第五章 排序

part 1: 基本概念与插入排序

张史梁
slzhang.jdl@pku.edu.cn

排序

- 排序是数据处理中经常使用的一种重要运算，
- 如何进行排序，特别是高效率排序，是计算机领域的重要课题之一

- 研究问题:
  - 如何进行排序
  - 如何进行高效率排序

排序

- 教学目的：
  - 排序方法的基本思想、排序过程、算法实现
  - 时间和空间性能的分析
  - 各种排序方法的比较和选择
- 教学重点：
  - 掌握插入排序、交换排序的归并排序
- 教学难点：
  - 排序算法的实现和应用

内容提要

- 排序的基本概念
- 插入排序
- 选择排序
- 交换排序
- 分配排序
- 归并排序

排序的基本概念

- 排序：
  - 设 $\{R_0,R_1,\ldots,R_{n-1}\}$ 是由n个记录组成的文件，
    $\{K_0,K_1,\ldots,K_{n-1}\}$ 是排序码集合，
    排序是将记录按排序码非递增(或非递减)的次序排列
- 排序码：记录中的一个(或多个)字段。
  - 排序码可以是关键码---此时按关键码排序；
  - 可能多个记录具有相同的排序码，排序的结果不唯一
  - 排序码的类型可以是整数类型，也可以是字符类型等
    （本章假设排序码为整形）

排序的基本概念

- 正序与逆序
  - “正序”序列：待排序序列正好符合排序要求
  - “逆序”序列：把待排序序列逆转过来，正好符合排序要求
  - 譬如，需要得到一个非递减序列？

- 正序：

- 逆序：

排序的基本概念

- 排序的稳定与不稳定：
  - 在待排序的文件中，若存在多个排序码相同的记录，
  - 经过排序后记录的相对次序保持不变，则这种排序方法称为是“稳定的”；
  - 否则，是“不稳定的”(只需列举出一组关键字说明不稳定即可)
  - 譬如：对下列数据进行排序，得到一个递增序列

23	19’	55	97	19	80

排序过程中出现

19’	19	23	97	55	80

排序的种类

- 按排序方法：
  - 插入排序、选择排序、交换排序、分配排序、归并排序
- 按排序中涉及的存储器不同：
  - 内排序：待排序记录在排序过程中全部存放在内存的
  - 外排序：如果排序过程中需要使用外存的

注：本章讨论的都是内排序的方法，但有些方法（特别是归并排序的思想）也可以用于外排序

排序的基本操作

- 比较两个排序码的大小
- 将一个记录从一个位置移动到另一个位置

排序算法的评价

- 评价排序算法好坏的标准
  - 执行算法所需的时间
  - 执行算法所需要的附加空间
  - 算法本身的复杂程度也是考虑的一个因素

- 执行算法所需的时间
  - 注：排序的时间开销是算法好坏的最重要的标志
  - 排序的时间开销衡量标准：
    - 算法执行中的比较次数
    - 算法执行中的移动次数

排序算法的评价-续

- 算法的时间复杂性
  - 一般按最坏情况或平均情况估算（在应用时要根据情况计算实际的开销，选择合适的算法）
  - 排序的两个基本操作：比较和交换

比较两个排序码（如23和11）的大小
交换两个记录（如23和11）的位置

- 算法的空间复杂性
  - 执行排序算法所需的附加空间一般不大，一般只给出结论

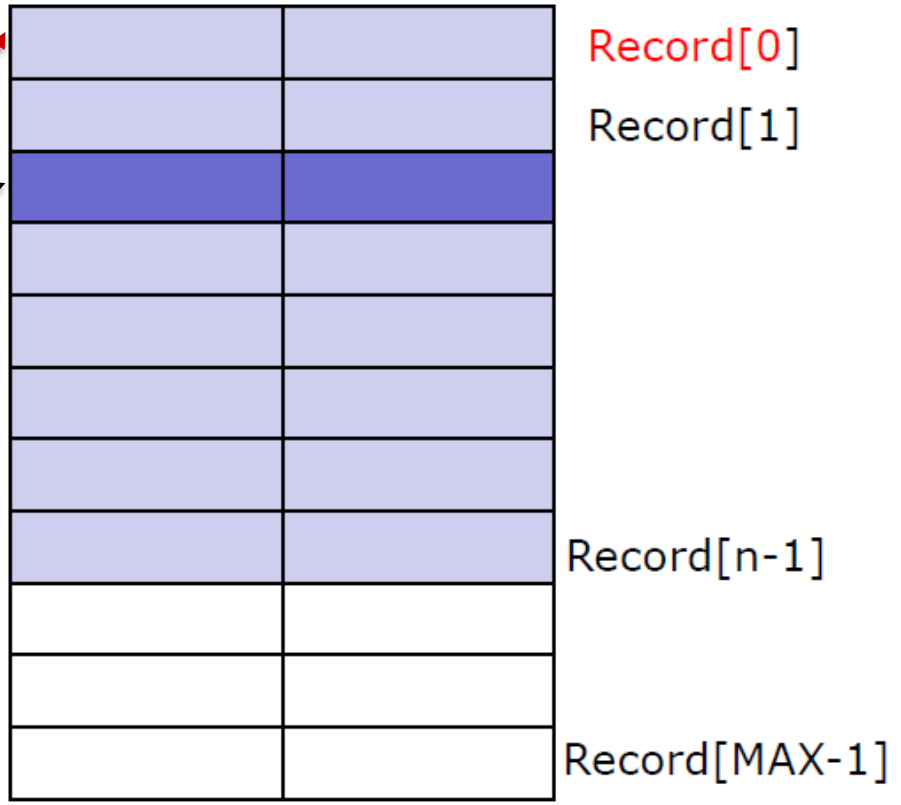
排序算法介绍

插入排序	1 直接插入排序 2 二分法插入排序 3 表插入排序 4 Shell排序
选择排序	5 直接选择排序 6 堆排序
交换排序	7 起泡排序 8 快速排序
分配排序	9 基数排序
归并排序	10 二路归并排序

本章假设-记录的数据结构

```
typedef int KeyType;
typedef int DataType;
typedef struct
{
    KeyType key; /* 排序码字段 */
    DataType info; /* 其他字段*/
}RecordNode;

typedef struct
{
    RecordNode record[MAX];
    int n;  // 记录个数，n<MAX
}SortObject;
```



内容提要

- 排序的基本概念
- 插入排序
- 选择排序
- 交换排序
- 分配排序
- 归并排序

插入排序

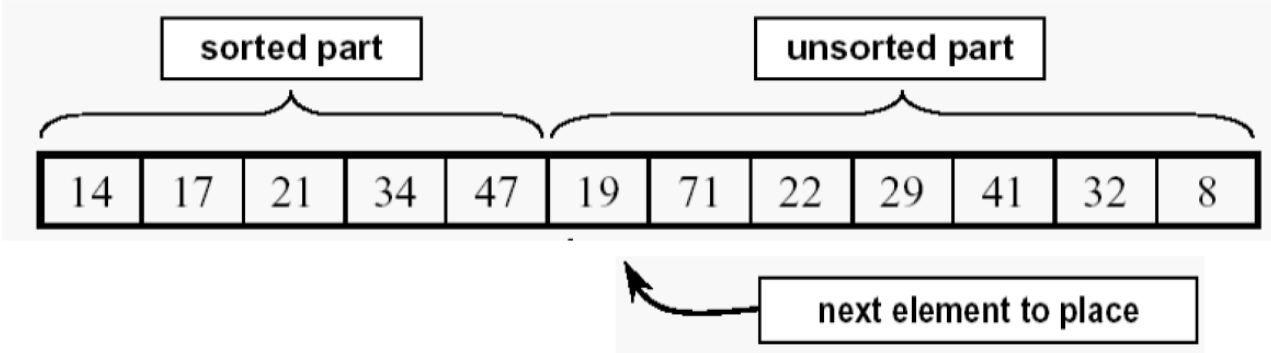
- 插入排序的基本方法：

  每一步将一个待排序的记录，按其排序码大小插到前面已经排序的文件中的适当位置，直到全部插入完为止。

  - 直接插入排序
  - 二分法插入排序
  - 表插入排序
  - Shell排序

直接插入排序

- 基本思想:
  - 逐个处理待排序的记录。每步将一个待排序的元素R_i按其排序码K_i大小插入到前面已排序表中的适当位置，直到全部插入完为止



直接插入排序

- 方法:
  - 假设待排序的n个记录$\{R_0,R_1,\ldots,R_{n-1}\}$存放在数组中，插入记录$R_i$时，记录集合被划分为两个区间$[R_0，R_{i-1}]$和$[R_i，R_{n-1}]$
  - 其中，前一个子区间已经排好序，后一个子区间是当前未排序的部分
  - 将排序码K_i与K_{i-1}，K_{i-2}，…，K_0依次比较，找出应该插入的位置，将记录R_i插入，原位置的记录向后顺移
- 直接插入排序采用顺序存储结构

示例

{23，11，55，97，19，80}

第一趟：{23}，11，55，97，19，80　　11

{11，23}

第二趟：{11，23}，55，97，19，80　　55

{11，23，55}

第三趟：{11，23，55}，97，19，80　　97

{11，23，55，97}

第四趟：{11，23，55，97}，19，80　　19

{11，19，23，55，97}

第五趟：{11，19，23，55，97}，80　　80

{11，19，23，55，80，97}

直接插入排序算法 （指针）

```
①  void  insertSort(SortObject  * pvector)
②  {
③       int  i,  j;                    /* 按递增序进行直接插入排序*/
④       RecordNode        temp;
⑤      for ( i = 1; i < pvector->n; i++ )    //从第二个元素开始
⑥      {
⑦          temp = pvector->record[i];
⑧          j = i-1;                          //从i 的前一个元素开始
⑨          while ((temp.key < pvector->record[j].key) && (j>=0) )
⑩          {
⑪                  pvector->record[j+1] = pvector->record[j];
⑫                  j--;
⑬            }
⑭          if( j!=(i-1) )      pvector->record[j+1] = temp;    //插入位置j+1
⑮      }
⑯  }
```

教学网提供代码：insertsort

直接插入排序算法 （数组）

```
① void ImprovedInsertSort (Record Array[], int n)
② {
③     Record Temp;              // 临时变量
④     for (int i=1; i<n; i++)
⑤     {
⑥         TempRecord = Array[i];
⑦         int j = i-1;                    //寻找记录 i 的正确位置
⑧          while ((j>=0) && (Temp < Array[j]))
⑨         {
⑩               Array[j+1] = Array[j];
⑪               j = j - 1;
⑫          }
⑬         //此时 j 后面就是记录 i 的正确位置，回填
⑭         if( j!=(i-1) )  Array[j+1] = TempRecord;
⑮     }
⑯ }
```

直接插入排序算法评价1

- 插入记录R_i时，比较次数C_i
  - 最多为i次，R_i被排在第一个记录的位置上；
  - 最少为1次，此时R_i的位置不变

$$C_{\max} = \sum_{i=1}^{n-1} i = \frac{1}{2} n(n-1) \approx \frac{n^2}{2} \qquad C_{\min} = n - 1 \approx n$$

- 移动次数M_i
  - 最大值为 i+2 (2: R_i->temp, temp->R_{j+1}) 次，
  - 最小值1次，包括算法中temp 的移动次数

$$M_{\max} = \sum_{i=1}^{n-1} (i+2) \approx \frac{n^2}{2} \qquad M_{\min} = n - 1 \approx n$$

直接插入排序算法评价2

- 平均情况下的比较次数：
  - 假定初始位置为K的元素，其插入的位置为j，而j的取值只能为1到K之间的任意一个。在找到最终的位置j之前需要进行次K-j+1比较，平均来说，放置第K个元素的比较次数为：

$$\frac{1}{K}\sum_{j=1}^{K}(K-j+1)=\frac{1}{K}\left[K^2-\frac{K(K+1)}{2}+K\right]=\frac{K+1}{2}$$

  - 故而，对N个元素的集合进行插入排序，平均总代价为：

$$\sum_{k=2}^{N}\left(\frac{K+1}{2}\right)=\sum_{k=1}^{N-1}\left(\frac{K+2}{2}\right)=\frac{1}{2}\left[\frac{(N+1)(N+2)}{2}\right]=\frac{1}{4}N^2+\frac{3}{4}N-1$$

直接插入排序算法评价3

- 平均情况下的移动次数：
  - 与比较的分析类似，唯一区别在于第K个元素的最终放置位置j 为[1..K-1]之间的某个值，故移动次数为K-j+2
  - 特例为元素不移动（仍在第K个位置上）：
  - 同比较的处理方式，可得到平均总的移动次数为：

$$\sum_{k=2}^{N}\left(\frac{K+3}{2}-\frac{2}{K}\right)<\sum_{k=1}^{N-1}\left(\frac{K+4}{2}\right)=\frac{1}{4}N^2+\frac{7}{4}N+3$$

直接插入排序算法评价4

- 文件初态不同时，直接插入排序所耗费的时间有很大差异
  - 若文件初态为正序，则算法的时间复杂度为O(n)
  - 若初态为反序，则时间复杂度为$O(n^2)$
  - 直接插入排序算法的时间复杂度为 $T(n) = O(n^2)$
- 算法引入了一个附加的记录空间temp，因此辅助空间为$S(n) = O(1)$
- 直接插入排序算法是稳定的

插入排序

- 插入排序的基本方法：

  每一步将一个待排序的记录，按其排序码大小插到前面已经排序的文件中的适当位置，直到全部插入完为止。

  - 直接插入排序
  - 二分法插入排序
  - 表插入排序
  - Shell排序

二分法插入排序

- 直接插入排序的算法简洁，容易实现，N较小时是一种很好的排序方法
- 通常文件中记录的数量都很大，则此时直接插入排序方法不适用
- 在直接插入排序的基础上减少比较次数，即在插入R_i时改用二分法比较找插入位置，便得到二分法插入排序

二分检索算法

- 分别用low和high表示当前查找区间的下界和上界
- 例如：在下列待检索字典中检索关键码为25和78的元素

0	1	2	3	4	5	6	7	8	9	10
05,	10,	18,	25,	27,	32,	41,	51,	68,	73,	99

教学网提供代码：二分检索

二分插入排序

- 基本思想：
  - 在直接插入排序的基础上减少比较的次数，即在插入R_i时改用二分法找插入位置。
  - 插入记录R_i时，记录集合中子区间[R_0，R_{i-1}]已经有序
  - Low =0; right =i-1; mid= (low+right)/2 带入二分检索
  - 找出应该插入的位置；
  - 将原位置的记录向后顺移，将记录R_i插入。
- 二分插入排序采用顺序存储结构

二分插入排序

```
void binSort(SortObject * pvector) /* 按递增序进行二分法插入排序 */
{
   int i, j, left, mid, right;
   RecordNode temp;
   for( i = 1; i < pvector->n; i++ )
   {
      temp = pvector->record[i];
      left = 0; right = i - 1;  /*置已排序区间的上、下界初值 */
      while (left <= right)  // 查找终止条件
      {
         mid=(left+right)/2;  /*mid指向已排序区间的中间位置*/
         if (temp.key<pvector->record[mid].key)
            right=mid-1;   /*插入元素应在左子区间*/
         else
            left=mid+1;     /*插入元素应在右子区间*/
      }                      // left就是要放的位置
      for(j=i-1; j>=left; j--)
         pvector->record[j+1] = pvector->record[j];
                /*将排序码大于Ki的记录后移*/
      if(left!=i) pvector->record[left] = temp;
   }
}
```

示例

初始序列为49, 38, 65, 97, 76, 13, 27, 49’，用二分插入排序法排序

采用二分法插入排序经过6趟排序后，序列的前7个元素组成一个有序序列，下面是第7趟排序的过程：

(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

二分插入排序的比较次数

- 二分插入排序的比较次数与待排序记录的初始状态无关，仅依赖于记录的个数，适应性差。插入第i个记录时
  - 如果 $i = 2^j (0 \le j \le \lfloor \log_2 n \rfloor$ 则无论排序码的大小，都恰好经过 j 次比较才能确定插入位置
  - 如果，$2^j < i \le 2^{j+1}$ 则比较次数为 j+1 （2^j个j+1）
  - 因此，将 n($n=2^k$) 个记录排序的总比较次数为

$$\sum_{i=1}^{n} \lceil \log_2 i \rceil = 0+1+2+2+\ldots+k+k+\ldots+k$$

（2^0个1，2^1个2，2^{k-1}个k）

$$= n\log_2 n - n + 1$$

$$\approx n\log_2 n$$

二分法插入排序方法性能分析

- 比较次数
  - 当n较大时，比直接插入排序的最大比较次数少得多。但最小比较次数大于直接插入排序的最小比较次数
- 算法的移动次数与直接插入排序算法的相同
  - 最坏为$O(n^2)$；最好为$O(n)$；平均移动次数为$O(n^2)$

- 二分法插入排序的平均时间复杂度为$T(n)= O(n^2)$
- 算法的辅助空间为$S(n)=O(1)$
  - 算法中增加了一个辅助空间temp
- 二分插入排序法是稳定的

插入排序

- 插入排序的基本方法：

  每一步将一个待排序的记录，按其排序码大小插到前面已经排序的文件中的适当位置，直到全部插入完为止。

  - 直接插入排序
  - 二分法插入排序
  - 表插入排序
  - shell排序

表插入排序

- 表插入排序是在直接插入排序的基础上减少移动次数
- 基本思想:
  - 在记录中设置一个指针字段，记录用链表连接
  - 插入记录R_i时，记录R_0至R_{i-1}已经排序，先将记录R_i脱链
  - 再采用顺序比较的方法找到R_i应插入的位置，将R_i插入链表（和直接插入查找顺序相反）

示例

例题：初始序列为49, 38, 65, 97, 76, 13, 27, 49’，
　　用表插入排序法排序

初始链表为

　　49→38→65→97→76→13→27→49’

pre↑now↑

(1) 插入第2个记录R_1

　　38→49→65→97→76→13→27→49’

　　pre↑now↑

设置指针指向前驱！

示例

(2) 插入第3个记录R_2

38→49→65→97→76→13→27→49’

pre↑now↑

(3) 插入第4个记录R_3

38→49→65→97→76→13→27→49’

pre↑now↑

(4) 插入第5个记录R_4

38→49→65→76→97→ 13→27→49’

pre↑now↑

示例

(5) 插入第6个记录R_5

13→38→49→65→76→97→27→49’

pre↑now↑

(6) 插入第7个记录R_6

13→27→38→49→65→76→97→ 49’

pre↑now↑

(7) 插入第8个记录R_7

13→27→38→49→49’→65→76→ 97

pre↑

表插入排序算法

- 表插入算法中记录的数据结构

```
struct Node;    /* 单链表结点类型 */
typedef struct Node ListNode;
struct Node
{  KeyType key;         /* 排序码字段 */
   DataType info;       /* 记录的其它字段 */
   ListNode *next;      /* 记录的指针字段 */
};
typedef ListNode * LinkList;
```

```
void listSort(LinkList * plist)   /* 链表中第一个结点为头结点。 */
{
   ListNode *now, *pre, *p, *q, *head;
   head=*plist;
   pre=head->next;
   if(pre==NULL) return; /* 为空链表 */
   now=pre->next;       // 从第二个数据开始
   while(now!=NULL) {
      q=head; p=head->next;
      while(p!=now && p->key<=now->key) {   //查找插入位置
         q=p;
         p=p->next;
       } /* 本循环结束时，已经找到了now的插入位置 */
      if(p==now)  {  /* now应放在原位置 */
         pre=pre->next;  now=pre->next;  continue;
       }
      pre->next=now->next;     /* 使now记录脱链 */
      q->next=now;
      now->next=p;       /* 将now记录插入链表中 */
      now=pre->next;
   }
}
```

教学网提供代码：表插入排序

表插入排序的算法性能分析

第i趟排序：最多比较次数i次，最少比较次数1次。

n-1趟总的比较次数：

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

最少： n-1

记录移动次数：0

时间效率： $O(n^2)$

辅助空间： O(n) [指针]

稳定性： p->key <= now->key 保证稳定的排序。

插入排序

- 插入排序的基本方法：

  每一步将一个待排序的记录，按其排序码大小插到前面已经排序的文件中的适当位置，直到全部插入完为止。

  - 直接插入排序
  - 二分法插入排序
  - 表插入排序
  - Shell排序

Shell（希尔）排序

- Shell排序法又称缩小增量法，由D.L.Shell在1959年提出，是对直接插入排序法的改进
- 从两个方面考虑

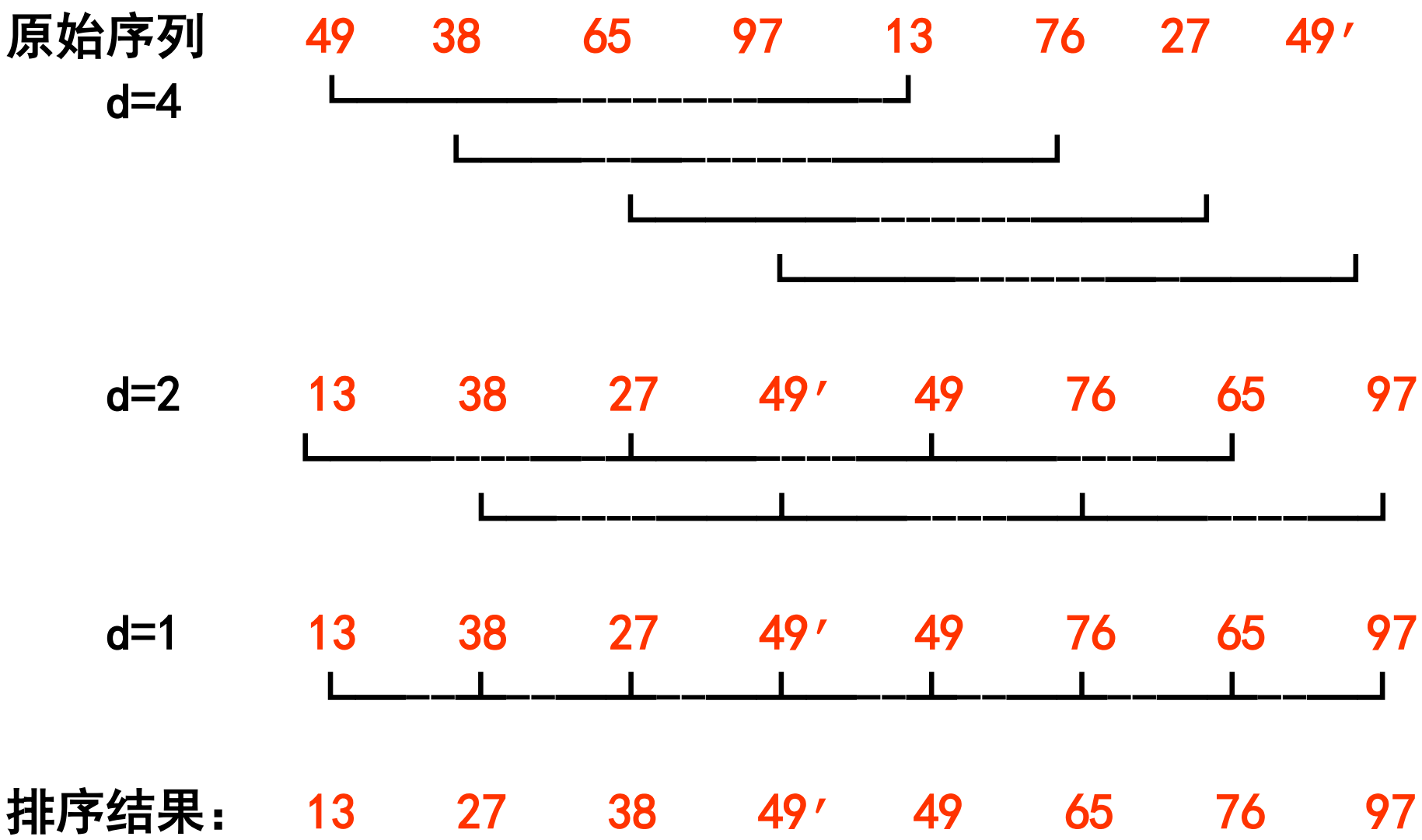
直接插入排序中，若初始序列基本有序时，大多数记录不需要插入，时间效率大大提高。

当记录数n较小时，n^2的值受n值影响不大。

Shell（希尔）排序

- 基本思想
  - 先取一个整数$d_1<n$，把全部记录分成d_1个组
  - 所有距离为d_1倍数的记录放在一组中，先在各组内排序
  - 然后取$d_2<d_1$重复上述分组和排序工作
  - 直到$d_i=1$，即所有记录放在一组中为止
- 各组内的排序可以采用直接插入法，也可以采用其它排序方法，如直接选择排序
- 对于增量的选择：
  - Shell提出 $d1=\left\lceil \frac{n}{2} \right\rceil, d_{i+1}=\left\lceil \frac{d_i}{2} \right\rceil$ ；Knuth提出 $d_{i+1}=\left\lceil \frac{d_i}{3} \right\rceil$
  - 还有其它选择，但何种最佳尚无法证明。
  - 无论如何，最后增量必须为1

示例：{49，38，65，97，13，76，27，49′}



```
void ShellSort(SortObject *pvector, int d)   /*按递增序列进行shell排序*/
{
    int i, j, increment;
   RecordNode temp;
   for (increment = d; increment > 0; increment /= 2)
                          /* increment为本趟shell排序增量 */
   {
       for (i = increment; i < pvector->n; i++) /* 每组直接插入排序 */
      {
          temp = pvector->record[i];    /* 保存待插入记录Ri*/
          j = i-increment;
         while (j >= 0 && temp.key < pvector->record[j].key)
                                          /*查找插入位置*/
         {
             pvector->record[j+increment] = pvector->record[j];
                                          /*记录后移*/
             j -= increment;
          }
         pvector->record[j+increment] = temp; /*插入记录Ri*/
      }
   }
}
```

教学网提供代码：Shell排序

Shell排序算法

```
17.  void ShellSort(Record Array[], int n)
18.  {
19.     int i, delta;    // 增量delta每次除以2递减
20.     for (delta = n/2; delta>0; delta /= 2)
21.          for (i = 0; i < delta; i++)
22.                    ModInsSort(Array , n, delta, i);
23.     // 如果增量序列不能保证最后delta间距为1
24.     // 可以安排下面这个扫尾性质的插入排序 ModInsSort(Array, n, 1);

25.  }
```

针对增量而修改的插入排序算法

```
①  void ModInsSort( Record Array[], int n, int delta, int k)
②  {                                          // 参数delta表示当前的增量
③     int i, j;
④    Record TempRecord ;
⑤     for (i = delta+k ; i < n; i += delta)   //从组内的第二个元素开始插入
⑥     {
⑦        TempRecord = Array[i];
⑧       int j = i-delta;                      //寻找记录 i 的正确位置
⑨        while ((j>=0) && (TempRecord < Array[j]))
⑩        {
⑪           Array[j+ delta] = Array[j];     //位置改变间隔为：delta
⑫           j = j - delta;
⑬        }
⑭        //此时 j 后面就是记录 i 的正确位置，回填
⑮        Array[j+ delta] = TempRecord;
⑯     }
```

Shell排序算法性能分析

- Shell排序算法的速度比直接插入排序快，其时间复杂度分析比较复杂，Shell排序的平均比较次数和平均移动次数都为$\mathbf{n^{1.3}}$左右

- Shell排序算法中增加了一个辅助空间temp，因此算法的辅助空间为S(n)=O(1)

- Shell排序是不稳定的

插入排序小结

- 直接插入排序简单，容易实现，但n较大时，时间效率低
- 所有的插入排序都首先确定位置，然后插入
- 其它插入排序方法都是从减少比较次数、移动次数出发对直接插入排序进行改进
  - 直接：顺序检索确定位置，记录移动实现插入；
  - 二分：减少比较次数（折半检索确定位置）；
  - 表插入：减少移动次数（采用链表存储）；
  - Shell排序：改变增量（当待排序序列基本有序，并且n较小时，提高了直接插入排序效率）；不稳定排序。

- 除Shell外，其它的插入排序的时间复杂度为$O(n^2)$，并且是稳定的

内容提要

- 排序的基本概念
- 插入排序
- 选择排序
- 交换排序
- 分配排序
- 归并排序