第八章 排序

8.1 排序的基本概念

8.2 插入排序

8.3 选择排序

8.4 交换排序

8.5 分配排序*

8.6 归并排序*

1

8.1 基本概念

排序是一种整理数据序列，使元素按特定顺序排列的操作。排序过程中序列里的数据元素没有变，但排列顺序可能改变了

一般定义:

假设考虑的数据集合是S，S 元素上有一个全序关系 *less*(*e*, *e*')

一个排序算法 sort 是从S的元素序列到S的元素序列的映射

- 对S的任意元素序列s，s' = sort(s) 是 s 中元素的一个排列
- 而且对 s' 中任意相邻的元素 e, e' 都有 *less*(e, e')

全序关系：集合S上的自反，传递，反对称关系*r*，而且对S中任意的e, e'，都有 *r*(e, e') 或者 *r*(e', e) 成立。

例：整数集合上的小于等于，字符串的字典序，等等。

2

排序是人们日常生活中经常进行的活动。

排序也是数据处理中最重要的工作。统计说明，在计算机数据处理中，很大比例的工作是在做某种排序。

排序可以使数据具有更好的结构性，有利于处理。

例：排序后的序列可以采用二分法查找（效率高）。在检索概率相同时，创建最优二叉检索树前需要把数据排序。

可以抽象地讨论排序问题，我们更关心的是在计算机上可以使用的排序方法，即**排序算法**。

由于排序的重要性，人们研究并提出了许多排序算法。

有些算法很直观朴素，描述简单，但通常效率较低。

也有些算法更深刻地反映了排序问题的某些本质，因此效率比较高，但通常也更复杂一些。

3

基于关键码的排序（排序的一种简单情况）

考虑某种数据记录，记录里有一个或几个可比较大小的关键码。基于关键码的排序就是根据某关键码（此时称之为**排序码**）的大小整理记录序列，使之成为按关键码排序的序列。

设F = (R_0, R_1, …, R_{n-1}) 是一个记录序列，相应排序码的序列是 (K_0, K_1, …, K_{n-1})。排序就是得到F的排列 ($R_{i(1)}$, $R_{i(2)}$, …, $R_{i(n-1)}$)，使对任何 i(j)，$0 \leqslant j \leqslant n-2$，都有$K_{i(j)} < K_{i(j+1)}$。

这是按排序码递增，也可能需要按递减的顺序排序。

在排序过程中待排序记录全部存在内存的称为**内排序**；排序中需要使用外存的称为**外排序**。有些算法更适合用于内排序，也有些可能适合处理外排序问题。

本章讨论的都是内排序算法，其中的归并排序算法是大部分外排序算法的基础。

4

排序中的基本操作:

1. 比较关键码大小（比较元素）
2. 移动记录（调整记录的顺序）

评价排序算法的主要标准是:

- 执行算法所需的时间（时间复杂性，基于基本操作描述）
- 执行算法所需要的附加空间（空间复杂性）。在考虑排序算法时，通常不计记录序列本身所占用的空间，因为这部分空间是原有的，总是需要的。这里只考虑排序操作中需要的临时性辅助空间量
- 算法本身的复杂程度也是一个需要考虑的因素。但算法的实现只需要做一次，因此这是一个次要因素

这些都是对任何算法时都需要考虑的性质

5

排序算法特有的一些性质:

稳定性：排序算法的一种重要性质，稳定的算法**更有用**。

可能出现 $K_i = K_j$ ($0 \leqslant i, j \leqslant n-1, i < j$)的情况，也就是说：两记录排序码相等，在待排序序列里 R_i 位于 R_j 之前 (因为 i<j)。

如果某排序算法能保证：只要 (R_i, R_j) 有上述性质，在排序后的序列里R_i必然位于R_j之前，那么就说这种排序算法是**稳定的**，否则（不能保证的话）它就是不稳定的。

原序列的顺序可能隐含一些信息，稳定排序算法维持这些信息

排序算法的另一有价值性质是**适应性**：如果实际的待排序序列比较接近排好序的形式，算法能否更快完成工作。

具有适应性的算法有时工作得更快（实际效率可能更高），因为实际中常常遇到接近排好序的序列。

6

一些书籍里把被排序的记录序列称为文件。

排序方法很多，可能按照许多方式对它们分类。

本教科书把排序方法分为如下几类：

一、插入排序　　二、选择排序

三、交换排序　　四、分配排序

五、归并排序　　六、外部排序

下面讨论中，对每类都给出一种或几种排序算法

下面大部分排序算法里使用的示例数据结构：

```
typedef int KeyType;
typedef int DataType;

typedef struct {
   KeyType key;    /* 排序码字段 */
   DataType info;  /* 记录的其它字段 */
} RecordNode;

typedef struct {
   int n;              /* 数组中实际的记录个数，n<MAXNUM */
   RecordNode record[MAXNUM];
} SortObject;
```


8.2 插入排序

排序的一种基本方法是维护一个已排序的子序列：

- 每步把一个待排序的记录按关键字大小插入已排序的部分序列中的适当位置；
- 一个记录的序列是排序的（作为排序工作的出发点）；
- 当所有记录都插入排序序列时，排序工作完成。

8.2.1 直接插入排序

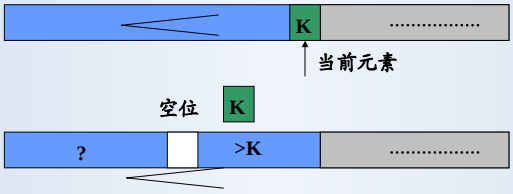
8.2.2 二分法插入排序

8.2.3 表插入排序

8.2.4 Shell排序*

8.2.1 简单插入排序 (Insert Sort)

待排序的n个记录 $\{R_0,R_1,\ldots,R_{n-1}\}$ 存在数组中，简单插入法在插入记录R_i(i=1,2…n-1)时，记录序列分为两个区间 $[R_0, R_{i-1}]$ 和 $[R_i, R_{n-1}]$，前一区间里的记录已排好序，后一区间尚未排序。对 R_i 的处理就是排序码 K_i 依次与 K_{i-1}，K_{i-2}，…，K_0 比较，找出适当位置将 R_i 插入，原位置的记录向后顺移。



算法：简单插入排序(递增序，简单插入排序)

```
void insertSort(SortObject * pvector) {int i, j;
   RecordNode temp;
   RecordNode *data = pvector->record;

   for( i = 1; i < pvector->n; ++i) { /* 依次插入R1, R2…Rn-1 */
      temp = data[i];
      for ( j = i-1; temp.key < data[j].key && j >= 0; j-- )
         /* 由后向前找插入位置，排序码大于ki的记录后移 */
         data[j+1] = data[j];
      if( j != i-1 ) data[j+1] = temp; /* j == i-1时不用复制 */
   }
}
```


初始：　[49] 38 65 97 76 13 27 49

i=1:　[38 49] 65 97 76 13 27 49

i=2:　[38 49 65] 97 76 13 27 49

i=3:　[38 49 65 97] 76 13 27 49

i=4:　[38 49 65 76 97] 13 27 49

i=5:　[13 38 49 65 76 97] 27 49

i=6:　[13 27 38 49 65 76 97] 49

i=7:　[13 27 38 49 49 65 76 97]

稳定排序（这里用颜色区分关键码相同的元素）

算法分析:

空间效率：只需要一个记录的辅助空间。

时间效率：

比较记录的次数：最小 n-1次，最大 n(n-1)/2次

移动记录的次数：最小为 n，最大：(n+2)(n-1)/2

平均比较次数 $\sum_{j=0}^{i-1} p_j c_j = (1/i)\sum_{j=0}^{i-1}(j+1) = (i+1)/2$

总比较次数 $\sum_{j=1}^{n-1}(j+1)/2 = O(n^2)$

平均情况：比较$O(n^2)$，移动$O(n^2)$

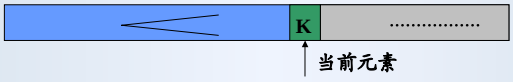
对于已有序的序列，简单插入排序只需要线性时间。它对接近排序的序列工作得更快（适应性）。

8.2.2 二分法插入排序

插入排序的基本操作是在有序序列（已经排序的前段）中进行查找，这种查找可以用二分查找实现。这样实现的排序称为二分法插入排序。

二分法插入排序只能用于顺序存储的数据。每步的工作：

1. 用二分法在已排序的前一段找出当前记录应插入的位置
2. 取出当前记录，将该位置之后的已排序记录顺序后移，腾出空位后将当前记录插入



当前元素


```
void binSort(SortObject * pvector) {     /* 二分法插入排序 */
   int i, j, left, mid, right;       RecordNode temp;
   RecordNode *data = pvector->record;
   for( i = 1; i < pvector->n; ++i ) {
      for ( left = 0, right = i-1; left <= right; ) { /*找插入位置*/
         mid = (left + right)/2;
         if (temp.key < data[mid].key)
            right = mid-1;         /* 应插入左子区间 */
         else left = mid+1;        /* 应插入右子区间 */
      }
      if (left == i) continue;     /* data[i] 位置正确 */
      for ( temp = data[i], j = i-1;  j >= left; --j )
         data[j+1] = data[j];      /* 排序码大于ki的记录后移 */
      data[left] = temp;
   }
}
```


(1) 13 27 38 49 65 76 97 49

left=0 mid=3 right=6

(2) 13 27 38 49 65 76 97 49

left=4 mid=5 right=6

(3) 13 27 38 49 65 76 97 49

left=4 mid=4 right=4

∵ 49<65 ∴ right=mid-1=3

left=4>right ∴ 已找到插入位置left=4

(4) 13 27 38 49 49 65 76 97

二分法插入排序示例

二分插入排序很容易实现稳定排序（与算法细节有关）。

算法分析:

空间效率：同直接插入排序

时间效率：比较次数：

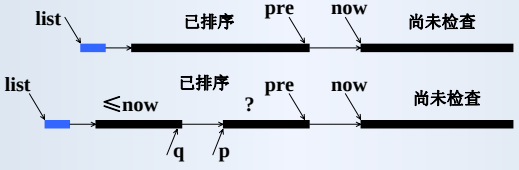
$$\sum_{i=1}^{n} \lceil \log_2 i \rceil \approx n \cdot \log_2 n$$

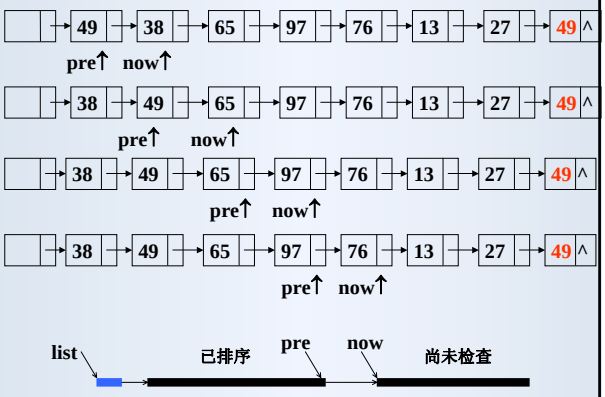
移动记录次数：最坏情况$n^2/2$，最好情况2n，平均为$O(n^2)$

8.2.3 表插入排序

这里讨论对链表的直接插入排序。

- 链表插入排序的主要优点是不需要搬动记录数据，只修改链接指针。如果记录很大，只修改指针可以节约时间。
- 基本思想：在插入记录R_i时，记录R_0至R_{i-1}已经排序。采用顺序比较的方法找到R_i应插入的位置，将记录R_i脱链后插入链表已排序部分中的合适位置。





19

类型声明:

```
typedef struct  Node Node, * LinkList;

struct  Node  {
   KeyType   key;
   DataType   info;
   ListList   *next;
};
```

20

```
void listSort (LinkList * plist) {/* 链表插入排序，有头结点 */
   LinkList now, pre, p, q, head = *plist;
   if (head->next == NULL || head->next->next == NULL)
      return;  /* 空链表或链表中只有一个结点 */
   for ( pre = head->next, now = pre->next; now != NULL; ) {
      q = head; p = head->next;
      while (p != now && p->key <= now->key) {
         q = p;  p = p->next;
      } /* 循环结束时已找到 now 的插入位置 */
      if (p != now) { /* 使now记录脱链，插入q之后 */
         pre->next = now->next; q->next = now; now->next = p;
      }
      else pre = now; /* now 位置正确 */
      now = pre->next;
   }
}
```

21

算法分析:

空间效率:

临时空间是常量的 O(1);

如果考虑链表用的额外空间（除保存数据之外的空间），每个记录增加了next字段，这部分空间 S(n) = O(n)。

时间效率:

用链表方式不需要记录移动，摘除结点和链入结点的次数为 O(n)，但关键码比较次数仍是 $O(n^2)$。因此算法的时间复杂度仍为 $O(n^2)$。

在找插入位置的循环里，条件用的是 p.key<=now.key，直至找到大于 now 的关键码的结点为止。

这样就保证了这个表插入排序是稳定的。

22

8.2.4 Shell排序*

Shell排序又称缩小增量排序(Diminishing Increment Sort)。

Shell (1959) 认为:

- 排序较慢的原因是元素移动太慢，可能经过许多步才能到达最终位置（如插入排序中位于前面的大关键码记录）
- 若能加大元素的移动步伐，就可能加快排序

Shell 提出的想法：把一个序列看成由间距 d 的元素构成的d个“子序列”。反复做:

1. 在每个子序列内部分别排序;
2. 缩小 d 之后重复第1步，直到 d 缩小为 1 时再做一次排序（这也就是整个序列排序）。

23

Shell 排序希望:

前面各遍排序可以使距离目标位置很远的元素大步迁移到目标位置的附近，后续排序主要是局部调整，希望局部调整可以用较少时间完成。

shell排序通过许多次排序实现整个序列的排序。

这一策略保证成功，因为只有最后一次排序就足够了。

非形式算法:

```
inc = d;              /*  d < n  */
while ( inc > 0 ) {
    对相距 inc 的各组排序;
    inc = decrease ( inc ) ;
};
```

24

shell排序可以看作一种排序模式，其中需要:

1. 选择一种基本排序算法做分组的排序。这一排序算法必须有适应性，因前面排序已把大元素移向右，小元素移向左，后面遇到的是接近排序的序列。例如采用插入排序
2. 确定一种减小间距值 d 的模式。常用每次间距 d 减小一半

非形式算法:

```
inc = d;            /*  d < n  */
while(inc > 0) {
    对相距 inc 的各组记录做插入排序;
    inc = inc / 2;
};
```



```
/* 按递增序的 Shell 排序算法 */

void  shellSort(SortObject * pvector, int d) {
   int i, j, inc;
   RecordNode temp, *data = pvector->record;

   for (inc = d; inc > 0; inc /= 2) {  /* inc 为当时增量 */
      for (i = inc; i < pvector->n; i++) { /*同时做inc个序列排序*/
         temp = data[i];       /* 保存待插入记录Ri*/
         /* 按间隔 inc 寻找插入点 */
         for (j = i - inc; j >= 0 && temp.key < data[j].key; j -= inc)
            data[j+inc] = data[j];   /* 大记录后移 */
         data[j+inc] = temp;         /* 插入记录Ri */
      }
   }
}
```


例：序列为49,38,65,97,13,76,27,49，用Shell排序法排序。

原始序列 49 38 65 97 13 76 27 49

1. d=4

2. d=2 13 38 27 49 49 76 65 97

3. d=1 13 38 27 49 49 76 65 97

排序结果 13 27 38 49 49 65 76 97

研究表明:

采用上述常见模式（插入排序和间距减半），Shell排序的平均比较次数和平均移动次数都为$O(n^{1.3})$左右。

Shell排序算法只需常量的辅助空间。

一般说，Shell排序不能实现稳定排序。

最佳间距序列?

最佳间距序列时的时间复杂性?

这里有数学问题和许多研究结果。可以参考 Knuth 的《The Art of Computer Programming》第3卷：排序与检索

中译本（国防工业出版社）

影印本（清华大学出版社）

8.3 选择排序

基本思想:

- 维护最小的 i 个记录的已排序序列
- 每次从剩余未排序的记录中选取关键码最小的记录，排在已排序序列之后，作为序列的第 i +1 个记录
- 空序列作为排序开始
- 被选序列只剩一个元素时（它必最大）排序即完成

如果需要从大到小排序，应该每次选取最大元素。

➢ 8.3.1 直接选择排序

➢ 8.3.2 堆排序*

8.3.1 直接选择排序

直接选择算法（数组）:

以空排序序列开始；每次从未排序记录中选排序码最小的记录，与未排序段的第一个记录交换；直到所有记录排好序。

直接选择排序的比较次数与文件初始状态无关。

已排序段中的最大元素小于等于所有未排序元素。

通过循环顺序比较，维护已找到的最小记录下标。循环结束时得到未排序段最小记录的下标。通过交换扩展已排序序列。

未排序段剩下一个元素时就不必再选择（工作完成）。

49 38 65 97 49 *13* 27 76

[13] 38 65 97 49 49 *27* 76

[13 27] 65 97 49 49 *38* 76

[13 27 38] 97 *49* 49 65 76

[13 27 38 49] 97 *49* 65 76

[13 27 38 49 49] 97 *65* 76

[13 27 38 49 49 65] 97 *76*

[13 27 38 49 49 65 76] *97*


```
/* 直接选择排序，按递增序 */
void  selectSort(SortObject * pvector) {
   int i, j, k;
   RecordNode temp, *data = pvector->record;

   for( i = 0; i < pvector->n-1; i++ ) {  /* 做n-1趟选择排序 */
      /* 在无序段找最小记录*/
      for (k = i, j = i+1; j < pvector->n; j++)
         if (data[j].key < data[k].key) k = j;
      if (k != i) {                /* 需要时交换记录 */
         temp = data[i];
         data[i] = data[k];
         data[k] = temp;
      }
   }
}
```


直接选择排序的时间复杂度：

记录复制：最好时 0　　最坏时 3×(n-1)

比较：n(n-1)/2（总是这样）

总的时间复杂度：$O(n^2)$

稳定性：不稳定

（其中的交换操作是导致不稳定的根源。如果改为移动前面未排序元素腾出空位后存入，可以把算法改为稳定的）

直接选择排序没有适应性，对任何序列都需要 $O(n^2)$ 次比较。

实际试验说明其平均排序效率低于插入排序算法。

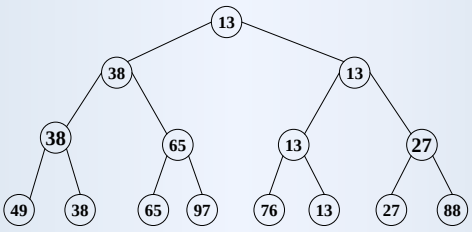
8.3.2 堆排序(Heap Sort)

选择排序的主要操作是比较，提高速度的关键是减少比较次数。

直接选择排序中没利用已做的比较，可能做许多重复比较，算法效率低。利用已有比较结果可能提高排序速度。

研究者提出树形选择排序(Tree Selection Sort)，其方法是：

- 先两两比较 n 个记录的关键码，然后在$\lceil n/2 \rceil$个较小者间再两两比较，如此重复直到选出最小关键字的记录为止。此过程可用一棵二叉树来表示（锦标赛方式）
- 反复从剩下的记录中选出关键字最小的记录，排入序列。比较树的信息有助于选出次小元素和后续元素。（只有败给冠军的那些人可能是亚军）



树形选择排序示例

树形选择排序方法的时间复杂度为$O(n\log_2 n)$。

采用直接树形选择的问题是如何有效表示算法，有效记录比较中得到的信息。可能需要用较多的辅助空间。

研究者提出了更自然的技术，直接树形选择后来被抛弃了。

1964 年 Williams 和 Floyd 提出了堆排序算法，这是一种可以有效地在数组上实现的树形选择排序算法。

n 个记录的数组可看作一棵完全二叉树。一棵完全二叉树称为堆，如果其结点的值满足堆序：

父结点的值小于等于其两个子结点（若存在）的值：

$$\begin{cases} k_i \le k_{2i+1} \\ k_i \le k_{2i+2} \end{cases} \quad i = 0, 1, \ldots, \lfloor (n-1)/2 \rfloor$$，设下标从 0 开始

上面定义的是“小顶堆”。也可定义“大顶堆”：

$$\begin{cases} k_i \ge k_{2i+1} \\ k_i \ge k_{2i+2} \end{cases}$$

注意：1，如果数组 $\{k_0, k_1, \ldots, k_{n-1}\}$ 是堆，堆顶元素必为数组元素的值最小的一个（对大顶堆是最大值）。

2，一个堆去掉堆顶，其余元素形成两个堆（如果不空）。

（数组）堆排序的基本过程（考虑从小到大的排序）：

1. 首先调整记录，把数组变成一个大顶堆；
2. 取出堆顶的最大元素后再调整数组，使数组里剩下的元素重新成为一个（大顶）堆
3. 反复选取和重建堆的操作，直至得到所有记录的有序序列

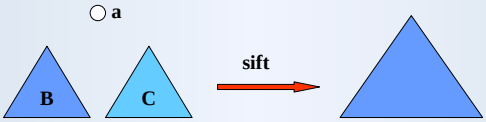
通过适当安排，这一排序过程可以有效地在数组上实现

在数组中做堆排序的基本格局（不变式）：

每次把选出的最大元素与堆的最后元素交换，使排序段增加一个元素（堆减少一个元素）。最终完成从大到小排序。

下面考虑大顶堆（小顶堆情况类似）。

建堆的基本操作是“筛选”(sift)：有两个堆B、C和一个元素a，sift 操作把它们做成一个堆：



方法：用a与B、C的顶元素比较，最大者作为整个堆的顶。如果a不是最大，最大的一定是B（或C）的顶元素。下面继续考虑把a 放入去掉堆顶的B（或C）的问题（也是两个堆加一个元素，设法把它们调整为堆）。两种结束情况：

1，a 在某次比较中为最大，以它为堆顶，这个局部成为堆

2，a 落到底，这时它本身就是堆

两种情况下，整个堆的构造都完成了。

数组的堆排序的基本过程：

一，初始建堆

1. 把数组中元素看作一棵完全二叉树的结点
2. 数组的后一半元素是叶，每个元素（自然）是一个堆（一个元素的完全二叉树都是堆）
3. 对前一半的元素，从后向前逐个考虑和处理
   - 遇到的每个元素 e 都是一棵子树的根，其左右子树已经是堆（是以其子元素为根的堆），通过一次筛选，可以把以 e 为根的子树调整为堆
   - 所有元素都处理完时（处理完数组的首元素时），整个数组里的全部元素就形成了一个（大顶）堆

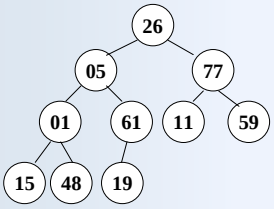
二，选择和堆的重建

1. 交换堆顶元素与堆中的最后一个元素并重建堆
   - 已排序的序列增加一个元素
   - 而且把堆的最后元素放在两个堆上
   - 重建堆。通过一次筛选，使数组前面一段重新成为一个新堆（堆里的元素减少了一个）
2. 反复做上一步，直至整个堆里只剩下一个元素时（它必然是所有元素里最小的）排序完成

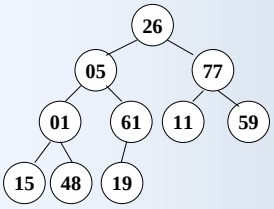
例：设初始序列为

26，5，77，1，61，11，59，15，48，19

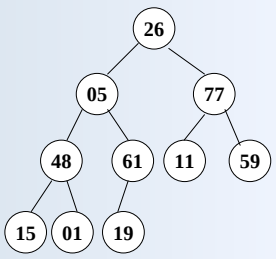
从下标为 (n-1)/2 的记录开始，从下向上逐步建堆，（其间下标逐步减小，对每个记录，把以它为根的子树调整为大顶堆）。



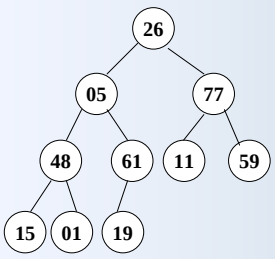
（1）初始完全二叉树



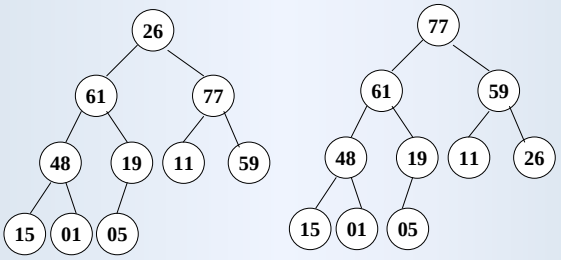
(2) 调整序号为4的结点



(3) 调整序号为3的结点

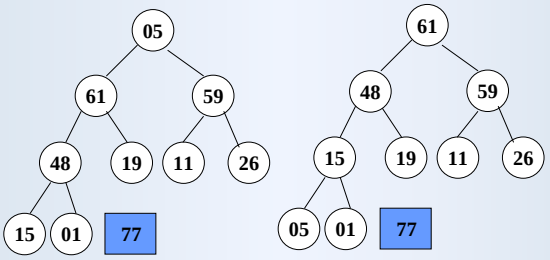


(4) 调整序号为2的结点



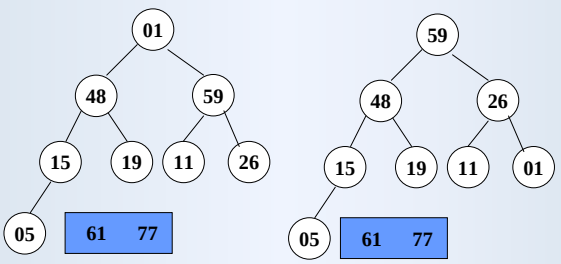
(5) 调整序号为1的结点　(6) 调整序号为0的结点

43



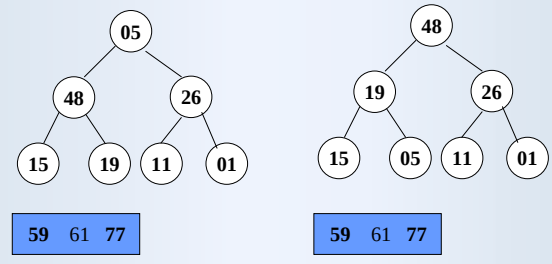
(7) 结点77与结点5互换　(8) 重建堆

44



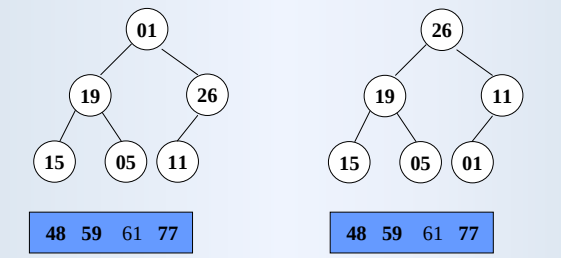
(9) 结点61与结点1互换　(10) 重建堆

45



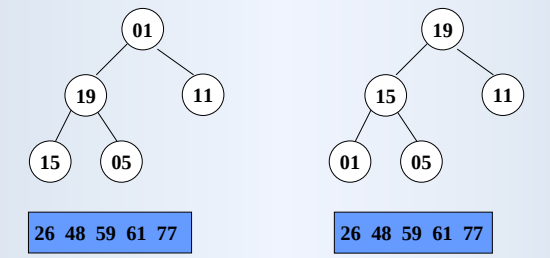
(11) 结点59与结点05互换　(12) 重建堆

46



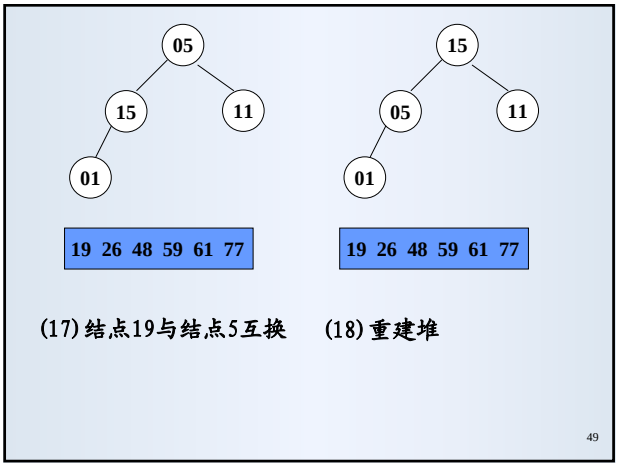
(12) 结点48与结点1互换　(13) 重建堆

47

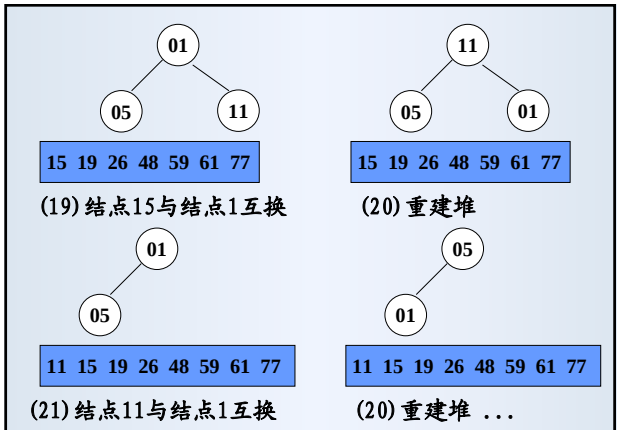


(15) 结点26与结点1互换　(16) 重建堆

48



(17) 结点19与结点5互换　(18) 重建堆



(19) 结点15与结点1互换　(20) 重建堆

(21) 结点11与结点1互换　(20) 重建堆 ...

算法：堆排序算法

```
void  heapSort(SortObject * pvector) {
   int i, n = pvector->n;
   RecordNode temp, *data = pvector->record;

   for (i = n/2-1; i >= 0; i--) /* 建立初始堆 */
      sift(pvector, i, n);      /* 对项i做筛选，在n-1范围内 */
   for (i = n-1; i > 0; i--) {  /* 进行n-1趟堆排序 */
      temp = data[0];           /* 当前堆顶和堆中最后记录互换 */
      data[0] = data[i];
      data[i] = temp;
      sift(pvector, 0, i);      /* 从R0到Ri-1重建堆 */
   }
}
```



```
void  sift(SortObject * pvector, int i, int n) {
   int cd;
   RecordNode *data = pvector->record, temp = data[i];
   for (cd = 2*i + 1; cd < n; ) { /*大于n时无子结点*/
      if (cd < n-1 && data[cd].key < data[cd+1].key)
         ++ cd; /* cd 指向Ri左右子女中排序码较大的结点 */
      if ( temp.key < data[cd].key ) {
         /* child换到父结点位置，继续调整*/
         data[i] = data[cd]; i = cd;  cd = 2*i + 1;
      }
      else break;          /* 调整结束 */
   }
   data[i] = temp;         /* 将记录Ri放入正确位置 */
}
```


初始建堆比较次数为：O(n)

排序中比较次数为： O(n *log* n)。（完全树高度是O(*log* n)）

移动次数小于比较次数。

最坏情况下的时间复杂度是O (n *log* n)。

且仅需一个记录大小的辅助空间。

堆排序是一种高效排序算法，适用于 n 值较大的情况。

堆排序不稳定（是其本质性弱点，堆序不保证不同分支中结点的大小关系。如不同分支中有关键码相同的结点，算法没法保证它们在最终排序序列里的位置关系）。

8.4 交换排序

基本观点：序列没排好序，是因为其中存在逆序。在发现逆序时交换有关记录的位置，得到的序列将更接近排序序列。

通过不断减少序列中的逆序，最终可以得到排序序列。

不同确定逆序的方式和交换方式，可以得到不同排序方法。

起泡排序是典型的交换排序。

许多教科书（包括本教科书）把快速排序归为交换排序，其实快速排序的基本想法是“划分”。

- 8.4.1 起泡排序
- 8.4.2 快速排序

8.4.1 起泡排序

基本想法:

- 顺序比较相邻记录，发现逆序就交换它们。
- 通过不断比较和交换，最终可以得到一个排序序列。

很容易证明:

只要每对相邻记录的顺序正确（前一记录不大于后一记录，假定要求按上升序排序），则整个序列为一个排序序列

这也是通过局部性质得到全局性质的一个实例

算法概要:

设待排序记录顺序存在$R_0,R_1,R_2,\ldots,R_{n-1}$中;

- 顺序比较(R_0,R_1)，$(R_1,R_2),\ldots,(R_{n-2},R_{n-1})$，遇到相邻记录的顺序不对则交换它们。一遍比较和交换的结果，将保证最大记录移到 R_{n-1}。称为一次起泡
- 再对存放在$R_0,R_1,R_2,\ldots,R_{n-2}$中n-1个记录作同样处理。结果将保证次大记录移到 R_{n-2}
- ……
- n-1次起泡一定能完成排序
- 可以增加一个标志noswap，用于记录本次起泡是否进行了交换。若无交换则表示排序已经完成

初始	1	2	3	4	5	6
49	38	38	38	38	13	13
38	49	49	49	13	27	27
65	65	65	13	27	38	38
97	76	13	27	49	49	49
76	13	27	49	49	49	49
13	27	49	65	65	65	65
27	49	76	76	76	76	76
49	97	97	97	97	97	97


```
void  bubbleSort(SortObject * pvector) {
   int i, j, noswap;
   RecordNode temp, *data = pvector->record;

   for(i = 0; i < pvector->n-1; i++) {        /* 做n-1次起泡 */
      noswap = TRUE;                          /* 置交换标志 */
      for (j = 0; j < pvector->n-i-1; j++)    /* 从前向后扫描 */
         if (data[j+1].key < data[j].key) {  /* 交换记录 */
            temp = data[j];
            data[j] = data[j+1];
            data[j+1] = temp;
            noswap = FALSE;
         }
      if ( noswap ) break; /* 一遍起泡未发生交换，算法结束 */
   }
}
```


起泡排序的性质:

- 最坏时间复杂度为$O(n^2)$，平均时间复杂度为$O(n^2)$。最好情况时间为 O(n)。
- 起泡排序算法中的辅助空间是 O(1)。
- 这一起泡排序算法是稳定的。
- 起泡排序算法具有适应性。

一遍起泡中“大记录”可以移动很远，小记录只移动一个位置

采用往复起泡可能改变这种情况，有时可以提高效率。但不会改变算法复杂性的量级

8.4.2 快速排序

在基于关键码比较的各种内排序算法中，快速排序是实践中平均速度最快的算法。

快速排序算法在 1960 年前后由英国计算机科学家 C.A.R. Hoare提出，作为最早的用递归描述的优美算法，展示了用递归方式描述算法的威力。

快速排序算法被说成是“20世纪Top-10最具影响力的算法”

快速排序的基本思想是划分:

- 设法把被排序序列按某种标准分为大小两组
- 而后可以递归地分别对两组记录采用同样方式排序
- 划分到每个子部分只包含一个记录，整个序列的排序完成

快速排序的数组实现：

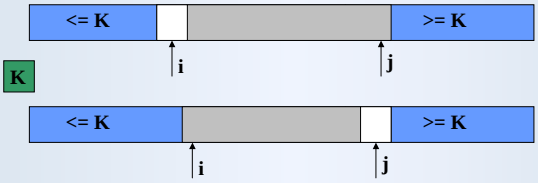
- 需要在数组内部完成排序，使用尽可能少的辅助空间
- 最简单的划分方式是取序列中第一个记录，以它的关键码为标准，把关键码小的记录移到数组一边，关键码大的记录移到数组另一边
- 一次划分完毕后，中间空位就是作为标准的记录的位置
- 而后对两边的记录序列用同样方式分别处理（递归）

存在不同的选择标准和移动记录的方式，形成了记录数组的快速排序的不同实现

不同的书籍上给出的算法可能不同，但基本思想一样，都是基于关键码的记录划分（分组）

一遍排序的一种做法：

- 设指针 i 和 j，初值分别是序列第一个和最后记录的位置；
- 取出第一个记录，设其排序码为 K（划分标准）
- 从 j 所指位置起向前搜索，找到第一个关键字小于 K 的记录并将其存入前面空位；从 i 所指位置起向后搜索，找到第一个关键字大于 K 的记录并将其存入上一步留下的空位
- 重复地交替进行上述两个动作直到 i 不小于 j 为止




```
void quickSort(SortObject * pvector, int l, int r) {
   int i, j;
   RecordNode temp, *data = pvector->record;
   if (l >= r) return;  /* 只有一个记录或无记录，则无须排序 */

   for (i = l,  j = r,  temp = data[i]; i < j; ) {/* 找Ri的最终位置 */
      while( i < j && data[j].key >= temp.key)
         j--;  /* 向左扫描找排序码小于temp.key的记录 */
      if (i < j) data[i++] = data[j];
      while( i < j && data[i].key <= temp.key)
         i++; /* 向右扫描找排序码大于temp.key的记录 */
      if (i < j) data[j--] = data[i];
   }
   data[i] = temp;                    /* 将 Ri 存入其最终位置 */
   quickSort(pvector, l, i-1);        /* 递归处理左区间 */
   quickSort(pvector, i+1, r);        /* 递归处理右区间 */
}
```


算法分析：

快速排序的记录移动次数不大于比较次数，所以其最坏时间复杂度应为$O(n^2)$，最坏情况出现在待排序序列为有序时。

最好时间复杂度为$O(n \log_2 n)$，如果每次划分能把序列分为长度差不多的两段，就可以得到 O(n log n)。

为减少最坏情况的出现，可采用“三者取中”规则，每趟划分前，比较R[l].key、R[r].key和R[(l+r)/2].key的大小，取中间的记录与R[l]交换，用它的关键码划分。

快速排序的平均时间复杂度是$T(n) = O(n\log_2 n)$。

算法需要栈空间实现递归。栈大小取决于递归深度，最多不超过n。若每次选较大一半进栈，处理短的一半，递归深度将不超过$\log_2 n$，所以快速排序的辅助空间为$O(\log_2 n)$。

常见（包括这里的）快速排序算法是不稳定的（但也有人提出了稳定的快速排序算法）。

8.5 分配排序

分配排序的思想是把排序码分解成若干部分，然后通过对各个部分排序码的分别排序，最终实现对整个排序码的排序。

书上实例：扑克牌排序（花色和点数）。先按点数分13组，收集到一起；后按花色分4组，再收集到一起。

另一实例：报纸排序（年、月、日），先按日期分31组；收集起来后再按月分12组；再收集起来后按年分组。

注意：分组时应维持当时现有的顺序。

- 8.5.1 概述
- 8.5.2 基数排序

8.5.1 概述

假设文件 F 里有n个记录， $F=(R_0, R_1, \ldots R_{n-1})$

记录 R_i 的排序码包含 d 个部分 $(k_i^0, k_i^1, \ldots, k_i^{d-1})$，文件 F 对排序码 $(k^0, k^1, \ldots, k^{d-1})$ 有序是指：对于文件中的任意两个记录 R_i 和 R_j $(0 \leqslant i \leqslant j \leqslant n-1)$，其关键码都满足词典序：

$$(k_i^0, k_i^1, \ldots, k_i^{d-1}) < (k_j^0, k_j^1, \ldots, k_j^{d-1})$$

这里的 k^0 称为最高位排序码，k^{d-1} 称为最低位排序码。

实现分配排序有两种方法：

1）从最高位排序码 k^0 开始，逐个针对各个排序码排序，这种方法称为高位优先法。

2）从最低位排序码k^{d-1}开始排序，称为低位优先法。

低位优先法比高位优先法简单:

采用高位优先排序，按一位排序码排序，就是将文件分割成若干子文件。要完成整个文件的排序，后续工作是做各子文件独立排序。这样逐层分割，数据组织比较麻烦

低位优先排序不划分子文件，对每位排序码 K^i 工作时都以整个文件作为排序对象，通过若干次“分配”和“收集”实现。针对各 K^i (0 <= i <= d-2) 的排序必须用稳定的排序方法

下面介绍的基数排序，是一种链表排序

其中采用的是排序码低位优先法

对排序码进行分解形成的排序方法

8.5.2 基数排序算法

首先把排序码看成 d 元组：

$$K_i = (K_i^0, K_i^1, \ldots, K_i^{d-1})$$

其中每个 K_i 都是集合$\{C_0, C_1, \ldots, C_{r-1}\}$ $(C_0<C_1<\ldots<C_{r-1})$ 的元素，即 $C_0 \leqslant K_i^j \leqslant C_{r-1} (0 \leqslant i \leqslant n-1, 0 \leqslant j \leqslant d-1)$

r 称为基数，排序需要做 d 遍

排序过程中:

- 首先按 K_i^{d-1} 把记录从小到大分配到 r 个组里，后依次收集
- 再按 K_i^{d-2} 把记录分配到 r 个组里
- 如此反复，直到对 K_i^0 的分配、收集，便得到排好序的序列

假定关键码为3位4进制数:

301, 212, 031, 320, 122, 203, 113, 020, 333, 001, 100, 311

按最低位分为4组:

320, 020, 100 | 301, 031, 001, 311 | 212, 122 | 203, 113, 333

收集: 320, 020, 100, 301, 031, 001, 311, 212, 122, 203, 113, 333

按第2位分为4组:

100, 301, 001, 203 | 311, 212, 113 | 320, 020, 122 | 031, 333

收集: 100, 301, 001, 203, 311, 212, 113, 320, 020, 122, 031, 333

按最高位分为4组:

001, 020, 031 | 100, 113, 122 | 203, 212 | 301, 311, 320, 333

收集: 001, 020, 031, 100, 113, 122, 203, 212, 301, 311, 320, 333


```
/* 下面基数排序算法是一种链表排序 */

#define D 3             /* D为排序码的最大位数 */
#define R 10            /* R为基数 */
typedef int KeyType, DataType;

typedef struct Node RadixNode, *RadixList;
struct Node {           /* 单链表结点类型 */
   KeyType key[D];   /* 关键码是数组 */
   /* DataType info;*/
   RadixNode *next;
};

typedef struct QueueNode { /* 用队列表示各个子序列 */
   RadixNode *f;      /* 队列的头指针 */
   RadixNode *e;      /* 队列的尾指针 */
} Queue;
```



```
void radixSort(RadixList * plist, int d, int r) {
   int i,j,k;   RadixNode *p, *head = (*plist)->next;
   Queue queue[r]; /* r 元的队列数组 */
   for(j = d-1; j >= 0; j--) {  /* 进行d次分配和收集*/
      for(i = 0; i < r; i++)
         queue[i].f = queue[i].e = NULL; /* 清空队列 */
      for (p = head; p != NULL; p = p->next) {
         k = p->key[j];      /* 按排序码的第j个分量分配*/
         if (queue[k].f == NULL)
            queue[k].f = p;  /*队列k为空，设置队头指针 */
         else (queue[k].e)->next = p;/* 接到队列k 尾*/
         queue[k].e = p;
      }
      for (i = 0; queue[i].f == NULL; i++)
          ;   /*找到第一个非空队列*/
```



```
      head = queue[i].f;   /* head为收集链表的头指针*/
      p = queue[i].e;
      for(i++; i < r; i++)  /* 收集其他非空队列 */
         if (queue[i].f != NULL) {
            p->next = queue[i].f;
            p = queue[i].e;
         }
      p->next = NULL;
   }
   (*plist)->next = head;
}
```

对整数关键码可以用基数排序，方法是每次取关键码中一位十进制数字。也可根据一个二进制位每趟把记录分为两组

对字符串关键码可以每次取一个字符（但要考虑和处理字符串不等长的情况）

算法分析:

- 基数排序算法中，时间耗费主要在修改指针上
- 一趟排序的时间为O(r+n)。总共要进行d趟排序，基数排序的时间复杂度是T(n) = O(d*(r+n))
- 当n较大、d较小，基数排序非常有效
- 采用链表排序方法只修改链接指针，操作效率不受记录的信息量大小的影响
- 排序中每个记录中增加了一个next字段（链表指针），还增加了一个 queue 数组，故辅助空间为S(n) = O(n+r)
- 基数排序是稳定的

8.6 归并排序

归并是一种操作，它把两个或两个以上的有序序列合并成一个有序序列。利用归并的思想可以实现排序:

- 初始时，可以把待排序序列的 n 个记录看成 n 个有序子序列，每个子序列的长度为 1
- 把当时的有序子序列两两归并，可得到长度加倍，数目减少一半的一组有序子序列
- 重复做子序列的两两归并，最终得到长度为 n 的有序序列

这种方法称为二路归并排序，还可考虑三路归并或多路归并。

由于归并操作是顺序处理，其中对于数据的访问具有局部性，符合外存特点，因此归并排序方法大量用于外排序。

例：初始序列为25, 57, 48, 37, 12, 82, 75, 29, 16，用二路归并排序法完成序列的排序

```
初始序列          25 57  48 37  12 82  75 29  16
                   \ /    \ /    \ /    \ /    |
第一趟归并后      25 57  37 48  12 82  29 75  16
                     \      /      \      /    |
第二趟归并后      25 37 48 57    12 29 75 82   16
                        \             /        |
第三趟归并后      12 25 29 37 48 57 75 82      16
                              \                /
第四趟归并后          12 16 25 29 37 48 57 75 82
```

排序后的结果为：12, 16, 25, 29, 37, 48, 57, 75, 82

下面讨论数组的两路归并排序算法

8.6.1 数组的两路归并排序

数组归并排序分三层实现（用一个同样大的辅助数组）：

1. 最下层：实现数组中相邻的一对有序序列的归并，归并结果存入另一数组里的相同位置
2. 第二层：基于 1 实现对整个数组中各对有序序列的归并，所有归并结果存入另一数组
3. 最高层：在两个数组之间往复进行操作 2；完成一遍归并后交换两数组的地位并重复这一操作，直至数组里只有一个有序序列时排序完成

下面从最下层开始介绍排序算法

由于数组长度未必是2的幂，因此需要处理一些不规范的情况

这里的RecordNode是要排序的记录。


```
/* from[low .. m]和from[m+1 .. high]是两个有序段 */
/* 把它们归并到 to[low .. high] 一段中 */

void merge (RecordNode from[], RecordNode to[],
            int low, int m, int high) {
   int i = low, j = m + 1, k = low;
   while ( i <= m && j <= high ) {
      /* 反复复制两段的首记录中较小的一个 */
      if (from[i].key <= from[j].key)  to[k++] = from[i++];
      else to[k++] = from[j++];
   }
   while (i <= m)    /* 复制第一段剩余记录 */
      to[k++] = from[i++];
   while (j <= high) /* 复制第二段剩余记录 */
      to[k++] = from[j++];
}
```



```
/* 对 from 做一趟归并，结果放入 to。当前有序段长 len */
void mergePass(RecordNode from[], RecordNode to[],
                    int n, int len) {
   int i = 0, j;
   while(i + 2*len - 1 < n) {
      merge(from, to, i, i+len-1, i + 2*len - 1);
                  /* 归并长len的两个子段*/
      i += 2*len;
   }
   if (i + len - 1 < n - 1)  /* 剩下两段，后段长度小于 len */
      merge(from, to, i, i+len-1, n-1);
   else                      /* 剩下一段，将它复制到数组r1 */
      for(j = i; j < n; j++) to[j] = from[j];
}
```

```
void mergeSort(SortObject * pvector) {
   RecordNode record[MAXNUM];
   int length = 1;

   while (length < pvector->n) {
      /* 一趟归并，结果存入辅助数组record */
      mergePass(pvector->record, record, pvector->n, length);
      length *= 2;
      /* 另一趟归并，结果存回原位 */
      mergePass(record, pvector->record, pvector->n, length);
      length *= 2;
   }
}
```


算法分析:

- 时间复杂度: 做了第 i 遍归并后，有序子序列的长度为 2^i，因此完成排序需要做不多于 $\log n+1$ 遍归并，每遍归并做O(n)次比较，总比较和移动次数都为 $O(n\log_2 n)$
- 空间复杂度: 和待排记录序列等量的空间（空间开销大）
- 二路归并算法是稳定的（这里关注了归并相同排序码的记录时的选择顺序）
- 一般情况下较少用于内部排序
- 容易实现基于归并的外存排序算法

8.6.2 基于归并排序算法的外排序（简介）

若待排序的记录存在外存文件，由于数据量太大无法同时全部放入内存，就需要考虑适合这种情况的排序算法。

归并排序的特点是顺序处理一系列（排序）记录并顺序生成更长的排序序列，适合外存顺序访问效率高、随机访问效率低的特点，因此成为实现外排序算法的基础。

最简单的外排序算法是照搬前面介绍的两路归并算法:

- 准备: 读入一组组适当数量的页块，在内存完成页块组内记录排序，将一组页块写入一个临时文件（内容已排序）
- 归并: 同时读入两个排序文件，读入过程中用归并方法生成一个更大的排序记录序列并写入另一临时排序文件。一遍归并使所有文件的长度加大一倍，数量减少一半
- 反复归并，直至所有记录都在一个排序文件里

排序方法	最坏时间复杂度	平均时间复杂度	辅助空间	稳定性
直接插入排序	$O(n^2)$	$O(n^2)$	O(1)	稳定
二分法插入排序	$O(n^2)$	$O(n^2)$	O(1)	稳定
表插入排序	$O(n^2)$	$O(n^2)$	O(n)	稳定
Shell排序	$O(n^{1.3})$	$O(n^{1.3})$	O(1)	不稳定
直接选择排序	$O(n^2)$	$O(n^2)$	O(1)	不稳定
堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	O(1)	不稳定
起泡排序	$O(n^2)$	$O(n^2)$	O(1)	稳定
快速排序	$O(n^2)$	$O(n\log_2 n)$	$O(\log_2 n)$	不稳定
基数排序	O(d(n+r))	O(d(n+r))	O(r+n)	稳定
归并排序	$O(n\log_2 n)$	$O(n\log_2 n)$	O(n)	稳定

结论:

1）平均时间性能：实践中快速排序法的平均速度最快，但其最坏情况是平方的，这方面不如堆排序和归并排序；在 n 较大时归并排序可能比堆排序快，但需要很大辅助空间。

2）朴素排序算法中直接插入排序最简单，当序列中的记录“基本有序”或n值较小时，应优先选用的直接插入排序。这种排序算法也常与其他排序方法结合使用（例如，一些实用快速排序算法在划分出的分段很短时，常转而用插入排序）。

3）当序列接近有序时，直接插入排序速度很快，起泡排序常常也很快（可能受到“小记录”与最终位置距离的影响）。

4）基数排序最适用于 n 值很大而关键字取值范围较小的序列。若关键字取值范围很大，而序列中大多数记录的“最高位关键字”均不同，则也可以先按“最高位关键字”不同将序列分成若干个子序列，而后用直接插入排序。

4）稳定性：大部分时间复杂度为$O(n^2)$的简单排序法都是稳定的，归并排序和基数排序是稳定的排序方法。然而，大部分时间性能较好的排序都是不稳定的，如快速排序、堆排序和希尔排序等。一般来说，排序过程中的比较是在相邻的两个记录关键字之间进行的排序方法是稳定的。

实际中记录的主关键字（例如各种唯一标示码，学号、身份证编号等）通常具有唯一性。如果是按主关键字排序，所用排序方法是否稳定就无关紧要。

如果需要按记录的次关键字（其他可能作为排序比较用的关键码，例如姓名，籍贯，年龄，成绩，）排序，则应根据问题所需慎重选择，有时需要用稳定算法。如果用了不稳定的排序算法，可能还需要对具有相同关键码的记录段再做排序。