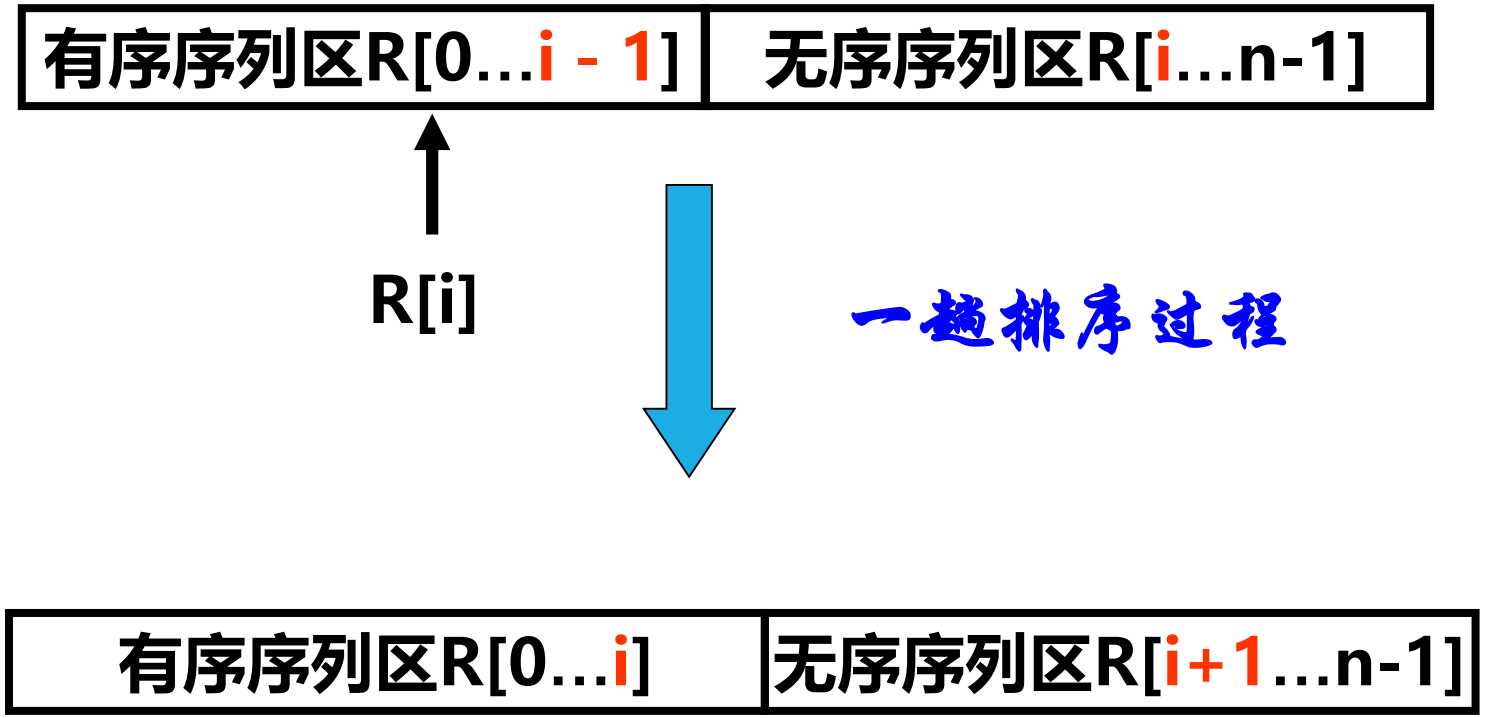
8.2 插入排序



直接插入排序

二分法插入排序

Shell排序

8.2 .1 直接插入排序

表 8-1 直接插入排序每趟结果

趟 \ 下标	0	1	2	3	4	5	6	7
初始序列	**15**	13(1)	9	46	4	18	13(2)	7
i = 1 (15)	**15**	13(1)	9	46	4	18	13(2)	7
i = 2 (13)	**13(1)**	**15**	9	46	4	18	13(2)	7
i = 3 (9)	**9**	**13(1)**	**15**	46	4	18	13(2)	7
i = 4 (46)	**9**	**13(1)**	**15**	**46**	4	18	13(2)	7
i = 5 (4)	**4**	**9**	**13(1)**	**15**	**46**	18	13(2)	7
i = 6 (18)	**4**	**9**	**13(1)**	**15**	**18**	**46**	13(2)	7
i = 7 (13)	**4**	**9**	**13(1)**	**13(2)**	**15**	**18**	**46**	7
i = 8 (7)	**4**	**7**	**9**	**13(1)**	**13(2)**	**15**	**18**	**46**

8.2 .1 直接插入排序

有序区　　无序区

i=8:　[4　9　13(1) 13(2) 15　18 46]　7

1. temp = sortArr->recordArr[i]
2. 寻找sortArr->recordArr[i]的插入位置（进行关键字的比较）；

   查找的方向由后向前进行；查找的同时记录向后移动；
3. 将sortArr->recordArr[i]复制到合适位置

算法8-2

```
1  void InsertSort(SortArr* sortArr) //直接插入排序
2  {
3          int i, j;
4          RecordType temp;
5          for( i = 1; i < sortArr->cnt; i++ )
6          {
7                  j = i-1; //j是已经排好顺序的数据最后一个元素下标
8                  temp = sortArr->recordArr[i]; //等待插入的数据temp
9                  //从j位置开始，从后向前在已经排好顺序的序列中找到插入位置
10                 while(temp.key < sortArr->recordArr[j].key && j>=0)
11                 {
12                         sortArr->recordArr[j+1] = sortArr->recordArr[j];
13                         j--;
14                 }
15                 //找到待插入位置为j+1
16                 //如果待插入位置正好就是要插入元素所在位置则可以不进行数据赋值
17                 if( j+1 != i )
18                 {
19                         sortArr->recordArr[j+1] = temp;
20                 }
21         } //end 5
22 }
```

直接插入算法分析

■ 若待排序记录按照关键字从小到大排序（正序）　1 2 3 4 5 6 7

比较次数　$C_{\min}=\sum_{i=1}^{n-1}1=n-1\approx \mathrm{n}$

移动次数　$M_{\min}=\sum_{i=1}^{n-1}1=(n-1)\approx \mathrm{n}$

O(n)

■ 若待排序记录按照关键字从大到小排序（逆序）　7 6 5 4 3 2 1

比较次数　$C_{\max}=\sum_{i=1}^{n-1}i=\frac{n(n-1)}{2}\approx \frac{\mathrm{n}^2}{2}$

移动次数　$M_{\max}=\sum_{i=1}^{n-1}(i+1)=\frac{n(n+1)}{2}\approx \frac{\mathrm{n}^2}{2}$

O(n²)

8.2 .1 直接插入排序

直接插入排序的特点

- 思路简单，算法简洁；
- 该排序法是稳定的；
  - 循环条件temp.key < sortArr->recordArr[j].key， 保证了算法的稳定性
- 适用于 n较小的排序；

8.2.2二分插入排序

i=8:　[4　9　13(1) 13(2) 15　18 46]　7

思考： 插入排序的基本操作是在**有序表**中进行查找合适的插入位置，直接插入是**顺序查找**sortArr->recordArr[i]的插入位置，时间消耗在比较次数上，有没有改进的方法呢?

yes! 在已形成的**有序表**中**折半查找**，并在适当位置插入，把原来位置上的元素向后顺移。由此实现的排序称为**二分插入排序**

优点：减少比较的次数

8.2.2二分插入排序

	low			mid			high	
Step1：	4	9	13(1)	13(2)	15	18	46	7

	low	mid	high					
Step2：	4	9	13(1)	13(2)	15	18	46	7

	low=mid=high=0							
Step3：	4	9	13(1)	13(2)	15	18	46	7

	low=1, mid=high=0							
Step4：	4	7	9	13(1)	13(2)	15	18	46

图 8-1 二分插入排序一趟过程

算法8-3

```
1  void BinSort(SortArr* sortArr) {  //二分插入排序
2          int i, j;  int low, mid, high ; RecordType temp;
3          for( i = 1; i < sortArr->cnt; i++ ) {
4                  temp = sortArr->recordArr[i];
5                  //二分查找法查找插入位置
6                  low = 0;  high = i-1;  //区间左右边界
7                  while (low <= high) {
8                          mid = (low + high)/2;
9                          if (temp.key < sortArr->recordArr[mid].key)
10                                 high = mid-1; //待排序的值比中间位置的值小，在前半区间查找
11                         else  low = mid+1;  //否则，在后半区间查找
12                 }
13                 //如果待插入数据正好在还要插入的位置上就不需要插入了
14                 if (low != i){
15                         //如果需要挪动数据，空出位置，插入数据
16                         //找到插入位置后，移动数据，空出地方给数据插入
17                         for (j = i-1;  j >= low;  j--)
18                                 sortArr->recordArr[j+1]= sortArr->recordArr[j];
19                         sortArr->recordArr[low] = temp; //插入数据
20                 }
21         }
22 }
```

8.2.2二分插入排序

- **空间效率**：同直接插入排序temp
- **时间效率**：
  - **比较次数**：在插入第 i 个对象时，需要经过$\lceil \log_2 i \rceil$次关键码比较，才能确定它应插入的位置。插入n个记录总的比较次数：

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

当n较大时，比直接插入排序的最大比较次数少得多；

  - **移动记录次数**：

最坏的情况为$n^2/2$；最好的情况为n；平均移动次数为$O(n^2)$

- **稳定排序**：temp.key < sortArr->recordArr[mid].key保证了算法的稳定性

8.2.2二分插入排序

若记录是链表结构，用直接插入排序行否？折半插入排序呢?

直接插入不仅可行，而且还无需移动元素，时间效率更高！

但链表无法“折半”！

8.2.3 Shell排序

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

基本思想：**先宏观调整后微观调整的思想**

先将整个待排记录序列分割成若干个子序列分别进行直接插入排序，待整个序列中的记录“基本有序“时，再对全体记录进行一次直接插入排序，就可以完成整个的排序工作

将n个记录分成d个子序列

R[1],R[1+d],R[1+2d]……..R[1+kd]

R[2],R[2+d],R[2+2d]…. ….R[2+kd]

…….

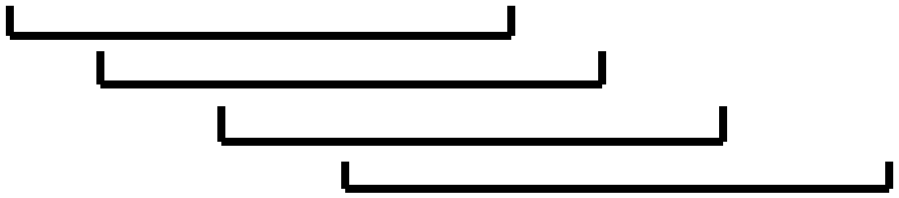
R[d],R[d+d],R[d+2d]…. .. R[kd] R[(1+k)d]

d为增量，d由大到小变化，直到d为1为止

8.2.23 Shell排序

shell排序演示

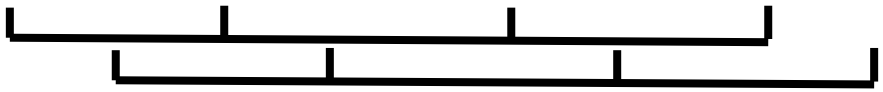
初始序列：7 18 46 15 13(1) 9 13(2) 4



d1=4分组：

一趟排序：7 9 13(2) 4 13(1) 18 46 15

d2=2分组：7 9 13(2) 4 13(1) 18 46 15



二趟排序：7 4 13(2) 9 13(1) 15 46 18

d3=1分组：7 4 13(2) 9 13(1) 15 46 18

三趟排序：4 7 9 13(2) 13(1) 15 18 46

8.2.3 Shell排序

表 8-2　shell 排序过程和每趟结果

下标 / 趟		0	1	2	3	4	5	6	7
初始序列		7	18	46	15	13(1)	9	13(2)	4
i = 1, 增量 d = 4	第一组	7				13(1)			
	第二组		18				9		
	第三组			46				13(2)	
	第四组				15				4
i = 1 排序后的结果		**7**	**9**	**13(2)**	**4**	**13(1)**	**18**	**46**	**15**
i = 2, 增量 d = 2	第一组	7		13(2)		13(1)		46	
	第二组		9		4		18		15
i = 2 排序后的结果		**7**	**4**	**13(2)**	**9**	**13(1)**	**15**	**46**	**18**
i = 3, 增量 d = 1	第一组	7	4	13(2)	9	13(1)	15	46	18
i = 3 排序后的结果		**4**	**7**	**9**	**13(2)**	**13(1)**	**15**	**18**	**46**

算法8-4

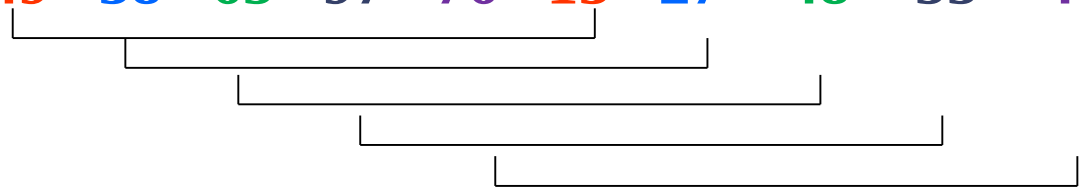
```
1  void ShellSort(SortArr *sortArr, int d) //shell排序
2   //d为初始的增量，以后每一趟为前一趟增量的一半
3  {
4          int i, j, increment; // increment记录当前趟的增量
5          RecordType temp;  //保存待排序记录
6          for (increment = d; increment>0; increment /= 2){
7                  for (i = increment; i<sortArr->cnt; i++){
8                          temp = sortArr->recordArr[i]; //保存待排序记录
9                          j = i - increment; //j按照增量进行变化
10                         while (j >= 0 && temp.key<sortArr->recordArr[j].key){
11                                 //记录按照增量间隔向后移动
12                                 sortArr->recordArr[j+increment]= sortArr->recordArr[j];
13                                 j -= increment; //j按照增量进行变化
14                         }
15                         sortArr->recordArr[j + increment]=temp; //插入待排序记录
16                 }//end 7
17         }//end 6
18 }
```

8.2.3 Shell排序：课堂练习，d的取值分别为5，3，1

实例 初始：49 38 65 97 76 13 27 48 55 4

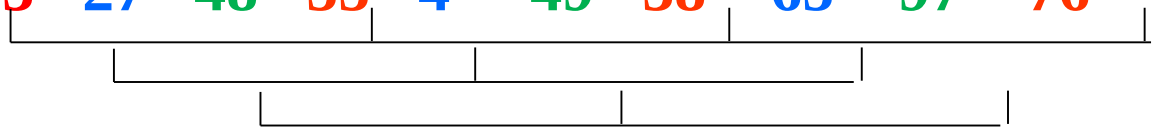
（7 18 46 15 13(1) 9 13(2) 4）

d1=5分组：49 38 65 97 76 13 27 48 55 4



一趟排序：13 27 48 55 4 49 38 65 97 76

d2=3分组：13 27 48 55 4 49 38 65 97 76



二趟排序：13 4 48 38 27 49 55 65 97 76

d3=1分组：13 27 48 55 4 49 38 65 97 76

三趟排序：4 13 27 38 48 49 55 65 76 97

算法分析

时间效率：

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

《The art of programming》《计算机程序设计艺术》

空间效率：

Shell排序算法中增加了一个辅助空间 temp

稳定性：

Shell排序是不稳定的

插入排序方法总结

时间复杂度:

直接插入排序$O(n^2)$　　如何对它改进?

减少比较次数：二分法插入排序$O(n^2)$；希尔排序$O(n^{1.3})$

稳定性:

稳定排序：直接插入、二分法插入

不稳定排序：希尔排序

空间复杂度：　S(n)=O(1)　temp