第10章 内部排序

- **主要内容：**
- 排序的基本概念
- 插入排序
- 交换排序
- 选择排序
- 归并排序
- 各种排序方法的比较

§10.1 概述

1. 基本概念

- 排序：将一个数据元素(或记录)的任意序列，重新排列成一个按关键字有序的序列的过程叫排序。
  - 设序列$\{R_1, R_2, \ldots, R_n\}$，相应关键字序列为$\{K_1, K_2, \ldots, K_n\}$，所谓排序是指重新排列$\{R_1, R_2, \ldots, R_n\}$为$\{R_{p1}, R_{p2}, \ldots, R_{pn}\}$，使得满足：$\{K_{p1} \leqslant K_{p2} \ldots\ldots \leqslant K_{pn}\}$ 或 $\{K_{p1} \geqslant K_{p2} \ldots\ldots \geqslant K_{pn}\}$
- 排序的稳定性：假设$K_i = K_j$，$(1 \leqslant i \leqslant n;\ 1 \leqslant j \leqslant n;\ i \neq j)$且在排序前的序列中$R_i$领先于$R_j$，如在排序后$R_i$仍领先于$R_j$，则称所用的排序方法是稳定的，反之则称排序方法是不稳定的。

2. 排序的分类

- 待排序记录所在位置
  - 内部排序：待排序记录存放在内存中
  - 外部排序：排序过程中需对外存进行访问的排序

内部排序适用于记录个数不很多的小文件；外部排序则适用于记录个数太多，不能一次将其全部放入内存的大文件

- 排序依据策略
  - 插入排序：直接插入排序，折半插入排序，希尔排序
  - 交换排序：冒泡排序，快速排序
  - 选择排序：简单选择排序，堆排序
  - 归并排序：2-路归并排序
  - 基数排序

3. 排序的基本操作

- 排序的基本操作：
  - 比较操作：比较两个关键字的大小
  - 改变指向记录的指针(逻辑关系)或将一个记录从一个位置移动到另一个位置

是否需要移动，与待排序记录的存储方式有关

4. 数据的存储形式

- 待排记录一般有三种存储形式
  - 存放在一组地址连续的存储单元中，类似顺序表
    - 实现排序必须借助移动记录
  - 存放在静态链表中
    - 实现排序只需修改指针
  - 存放在一组地址连续的存储单元中，同时另设一个指示各个记录存储位置的地址向量
    - 实现排序时修改地址向量中的地址，排序结束时统一调整记录的存储位置

注意：本章中排序的记录以第一种方式存储

顺序存储结构定义

```
#define MAXSIZE  20    //顺序表的长度
typedef int KeyType;   //关键字类型为整数类型

typedef struct{
   KeyType  key;          //关键字项
   InfoType  otherinfo;   //其它数据项
}RedType;                 //记录类型

type struct {
   RedType r[MAXSIZE+1]; //r[0]空作为哨兵
   int  length;           //顺序表长度
}SqList;                  //顺序表类型
```

5. 算法复杂度分析

- 评价排序算法的标准：
  - 执行时间
  - 所需辅助空间
  - 稳定性
- 排序算法的时间开销
  - 主要是关键字的比较次数和记录的移动次数。
  - 有的排序算法其执行时间不仅依赖于问题的规模，还取决于输入实例中数据的状态。
- 排序算法的空间开销
  - 若所需辅助空间不依赖于问题的规模n，即辅助空间为O(1)，则称为就地排序。
  - 非就地排序所要求的辅助空间一般为O(n)

排序所需时间
- 简单排序：T(n)=O(n²)
- 先进的排序：T(n)=O(logn)
- 基数排序：T(n)=O(d.n)

§10.2 插入排序

1. 插入排序的基本思想

- 基本思想：设R={R_1，R_2，……，R_n}为原始序列，R'={ }初始为空。插入排序就是依次取出R中的元素R_i，然后将R_i有序地插入到R'中。
- 例如：

	有序插入	
R={5,2,10,2}	➡	R'={ }
R={2,10,2}	5	R'={5}
R={10,2}	2	R'={2,5}
R={2}	10	R'={2,5,10}
R={ }	2	R'={2,2,5,10}

2. 直接插入排序

- 排序过程：整个排序过程为n-1趟插入
  - 将序列中第1个记录看成是一个有序子序列
  - 从第2个记录开始，逐个进行插入，直至整个序列有序

R_1	R_2	……	R_n

R'={R_1}　R={ R_2，R_3，……，R_n }

- 若R[1..*i*-1]为有序区，寻找R[*i*]插入位置的方法:
  - 在R[0]处设置监视哨，即令：R[0]=R[*i*]
  - 从j=*i*-1的记录位置开始依次向前比较
  - 若R[*i*].key<R[j].key，R[j]后移，否则插入位置找到，将R[*i*]插入到第j+1个位置

R[0]既有哨兵的作用，也暂存了R[*i*]的值，使其不会因记录后移而丢失

举例

- 例：已知某下列的关键字为{49 38 65 97 76 13 27 49}，试采用直接插入排序方法进行排序

i=1:	(38)	(49)	38	65	97	76	13	27	49
i=2:	(65)	(38	49)	65	97	76	13	27	49
i=3:	(97)	(38	49	65)	97	76	13	27	49
i=4:	(76)	(38	49	65	97)	76	13	27	49
i=5:	(13)	(38	49	65	76	97)	13	27	49
i=6:	(27)	(13	38	49	65	76	97)	27	49
i=7:	(49)	(13	27	38	49	65	76	97)	49
i=8:		(13	27	38	49	49	65	76	97)

算法程序

```
void Dinsertsort(Sqlist  &L)
{  for (i=2;i<=L.length;++i)   //对每一个Ri∈R
     //小于时，需将L.r[i]插入到有序表
     if LT(L.r[i].key, L.r[i-1].key)
     {  L.r[0]=L.r[i];       //复制为哨兵
        /*寻找插入位置j*/
        for (j=i-1; LT(L.r[0].key, L.r[j].key); --j)
            L.r[j+1]=L.r[j];  //将j……i-1的记录后移一格
        L.r[j+1]=L.r[0];   }   //将Ri插入到位置j+1
}
```


算法效率

特点：n较小时，排序较快；待排序列基本正序时，排序时间为O(n)，是稳定的排序方法

- 时间复杂度
  - 待排序记录按关键字从小到大排列(正序)
    比较次数: $\sum_{i=2}^{n} 1 = n-1$　移动次数: 0
  - 待排序记录按关键字从大到小排列(逆序)
    比较次数: $\sum_{i=2}^{n} i = \frac{(n+2)(n-1)}{2}$　移动次数: $\sum_{i=2}^{n} (i+1) = \frac{(n+4)(n-1)}{2}$
  - 待排序记录随机，取平均值
    比较次数: $\approx \frac{n^2}{4}$　移动次数: $\approx \frac{n^2}{4}$
  - 总的时间复杂度:T(n)=O(n²)
- 空间复杂度：S(n)=O(1)

3. 折半插入排序

- 排序过程：用折半查找方法确定插入位置。
- 举例：

i=1: (38) (49) 38 65 97 76 13 27 49

LMH

H LM

H+1:插入位置

i=2: (65) (38 49) 65 97 76 13 27 49

LM H

LMH

MH L

……

H+1:插入位置

i=8: (13 27 38 49 49 65 76 97)

算法程序

```
void  BInsertSort ( Sqlist  &L)
{  for( i=2; i<=L.length; ++i ) {
     L.r[0]=L.r[i];
     low=1;  high=i-1;
     while (low<=high){    //折半查找位置
        m=(low+high)/2;
        if (L.r[0].key<L.r[m].key) high=m-1;
        else  low=m+1;  }
     for (j=i-1; j>=high+1; --j)
        L.r[j+1]=L.r[j];
     L.r[high+1]=L.r[0];
  }  }
```

算法效率

时间复杂度(只改善了比较): $T(n)=O(n^2)$

空间复杂度: $S(n)=O(1)$

折半插入排序是稳定的排序

4. 希尔排序

直接插入排序什么时候效率高？

若待排记录为“正序”时，时间复杂度为O(n)，即待排记录基本有序时，效率高；当n很小时，效率也很高。

- 希尔排序(Shell’s Sort)又称“缩小增量排序”，基本思想为：把一个较长的待排序列分成若干段，$n=n_1+n_2+\ldots\ldots+n_k$；然后，分别对各段进行直接插入排序，使得整个序列基本有序；最后对整个序列进行一次直接插入排序。

需解决的关键问题：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？

希尔排序过程

- 分割待排序记录的目的
  - 减少待排序记录个数；使整个序列向基本有序发展

基本有序：{1, 2, 8, 4, 5, 6, 7, 3, 9}

局部有序：{6, 7, 8, 9, 1, 2, 3, 4, 5}

注意：局部有序不能提高直接插入排序算法的时间性能，因此子序列的构成不能是简单地“逐段分割”，而是将相隔某个增量的记录组成一个子序列

- 希尔排序过程：先取一个正整数间隔$d_1<n$，把所有相隔d_1的记录放一组，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操作；直至$d_i=1$，即所有记录放进一个组中排序为止

举例

- {49 38 65 97 76 13 27 49 55 04}

1趟分组(d_1=5)	49	38	65	97	76	13	27	49	55	4
1趟排序结果：	13	27	49	55	4	49	38	65	97	76
2趟分组(d_2=3)	13	27	49	55	4	49	38	65	97	76
2趟排序结果：	13	4	49	38	27	49	55	65	97	76
3趟分组(d_3=1)	13	4	49	38	27	49	55	65	97	76
3趟排序结果：	4	13	27	38	49	49	55	65	76	97

最后一趟移动记录5次, 共移动记录10次; 直接插入排序共移动27次

算法效率

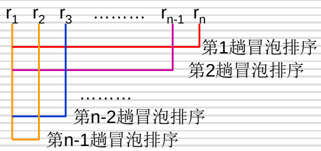
- 希尔排序当$d[k]=2^{(t-k+1)}-1$时(t为排序的总趟数，k为第k趟排序)，可提高排序速度为$O(n^{3/2})$，因为：
  - 插入排序时$T(n)=O(n^2)$，而分组后n值减小，n^2更小
  - 关键字较小的记录跳跃式前移，在进行最后一趟增量为1的插入排序时，序列已基本有序
  - 根据经验统计，希尔排序所需比较和移动次数约为$n^{1.3}$，当$n\to\infty$时，可减少到$n(\log_2 n)^2$。
- 希尔排序不稳定
- 增量序列取法
  - 无除1以外的公因子
  - 希尔给出：$d_1=n/2$，$d_{i+1}=d_i/2$
  - 最后一个增量值必须为1

§10.3 交换排序

- 1. 交换排序的基本思想
- 基本思想：在待排序列中选两个记录，将它们的关键码相比较，如果反序(即排列顺序与排序后的次序正好相反)，则交换它们的存储位置。
  - 特点：通过交换，将关键字值较大的记录向序列的后部移动，关键字较小的记录向前移动。
- 典型算法
  - 冒泡排序
  - 快速排序

2. 冒泡排序

- 基本思想
  - 在n个记录中，将相邻两个记录进行比较，若r[i].key>r[i+1].key (i=1,2,...,n)，则交换(第一趟冒泡排序)，结果关键字最大的记录被放在r_n。
  - 在n-1个记录中，若r[i].key>r[i+1].key (i=1,2,...,n-1)，则交换，结果关键字次大的记录被安置在r_{n-1}
  - 以此类推，直到"在一趟排序过程中没有进行过交换记录的操作"为止



举例

- 已知某序列的关键字为{ 49 38 65 97 76 13 27 49 }，试采用冒泡排序法对其进行排序

初始关键字	49	38	65	97	76	13	27	49
第1趟排序后	38	49	65	76	13	27	49	97
第2趟排序后	38	49	65	13	27	49	76	97
第3趟排序后	38	49	13	27	49	65	76	97
第4趟排序后	38	13	27	49	49	65	76	97
第5趟排序后	13	27	38	49	49	65	76	97
第6趟排序后	13	27	38	49	49	65	76	97

结束标志：在一趟排序过程中没有进行过交换记录的操作

算法程序

```
void Bubble_sort(Sqlist  &L)
{  for (i= L.length; i>1; --i)
   {   exchange=false;   //设置开关
       for(j=1; j<i; ++j)
           if GT(L.r[j].key, L.r[j+1].key)//一趟冒泡
           {  L.r[j] ←→ L.r[j+1];  //移动3次
              exchange=true;   }
       if (!exchange) exit;
   }
}
```


性能分析

- 时间复杂度
  - 最好情况(正序)：比较1趟n-1次，不移动
  - 最坏情况(逆序)：
    - 比较次数：$\sum_{i=1}^{n-1}(n-i)=\frac{1}{2}(n^2-n)$
    - 移动次数：$3\sum_{i=1}^{n}(n-i)=\frac{3}{2}(n^2-n)$
  - 总的时间复杂度：$O(n^2)$
- 空间复杂度：$S(n)=O(1)$
- 冒泡排序是稳定的排序

冒泡排序的改进

- 改进不对称性

1	2	3	4	5	需扫描1趟	5	4	3	2	1	需扫描n-1趟
5	1	2	3	4	需扫描2趟	2	3	4	5	1	需扫描n-2趟

造成不对称的原因是什么？

每趟扫描仅能使最轻气泡"下沉"一个位置，因此使位于顶端的最轻气泡下沉到底部时，需做n-1趟扫描

- **双向冒泡排序**：在排序过程中交替改变扫描方向

2 3 4 5 1 →

2 3 4 1 5 ←

1 2 3 4 5

改进方案2：记录的比较和移动是在相邻单元中进行，记录的每次交换只能移动一个单元。因此要减少扫描次数，可以增大比较和移动的距离，并缩小序列规模

2. 快速排序

- 基本思想：选择一个枢轴，通过一趟排序，将待排序记录分割成独立的两部分，其中一部分记录的关键字均比另一部分记录的关键字小，然后分别对这两部分记录进行排序，以达到整个序列有序。

快速排序方法的实质是将一组关键字进行分区交换排序

- 排序过程：
  - 设序列为r_1，r_2，…，r_n，定r_1为枢轴
  - 设low，high指针分别从两端与枢轴比较，把比r_1小的记录放前，比r_1大的记录放后，得到一次划分：

    r_{s1}，r_{s2}，…，r_{sk}，r_1，r_{t1}，r_{t2}，…，r_{tj}
  - 然后分别对两序列r_s和r_t再进行划分，直到划分后的序列剩一个元素为止，这是一个递归的过程

一趟分割过程

- 先从high所指记录向前搜索，找到第1个小于key的记录与枢轴交换。然后从low起向后搜索，找到第一个比key大的记录与low互换。重复上面两步，直到low=high为止(该位置即为枢轴的位置)

初始关键字	49	38	65	97	76	13	27	49
	L						H	H
1次交换之后	27	38	65	97	76	13	49	49
	L	L	L				H	
2次交换之后	27	38	49	97	76	13	65	49
			L			H	H	
3次交换之后	27	38	13	97	76	49	65	49
			L	L		H		
4次交换之后	27	38	13	49	76	97	65	49
				LH	H	H		
完成一趟排序	27	38	13	49	76	97	65	49

快速排序整个过程

初始关键字	49	38	65	97	76	13	27	49
第1趟排序后	27	38	13	49	76	97	65	49
第2趟排序后	13	27	38	49	49	65	76	97
	结束		结束					结束
第3趟排序后	13	27	38	49	49	65	76	97
						结束		
有序序列	13	27	38	49	49	65	76	97

快速排序是不稳定的排序

初始关键字	38	49	65	97	76	13	27	49
有序序列	13	27	38	49	49	65	76	97


```
int Partition(Sqlist &L, int low, int high)
//以L.r[low]为主记录，对子系列L.r[low...high]的一趟划分
{  temp= L.r[low];
   while (low<high)     //进行一趟划分
   {  while (low<high &&L.r[high].key>=temp.key) --high;
      L.r[low]=L.r[high];
      while (low<high &&L.r[low].key<=temp.key) ++low;
      L.r[high]=L.r[low];   }
   L.r[low]=temp;        //找到主记录的位置low
   return low;  }

void Qsort(Sqlist &L,int low,int high) //递归算法实现
{  if (low<high)                  //长度大于1
   {  loc=Partition(L,low,high);      //将L.r[low...high]一分为二
      Qsort(L,low,loc-1);            //对低子表递归排序
      Qsort(L,loc+1,high);   } } //对高子表递归排序

void QuickSort(SqList L) //对顺序表L做快速排序
{  Qsort(L,1,L.length);  }
```


算法效率分析

- 时间复杂度
  - 最好情况：每次总是选到中间值作枢轴，则形成二叉递归树：$T(n)=O(n\log_2 n)$
  - 最坏情况：每次总是选到最小或最大元素作枢轴，则退化为冒泡排序：$T(n)=O(n^2)$
  - 平均时间复杂度：$T_{avg}(n)=kn\ln n$；k为某个常数
  - 经验证明，在所有同数量级O(nlogn)的此类排序中，快速排序的常数因子k最小。因此，就平均时间而言，快速排序是速度最快的排序
  - 若枢轴记录取r[low]、r[(low+high)/2]和r[high]中关键字的中值的记录，并与r[low]互换，可以大大改善最坏情况的快速排序
- 空间复杂度：使用栈空间实现递归
  - 最坏情况：$S(n)=O(n)$；一般情况：$S(n)=O(\log_2 n)$

§10.4 选择排序

- 基本思想：每一趟在n-i+1(i=1，2，…，n-1)个记录中选取关键字最小的记录作为有序序列中第i个记录。
- 1. 简单选择排序
- 排序过程：
  - 设待排序列为：a_1，a_2，…，a_n
  - 在a_1，a_2，…，a_n中，找最小值记录与a_1交换
  - 在a_2，a_3，…，a_n中，找最小值记录与a_2交换
  - 重复上述操作，共进行n-1趟排序后，排序结束
- 简单选择排序是不稳定的排序

5	5	1	2	3	➡	1	2	3	5	5

算法程序和效率分析

```
void  Selectsort(Sqlist &L)
{  for(i=1; i<L.length; ++i)
   {   for (j=i+1, k=i; j<=L.length; ++j)
         if (L.r[i].key>L.r[j].key)  k=j;
       if(k!=i) L.r[i] ↔L.r[k];   }  }
```

- 时间复杂度
  - 移动次数：最好(正序)：0；最坏(逆序)：3(n-1)
  - 比较次数：$\sum_{i=1}^{n-1}(n-i)=\frac{1}{2}(n^2-n)$
  - 总的时间复杂度：$T(n)=O(n^2)$
- 空间复杂度：$S(n)=O(1)$

改进方案：若在每趟比较时，既找出键值最小的记录，也找出键值较小的记录，则可减少后面的选择中所用的比较次数，并缩小序列规模

2. 堆排序

- (1) 堆和堆排序
- 堆：n个元素的序列(a_1, a_2, …, a_n)，其关键值序列为：(k_1, k_2, …, k_n)，当且仅当其关键值满足下式称之为堆：

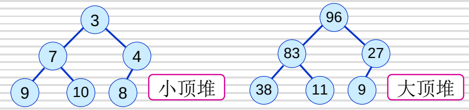
$$\begin{cases} k_i \le k_{2i} \\ k_i \le k_{2i+1} \end{cases} \text{或} \begin{cases} k_i \ge k_{2i} \\ k_i \ge k_{2i+1} \end{cases} \quad \text{其中} i=1, 2, \dots, \lfloor n/2 \rfloor$$

- 满足前一表达式的堆称为小顶堆，即堆顶元素为所有元素中最小值；符合后一组不等式的堆称为大顶堆，即堆顶元素为所有元素中最大值。

考虑完全二叉树的顺序存储，若将其解释成堆如何？

举例

- {3, 7, 4, 9, 10, 8}，{96, 83, 27, 38, 11, 9}



注意：小顶堆中较小结点靠近根结点，但不绝对。大顶堆中较大结点靠近根结点，但也不绝对

- 堆是具有如下性质的完全二叉树：若树中每个结点的值都小于等于(或大于等于)其左、右孩子结点的值，则从根结点开始按结点编号排列所得的结点序列就是一个堆。

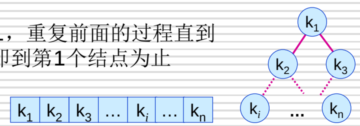
堆排序

- 堆排序：将无序序列建成一个堆，得到关键字最小(或最大)的记录；输出堆顶的最小(大)值后，使剩余的n-1个元素重又建成一个堆，则可得到n个元素的次小值；重复执行，得到一个有序序列，这个过程就叫堆排序。
- 排序过程(小顶堆)
  - 若a_1，a_2，……，a_n为堆，则a_1最小，交换a_1，a_n，输出a_n
  - 将剩余a_1，a_2，……，a_{n-1}调整为堆，当作新的序列
  - 重复这个过程直到序列只剩一个元素

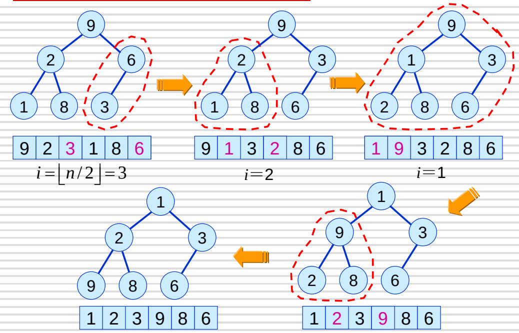
面临问题：
1. 如何由一个n个元素的无序序列建成一个堆？
2. 输出堆顶元素后，剩下的n-1个元素如何调整才能成为一个新的堆？

(2) 建立堆

- 算法描述(小顶堆)
  - 首先把给定序列看成是一棵完全二叉树
  - 从第$i=\lfloor n/2 \rfloor$结点(最后一个非终端结点)开始与其子树结点比较，若其直接子树结点中较小者小于i结点，则交换。若该直接子树结点交换后大于其孩子结点，继续交换，直到叶结点或不再交换
  - 令$i=i-1$，重复前面的过程直到$i=1$，即到第1个结点为止



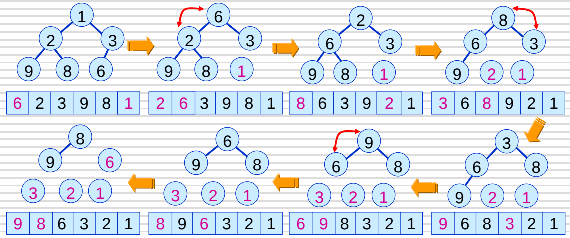
举例：9 2 6 1 8 3



(3) 调整堆—筛选

- 筛选：输出堆顶元素后，剩余元素重新调整为堆的过程。
- 调整步骤(小顶堆)
  - 将该完全二叉树中最后一个元素替代已输出的结点
  - 若新的完全二叉树的根结点小于左右子树的根结点，则直接输出。反之，则比较左右子树根结点的大小。若左子树的根结点小于右子树的根结点，则将左子树的根结点与该完全二叉树的根结点交换，否则将右子树与其交换。
  - 重复上述过程，调整左子树(或右子树)，直至叶子结点，则新的二叉树满足堆的条件。

举例



堆排序的过程就是一个不断从堆顶到叶子的筛选过程

当堆顶元素为所有元素中最小值时，输出的序列为逆序，为了符合正序，可采用大顶堆，即堆顶元素(根结点)为最大值。

算法程序

```
void  Heapadjust(Sqlist  &H,int s,int m)
// 除结点H.r[s]外,H.r[s+1..m]为大顶堆,将s结点调整到适当位置
{  temp=H.r[s];
   for (j=2*s; j<=m; j*=2) //沿k值较大的孩子筛选
   {  if ( j<m && LT(H.r[j].key,H.r[j+1].key)  ++j; //结点值较大的为j
      if (LT(temp.key,H.r[j].key)
      {  H.r[s]=H.r[j];  s=j;  }   //若s结点小于j,则j覆盖s,s下移一层
      else  exit;   }
   H.r[s]=temp;   } //将s放入合适的位置

void  Heapsort(Sqlist  &H)
{  for (i=H.length/2; i>0; --i;)
      Heapadjust(H,i,H.length); //建立初始堆
      for (i=H.length; i>1; --i;)
      {   H.r[1] ↔ H.r[i];      //输出第i个小元素
          Heapadjust(H,1,i-1);  } } //剩余1..i-1个元素调整为堆
```


算法效率分析

- 时间复杂度
  - 对深度为k的堆，筛选算法中进行关键字的比较次数至多为2(k-1)。建堆时，n个元素所需的比较次数不超过4n。总的比较次数：$T(n) < 2n\lfloor \log_2 n \rfloor$
  - 在最坏情况下，其时间复杂度为O(n log n)
- 空间复杂度
  - 仅需一个记录大小的辅助存储空间。
- 堆排序是不稳定的排序。

堆排序的运行时间主要耗费在建立初始堆和筛选上，当记录较少时，不值得提倡。当n很大时效率高，是常用算法。

§10.5 归并排序

- 基本思想：将若干有序序列逐步归并，最终得到一个有序序列。
- 归并：将两个或两个以上的有序表合并成一个新的有序表的过程。

归并排序有多路归并排序、两路归并排序，可用于内排序，也可以用于外排序。

- 2-路归并排序
  - 设初始序列含有n个记录，看成n个有序的子序列，每个子序列长度为1
  - 两两合并，得到$\lceil n/2 \rceil$个长度为2或1的有序子序列
  - 对n/2个有序表两两合并，得到n/4个长度为4或3的有序表
  - 重复这个过程，直至得到一个长度为n的有序序列

举例

- 已知某序列为{49, 38, 65, 97, 76, 13, 27}，试用归并排序，将其按从小到大顺序排列。

初始关键字	[49]	[38]	[65]	[97]	[76]	[13]	[27]	7个序列
1趟归并后	[38	49]	[65	97]	[13	76]	[27]	4个序列
2趟归并后	[38	49	65	97]	[13	27	76]	2个序列
3趟归并后	[13	27	38	49	65	76	97]	1个序列

归并排序可否就地进行？

对顺序表进行归并有可能破坏原来的表，因此需要一个与原来长度相同的表存放归并结果，如果采用链表存储可以就地排序

```
void Merge ( RcdType SR[], RcdType TR[], int i, int m, int n )
{   //将有序序列SR[i..m]和SR[m+1..n]归并为有序序列TR[i..n]
   for ( j=m+1, k=i; i<=m&&j<=n; ++k )
   {  if (SR[i].key<=SR[j].key )  TR[k]=SR[i++];
      else  TR[k]=SR[j++];   }
   if ( i<=m )  TR[k..n]=SR[i..m];     //将剩余元素复制到TR中
   if ( j<=n )   TR[k..n]=SR[j..n];    }

void  MSort( RcdType SR[],  RcdType &TR1[],  int s,  int t )
{   //将SR[s..t]归并到TR1[s..t]中
   if ( s==t )    TR1[s]=SR[s];  //子序列长度为1
   else
   {  m=(s+t)/2;     //将SR[s..t]平分成SR[s..m]和SR[m+1..t]
      MSort ( SR, TR2, s, m );
      MSort ( SR, TR2, m+1, t );
      Merge ( TR2, TR1, s, m, t );  } }
```

```
void  MergeSort ( SqList &L ) //对顺序表作归并排序
{     MSort ( L.r, L.r, 1, L.length );      }
```


算法效率

- 时间复杂度
  - 对任意两个有序序列，长度分别为 m，n，可以在O(m+n)时间内完成有序归并，因此每趟归并的时间复杂度为O(n)，整个算法需$\log_2 n$趟。总的时间复杂度：$O(n \log_2 n)$
- 空间复杂度
  - S(n)=O(n)
- 归并排序是稳定的排序

归并排序算法虽简单，但占用辅助空间大，实用性差

§10.7 各种内部排序方法的比较

排序方法	平均时间	最坏情况	辅助空间	稳定性
插入排序	$O(n^2)$	$O(n^2)$	O(1)	稳定
冒泡排序	$O(n^2)$	$O(n^2)$	O(1)	稳定
简单选择排序	$O(n^2)$	$O(n^2)$	O(1)	不稳定
希尔排序	$O(n^2)$	$O(n^2)$	O(1)	不稳定
快速排序	O(n log n)	$O(n^2)$	O(log n)	不稳定
堆排序	O(n log n)	O(n log n)	O(1)	不稳定
归并排序	O(n log n)	O(n log n)	O(n)	稳定
基数排序	O(d(n+rd))	O(d(n+rd))	O(rd)	稳定

总结

- 从平均时间看，快速排序最佳，所需时间最省，但快速排序在最坏情况下的时间性能不如堆排序和归并排序。而后两者相比较，在n较大时，归并排序所需时间较堆排序省
- 简单排序中，直接插入最简单，当序列基本有序或n较小，其最佳
- 基数排序的时间复杂度可写成O(d×n)，它最适用于n值很大而关键字较小的序列
- 从稳定性来比较，基数排序是稳定的，一般来说，时间复杂度为$O(n^2)$的简单排序也是稳定的，而快速排序、堆排序和希尔排序等时间性能较好的排序方法是不稳定的