

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
- ◆ 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





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 $\{r_1, r_2, \dots, r_n\}$, 其相应的关键字序列为 $\{k_1, k_2, \dots, k_n\}$ 。排序是将这些记录排列成顺序为 $\{r_{s1}, r_{s2}, \dots, r_{sn}\}$ 的一个序列, 使得相应的关键字的值满足 $k_{s1} \leq k_{s2} \leq \dots \leq k_{sn}$ (称为升序) 或 $k_{s1} \geq k_{s2} \geq \dots \geq k_{sn}$ (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。





6.1 基本概念 (cont.)

→ 排序算法及其存储结构:

- 从操作角度看，排序是线性结构的一种操作，待排序记录可以用顺序存储结构或链接存储结构存储。我们假定采用顺序存储结构：

```
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 (cont.)

► 算法的实现:

```
void BubbleSort ( int n , LIST &A )
```

```
{   for ( int i =1; i <= n-1; i++ )
```

```
    for ( int j =n; j >= i+1; j-- )
```

```
        if ( A[j].key < A[j-1] )
```

```
            swap (A[j], A[j-1])
```

```
}
```

```
void swap(records &x, records &y)
```

```
{   records w ;
```

```
    w = x ;      x = y ;      y = w ;
```

```
}
```





6.2 气泡排序 (cont.)

```
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+1; 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;
    }
}
```





6.2 气泡排序 (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：

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

$$\sum_{i=1}^{n-1} 3(n-i) = \frac{3n(n-1)}{2}$$

- 移动次数：

- 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： $O(1)$ 。





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





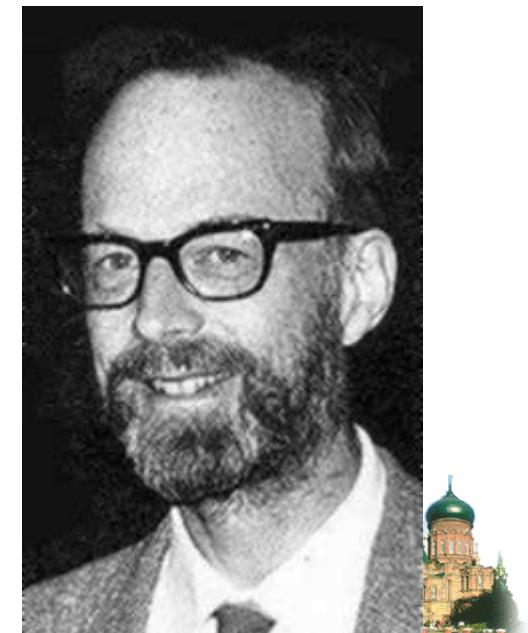
6.3 快速排序 (cont.)

◆ 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.)

→ 算法的实现步骤:

设被排序的无序区为 $A[i], \dots, A[j]$

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 $A[i], \dots, A[j]$ 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.)

→ 基准元素的选取:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 3 | 1 | 4 | 1 | 5 | 9 | 2 | 6 | 5 | 3 |
|---|---|---|---|---|---|---|---|---|---|

- 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 $\text{FindPivot}(i, j)$ 是求 $A[i].key, \dots, A[j].key$ 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 $A[i].key, \dots, A[j].key$ 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 $A[i], \dots, A[j]$

(1) 扫描：

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 3 | 1 | 4 | 1 | 5 | 9 | 2 | 6 | 5 | 3 |
|---|---|---|---|---|---|---|---|---|---|

- 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 $A[l].key \geq v$ ；
- 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 $A[i], \dots, A[j]$ 被划分成为满足条件的两部分 $A[i], \dots, A[l - 1]$ 和 $A[l], \dots, A[j]$ 。





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[l].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 );
```

```
}
```

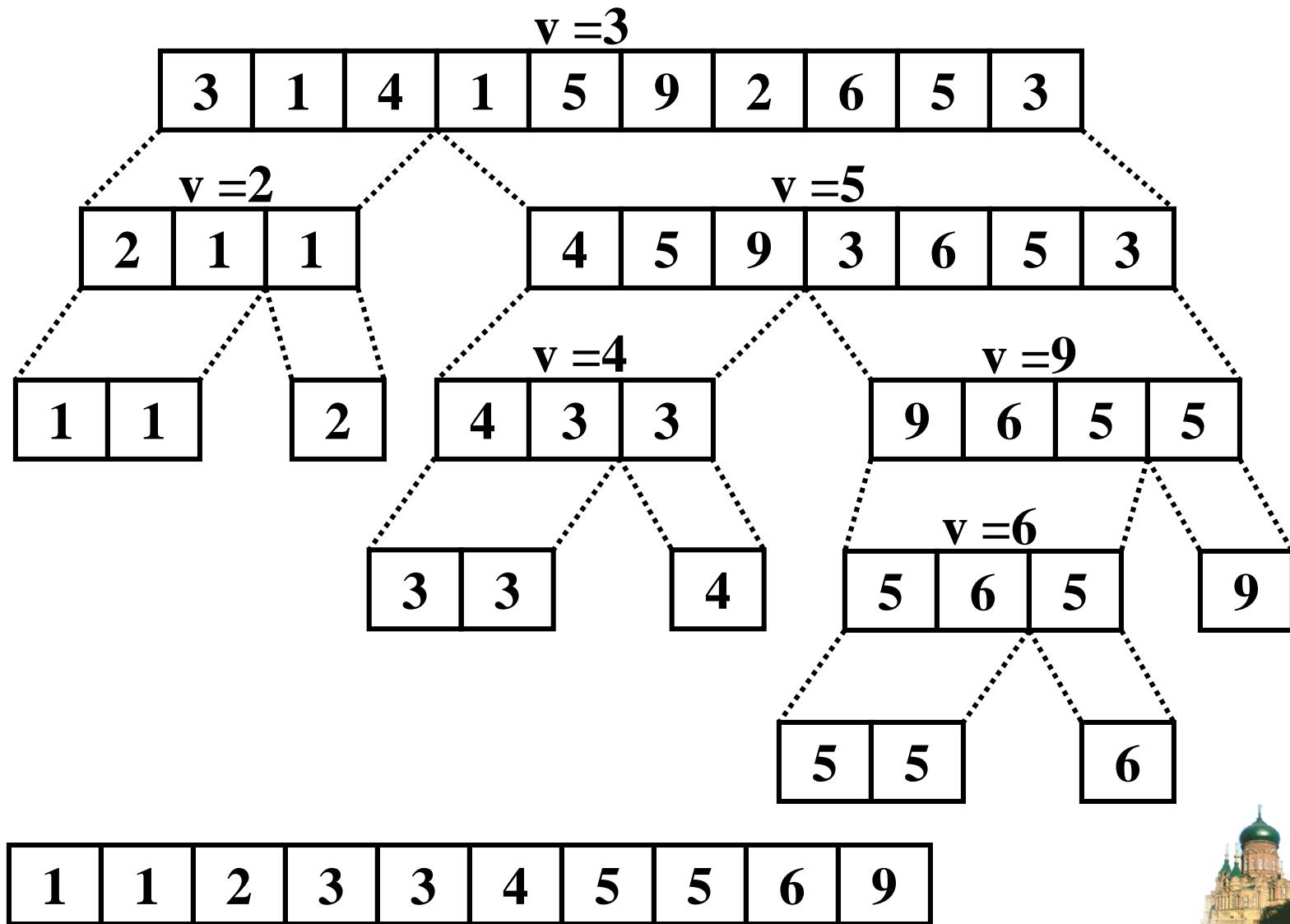
```
} //对数组A[1],...,A[n]进行快速排序可调用QuickSort(1,n)实现
```





6.3 快速排序 (cont.)

示例





6.3 快速排序 (cont.)

| pivotkey 49 38 65 97 76 13 27 49 | | | | | | | | | |
|--|--|--|--|--|--|--|--|--|--|
| 初始关键字 | | | | | | | | | |
| 进行 1 次交换之后 | | | | | | | | | |
| 进行 2 次交换之后 | | | | | | | | | |
| 进行 3 次交换之后 | | | | | | | | | |
| 进行 4 次交换之后 | | | | | | | | | |
| 完成一趟排序 | | | | | | | | | |
| (a) | | | | | | | | | |
| 初始状态 | | | | | | | | | |
| {49 38 65 97 76 13 27 49} | | | | | | | | | |
| 一次划分之后 | | | | | | | | | |
| {27 38 13} {49} {76 97 65 49} | | | | | | | | | |
| 分别进行快速排序 | | | | | | | | | |
| {13} {27} {38} {49} {76} {97} | | | | | | | | | |
| 结束 结束 结束 {49} {65} {76} {97} 结束 | | | | | | | | | |
| 有序序列 | | | | | | | | | |
| {13 27 38 49 49 65 76 97} | | | | | | | | | |
| (b) | | | | | | | | | |





6.3 快速排序 (cont.)

```
int PartSort( int* array , int left , int right)
{
    int key = array[right];
    while(left < right)
    {
        while(left < right && array[left] <= key)
            {           ++left;           }
        array[right] = array[left];
        while(left < right && array[right] >= key)
            {           --right;           }
        array[left] = array[right];
    }
    array[right] = key;
    return right;
}
```





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 $O(n \log_2 n)$ 。
- $T(n) \leq 2T(n/2) + n$

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

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

...

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

- 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 $O(n)$



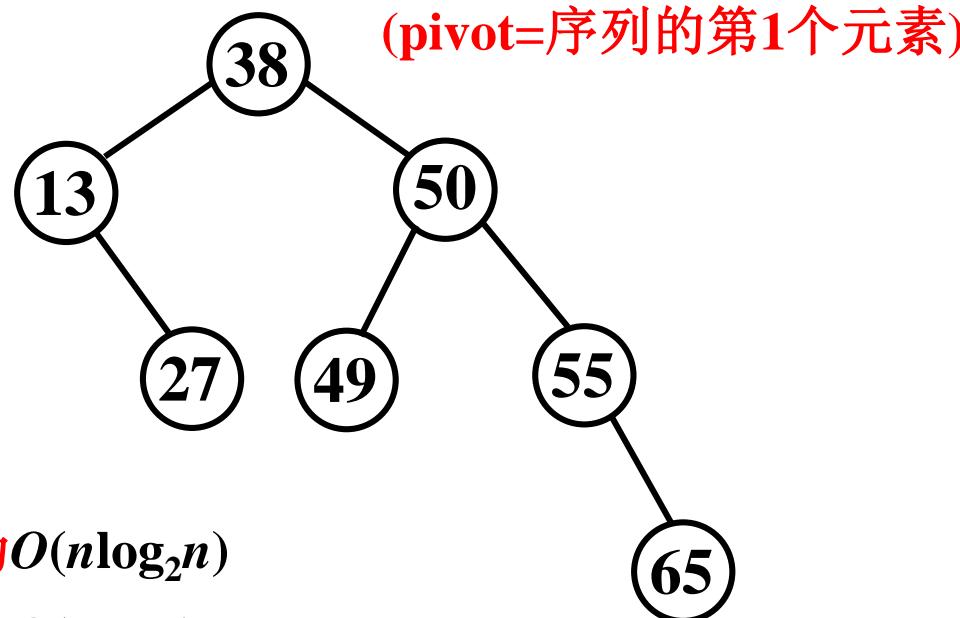


6.3 快速排序 (cont.)

◆ 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 $O(\log_2 n)$





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 $A[1\dots n]$ 中最小者与 $A[1]$ 交换位置，第2遍处理是将 $A[2\dots n]$ 中最小者与 $A[2]$ 交换位置，.....，第 i 遍处理是将 $A[i\dots n]$ 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 (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++)
            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 ;
            }
        swap(A[i], A[lowindex]) ;
    }
}
```





6.4 直接选择排序 (cont.)

► 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 $O(1)$ 。





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





6.5 堆排序 (cont.)

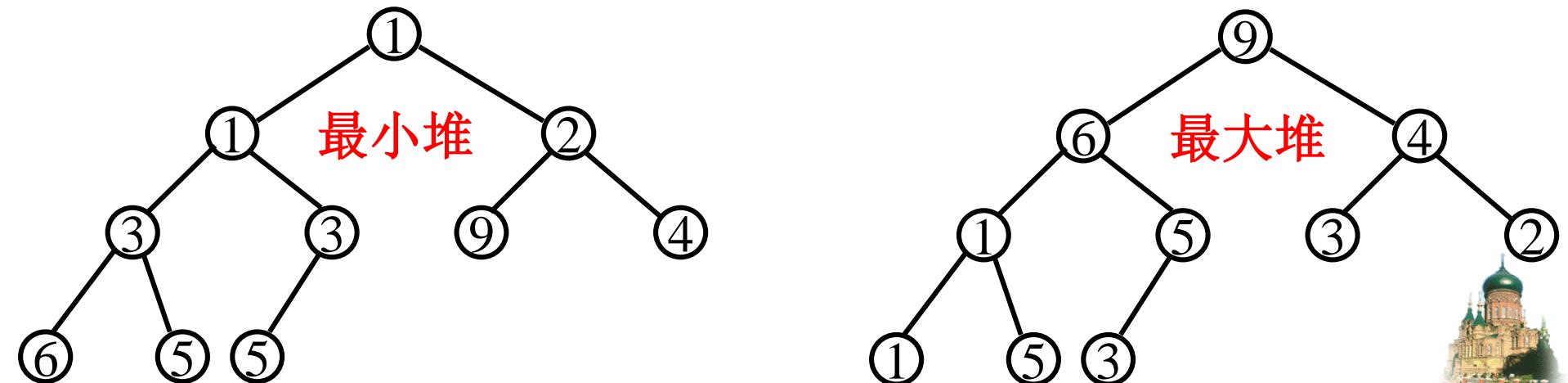
堆的定义

把具有如下性质的数组A表示的完全二叉树称为（最小）堆：

- (1) 若 $2*i \leq n$ ， 则 $A[i].key \leq A[2*i].key$ ；
- (2) 若 $2*i+1 \leq n$ ， 则 $A[i].key \leq A[2*i+1].key$ 。

把具有如下性质的数组A表示的完全二叉树称为（最大）堆：

- (1) 若 $2*i \leq n$ ， 则 $A[i].key \geq A[2*i].key$ ；
- (2) 若 $2*i+1 \leq n$ ， 则 $A[i].key \geq A[2*i+1].key$ 。





6.5 堆排序 (cont.)

◆ 堆的性质

- 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 $A[i/2].key \leq A[i].key \quad 1 \leq i/2 < i \leq n$ 。
- 在堆中，以任意结点为根的子树仍然是堆。特别地，每个叶结点也可视为堆。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**。
- 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**。

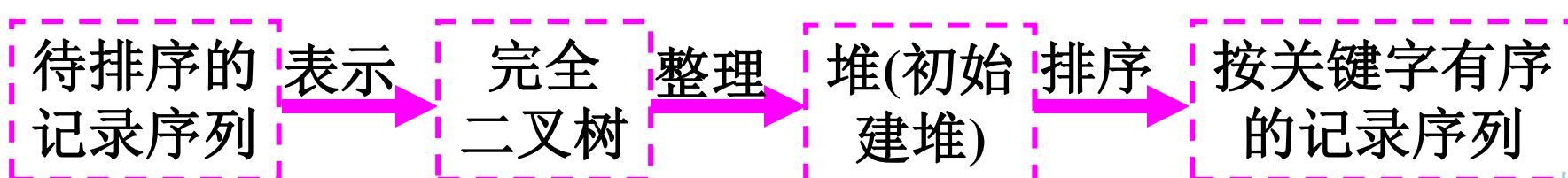




6.5 堆排序 (cont.)

▶ 堆排序的基本思想:

- 首先将待排序的记录序列用完全二叉树表示;
- 然后完全二叉树构造成一个堆，此时，选出了堆中所有记录关键字的最小者；
- 最后将关键字最小者从堆中移走，并将剩余的记录再调整成堆，这样又找出了次小的关键字记录，以此类推，直到堆中只有一个记录。





6.5 堆排序 (cont.)

→ 堆排序的实现步骤:

- 把待排序的记录序列用完全二叉树的数组存储结构A表示
- **初始建堆:** 把数组所对应的完全二叉树以**堆不断扩大的方式整理成堆**。令 $i = n/2, \dots, 2, 1$ 并分别把以 $n/2, \dots, 2, 1$ 为根的完全二叉树整理成堆，即执行算法 $\text{PushDown}(i, n)$ 。
- **堆排序:** 令 $i = n, n-1, \dots, 2$
 - 1.**交换:** 把堆顶元素(当前最小的)与位置 i (当前最大的叶结点下标)的元素交换，即执行 $\text{swap}(A[1], A[i])$;
 - 2.**整理:** 把剩余的 $i-1$ 个元素整理成堆，即执行 $\text{PushDown}(1, i-1)$;
 - 3.**重复** 执行完这一过程之后，则 $A[1], A[2], \dots, A[n]$ 是按关键字**不增顺序**的有序序列。





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*/  
    }  
}
```





6.5 堆排序 (cont.)

◆ 整理堆算法: PushDown(first , last)

- 把以A[first]为根, 以A[last]为最右边叶结点的完全二叉树整理成堆。根据堆的定义, 它要完成的功能是, 把完全二元树中的关键字最小的元素放到堆顶, 而把原堆顶元素下推到适当的位置, 使(A[first],...,A[last])成为堆。
- 那么, 怎样把关键字最小的元素放到堆顶, 把堆顶元素下推到适当位置呢?
- 具体操作(要点)如下:
 - 把完全二元树的根或者子树的根与其左、右儿子比较, 如果它比其左 / 右儿子大, 则与其中较小者交换(若左、右儿子相等, 则与其左儿子交换)。重复上述过程, 直到以A[first]为根的完全二元树是堆为止。
- PushDown(first , last)算法实现如下:

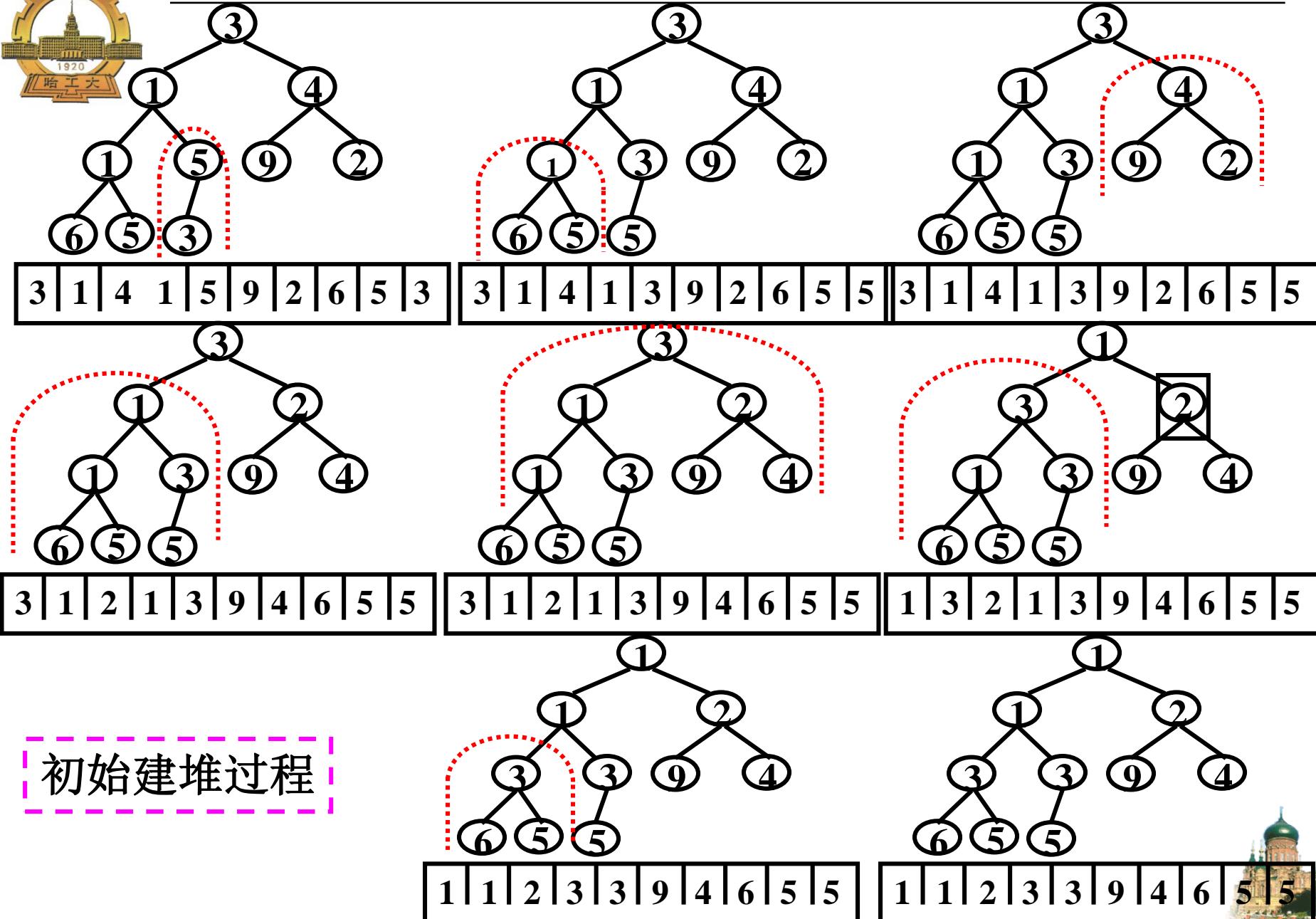




6.5 堆排序 (cont.)

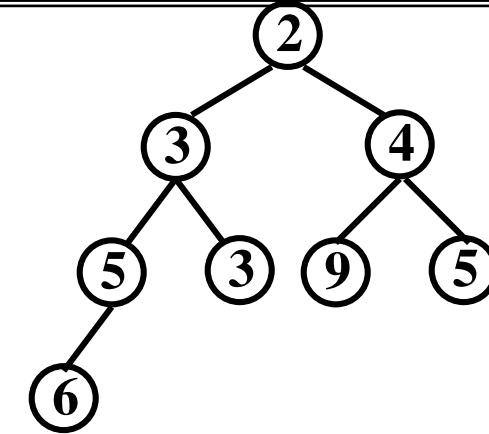
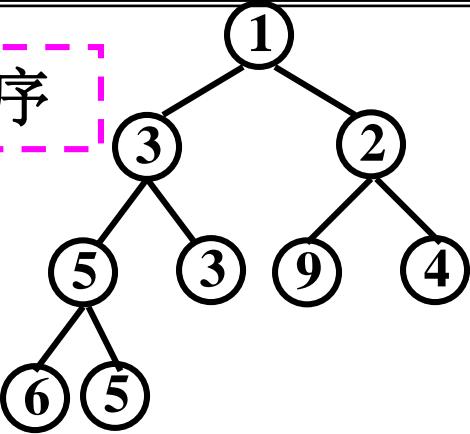
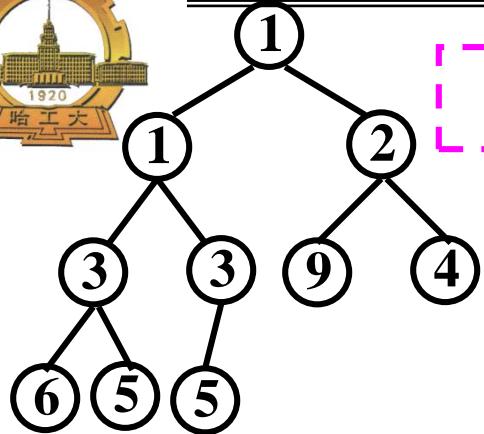
```
void PushDown(int first,int last)
{
    /*整理堆:A是外部数组,把A[first]下推到完全二元树的适当位置*/
    int r=first; /* r是被下推到的适当位置, 初始值为根first*/
    while(r<=last/2) /* A[r]不是叶, 否则是堆 */
        if((r==last/2) && (last%2==0)) /* r有一个儿子在2*r上且为左儿子*/
            if(A[r].key>A[2*r].key)
                swap(A[r],A[2*r]);/*下推*/
            r=last; /*A[r].key小于等于A[2*r].key或者"大于", 交换后到叶, 循环结束*/
        } else if((A[r].key>A[2*r].key)&&(A[2*r].key<=A[2*r+1].key)) {
            /*根大于左儿子, 且左儿子小于或等于右儿子*/
            swap(A[r],A[2*r]);      /*与左儿子交换*/
            r=2*r;                  /*下推到的位置也是下次考虑的根*/
        } else if((A[r].key>A[2*r+1].key)&&(A[2*r+1].key<A[2*r].key)) {
            /*根大于右儿子, 且右儿子小于左儿子*/
            swap(A[r],A[2*r+1]);   /*与右儿子交换*/
            r=2*r+1;               /*下推到的位置也是下次考虑的根*/
        }else /*A[r]符合堆的定义, 不必整理, 循环结束*/
            r=last;
}/*PushDown*/
```







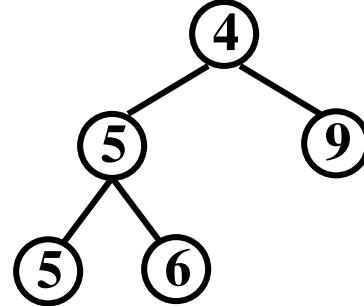
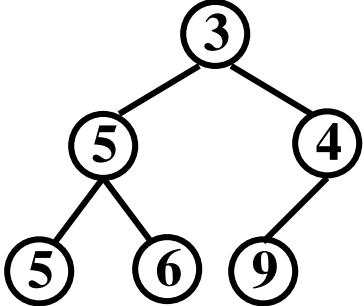
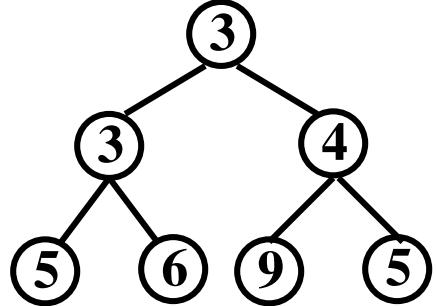
堆排序



| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 2 | 3 | 3 | 9 | 4 | 6 | 5 | 5 |
|---|---|---|---|---|---|---|---|---|---|

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 3 | 2 | 5 | 3 | 9 | 4 | 6 | 5 | 1 |
|---|---|---|---|---|---|---|---|---|---|

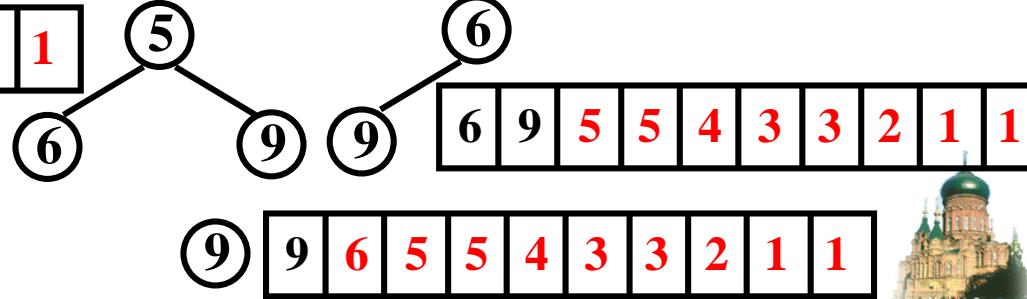
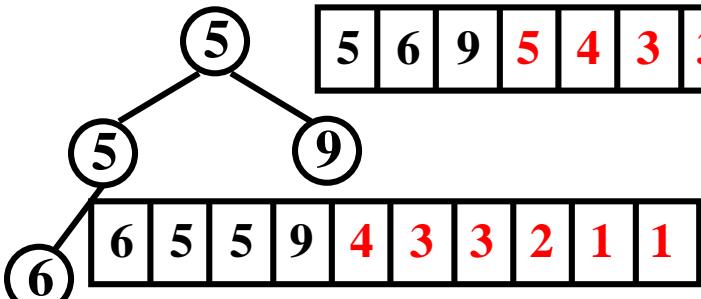
| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 3 | 4 | 5 | 3 | 9 | 5 | 6 | 1 | 1 |
|---|---|---|---|---|---|---|---|---|---|



| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 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 |
|---|---|---|---|---|---|---|---|---|---|





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*2ⁱ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 $O(\log(last/first))=O(\log_2n)$ 。
- 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： $O(1)$

思考:建立堆的时间复杂度是多少?

- (1) 自顶向下：逐个插入法； $O(n * \log n)$
- (2) 自底向上：调整法。 $O(n)$





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。
- 经过 $i-1$ 遍处理后， $A[1\dots i-1]$ 已排好序。第*i*遍处理仅将 $A[i]$ 插入 $A[1\dots i-1, i]$ 的适当位置，使得 $A[1\dots i]$ 又是排好序的序列。要达到这个目的，可以用顺序比较的方法。首先比较 $A[i]$ 和 $A[i-1]$ 的关键字，如果 $A[i-1].key \leq A[i].key$ ，由于 $A[1\dots i]$ 已排好序，第*i*遍处理就结束了；否则交换 $A[i]$ 与 $A[i-1]$ 的位置，继续比较 $A[i-1]$ 和 $A[i-2]$ 的关键字，直到找到某一个位置 $j (1 \leq j \leq i-1)$ ，使得 $A[j].key \leq A[j+1].key$ 时为止。





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=1; i<=n; i++) {  
        j=i;  
        while(A[j].key<A[j-1].key) { //从2开始计算  
            swap(A[j],A[j-1]) ;  
            j=j-1;  
        }  
    }  
}
```





6.6 (直接) 插入排序 (cont.)

◆ 算法的性能分析

■ 最好情况下 (正序) :

- 比较次数: $n-1$
- 移动次数: 0
- 时间复杂度为 $O(n)$ 。

■ 最坏情况下 (反序) :

- 比较次数:
- 移动次数:
- 时间复杂度为 $O(n^2)$ 。

■ 平均情况下 (随机排列)

- 比较次数:
- 移动次数:
- 时间复杂度为 $O(n^2)$ 。

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

$$\sum_{i=2}^n (i+1) = \frac{(n+4)(n-1)}{2}$$

$$\sum_{i=2}^n i/2 = \frac{(n+2)(n-1)}{4}$$

$$\sum_{i=2}^n (i+1)/2 = \frac{(n+4)(n-1)}{4}$$





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。





6.7 希尔排序----分组插入排序

- 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 (cont.)

→ 示例：缩小增量排序

| | 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++) { //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指向所属于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。
- 希尔排序算法的时间性能是所取**增量**的函数，而到目前为止尚未有人求得一种最好的增量序列。
- 研究表明，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 $O(n^{1.3})$





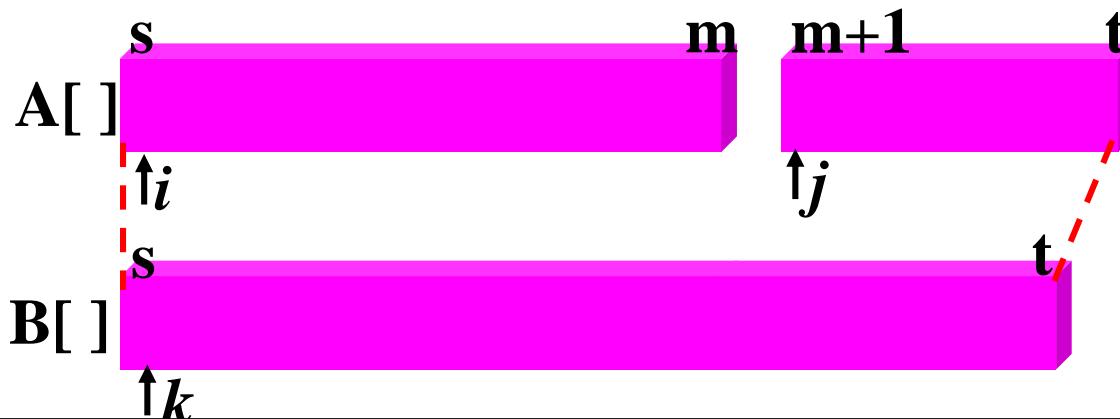
6.8 (二路) 归并排序

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 $A[s] \sim A[m]$ 和 $A[m+1] \sim A[t]$ ，归并成一个按关键字有序序列 $B[s] \sim B[t]$





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 */
```

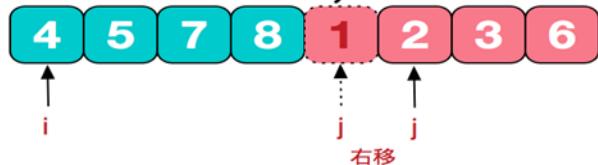
```
while ( j <= t )  B[k++] = A[j++];
```

```
}/*时间复杂度： O( t-s+1 ); 空间复杂度： O( t-s+1 ) */
```



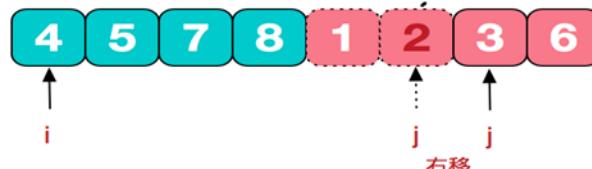


1<4, 将1填入temp数组, 右移j



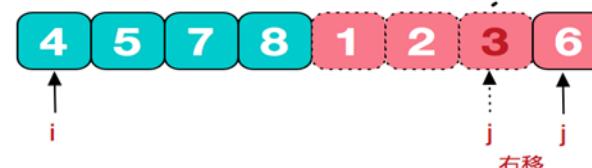
temp

2<4, 将2继续填入temp数组, 右移j



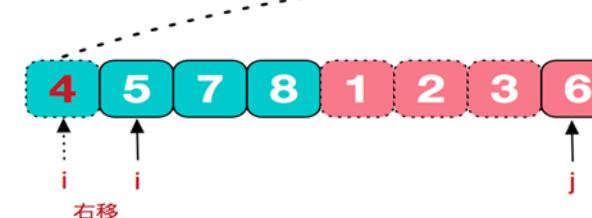
temp

3<4, 将3填入temp数组, 右移j



temp

4<6, 此时将4填入temp数组, 右移i



temp

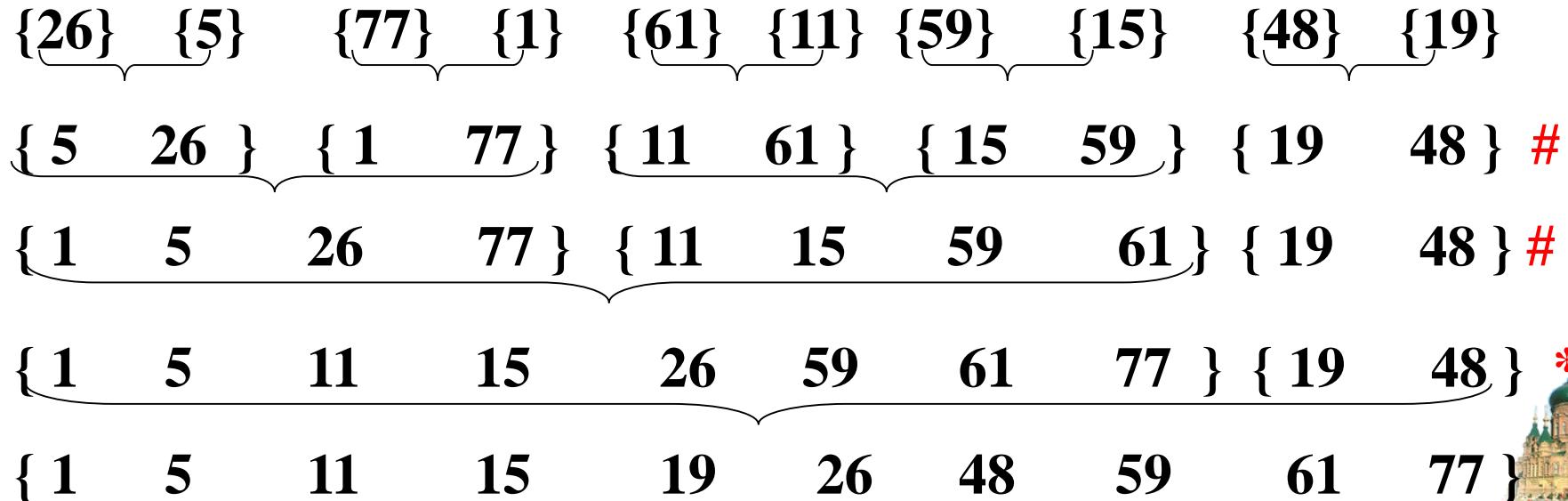




6.8 (二路) 归并排序 (cont.)

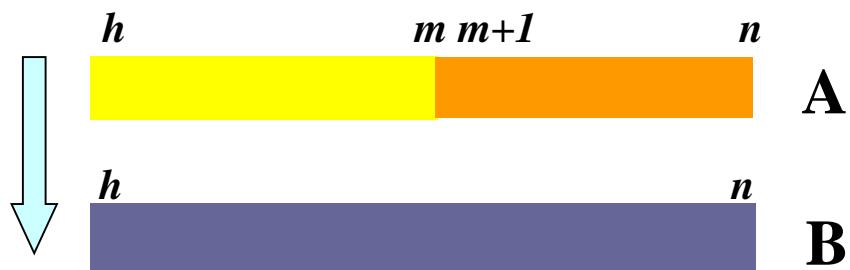
→ 二路归并排序的基本思想 (自底向上的非递归算法)

- 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 $\lceil n/2 \rceil$ 个长度为2的有序序列；
- 再进行两两归并，得到 $\lceil n/4 \rceil$ 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。

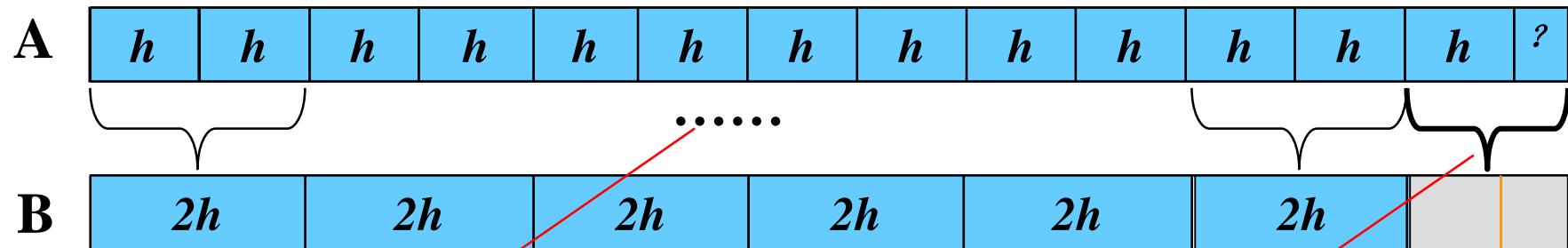




Void merge (h , m, n, A, B)



Void mpass (n, h, A, B)



for ($i=1 ; i+2*h-1 \leq n ; i+=2*h$)
merge ($i, i+h-1, i+2*h-1, A, B$)

$if ((i+h-1) < n)$
merge ($i, i+h-1, n, A, B$);
else
for ($t = i ; t \leq n ; t++$) $B[t] = A[t];$



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 ;      | 开始时,有序序列的长度h=1;结束时,有序序列的长度h=n,
```

```
    while (h < n){ | 用有序序列的长度来控制排序的结束.
```

```
        MergePass (n , h , A , B ) ;
```

```
        h = 2*h ;
```

```
        MergePass (n , h , B , A ) ;/* A、B互换位置 */
```

```
        h = 2*h ;
```

```
}
```

```
 }/* MergeSort */
```





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 $A[1] \sim A[n]$ 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 $B[1] \sim B[n]$ 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 $O(n)$ 。
- 辅助数组是一个公用的数组。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.)

◆ (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 $A[low], \dots, A[high]$ 一分为二，即求分裂点 $mid = (low + high) / 2$ ；
- 求解：递归地对序列 $A[low], \dots, A[mid]$ 和 $A[mid+1], \dots, A[high]$ 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。

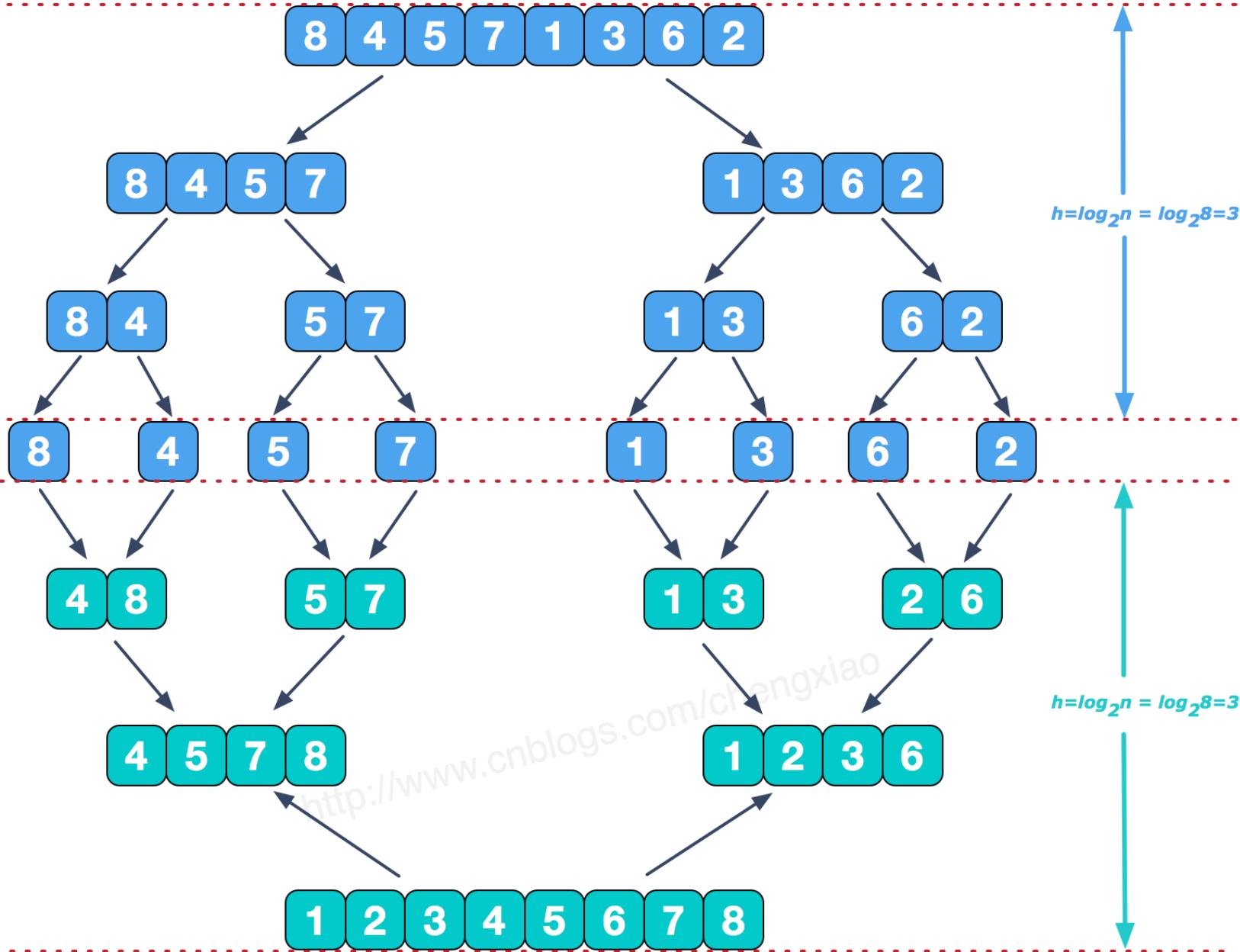
■ 递归的终止条件

- 是子序列长度为 1，
因为一个记录自然有序。

■ 算法实现



分



治

<http://www.cnblogs.com/chengxiao/>



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 */  
{   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 , hight , A , B) ;  
    }  
}/* MergeSort */
```





例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 $(len \gg 1) + 1$ 大的数，如果 len 为偶数，就是第 $(len \gg 1)$ 和 $(len \gg 1) + 1$ 两个数的平均数。

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

例3：归并排序最为经典的一个应用就是求一个整数序列中逆序对数了。

设 $a[1 \dots n]$ 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 $O(n \log(n))$ 的算法来求解数组 a 中逆序对数。

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 $a \leq b$ ，直接拿掉 a ，否则， a 以及它后面的所有元素跟 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……





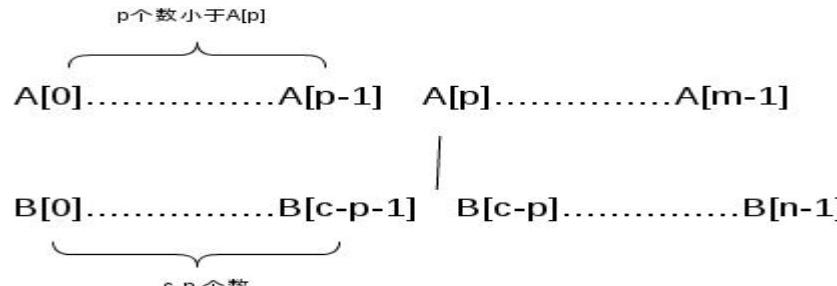
例1. 设数组A的长度为m, 数组B的长度为n, 两个数组都是递增有序的。求这两个数组的中位数。

中位数: 一个大小为n的数组, 如果n是奇数, 则中位数只有一个, 数组中恰好有 $(n-1)/2$ 个元素比中位数小。

如果n是偶数, 则中位数有两个(下中位数和上中位数), 设只求下中位数, 对于下中位数, 数组中恰好有 $(n-1)/2$ 个元素比下中位数小。

若中位数只有一个, 它前面有 $c = (m+n-1)/2$ 个数比它小。中位数要么出现在数组A中, 要么出现在数组B中。考察数组A中的一个元素A[p], 在数组A中, 有 p 个数比A[p]小, 如果数组B中恰好有 $c-p$ 个数比 A[p] 小, 则俩数组合并后就恰好有 c 个数比A[p]小, 于是A[p]就是要找的中位数。

如下图所示:



求两个有序数组的中位数

如果A[p] 恰好位于 B[c-p-1] 和 B[c-p] 之间, 则 A[p] 是中位数

如果A[p] 小于 B[c-p-1], 说明A[p] 太小了, 接下来从 A[p+1] ~ A[m-1] 开始找
如果A[p] 大于 B[c-p], 说明A[p] 太大了, 接下来从 A[0] ~ A[p-1] 开始找。

如果数组A没找到, 就从数组B找。





M个长度为N的排序好的数组 求中位数

目前有两个比较好的算法

算法一

- ◆ (1) 取所有数组的最小值和最大值, 求出全体的最大值和最小值 $\min \max$
然后取 $m = (\min + \max) / 2$
- ◆ (2) 用 m 在所有数组中搜索 找到比 m 小的数的个数 n_1 若 n_1 大于 $M * N / 2$ 则
 $mid = (\min + mid) / 2$ 否则 $mid = (mid + \max) / 2$
- ◆ (3) 重复上述搜索一共循环 $\log(\max - \min)$ 次

算法二

- ◆ (1) 取所有数组的中值排序
- ◆ 弃掉这个最大值所在数组的后半部分和最小值所在数组的前半部分 再
将这两个只剩一半的数组合并 $O(N)$
- ◆ (2) 从合并后的数组中找出中值插入到之前的中值排序数组里 $O(\log N)$
- ◆ (3) 重复 (1) (2), 每次循环可以舍掉一个数组, 一共需要循环 M 次





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：**
 - 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 $0,1,\dots,9$ 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



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=0;i<=9;i++) {
    Concatenate(Q[1], 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:

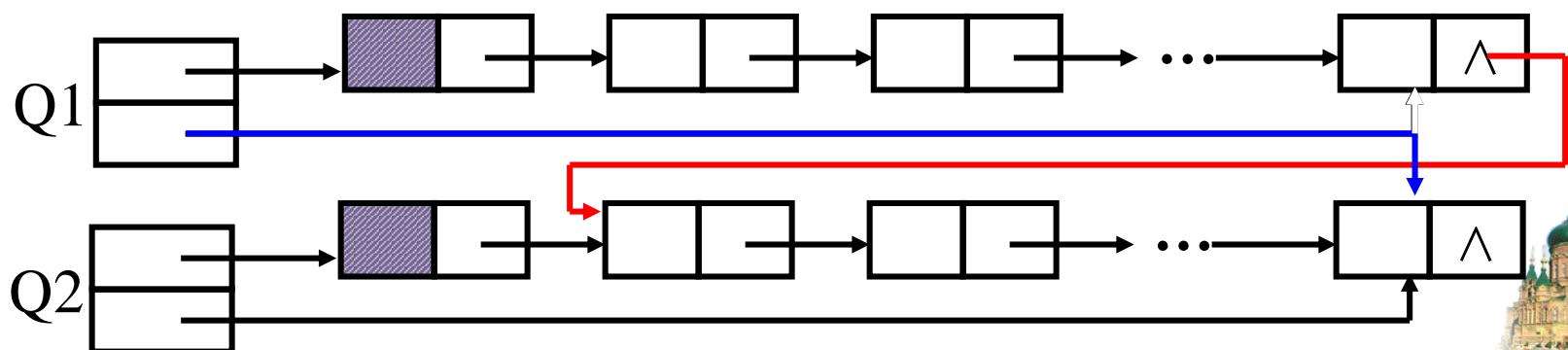
- 由于每个桶（箱）存放多少个关键字分量相同的记录个数无法预料，即队列 $Q[0], Q[1], \dots, Q[9]$ 长度很难确定，故桶一般设计成链式排队，两个排队链在一起的方法如下：

```
void Concatenate(Queue Q1, Queue Q2)
```

```
{  if ( !Empty( Q2 ) ) {  
    Q1.rear->next=Q2.front->next;  
    Q1.rear=Q2.rear;
```

```
}
```

```
}
```





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。





计数排序

- 假设：有n个数的集合，而且n个数的范围都在 $0 \sim k$ ($k = O(n)$) 之间。
- 运行时间： $\Theta(n+k)$

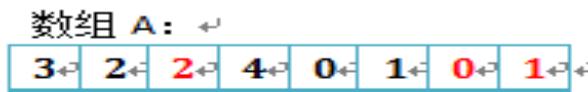


图 2.1

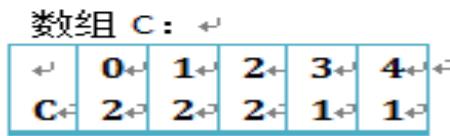


图 2.2

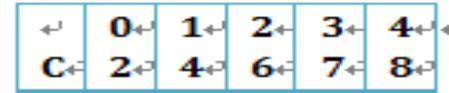


图 2.3

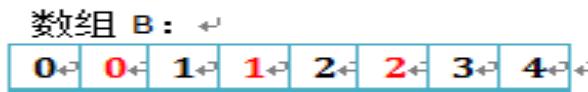


图 2.4

- 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 ($0 \leq i < k$) 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 *C = new int[k];
- int i;
- for (i = 0; i < k; i++)
- C[i] = 0;
- for (i = 0; i < len; i++)
- C[A[i]]++;
- for (i = 1; i < k; i++)
- C[i] += C[i-1];
- for (i = len-1; i >= 0; i--) // 从右至左保证算法的稳定性
- { B[C[A[i]]-1] = A[i]; C[A[i]]--; }
- }
- 所以总的运行时间为 $\Theta(2(n+k)) = \Theta(n+k)$ 。

1. 当数列最大最小值差距过大时，并不适用于计数排序

比如给定20个随机整数，范围在0到1亿之间，此时如果使用计数排序的话，就需要创建长度为1亿的数组，不但严重浪费了空间，而且时间复杂度也随之升高。

2. 当数列元素不是整数时，并不适用于计数排序

如果数列中的元素都是小数，比如3.1415，或是0.00000001这样子，则无法创建对应的统计数组，这样显然无法进行计数排序。

正是由于这两大局限性，才使得计数排序不像快速排序、归并排序那样被人们广泛适用。

• 如果不考虑稳定性，如何实现？





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 |
|----------|-----------------|-----------------|-----------------|
| 直接插入排序 | $O(n^2)$ | $O(n)$ | $O(n^2)$ |
| 希尔排序 | $O(n \log_2 n)$ | $O(n^{1.3})$ | $O(n^2)$ |
| 起泡排序 | $O(n^2)$ | $O(n)$ | $O(n^2)$ |
| 快速排序 | $O(n \log_2 n)$ | $O(n \log_2 n)$ | $O(n^2)$ |
| 直接选择排序 | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ |
| 堆排序 | $O(n \log_2 n)$ | $O(n \log_2 n)$ | $O(n \log_2 n)$ |
| (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 |
|----------|-------------------------|-------------------|
| 直接插入排序 | $O(1)$ | 是 |
| 希尔排序 | $O(1)$ | 否/3,2,2'(d=2,d=1) |
| 起泡排序 | $O(1)$ | 是 |
| 快速排序 | $O(\log_2 n) \sim O(n)$ | 否/2,2',1 |
| 直接选择排序 | $O(1)$ | 否/2,2',1 |
| 堆排序 | $O(1)$ | 否/1,2,2'(最小堆) |
| (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，包括直接插入排序、直接选择排序和起泡排序；
- 另一类是改进后的算法，包括希尔排序、堆排序、快速排序和归并排序，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 | O | $O(n^2)$ | $O(n^2)$ |
| 起泡排序 | O | $O(n^2)$ | $O(n^2)$ |
| 直接选择排序 | O | $O(n)$ | $O(n)$ |





本章小结--各种排序方法的比较(Cont.)

→ (7)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。





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





问题：若文件初态是反序的，则直接插入，直接选择和冒泡排序哪一个更好？

应选直接选择排序为更好。分析如下：

(1) 在直接插入排序算法中，反序输入时是最坏情况，此时：

$$\text{关键字的比较次数: } C_{max} = (n+2)(n-2)/2$$

$$\text{记录移动次数为: } M_{max} = (n-1)(n+4)/2$$

$$T_{max} = n^2 - 4n - 3 \text{ (以上二者相加)}$$

(2) 在冒泡排序算法中，反序也是最坏情况，此时：

$$C_{max} = n(n-1)/2 \quad M_{max} = 3n(n-1)/2$$

$$T_{max} = 2n^2 - 2n$$

(3) 在选择排序算法中，

$$C_{max} = n(n-1)/2 \quad M_{max} = 3(n-1)$$

$$T_{max} = n^2/2 - 5n/2 - 3$$

虽然它们的时间复杂度都是 $O(n^2)$, 但是选择排序的常数因子为 $1/2$,
因此选择排序最省时间。