和移动次数较多。

减少总的比较次数和移动次数

↓

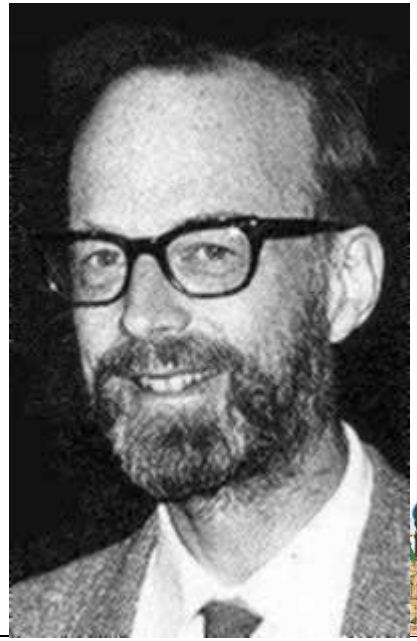
增大记录的比较和移动距离

↓

较大记录从前面直接移动到后面
较小记录从后面直接移动到前面

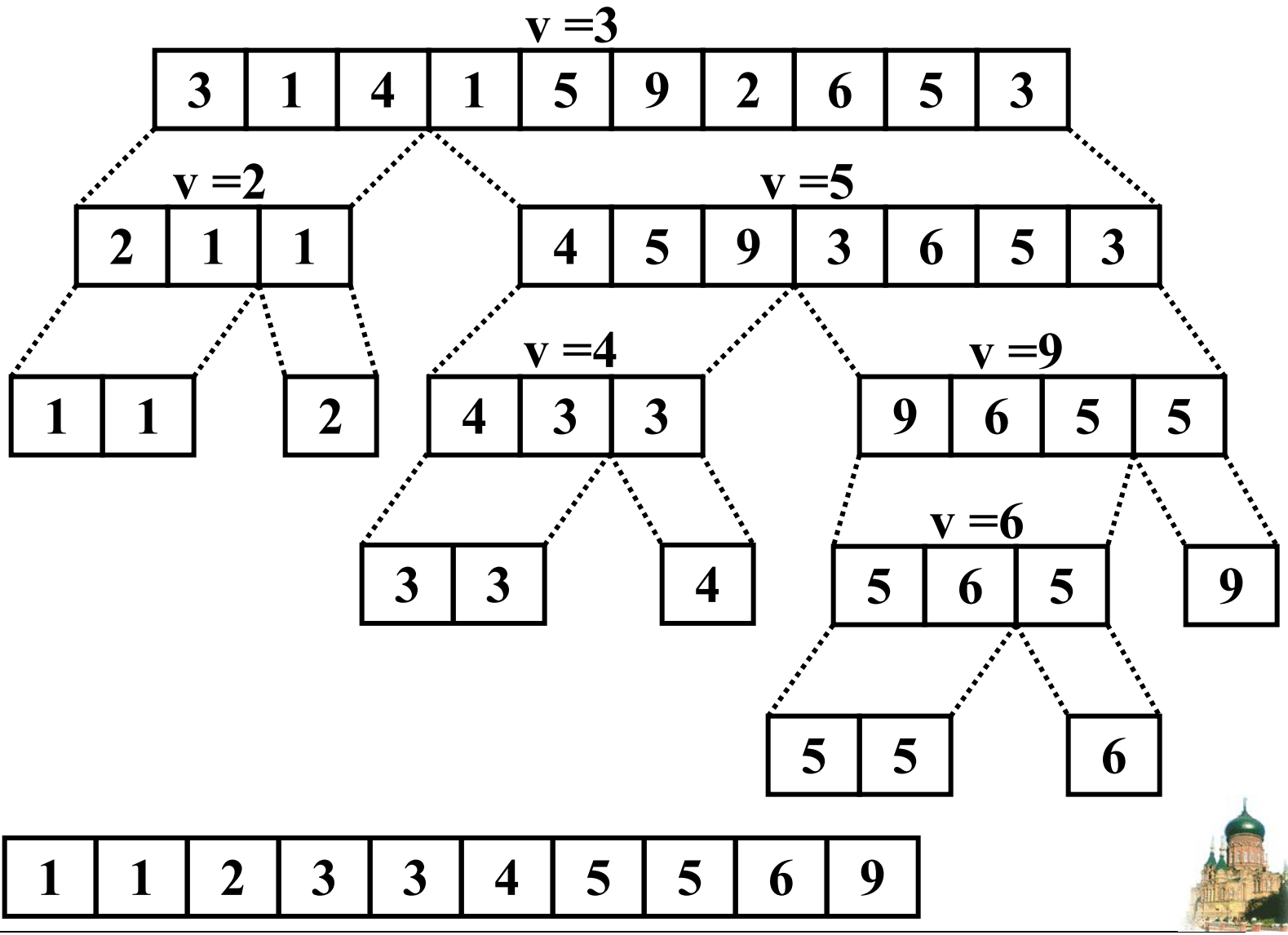
6.3 快速排序（cont.）

- 算法的基本思想：
  - 是C.R.A.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)，以减少排序过程之中的比较次数。
  - 通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的所有数据都比另外一部分的所有数据都要小，然后再按此方法对这两部分数据分别进行快速排序，整个排序过程可以递归进行，以此达到整个数据变成有序序列。
- 分治法的基本思想
  - 分解(划分): 将原问题分解为若干个与原问题相似的子问题；
  - 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。



6.3 快速排序（cont.）

- 示例



6.3 快速排序（cont.）

- 算法的实现步骤：

设被排序的无序区为A[*i*],……,A[*j*]

1. 基准元素选取：选择其中的一个记录的关键字*v* 作为基准元素(控制关键字);(怎么选择？)
2. 划分：通过基准元素*v* 把无序区A[*i*],……,A[*j*]划分成左、右两部分，使得左边的各记录的关键字都小于*v* ；右边的各记录的关键字都大于等于*v* ；(如何划分？)
3. 递归求解：重复(1)~(2)，分别对左边和右边部分递归地进行快速排序；
4. 组合：左、右两部分均有序，整个序列有序。

3	1	4	1	5	9	2	6	5	3

6.3 快速排序（cont.）

3	1	4	1	5	9	2	6	5	3

- 基准元素的选取：
  - 基准元素的选取是任意的，但不同的选取方法对算法性能影响很大；
  - 一般原则：是每次都能将表划分为规模相等的两部分（最佳情况）。此时，划分次数为$\log_2 n$，全部比较次数$n\log_2 n$，交换次数$(n/6)\log_2 n$ 。
  - 设 FindPivot(*i* , *j*)是求A[*i*].key,…,A[*j*].key的基准元素*v*=A[*k*]，返回其下标*k* 。
    - *v* =(A[*i*].key， A[(*i*+*j*)/2].key，A[*j*].key的中值)
    - *v* =从A[*i*].key到A[*j*].key 最先找到的两个不同关键字中的最大者。（若A[*i*].key,…A[*j*].key之中至少有两个关键字不相同）优点：若无两个关键字不同，则*A*[*i*]到A[*j*]已有序，排序结束。

6.3 快速排序（cont.）

```
int   FindPivot( int  i, int  j ) /* 设A是外部数组 */
/*若A[i],…A[j]的关键字全部相同，则返回0；
否则，左边两个不同关键字中的较大者的下标。*/
{    keytype   firstkey = A[i].key ; /* 第1个关键字的值A[i].key */
     int  k ;                    /*  从左到右查找不同的关键字  */
     for ( k=i+1 ;  k<=j; k++ ) /* 扫描不同的关键字 */
          if ( A[k].key > firstkey ) /* 选择较大的关键字 */
                    return  k ;
          else if ( A[k].key < firstkey )
                    return  i ;
     return  0 ;
}
```

3	1	4	1	5	9	2	6	5	3

6.3 快速排序（cont.）

- 无序区划分（分割）：

设被排序的无序区为A[*i*],……,A[*j*]

3	1	4	1	5	9	2	6	5	3

(1) 扫描：

- 令游标 *l* 从左端(初始时*l* = *i*)开始向右扫描，越过关键字小于*v* 的所有记录，直到遇到A[*l*].key≥*v*；
- 又令游标 *r* 从右端(初始时 *r* = *j*)开始向左扫描，越过关键字大于等于*v* 的所有记录，直到遇到A[*r*].key<*v*；

(2) 测试*l* 和 *r*：若*l* < *r*，则转(3)；否则(*l* > *r*, 即 *l*= *r* +1)转(4)；

(3) 交换：交换A[*l*]和A[*r*]，转(1)；(目的是使 *l* 和 *r* 都至少向其前进方向前进一步)

(4)此时A[*i*],…A[*j*]被划分成为满足条件的两部分A[*i*],…A[*l* -1]和A[*l*],…,A[*j*]。

6.3 快速排序（cont.）

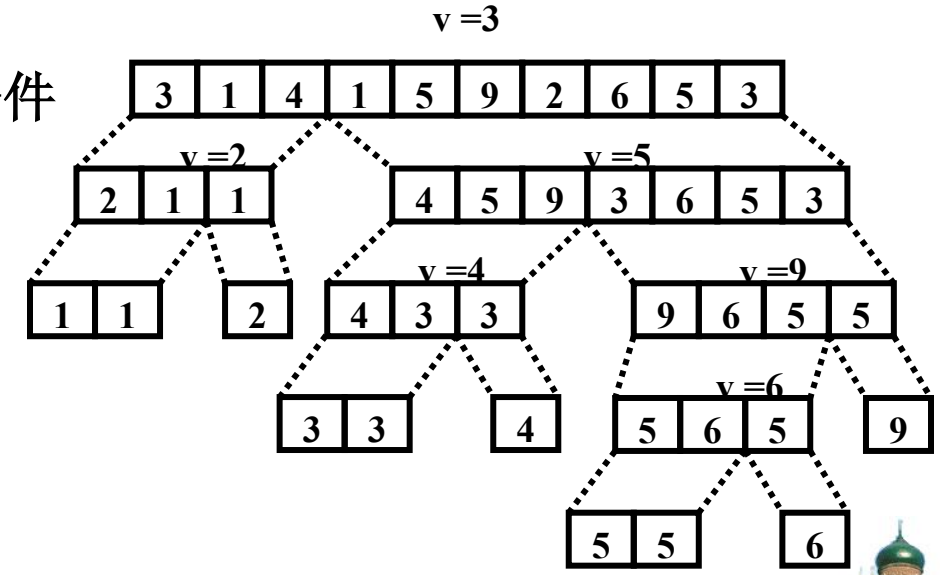
```
int Partition ( int i , int j , keytype pivot )
/*划分A[i],…,A[j]，是关键字 < pivot 的在左子序列，
  关键字≥pivot 的在右子序列，返回有子序列的起始下标*/
{   int l , r ;
    do{
        for( l = i ; A[l].key  <  pivot ; l++ ) ;
        for( r = j ; A[r].key >= pivot ; r--)  ;
        if( l < r )  swap(A[l],A[r]);
    } while( l <= r );
    return   l ;
}
```

3	1	4	1	5	9	2	6	5	3

6.3 快速排序（cont.）

- 快速排序的实现：/*对外部数组A 的元素A[i],…,A[j]进行快速排序*/

```
void QuickSort ( int i , int  j )
{    keytype pivot;
     int k; //关键字大于等于pivot的记录在序列中的起始下标
     int pivotindex ; //关键字为pivot的记录在数组A中的下标
     pivotindex = FindPivot ( i , j );
     if( pivotindex != 0 ) {  //递归终止条件
          pivot=A[pivotindex].key;
          k=Partition ( i , j , pivot );
          QuickSort ( i , k-1);
          QuickSort ( k  , j );
     }
} //对数组A[1],…,A[n]进行快速排序可调用QuickSort(1,n)实现
```



6.3 快速排序（cont.）

快速排序（时间）性能分析

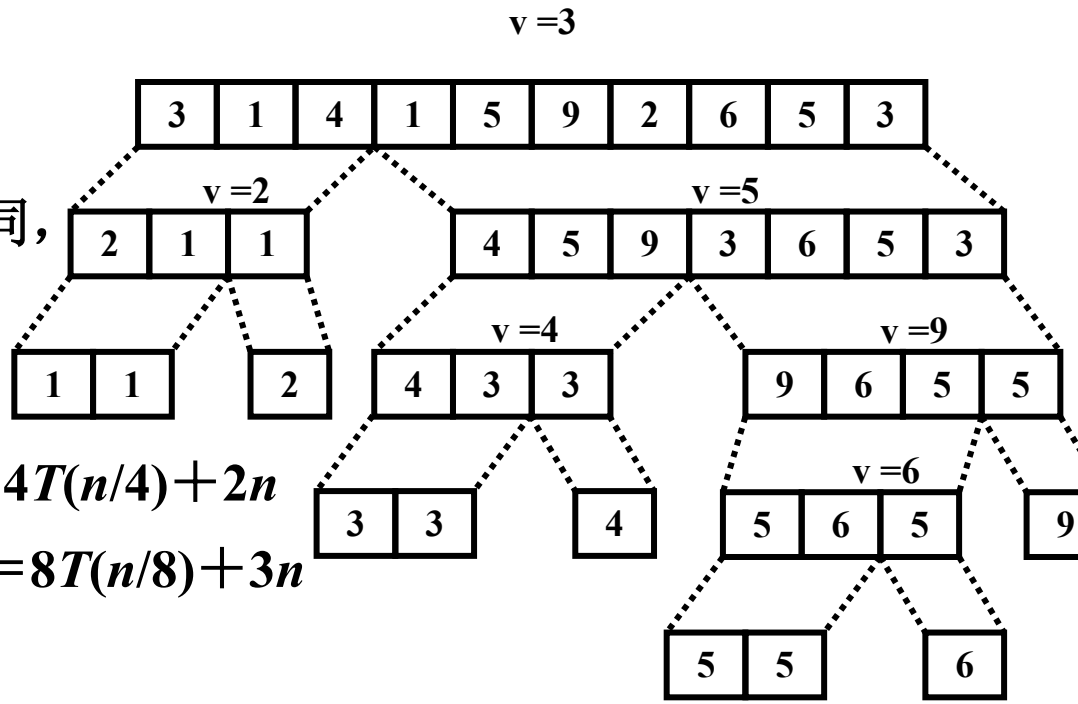
- 最好情况：
  - 每一次划分后，划分点的左侧子表与右侧子表的长度相同，则有，为$O(n\log_2 n)$。
  - $T(n)\leq 2T(n/2)+n$

$$\leq 2(2T(n/4)+n/2)+n=4T(n/4)+2n$$

$$\leq 4(2T(n/8)+n/4)+2n=8T(n/8)+3n$$

$$\cdots\cdots\cdots$$

$$\leq nT(1)+n\log_2 n=O(n\log_2 n)$$



  - 时间复杂度为$O(n\log_2 n)$
  - 空间复杂度为$O(\log_2 n)$

6.3 快速排序（cont.）

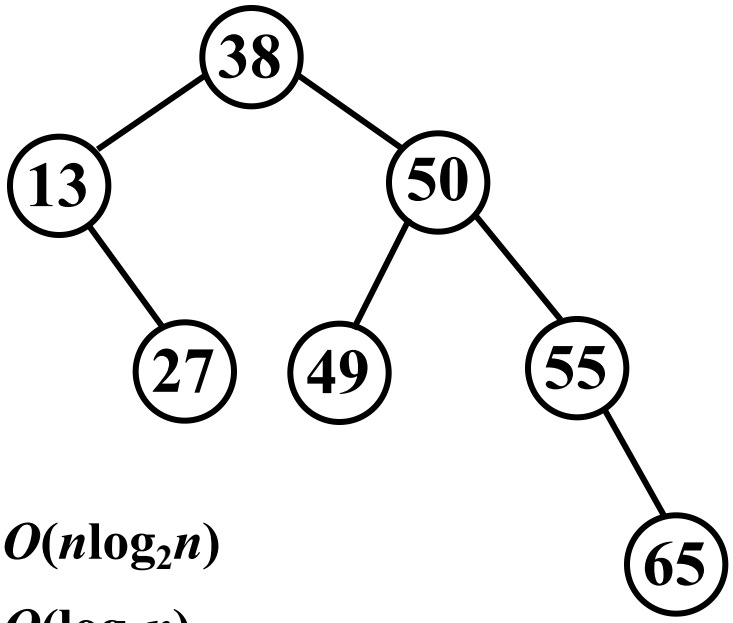
- 快速排序（时间）性能分析
  - 最坏情况：
    - 每次划分只得到一个比上一次划分少一个记录的子序列（另一个子序列长度为1），则有

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

    - 时间复杂度为$O(n^2)$
    - 空间复杂度为$O(n)$

6.3 快速排序（cont.）

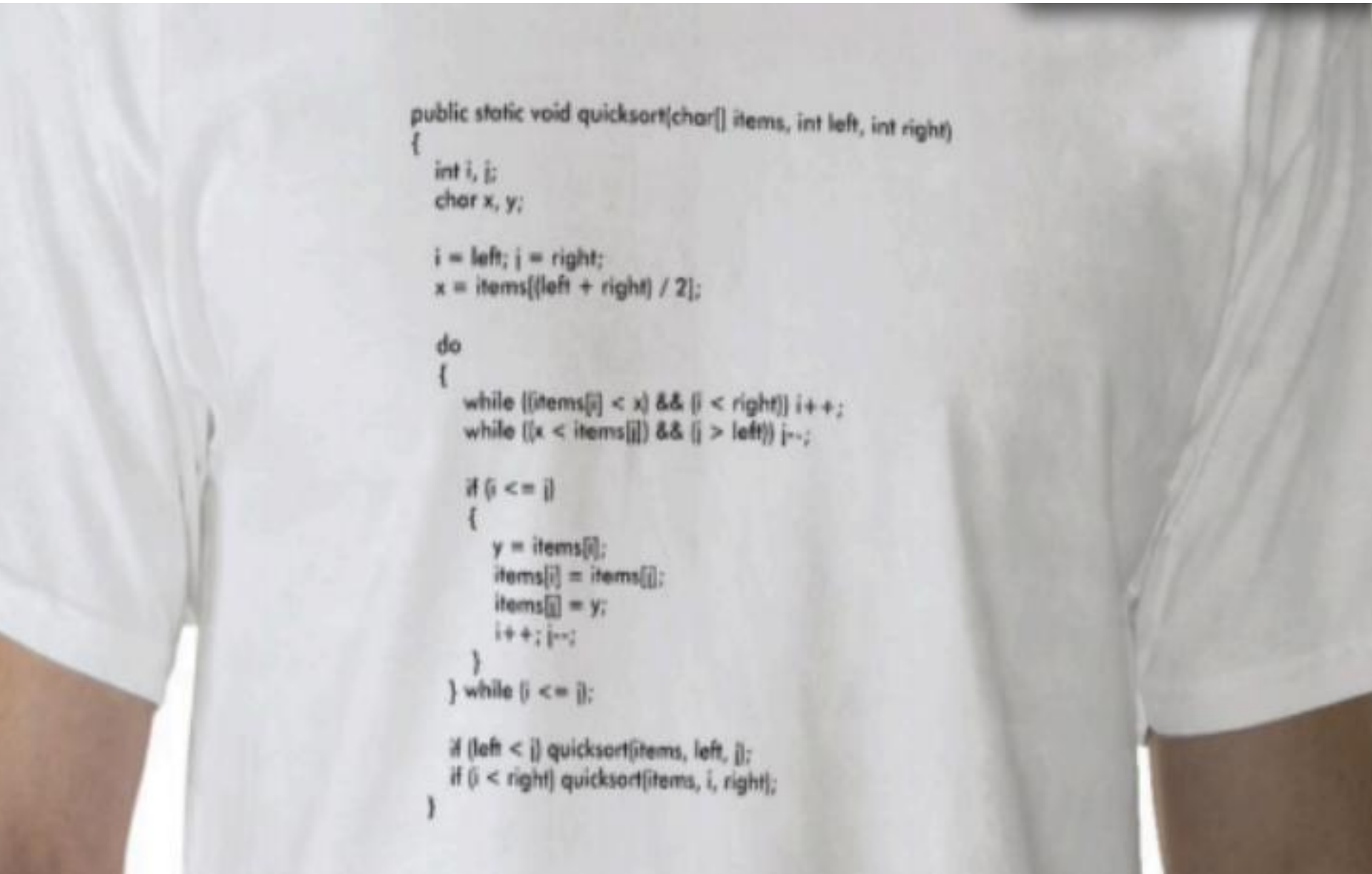
- 快速排序（时间）性能分析
  - 平均情况：
    - 快速排序的递归执行过程可以用递归树描述。
    - 例如，序列 {38, 27, 55, 50, 13, 49, 65}的快速排序递归树如下：



    - 时间复杂度为$O(n\log_2 n)$
    - 空间复杂度为$O(\log_2 n)$

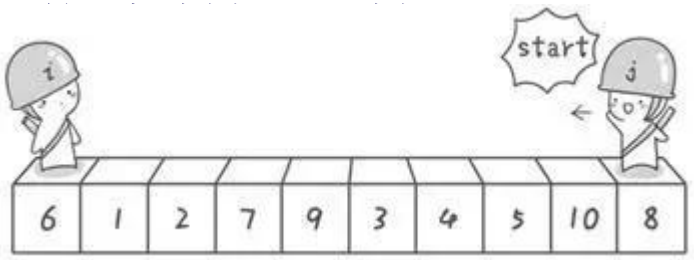
稳定性：不稳定（2 2’ 1）

Quicksort t-shirt

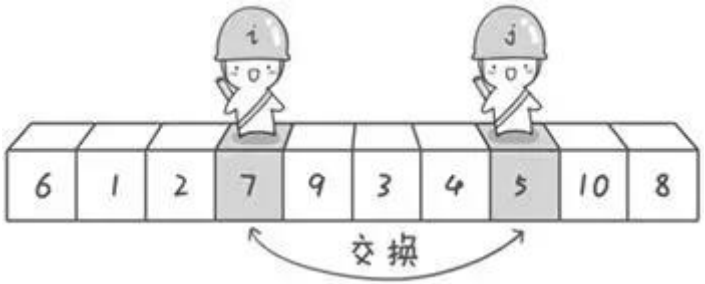


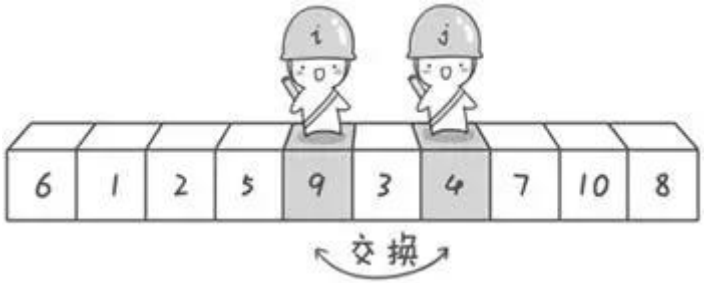
Quicksort

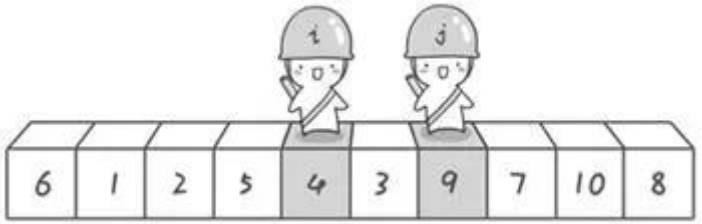
如何使得基准元素在边界处？

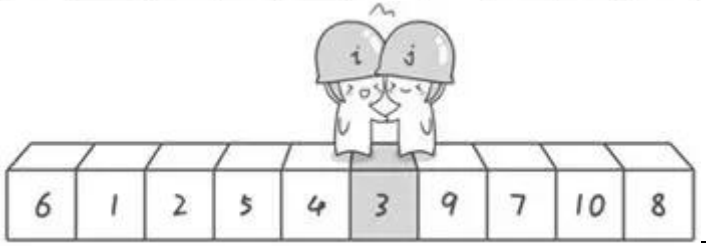


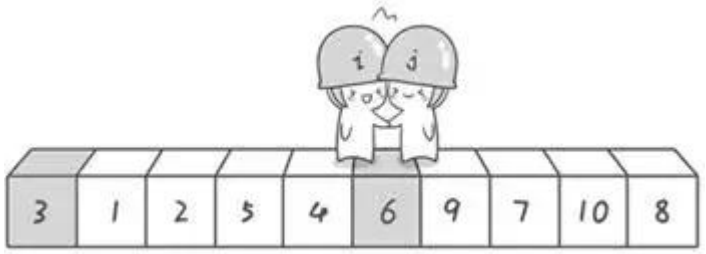
Pivot放在序列的第1个元素

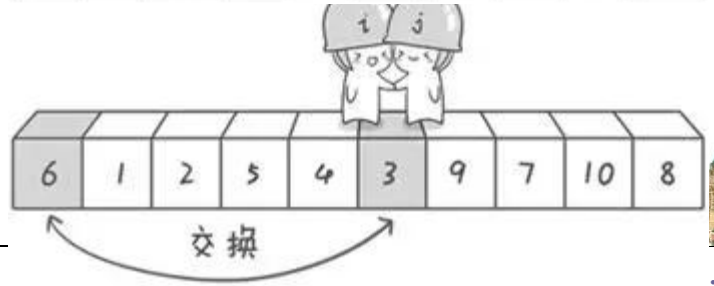












练习题：对给定的序号 j（1＜j＜n），要求在无序记录A[1]~A[n]中找到按关键码从小到大排在第 j 位上的记录，试利用快速排序的划分思想设计算法实现上述查找。

6.4 直接选择排序

- 算法的基本思想
  - 选择排序的主要操作是选择，其主要思想是：每趟排序在当前待排序序列中选出关键字值最小（最大）的记录，添加到有序序列中。
  - 直接选择排序，对待排序的记录序列进行n-1遍的处理，第1遍处理是将A[1…n]中最小者与A[1]交换位置，第2遍处理是将A[2…n]中最小者与A[2]交换位置，……，第i遍处理是将A[i…n]中最小者与A[i]交换位置。这样，经过i遍处理之后，前i个记录的位置就已经按从小到大的顺序排列好了。
  - 直接选择排序与气泡排序的区别在：气泡排序每次比较后，如果发现顺序不对立即进行交换，而选择排序不立即进行交换，而是找出最小关键字记录后再进行交换。

6.4 直接选择排序

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

直接选择排序与气泡排序的区别在：气泡排序每次比较后，如果发现顺序不对立即进行交换，而选择排序不立即进行交换，而是找出最小关键字记录后再进行交换。

6.4 直接选择排序（cont.）

算法的实现

```
void  SelectSort (int n, LIST A )
{    keytype lowkey;      //当前最小关键字
     int  i, j，  lowindex; //当前最小关键字的下标
     for(i=1; i<n; i++) {  //在A[i…n]中选择最小的关键字，与A[i]交换
          lowindex = i ;
          lowkey = A[i].key ;                    选择树排序（锦标赛排序），见第3章
          for ( j = i+1; j<=n; j++) //SelectMinKey( int i , List A)
               if (A[j].key<lowkey) { //用当前最小关键字与每个关键字比较
                    lowkey=A[j] ;
                    lowindex = j ;
               }
          if ( i != lowindex ) swap(A[i], A[lowindex]) ;//减少不必要交换
     }
}
```

性能分析

- 移动次数：
  - 最好情况（正序）：0次
  - 最坏情况：3(*n*−1)次

比较次数：

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

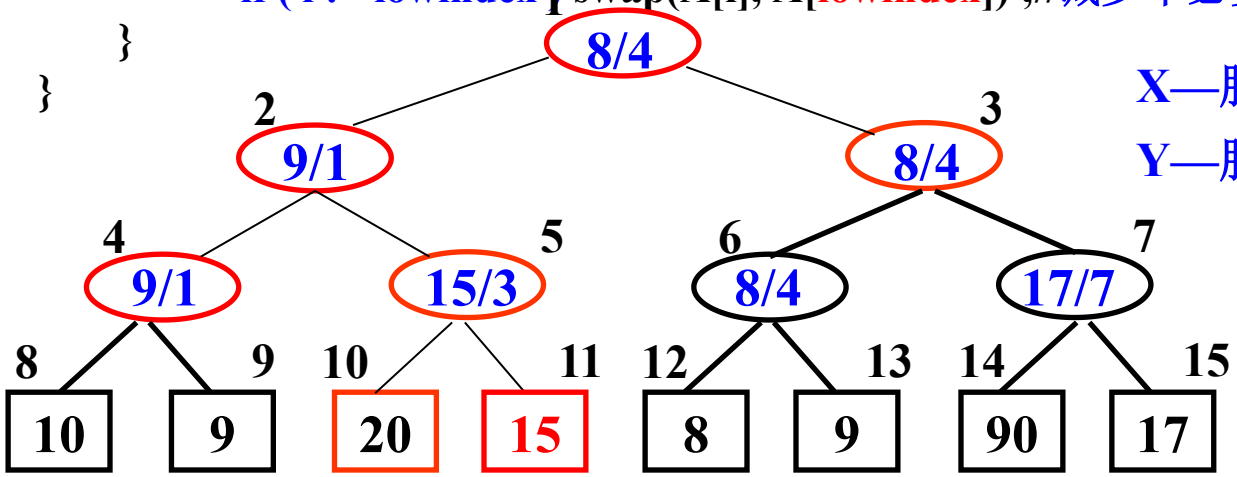
时间复杂度为$O(n^2)$。

空间复杂度为$O(1)$。　稳定性：否/2,2',1

6.4 直接选择排序（cont.）

算法的实现

```
void  SelectSort (int n, LIST A )
{     keytype lowkey;     //当前最小关键字
      int  i, j，lowindex; //当前最小关键字的下标
      for(i=1; i<n; i++) {  //在A[i…n]中选择最小的关键字，与A[i]交换
            lowindex = i ;
            lowkey = A[i].key ;
            for ( j = i+1; j<=n; j++) //SelectMinKey( int i , List A)
                if (A[j].key<lowkey) { //用当前最小关键字与每个关键字比较
                      lowkey=A[j] ;
                      lowindex = j ;
                }
            if ( i != lowindex ) swap(A[i], A[lowindex]) ;//减少不必要交换
      }
}
```



X—胜者的关键字

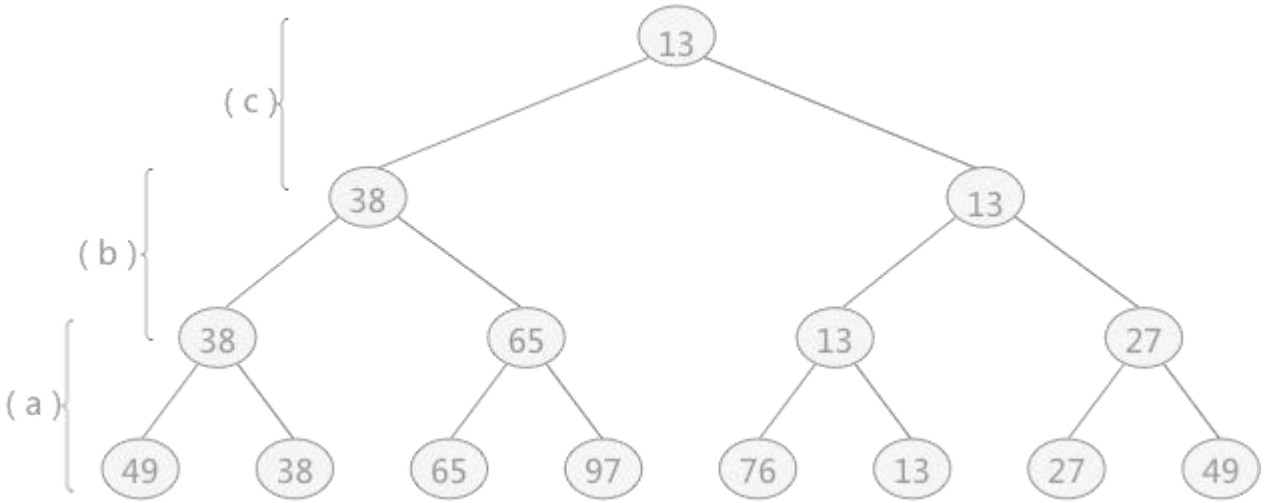
Y—胜者的下标

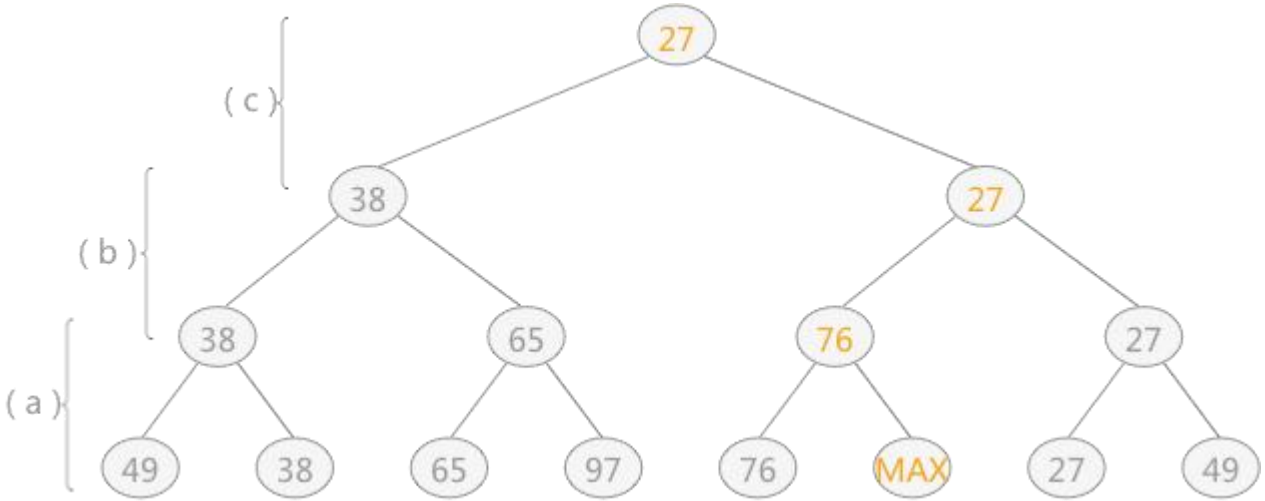
选择树排序：

时间复杂度为$O(nlogn)$。

空间复杂度为$O(n)$。

稳定排序


(c)
(b)
(a)
13
38
13
38
65
13
27
49
38
65
97
76
13
27
49


(c)
(b)
(a)
27
38
27
38
65
76
27
49
38
65
97
76
MAX
27
49

6.5 堆排序

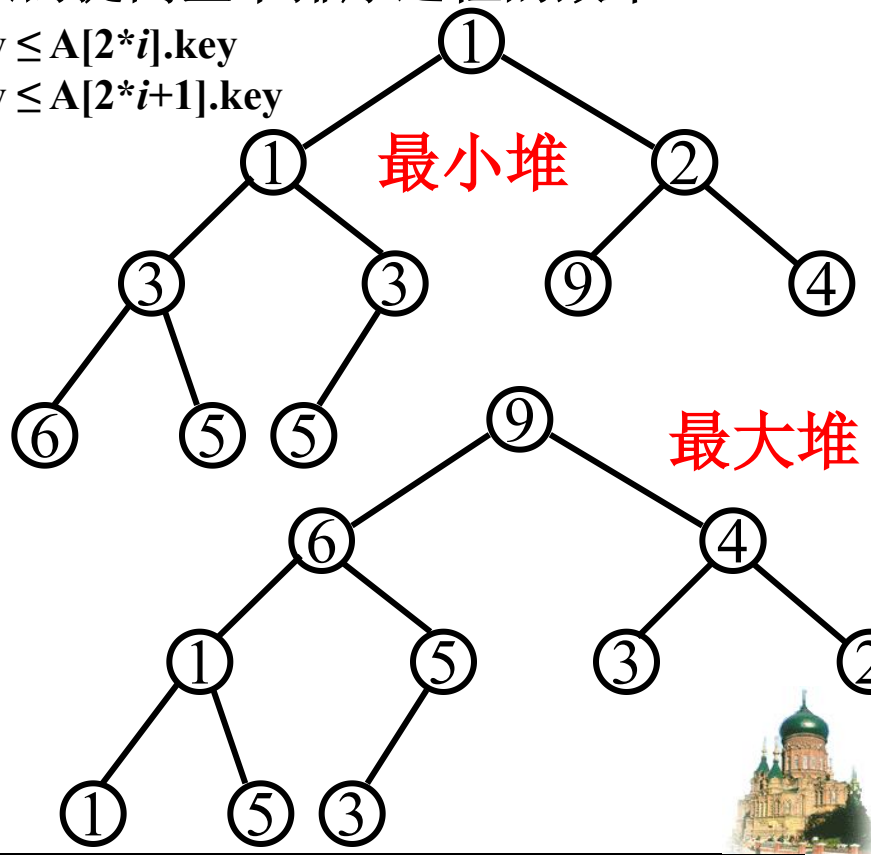
- 堆排序是对直接选择排序的改进，改进的着眼点：
  - 如何减少关键字之间的比较次数。若能利用每趟比较后的结果，也就是在找出关键字值最小记录的同时，也找出关键字值较小的记录，则可减少后面的选择中所用的比较次数，从而提高整个排序过程的效率。

减少关键字之间的比较次数

↓

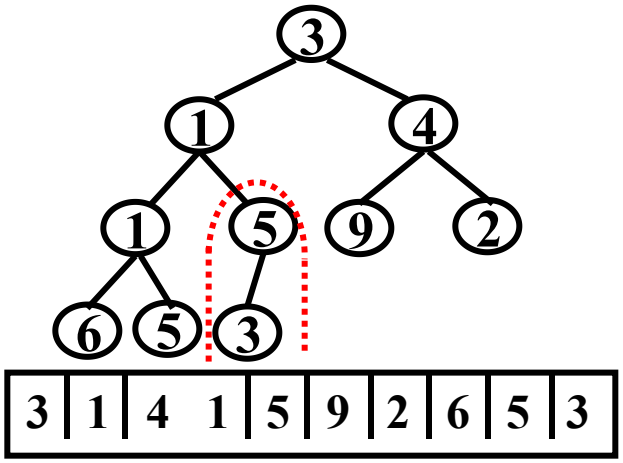
查找最小值的同时，找出较小值

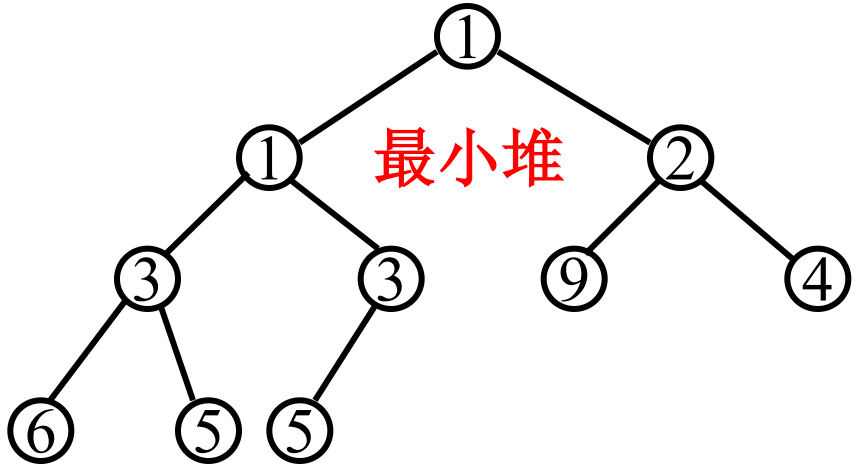
$A[i].key \le A[2*i].key$
$A[i].key \le A[2*i+1].key$



6.5 堆排序

- 堆排序是对直接选择排序的改进，改进的着眼点：
  - 如何减少关键字之间的比较次数。若能利用每趟比较后的结果，也就是在找出关键字值最小记录的同时，也找出关键字值较小的记录，则可减少后面的选择中所用的比较次数，从而提高整个排序过程的效率。





待排序的记录序列 → 表示 → 完全二叉树 → 整理 → 堆(初始建堆) → 排序 → 按关键字有序的记录序列

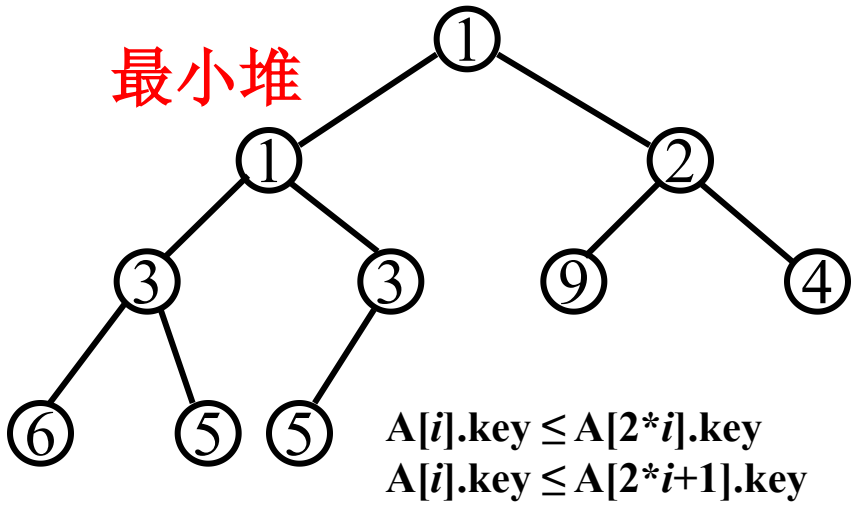
6.5 堆排序

- 堆排序是对直接选择排序的改进，改进的着眼点：
  - 如何减少关键字之间的比较次数。若能利用每趟比较后的结果，也就是在找出关键字值最小记录的同时，也找出关键字值较小的记录，则可减少后面的选择中所用的比较次数，从而提高整个排序过程的效率。

减少关键字之间的比较次数

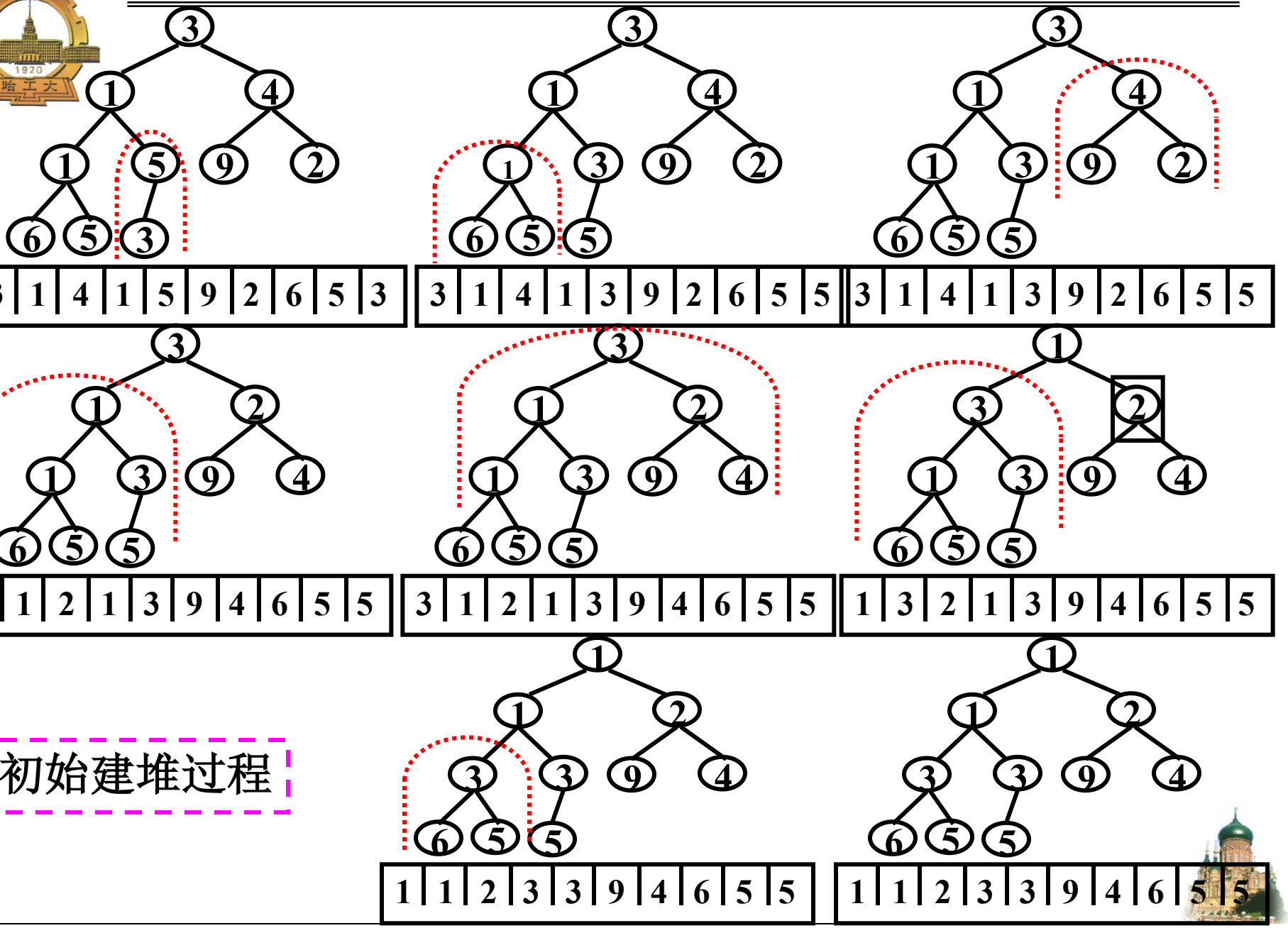
↓

查找最小值的同时，找出较小值



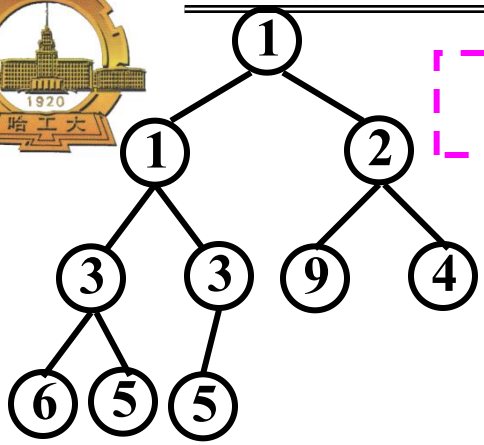
$A[i].key \le A[2*i].key$
$A[i].key \le A[2*i+1].key$

待排序的记录序列 → 表示 → 完全二叉树 → 整理 → 堆(初始建堆) → 排序 → 按关键字有序的记录序列

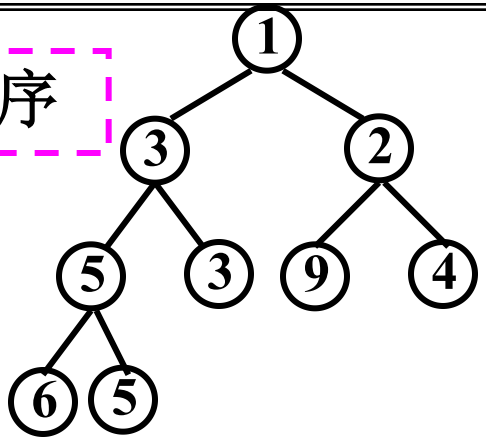

3 1 4 1 5 9 2 6 5 3
3 1 4 1 3 9 2 6 5 5
3 1 4 1 3 9 2 6 5 5
3 1 2 1 3 9 4 6 5 5
3 1 2 1 3 9 4 6 5 5
1 3 2 1 3 9 4 6 5 5
1 1 2 3 3 9 4 6 5 5
1 1 2 3 3 9 4 6 5 5

初始建堆过程

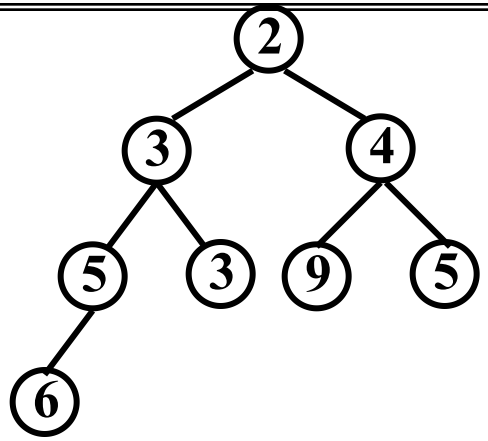
堆排序



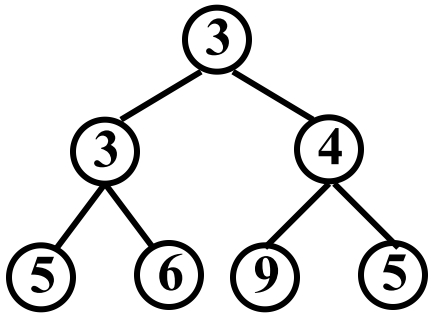
1	1	2	3	3	9	4	6	5	5



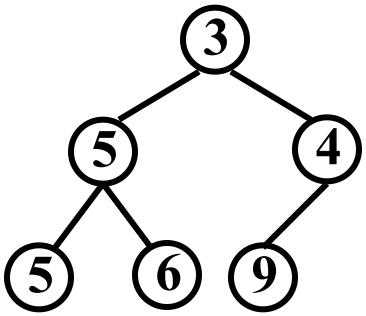
1	3	2	5	3	9	4	6	5	1



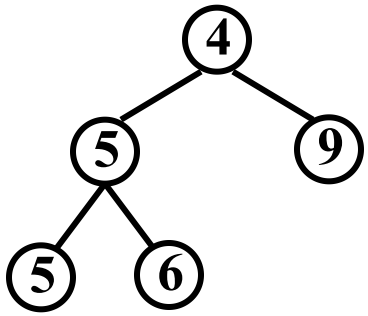
2	3	4	5	3	9	5	6	1	1



3	3	4	5	6	9	5	2	1	1

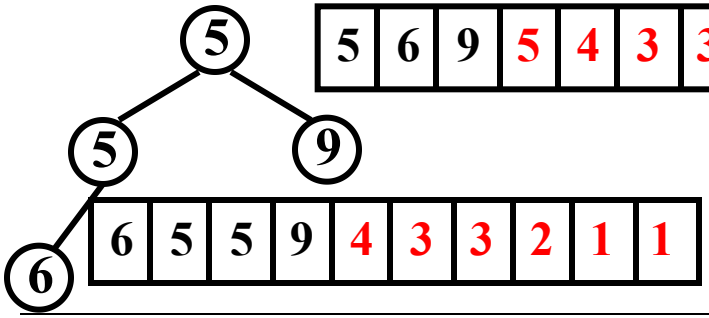


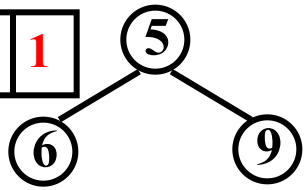
3	5	4	5	6	9	3	2	1	1

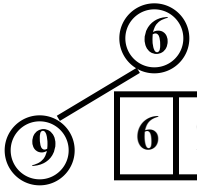


4	5	9	5	6	3	3	2	1	1

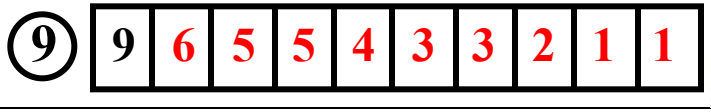
5	6	9	5	4	3	3	2	1	1



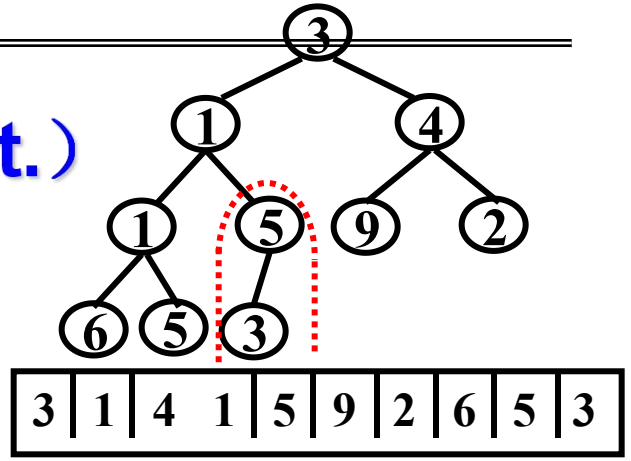




6	9	5	5	4	3	3	2	1	1

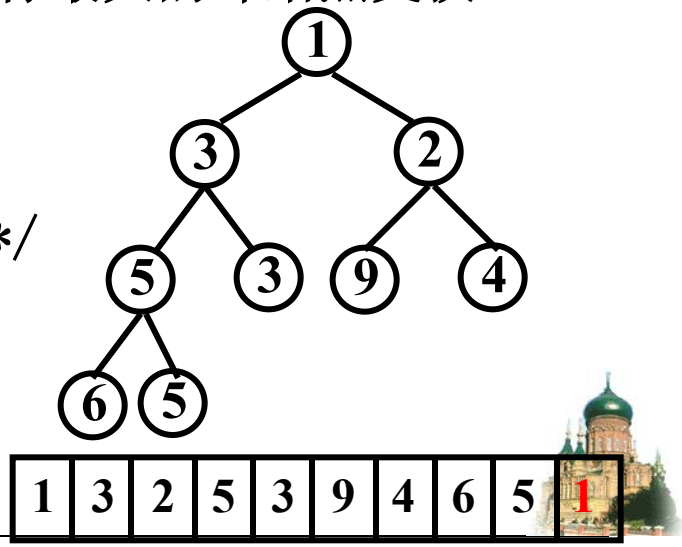


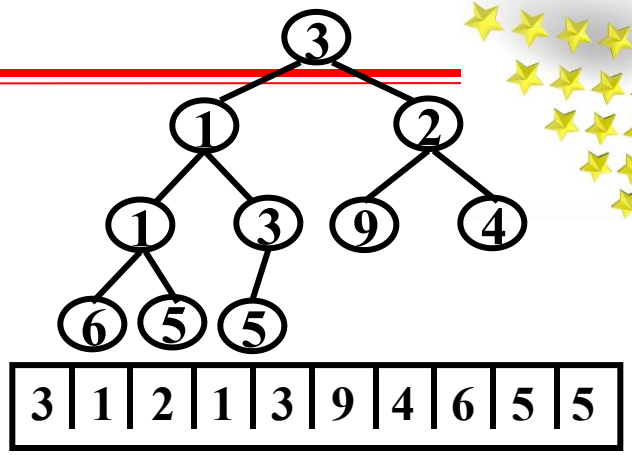
6.5 堆排序（cont.）



- 堆排序的实现：

```
void HeapSort ( int n , LIST A )
{    int i;
     for( i=n/2; i<=1; i--)  /*初始建堆，从最右非叶结点开始*/
          PushDown( i, n);  /*整理堆，把以i为根，最大下标的叶为n*/
    for( i=n; i<=2; i--) {
          swap(A[1],A[i]); //堆顶与当前堆中的下标最大的叶结点交换
          PushDown( 1, i-1 );
      /*整理堆把以1为根，
        最大叶下标为i-1的完全二元树整理成堆*/
    }
}
```






```
Void  pushdown( first, last )
{   int r ;
    r = first  ; /* r是被下推到的适当位置，初始值为根first*/
    while ( r <= last/2 )
      if ( ( r == last/2 ) && ( last%2 == 0 ) )//有度为1的节点，只能是l
        {  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 ;
}
```

6.5 堆排序（cont.）

算法性能分析

- 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 $O(\log(last/first))=O(\log_2 n)$。
- 所以，HeapSort时间复杂度： $O(n \log_2 n)$。
- 这是堆排序的最好、最坏和平均的时间代价。
- HeapSort空间复杂度： O(1)。
- 稳定性：否/1,2,2’(最小堆)

$$\frac{n}{2}\log_2 n + (n-1)\log_2 n$$

6.5 堆排序（cont.）

- 算法性能分析

排序方法	平均情况	最好情况	最坏情况	辅助空间
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$
直接选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
快速排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n^2)$	$O(\log_2 n)$ ~$O(n)$
堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$

6.6 （直接）插入排序

- 算法的基本思想：
  - 插入排序的主要操作是插入，其基本思想是：每次将一个待排序的记录按其关键字的大小插入到一个已经排好序的有序序列中，直到全部记录排好序为止。

6.6 （直接）插入排序

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

6.6 （直接）插入排序（cont.）

算法的实现

```
void  InsertSort (int n, LIST &A )
{   int  i, j ;
    A[0].key = -∞ ;//哨兵
    for(i=2; i<=n; i++) {
        j=i;
        while(A[j].key<A[j-1].key) {
            swap(A[j],A[j-1]) ;
            j=j-1;
        }
    }
} //不必检查当前位置是否为1
```

算法的性能分析

- 最好情况下（正序）：
  - 比较次数：n
  - 移动次数：0
  - 时间复杂度为$O(n)$。 $\sum_{i=2}^{n} i = \frac{(n+2)(n-1)}{2}$
- 最坏情况下（反序）：
  - 比较次数：
  - 移动次数： $\sum_{i=2}^{n}(i\text{-}1) = \frac{(n+4)(n-1)}{2}$
  - 时间复杂度为$O(n^2)$。
- 平均情况下（随机排列） $\sum_{i=2}^{n} i/2 = \frac{(n+2)(n-1)}{4}$
  - 比较次数：
  - 移动次数：
  - 时间复杂度为$O(n^2)$。 $\sum_{i=2}^{n}(i\text{-}1)/2 = \frac{(n+4)(n-1)}{4}$

稳定性：是

6.6 （直接）插入排序（cont.）

- 算法的性能分析
  - 空间复杂度：O(1)
  - 直接插入排序算法简单、容易实现，适用于待排序记录基本有序或待排序记录较小时。
  - 当待排序的记录个数较多时，大量的比较和移动操作使直接插入排序算法的效率降低。
- 改进的直接插入排序-----折半插入排序
  - 直接插入排序，在插入第 i（$i>1$）个记录时，前面的 i-1 个记录已经排好序，则在寻找插入位置时，可以用折半查找来代替顺序查找，从而较少比较次数。
  - 请大家自己写出这个改进的直接插入排序算法，并分析时间性能。

第8趟： (076 129 265 301 694 742 751 863 937) 438

6.7 希尔排序----分组插入排序

增量递减排序

Invented by Donald Shell in 1959

- 希尔排序是对直接插入排序的改进，改进的着眼点：
  - 若待排序记录按关键字值基本有序时，直接插入排序的效率可以大大提高；+
  - 由于直接插入排序算法简单，则在待排序记录数量*n*较小时效率也很高。
- 希尔排序的基本思想：
  - 将整个待排序记录分割成若干个子序列，在子序列内分别进行直接插入排序，待整个序列中的记录基本有序时，对全体记录进行直接插入排序。
- 需解决的关键问题？
  - 分组：应如何分割待排序记录，才能保证整个序列逐步向基本有序发展？
  - 组内直接插入排序：子序列内如何进行直接插入排序？

6.7 希尔排序----分组插入排序

增量递减排序

- 示例：缩小增量排序

Invented by Donald Shell in 1959

	1	2	3	4	5	6	7	8	9
初始序列	40	25	49	25*	16	21	08	30	13
d = 4	40	25	49	25*	16	21	08	30	13
	13	21	08	25*	16	25	49	30	40
d = 2	13	21	08	25*	16	25	49	30	40
	08	21	13	25*	16	25	40	30	49
d = 1	08	21	13	25*	16	25	40	30	49
	08	13	16	21	25*	25	30	40	49

6.7 希尔排序----分组插入排序(cont.)

- 算法的实现

```
void ShellSort(int n, LIST A)
{   int i, j, d;
    for (d=n/2; d>=1; d=d/2) {
        for (i=d+1; i<=n; i++) {
            A[0].key= A[i].key;
            j=i-d;
            while (j>0 && A[0].key< A[j].key) {
                A[j+d]= A[j];
                j=j-d;
            }
            A[j+d]= A[0];
        }
    }
}
```

直接插入排序算法的实现

```
void  InsertSort (int n, LIST A )
{  int  i, j ;
   A[0].key = -∞ ;//哨兵
   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;
      }
   }
} //不必检查当前位置是否为1
```

6.7 希尔排序----分组插入排序(cont.)

- 算法的实现

```
void ShellSort(int n, LIST A)
{   int i, j, d;
    for (d=n/2; d>=1; d=d/2) {
        for (i=d+1; i<=n; i++) {     //将A[i]插入到所属的子序列中
            A[0].key= A[i].key;      //暂存待插入记录
            j=i-d;                   //j指向所属子序列的最后一个记录
            while (j>0 && A[0].key< A[j].key) {
                A[j+d]= A[j];        //记录后移d个位置
                j=j-d;               //比较同一子序列的前一个记录
            }
            A[j+d]= A[0];
        }
    }
}
```

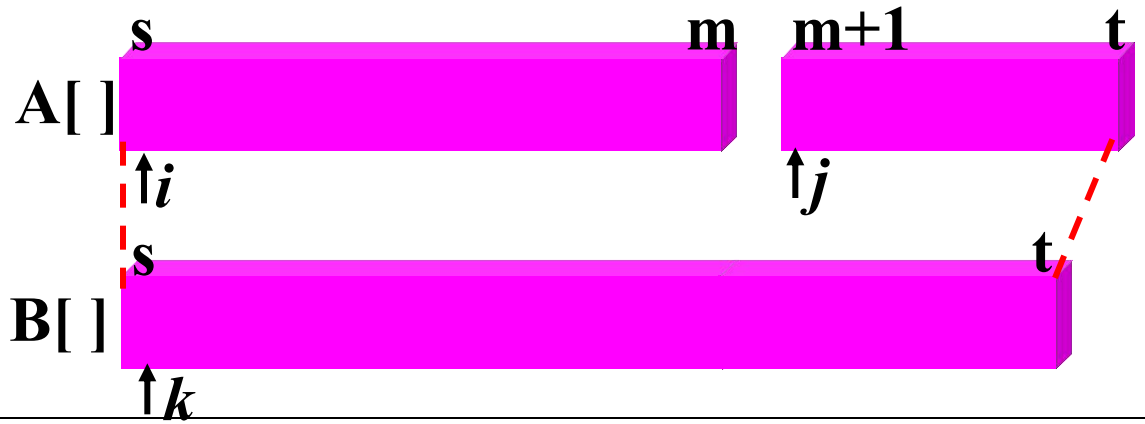
6.7 希尔排序----分组插入排序(cont.)

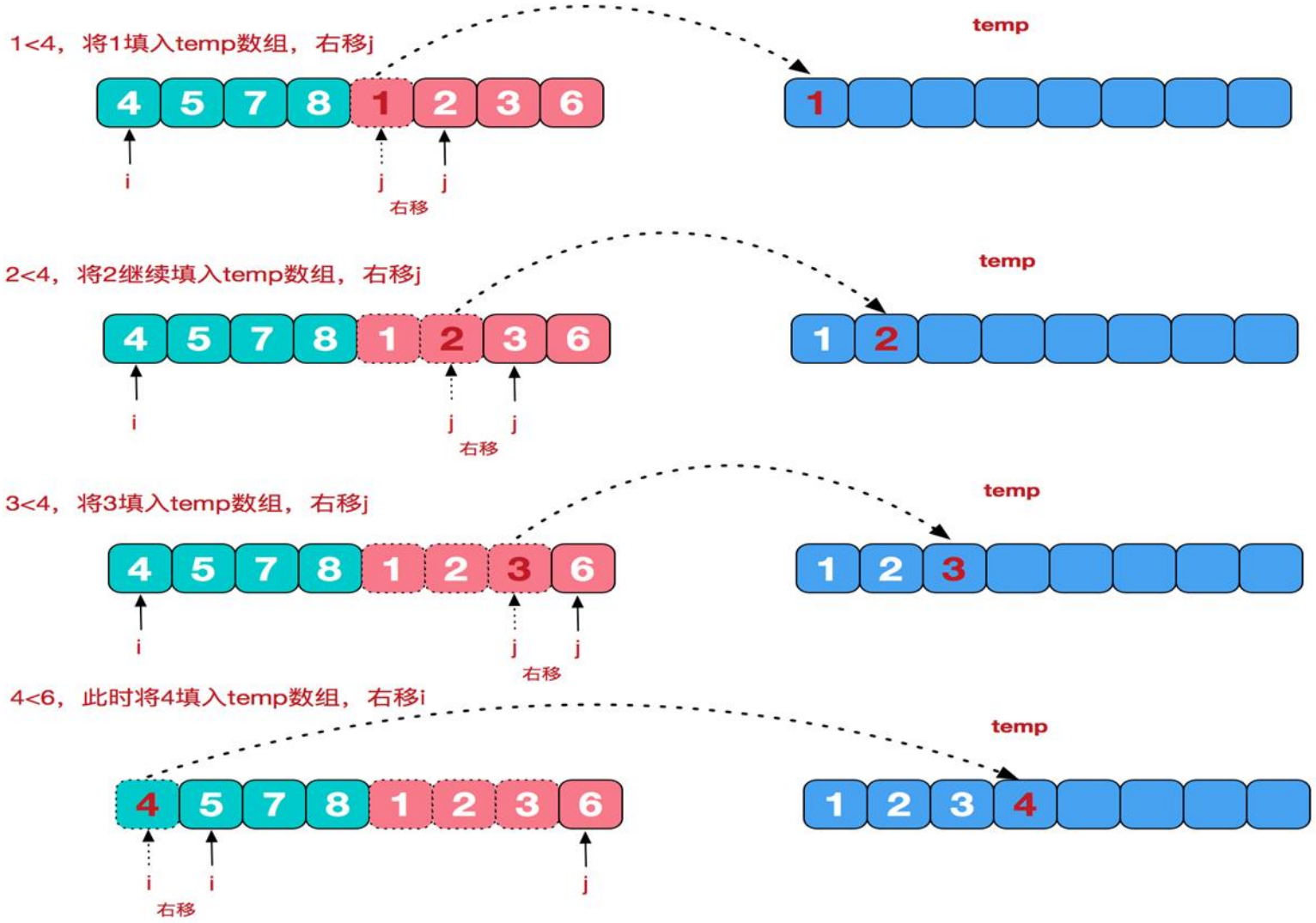
算法的性能分析

- 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。
- 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）。
- 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,...)
- 希尔排序的时间性能在$O(n^2)$和$O(nlog_2n)$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$O(n^{1.3})$。

6.8 （二路）归并排序

- 归并排序
  - 归并：将两个或两个以上的有序序列合并成一个有序序列的过程。
  - 归并排序的主要操作是归并，其主要思想是：将若干有序序列逐步归并，最终得到一个有序序列。
- 如何将两个有序序列合成一个有序序列？（二路归并基础）
  - 设相邻的按关键字有序序列为A[s] ~ A[m]和A[m+1] ~ A[t]，归并成一个按关键字有序序列B[s] ~ B[t]




1<4，将1填入temp数组，右移j
temp
4 5 7 8 1 2 3 6
1
i
j
j
右移
2<4，将2继续填入temp数组，右移j
temp
4 5 7 8 1 2 3 6
1 2
i
j
j
右移
3<4，将3填入temp数组，右移j
temp
4 5 7 8 1 2 3 6
1 2 3
i
j
j
右移
4<6，此时将4填入temp数组，右移i
temp
4 5 7 8 1 2 3 6
1 2 3 4
i
i
j
右移

6.8 （二路）归并排序（cont.）

```
void Merge (int s , int m , int t , LIST  A , LIST  B)
/*将有序序列A[s],…,A[m]和A[m+1],…,A[t]合并为一个有序序列 B[s],…,B[t]*/
{    int i = s ;  j = m+1 , k = s ;//置初值
       /* 两个序列非空时，取小者输出到B[k]上 */
     while ( i <= m && j <= t )
          B[k++] = ( A[ i ].key <= A[ j ].key) ? A[i++] : A[j++] ;
       /* 若第一个子序列非空(未处理完)，则复制剩余部分到B */
     while ( i <= m )    B[k++] = A[i++] ;
       /* 若第二个子序列非空(未处理完)，则复制剩余部分到B */
     while ( j <= t )     B[k++] = A[j++] ;
}/*时间复杂度：O( t－s＋1 );     空间复杂度：O( t－s＋1 ) */
```

6.8 （二路）归并排序（cont.）

- 二路归并排序的基本思想（自底向上的非递归算法）
  - 将一个具有*n*个待排序记录的序列看成是*n*个长度为1的有序序列；
  - 然后进行两两归并，得到⌈*n*/2⌉个长度为2的有序序列；
  - 再进行两两归并，得到⌈*n*/4⌉个长度为4的有序序列；
  - ……，
  - 直至得到1个长度为n的有序序列为止。

A[ ] {26} {5} {77} {1} {61} {11} {59} {15} {48} {19}

B[ ] { 5 26 } { 1 77 } { 11 61 } { 15 59 } { 19 48 } #

A[ ] { 1 5 26 77 } { 11 15 59 61 } { 19 48 } #

B[ ] { 1 5 11 15 26 59 61 77 } { 19 48 } *

A[ ] { 1 5 11 15 19 26 48 59 61 77 }

6.8 （二路）归并排序（cont.）

- 怎样完成一趟归并？

```
/*把A中长度为 h 的相邻序列归并成长度为 2h 的序列*/
void  MergePass (int n , int h , LIST A , LIST B)
{   int i , t ;
    for ( i=1 ; i+2*h-1<= n ; i+=2*h )
        Merge(i, i+h-1, i+2*h-1, A, B) ;//归并长度为h的两个有序子序列
    if ( i+h-1< n )   /* 尚有两个子序列，其中最后一个长度小于 h  */
        Merge( i, i+h-1, n , A, B) ;    /* 归并最后两个子序列 */
    else /* 若i<= n且i+h-1>= n 时，则剩余一个子序列轮空，直接复制 */
        for ( t= i ; t<= n ; t++ )
          B[t] = A[t] ;
} /* Mpass */
```

6.8 （二路）归并排序（cont.）

- （二路）归并排序算法：如何控制二路归并的结束？

```
void MergeSort ( int n , LIST A )
{   /* 二路归并排序 */
    int h = 1 ;/* 当前归并子序列的长度，初始为1 */
    LIST B ;
    while (h< n){
        MergePass (n , h , A , B) ;
        h = 2*h ;
        MergePass (n , h , B , A) ; /* A、B互换位置 */
        h = 2*h ;
    }
}/* MergeSort */
```

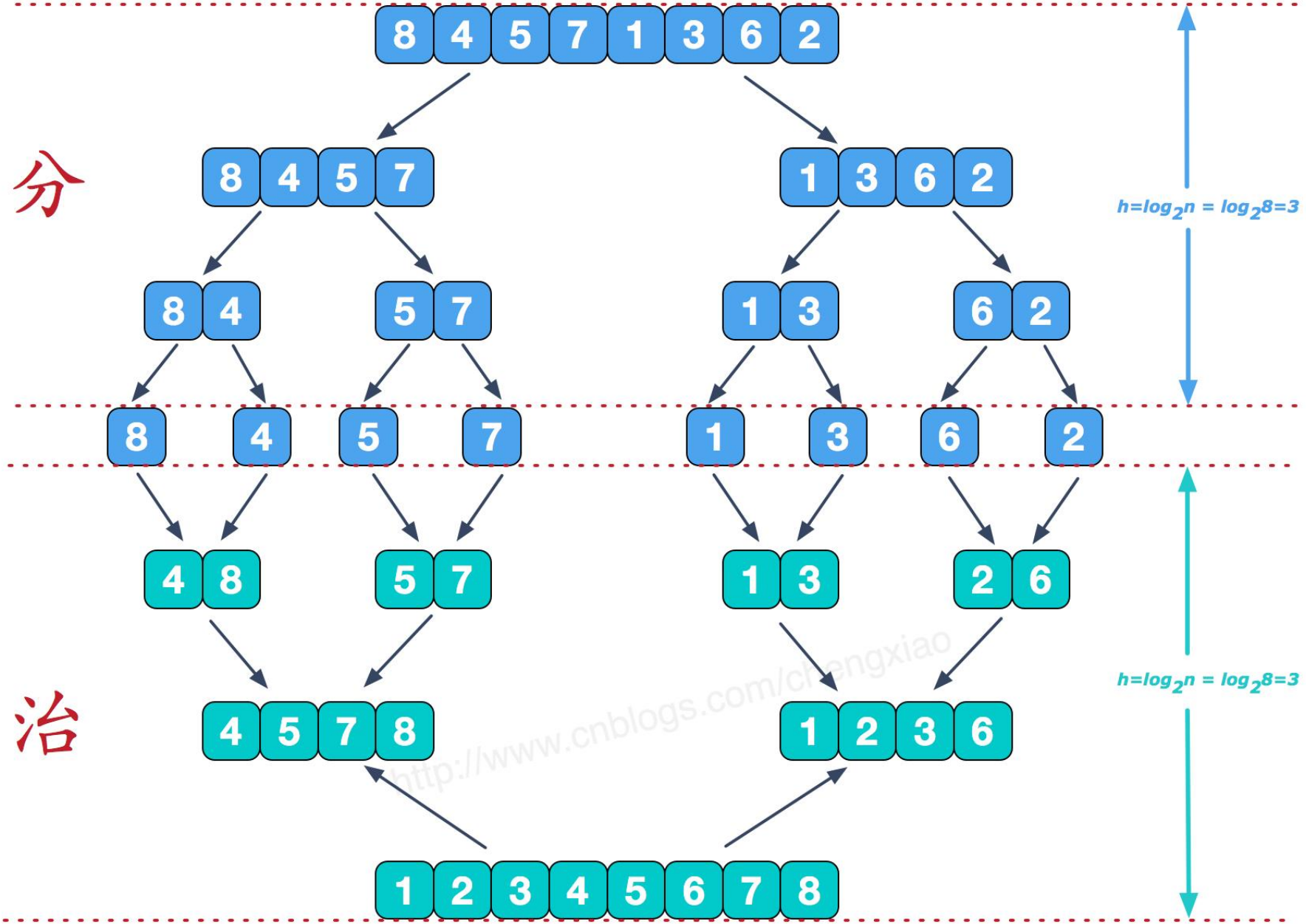
开始时,有序序列的长度h=1;结束时,有序序列的长度*h*=*n*,
用有序序列的长度来控制排序的结束.

6.8 （二路）归并排序（cont.）

- （二路）归并排序算法性能分析
  - 时间性能：
    - 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，并把结果存放到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$O(n\log_2 n)$。这是归并排序算法的最好、最坏、平均的时间性能。
  - 空间性能：
    - 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$O(n)$。
    - 辅助数组是一个共用的数组。如果在每个归并的过程中都申请一个临时数组会造成比较大的时间开销。
    - 归并的过程需要将元素复制到辅助数组，再从辅助数组排序复制回原数组，会拖慢排序速度。

6.8 （二路）归并排序（cont.）
自顶向下

- （二路）归并排序分治算法
  - 算法的基本思想
    - 分解：将当前待排序的序列A[low], …, A[high]一分为二，即求分裂点 mid = (low + high) / 2 ；
    - 求解：递归地对序列 A[low], …, A[mid]和A[mid+1], …, A[high]进行归并排序；
    - 组合：将两个已排序子序列归并为一个有序序列。
  - 递归的终止条件
    - 是子序列长度为 1，
      因为一个记录自然有序。
  - 算法实现



6.8 （二路）归并排序（cont.）

- （二路）归并排序分治算法
  - 算法实现

```
void MergeSort ( LIST A , LIST B , int low , int high )
/* 用分治法对A[low], …, A[high]进行二路归并排序 */
{   int mid = (low+high)/2 ;
    if (low<high){ /* 区间长度大于 1 ， high-low>0 */
        MergeSort (A , B , low , mid) ;
        MergeSort (A , B , mid+1 , high) ;
        Merge (low , mid , high , A , B) ;
    }
}/* MergeSort */
```

$T(n) = 2T(n/2) + n$

- 自底向上的归并算法效率较高，但可读性差。
- 若采用分治技术自顶向下的算法，形式简洁。

对A中所有记录排序调用函数：MergeSort(A,B,1,n)

练习1：给定两个有序的数组，求中位数

练习2： 假设两个有序数组 a 和 b，长度分别是 m 和 n，求第 k 小/大数。

练习3：归并排序最为经典的一个应用就是求一个整数序列中的逆序对的数目。
设a[1...n]是一个包含n个不同数的数组，若当i<j时，有a[i] >a[j]，则称(i,j)就是一个逆序对，现在要求设计一个O(nlog(n))的算法来求解数组a中的逆序对数。

比如{7, 5, 6, 4}中，一共有5个逆序对：{7,5}, {7,6}, {7,4}, {5,4}, {6,4}。

注意：当序列中存在2个数相等时也要认为是逆序，序列1，2，2中的逆序对为1个，而不是0个

练习1：给定两个有序的数组，求中位数

解法：给出两个有序数组，假设两个数组的长度和是 len，如果 len 为奇数，那么我们求的就是两个数组合并后的第（len >>1）+ 1 大的数，如果 len 为偶数，就是第（len >>1）和（len >>1）+ 1 两个数的平均数。

练习2： 假设两个有序数组 a 和 b，长度分别是 m 和 n，求第 k 小/大数。

解法：假设在 a 中取 x 个，那么 b 数组中取的个数也就确定了，为 k - x 个，据此可以将两个数组分别一分为二，根据两边的边界值判断此次划分是否合理。而对于 x 的值，可以用二分查找。

练习3：归并排序最为经典的一个应用就是求一个整数序列中的逆序对的数目。

设a[1...n]是一个包含n个不同数的数组，若当i<j时，有a[i] >a[j]，则称(i,j)就是一个逆序对，现在要求设计一个O(nlog(n))的算法来求解数组a中的逆序对数。

比如{7, 5, 6, 4}中，一共有5个逆序对：{7,5}, {7,6}, {7,4}, {5,4}, {6,4}。

注意：当序列中存在2个数相等时也要认为是逆序，序列1，2，2中的逆序对为1个，而不是0个

思路：逆序对是两个数之间关系的体现，所以最开始可以先看看两个相邻数字之间的关系：

比如{7, 5, 6, 4}中，{7,5}, {6,4}首先构成逆序对。找出这两对逆序对之后，两个相邻数是否构成逆序对的问题就解决了，接下来考虑4个数，则只需要考虑前两个数与后两个数之间的关系了，为避免重复统计，我们将两个子序列内部排好序。

序列变为{5,7,4,6}，第一个子序列中的元素大于第二个子序列中的元素，则构成逆序对。

性质：子序列是已排好序的，所以，第一个子序列中有一个元素m大于第二个子序列中的元素n，那么元素m后面的所有元素必然也将大于元素n。

具体做法：如果子序列的元素a <= 子序列的元素b，直接拿掉a，否则，a以及它后面的元素跟b构成逆序对，记录下来，然后b已经没有利用价值了，拿掉，继续比较……

6.9 基数排序----多关键字排序

- 理论上可以证明，对于基于关键字之间比较的排序，无论用什么方法都至少需要进行$\log_2 n!$次比较。
- 由Stirling公式可知，$\log_2 n! \approx n\log_2 n - 1.44n + O(\log_2 n)$。所以基于关键字比较的排序时间的下界是$O(n\log_2 n)$。因此不存在时间复杂性低于此下界的基于关键字比较的排序!
- 只有不通过关键字比较的排序方法,才有可能突破此下界。
- 基数排序（时间复杂性可达到线性级O(n)）
  - 不比较关键字的大小，而根据构成关键字的每个分量的值，排列记录顺序的方法，称为分配法排序（基数排序）。
  - 而把关键字各个分量所有可能的取值范围的最大值称为基数或桶或箱，因此基数排序又称为桶排序。

基数排序的适用范围：

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

  - 显然，要求关键字分量的取值范围必须是有限的，否则可能要无限的箱。

6.9 基数排序----多关键字排序（cont.）

- 算法示例：

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

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

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

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

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

6.9 基数排序----多关键字排序（cont.）

- 算法的基本思想
  - 设待排序的序列的关键字都是位相同的整数（不相同时,取位数的最大值），其位数为figure，每个关键字可以各自含有figure个分量，每个分量的值取值范围为0,1,…,9即基数为10。依次从低位考查每个分量
  - 首先把全部数据装入一个队列A，然后按下列步骤进行:

1.初态:设置10个队列，分别为Q[0],Q[1],…,Q[9]，并且均为空

2.分配:依次从队列中取出每个数据data；第pass遍处时，考查data.key右起第pass位数字，设其为r，把data插入队列Q[r]，取尽A，则全部数据被分配到Q[0],Q[1],…,Q[9].

3.收集:从Q[0]开始，依次取出Q[0],Q[1],…,Q[9]中的全部数据，并按照取出顺序，把每个数据插入排队A。

4.重复1,2,3步，对于关键字中有figure位数字的数据进行figure遍处理，即可得到按关键字有序的序列。

6.9 基数排序----多关键字排序（cont.）

- 算法实现：

```
void   RadixSort( 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++ )  /*置空队列*/
            MakeNull( Q[i] ) ;
      while ( !Empty( A ) ){/* 分配 */
          data = Front ( A ) ;
          DeQueue ( A );
          r = Radix(data.key, pass) ;
          EnQueue( data , Q[r] ) ; }
      for ( i=0 ; i <=9 ; i++ ) /*  收集 */
          while ( !Empty( Q[i] ) ) {
               data = Front ( Q[i] ) ;
               DeQueue( Q[i] ) ;
               EnQueue( data, A );}
   }
}
```

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

```
for (i=1;i<=9;i++) {
    Concatenate(Q[0], Q[i]);
     A=Q[0];
}/*大大缩短收集操作的时间*/
```

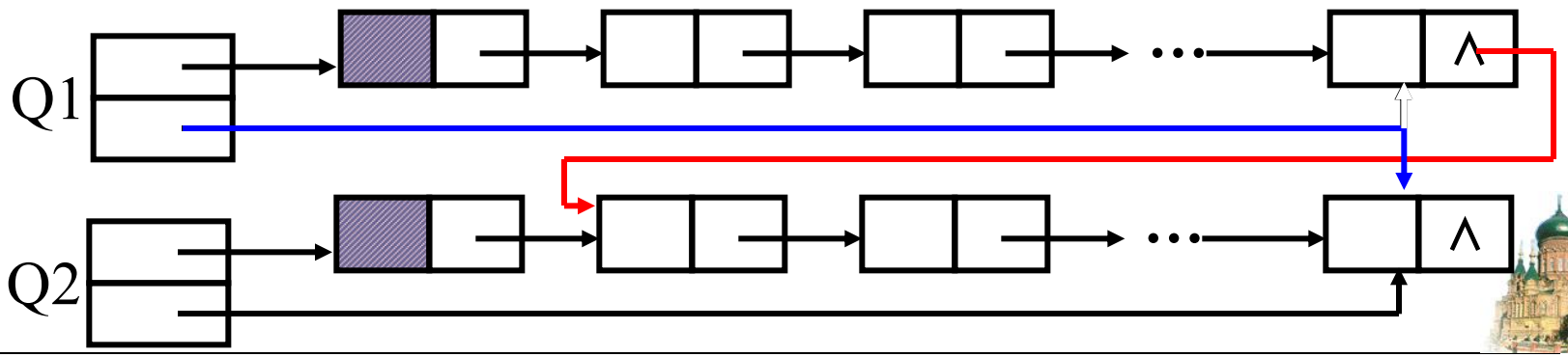
6.9 基数排序----多关键字排序（cont.）

- 算法的改进：
  - 由于每个桶（箱）存放多少个关键字分量相同的记录个数无法预料，即队列Q[0]，Q[1],...,Q[9]长度很难确定，故桶一般设计成链式排队，两个排队链在一起的方法如下：

```
void Concatenate(Queue Q1, Queue Q2)
{   if ( !Empty( Q2 ) ) {
        Q1.rear->next=Q2.front->next;
        Q1.rear=Q2.rear;
    }
}
```

```
//结点的类型：
struct celltype {
    ElemType data;
    celltype *next ;
} ;
```

```
队列的 类型：
struct QUEUE {
    celltype *front ;
    celltype *rear ;
};
```

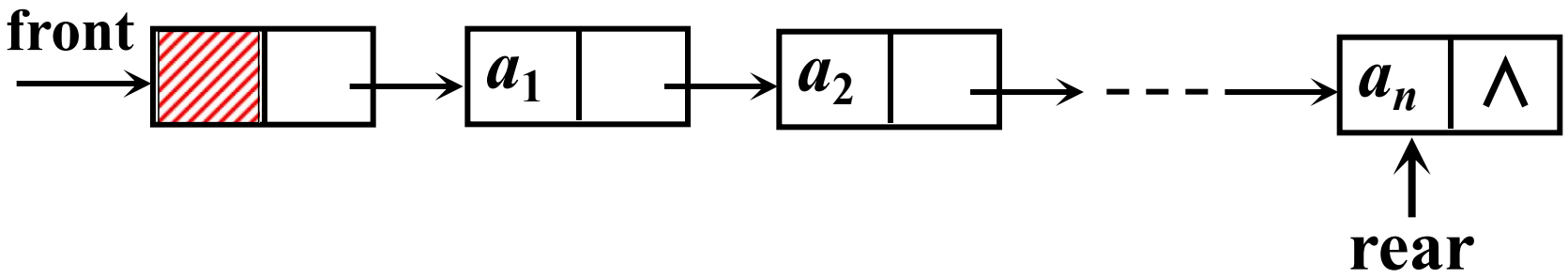


2.4.1 队列的指针实现(Cont.)

- 队列的链接存储结构及实现
  - 存储结构定义

```
//结点的类型：
struct  celltype  {
     ElemType  data;
      celltype  *next ;
} ;
```

```
队列的 类型：
struct  QUEUE {
      celltype  *front ;
      celltype  *rear ;
};
```



6.9 基数排序----多关键字排序（cont.）

- 算法实现：

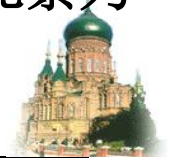
```
void   RadixSort( int figure, Queue &A)
{    Queue Q[10]; records  data ;
     int  pass, r, i ;
     for ( pass=1; pass<=figure ; pass++ ){
       for ( i=0 ; i<=9 ; i++ )  /*置空队列*/
            MakeNull( Q[i] ) ;
      while ( !Empty( A ) ){/* 分配 */
          data = Front ( A ) ;
          DeQueue ( A );
          r = Radix(data.key, pass) ;
          EnQueue( data , Q[r] ) ; }
      for (i=1;i<=9;i++) {
              Concatenate(Q[0], Q[i]);
               A=Q[0];
       }/*大大缩短收集操作的时间*/
   }
}
```

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

```
void Concatenate(Queue Q1, Queue Q2)
{    if ( !Empty( Q2 ) ) {
        Q1.rear->next=Q2.front->next;
        Q1.rear=Q2.rear;
    }
}
```

6.9 基数排序----多关键字排序（cont.）

- 算法性能分析
  - n----记录数，d----关键字（分量）个数，r----基数
  - 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))
  - 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
  - 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。
  - 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。
  - 当遇到负数的情况时可以将所有元素都加上一个常数使得所有元素为自然数后再排序，输出时再减去这个常数即可。



- 练习题：
  - 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

计数排序

- 假设：有n个数的集合，而且n个数的范围都在0~k(k = O(n))之间。

- 运行时间：Θ(n+k)

数组 A：

3	2	2	4	0	1	0	1

图 2.1

数组 C：

	0	1	2	3	4
C	2	2	2	1	1

图 2.2

	0	1	2	3	4
C	2	4	6	7	8

图 2.3

数组 B：

0	0	1	1	2	2	3	4

图 2.4

- 待排序数组A如图2.1所示，需要辅助数组B(存储最后排序结果)，数组C(存储元素的个数)。基于上述的假设，数组C的大小为k，C[i]表示数组A中元素i(0 <= i < k)的个数(如图2.2所示)，为了保证计数排序的稳定性，数组C变化为图2.3，C[i]表示小于或者等于i的个数。

```
// 输入：待排序数组A,存储排序后的数组B，数组A的大小，数组C的大小
// 功能：计数排序
void CountingSort(int A[], int B[], int len, int k)
{   int *CountArr = new int[k];
    int i;
    for (i = 0; i < k; i++)
      CountArr[i] = 0;
    for (i = 0; i < len; i++)
       CountArr[A[i]]++;
   for (i = 1; i < k; i++)
       CountArr[i] += CountArr[i-1];
    // 从右至左保证算法的稳定性
    for (i = len-1; i >=0; i--)
  { B[CountArr[A[i]]-1] = A[i];   CountArr[A[i]]--;   }
  }
```

所以总的运行时间为$\Theta(2(n+k)) = \Theta(n+k)$。

6.9 基数排序----多关键字排序（cont.）

- 算法示例：

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

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

890 210 321 901 432 012 123 543 765 986 987 018 678 098 789 019

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

本章小结--各种排序方法的比较

- 排序分类：
  - 按照排序时排序对象存放的设备，分为，
    - 内部排序:排序过程中数据对象全部在内存中的排序。
    - 外部排序:排序过程数据对象并非完全在内存中的排序
  - 按照排序是的基本操作是否基于关键字的比较？分为，
    - 基于比较：基本操作——关键字的比较和记录的移动，其最好时间下限已经被证明为$\Omega(n\log_2 n)$。
      - 交换排序（气泡、快速排序）；
      - 选择排序（直接选择、选择树排序、堆排序）；
      - 插入排序（直接插入、折半插入、希尔排序）、
      - 归并排序（二路归并排序）。
    - 不基于比较：根据根据组成关键字的分量及其分布特征。
      - 分配法排序（基数排序）。

本章小结--各种排序方法的比较

- 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。

本章小结--各种排序方法的比较(cont.)

- (1)时间复杂度比较：

排序方法	平均情况	最好情况	最坏情况
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$
直接选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$
直接插入排序	$O(n^2)$	$O(n)$	$O(n^2)$
快速排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n^2)$
堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$
希尔排序	$O(n^2)$~ $O(n\log_2 n)$	$O(n^{1.3})$	$O(n^2)$
(二路)归并排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$
基数排序	$O(d(n+r))$	$O(d(n+r))$	$O(d(n+r))$

本章小结--各种排序方法的比较(Cont.)

- (2)空间复杂度比较和(3)稳定性比较：

排序方法	辅助空间	稳定性/不稳定举例
冒泡排序	$O(1)$	是
直接选择排序	$O(1)$	否/2,2',1
直接插入排序	$O(1)$	是
快速排序	$O(\log_2 n)$ ~$O(n)$	否/2,2',1
堆排序/选择树	$O(1)/O(n)$	否/1,2,2'(最小堆)
希尔排序	$O(1)$	否/3,2,2'(d=2,d=1)
(二路)归并排序	$O(n)$	是
基数排序	$O(n+r)$	是

本章小结--各种排序方法的比较(Cont.)

- (4)算法简单性比较：从算法简单性看，
  - 一类是简单算法，包括起泡排序、直接选择排序和直接插入排序；
  - 另一类是改进后的算法，包括快速排序、堆排序、希尔排序、和归并排序，这些算法都很复杂。

- (5)待排序的记录个数比较：从待排序的记录个数n的大小看，
  - n越小，采用简单排序方法越合适；
  - n越大，采用改进的排序方法越合适。
  - 因为n越小，$O(n^2)$同$O(n\log_2 n)$的差距越小，并且输入和调试简单算法比输入和调试改进算法要少用许多时间。

本章小结--各种排序方法的比较(Cont.)

- (6)记录本身信息量比较：
  - 记录本身信息量越大，移动记录所花费的时间就越多，所以对记录的移动次数较多的算法不利。

排序方法	最好情况	最坏情况	平均情况
直接插入排序	0	$O(n^2)$	$O(n^2)$
起泡排序	0	$O(n^2)$	$O(n^2)$
直接选择排序	0	$O(n)$	$O(n)$

本章小结--各种排序方法的比较(Cont.)

- ⑺关键字值的分布情况比较：

  当待排序记录按关键的值有序时，

  - 插入排序和起泡排序能达到$O(n)$的时间复杂度；
  - 对于快速排序而言，这是最坏的情况，此时的时间性能蜕化为$O(n^2)$；
  - 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。

查·论·编	排序算法
理论	计算复杂性理论 · 大O符号 · 全序关系 · 列表 · 稳定性 · 比较排序 · 自适应排序 · 排序网络 · 整数排序
交换排序	冒泡排序 · 鸡尾酒排序 · 奇偶排序 · 梳排序 · 侏儒排序 · 快速排序 · 臭皮匠排序 · Bogo排序
选择排序	选择排序 · 堆排序 · Smooth排序 · 笛卡尔树排序 · 锦标赛排序 · 循环排序
插入排序	插入排序 · 希尔排序 · 二叉查找树排序 · 图书馆排序 · Patience排序
归并排序	归并排序 · 梯级归并排序 · 振荡归并排序 · 多相归并排序 · Strand排序
分布排序	美国旗帜排序 · 桶排序 · 鸽排序 · 爆炸排序 · 计数排序 · 鸽巢排序 · 相邻图排序 · 基数排序 · 闪电排序 · 插值排序
并发排序	双调排序器 · Batcher归并网络 · 两两排序网络
混合排序	Tim排序 · 内省排序 · Spread排序 · 反移排序 · J排序
其他	拓扑排序 · 煎饼排序 · 意粉排序