第7章 排序算法

知识要点

(1) 掌握排序的基本概念；
(2) 熟练掌握常见排序方法的基本思想；
(3) 了解不同的排序方法的时间复杂度和空间复杂度分析方法。

薄钧戈
2021年4月28日

7.1 排序的基本概念

- 假定文件由一组数据元素（或记录）组成，数据元素又由若干个数据项组成，其中用来唯一标识一个数据元素的数据项称为**关键字项**，该数据项的值称为**关键字**。
- **排序**就是将数据元素（或记录）的任意序列，按关键字重新排序成**有序序列**。

7.1 排序的基本概念

- 设有一组数据元素序列：$(R_1, R_2, R_3, \ldots, R_n)$
  - 对应的关键字分别为：$(K_1, K_2, K_3, \ldots, K_n)$
  - 将这组数据元素按关键字重新排序，序列为：

    $(R_1', R_2', R_3', \ldots, R_n')$
  - 使得 $(K_1' \le K_2' \le K_3' \le \ldots \le K_n')$ 或 $(K_1' \ge K_2' \ge K_3' \ge \ldots \ge K_n')$
- **排序算法**就是重新排列一组记录，使其关键字按非递增（或非递减）有序。

7.1 排序的基本概念

算法7.1：待排序记录的形式定义

```
1.  #define Element RecType
2.  const int DefaultSize=100;
3.  typedef struct RecType{
4.      keytype key;              //关键字码
5.      Itemtype otherinfo;   //记录中的其他数据
6.  }RecType;
```

```
8.  class recordList{//用顺序表来存储待排序的记录
9.  private:
10. Element  *R;
11. int MaxSize;
12. int CurrentSize;
13. public:
14. recordList(int MaxSz=DefaultSize){
15.         R=new Element[MaxSz];
16.         MaxSize=MaxSz;
17.         CurrentSize=0;
18. }
19. ~recordList(){delete [] R;}
20. }
```

7.1 排序的基本概念-稳定性

- 如果数据元素中的关键字是唯一的，则排序后所得序列也是唯一的。
- 如待排数据元素中存在多个相同的关键字，则排序结果可能不唯一。
  - 排序稳定性：如果$i < j$且$K_i = K_j$，经过排序后，R_i先于R_j，即具有相同关键字的数据元素的相对位置没有发生变化，称这种排序算法是稳定的。反之，则称算法是不稳定的。
  - 只有所有可能的待排序数据元素序列执行排序算法后得到的结果均符合稳定性要求时，该排序算法才是稳定的。
  - ★注意：排序算法的稳定性由算法本身决定，而不是由待排的数据元素序列决定的。

7.1 排序的基本概念-时间和空间复杂度

- 排序算法的时间复杂度通常考虑两个因素：
  - 关键字比较次数；
  - 记录在表中的移动次数。
  - 大多数排序方法都包含第一种操作，而第二种操作需要根据记录序列的存储方式来决定。
- 通常时间复杂度和空间复杂度（即所需的额外空间）是评价一个排序算法好坏的主要标准；
- 此外算法本身的复杂程度和稳定性也是需要考虑的因素。

7.1 排序的基本概念-存储方式

- 通常的存储方式有以下三种：
  - 以顺序表作为存储结构，即记录之间的次序关系是由其存储时的相对位置决定的，排序过程中必然会涉及到记录的移动。
  - 以链表作为存储结构，即记录之间的次序关系是由链表指针指示的。在排序过程中，仅需要修改指针，而不用移动记录。
  - 以二叉链表等树形存储，即查找表以二叉树形式存放，方便动态查找算法。比如：二叉查找树、平衡二叉树、树形选择排序、堆排序等。

7.1 排序的基本概念-分类

- 稳定排序VS.不稳定排序
- 直接排序VS.间接排序：
  - 直接排序时，直接移动记录，使得数据有序；
  - 为减少数据移动量，可以对指向各个元素的指针进行排序（指针排序算法）或者进行索引排序——间接排序

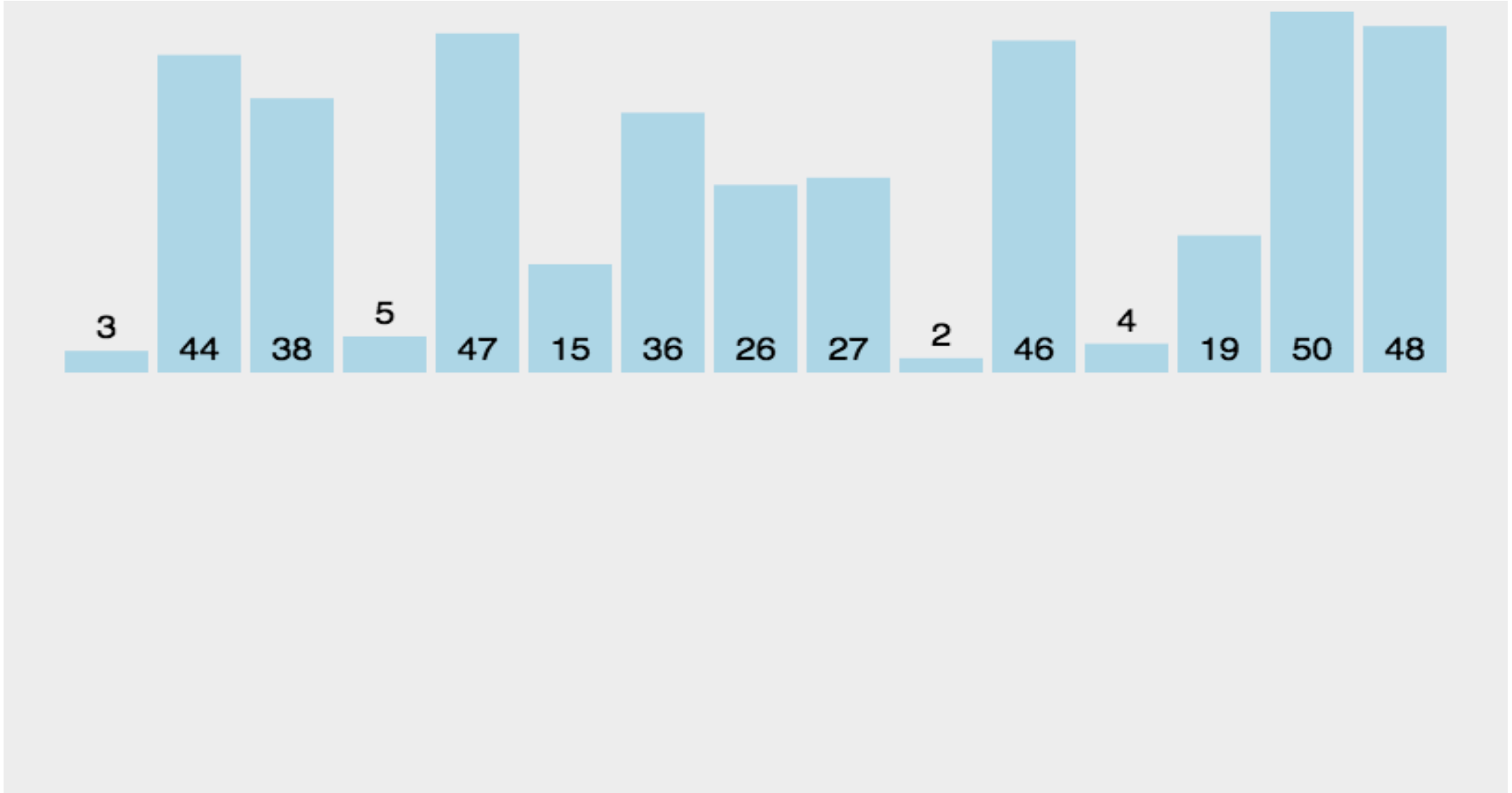
7.1 排序的基本概念-分类

- 内部排序VS.外部排序
  - 待排序列完全存放在内存中所进行的排序过程，适合数目不太多的元素序列——内部
  - 因待排序列元素足够多，不能完全放入内存，排序过程中还需访问外存储器——外部
- 原地(原址、就地)排序VS.非原地排序
  - 排序过程所需的辅助空间不依赖于问题的规模n，即辅助空间为O(1)——原地
  - 排序过程所需的辅助空间依赖于问题的规模n——非原地

7.2 简单排序

- 简单排序算法时间复杂度均为$O(n^2)$

7.2.1 简单插入排序



7.2.1 简单插入排序

- 简单插入排序通过顺序查找来确定待插入记录的位置。

① 如果已经有若干个记录按照非递减排好序，则将下一个待插入记录与已排好序的记录从后往前依次逐个进行比较。

② 直到找到第一个不大于待排序记录的值，从而将待插记录插入到其后的位置。

③ 按照上述方法把待排序记录序列全部插入到已经排好序的记录序列中。

7.2.1 简单插入排序-步骤

- 简单插入排序过程可分为三步：

定位 ① 寻找插入位置。查找R[i]在序列{R'[0]，……，R'[i-1]}中的插入位置j，需要满足R[j-1].key ≤ R[i].key < R[j].key。

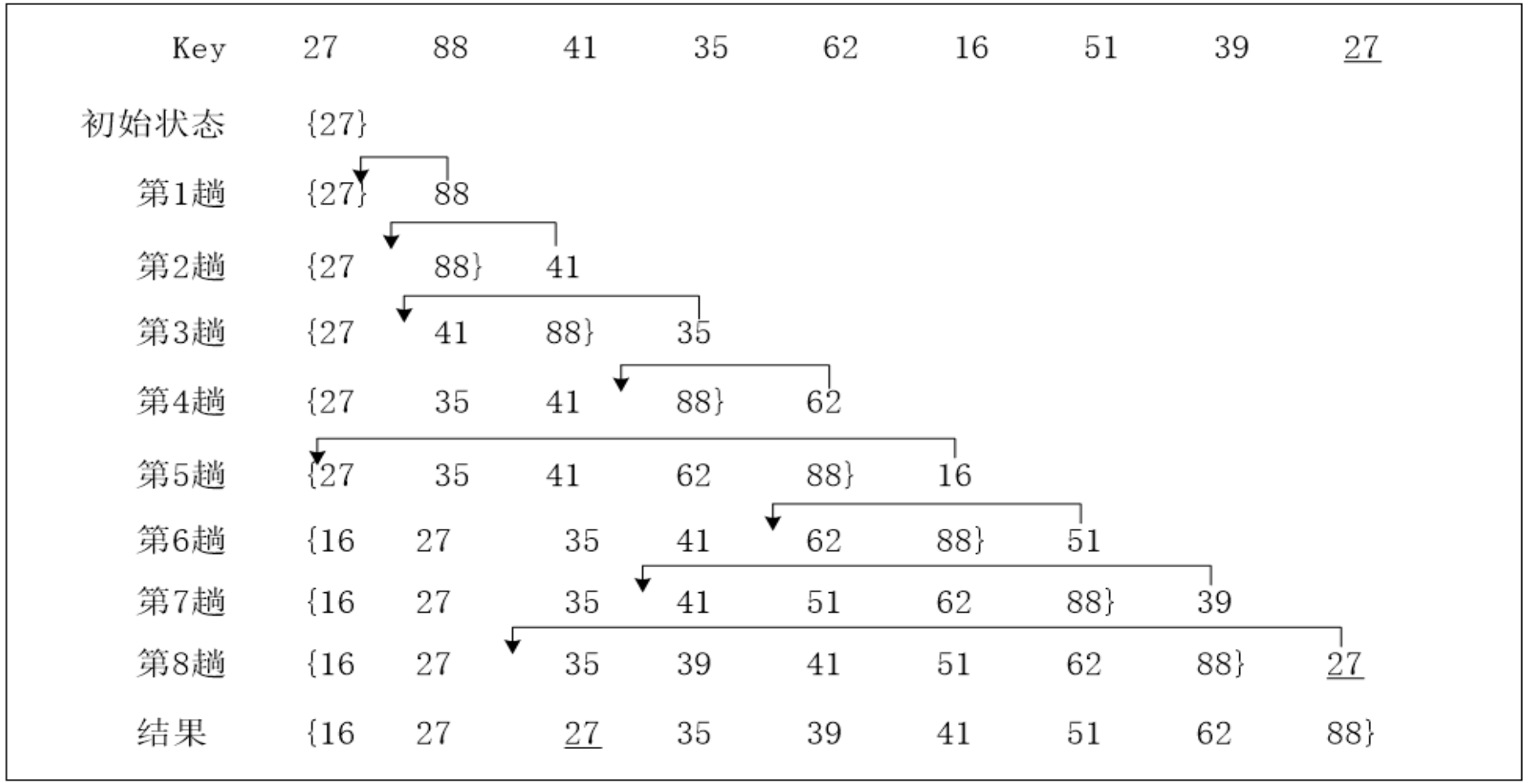
挤空 ② 将{R[j]，……，R[i-1]}中的所有记录依次往后移动一个位置。从后往前比较，保证稳定性，即对于相等的记录保证其先后顺序不变。

插入 ③ 把R[i]插到位置j上。

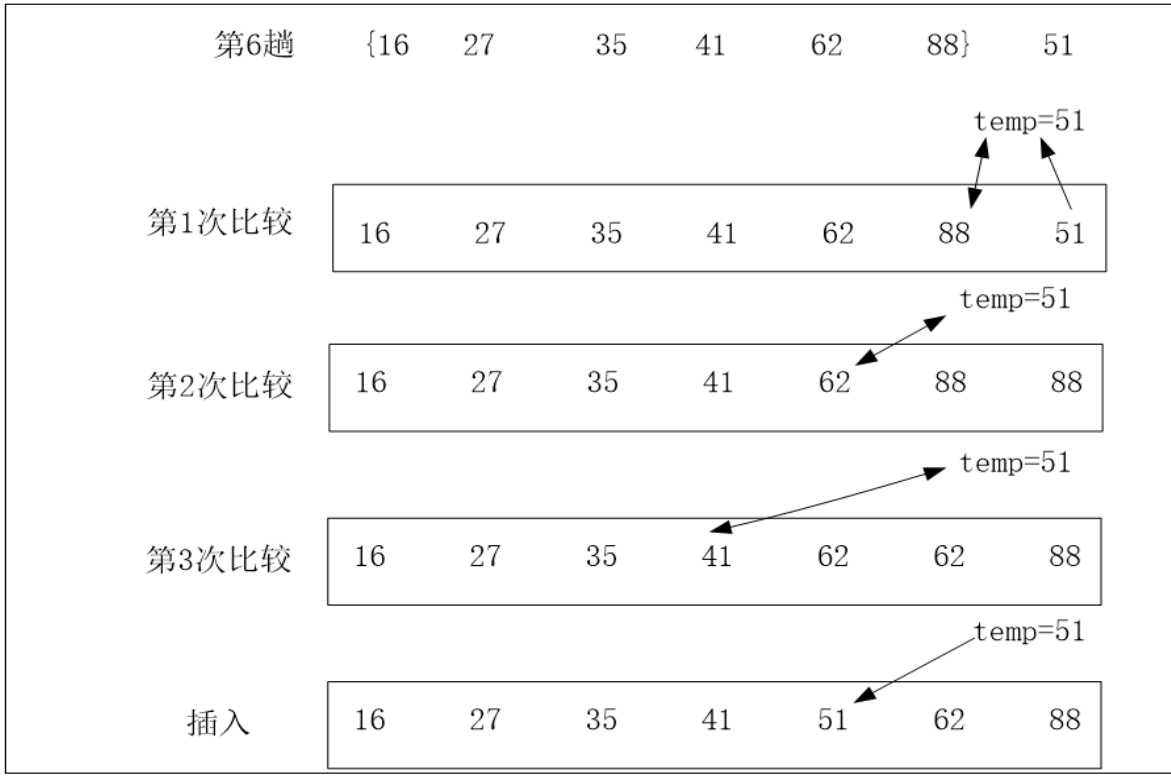
- ★注：为提高算法的效率，可一边查找，一边移动。

7.2.1 简单插入排序-例

【例7.1】有9个待排序记录，其关键字依次是27，88，41，35，62，16，51，39，27。其中“27”是为了区分关键字值相同的记录。



7.2.1 简单插入排序



- 以第6趟排序为例解释简单插入算法的具体执行过程：

① 首先将待插入记录51赋给temp变量；

② temp与已排好的序列中最后一个元素R[j-1]=88进行比较，因88>51，则将88后移一个位置，即R[j]=R[j-1]。完成第一次比较与移动；

③ temp与R[j-2]=62比较，因51<62，则将62后移一个位置，即R[j-1]=R[j-2]。完成第二次比较与移动。

④ temp与R[j-3]=41比较，因51>41，则找到插入位置将51插入，即R[j-2]=temp。完成本趟的插入过程。

7.2.1 简单插入排序-代码

算法7.2：直接插入排序的算法

```
1.  //按照关键字key 非递减顺序对待排序记录进行直接插入排序
2.  void recordList::InsertSort(RecType R[],int n){
3.      int i,j;
4.      for(i=1; i<n; i++){
5.          Element temp=R[i];      //将待插入记录暂时存放
6.          for(j=i; j>0; j--){    // 从后往前顺序比较
7.              if(temp.key<R[j-1].key)
8.                  R[j]=R[j-1];  //进行比较与移动
9.              else break;
10.         }
11.         if (i!=j)  R[j]=temp;      // 此时j就是记录i的正确位置，回填
12.     }
13. }
```

7.2.1 简单插入排序

- 对简单插入排序算法的性能进行分析：
  - 空间代价：算法中用到了一个辅助存放待插入记录的临时变量，故空间代价为一个记录的大小，即O(1)。
  - 时间代价：在简单插入排序算法中，基本操作是关键字大小的比较和记录的移动。

（1）平均情况：假设待排记录序列有k个元素，待插记录应在已排序的序列中第j个位置上，j∈1...k，需要比较k–j+1次。通常待排序记录可能出现的各种排列概率是相同的即1/k。因此，第k个记录平均情况下的比较次数为：

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

7.2.1 简单插入排序

- 对具有n个记录的列表进行简单插入排序所需的平均总代价为：

$$\sum_{k=2}^{n}\left[\frac{k+2}{2}\right]=\sum_{k=1}^{n-1}\left[\frac{k+4}{2}\right]=\frac{1}{4}n^2+\frac{7}{4}n+1$$

  - 约为$n^2/4$。因此其平均时间复杂度为$O(n^2)$。

（2）最好情况（正序）：

- 当初始待排序记录序列的关键字已经按照非递减顺序排列时，每次第i个记录一进入内层循环就退出，无需记录移动，不进行迭代。在一趟排序中，进行1次关键字的比较。总共比较次数为n-1 次，移动了(n-1)次。因此最好情况下插入排序的时间代价为O(n)。

7.2.1 简单插入排序

(3) 最坏情况(逆序):

- 当初始待排序记录序列的关键字已经按照非递增顺序排列时，每次第i个记录进入内层循环，需要进行i次迭代。在一趟排序中，进行i次关键字的比较，每比较一次需要移动一次，故进行了i次移动。记录保存在临时变量中1次，回填1次，所以移动i+2次。外层循环进行了n-1次，故比较次数和移动次数分别为：

$$\text{比较次数} = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \qquad \text{移动次数} = \sum_{i=1}^{n-1} (i+2) = \frac{(n+4)(n-1)}{2}$$

- 从排序的稳定性来看，简单插入排序是稳定的。
- 简单插入排序算法适用于待排序记录数目较小的情况。

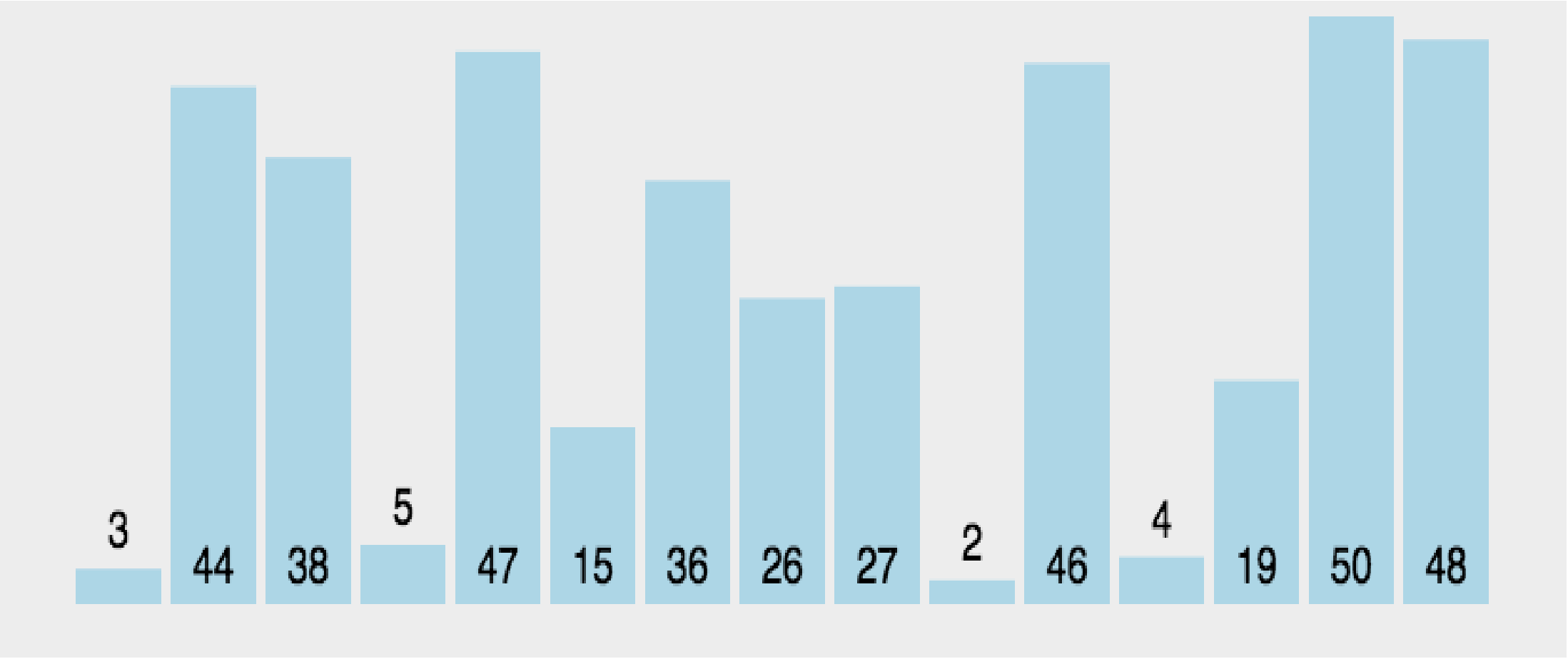
7.2.1 简单插入排序

表1 简单插入排序算法的性能

时间复杂度			空间复杂度	稳定性	复杂性
平均情况	最坏情况	最好情况			
$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定	简单

分析：用简单插入排序，数据序列越接近有序，比较次数越少。

7.2.2 冒泡排序



7.2.2 冒泡排序-基本思想

- 冒泡排序又称起泡排序。
  - 基本思想：对待排记录，自底向上不断地对相邻记录进行比较，如果不满足排序要求，则交换相邻的记录，一趟结束时待排序列中一个最小记录到位，重复以上过程直到所有的记录都已经排好序为止。
  - *在冒泡排序的过程中，关键字较小的元素像水中的气泡在逐趟向上漂浮，关键字较大的元素像石块一样相对往下沉，并且每趟排序都有一个最小的气泡（即此趟排序的最小元素）漂到最上边。*

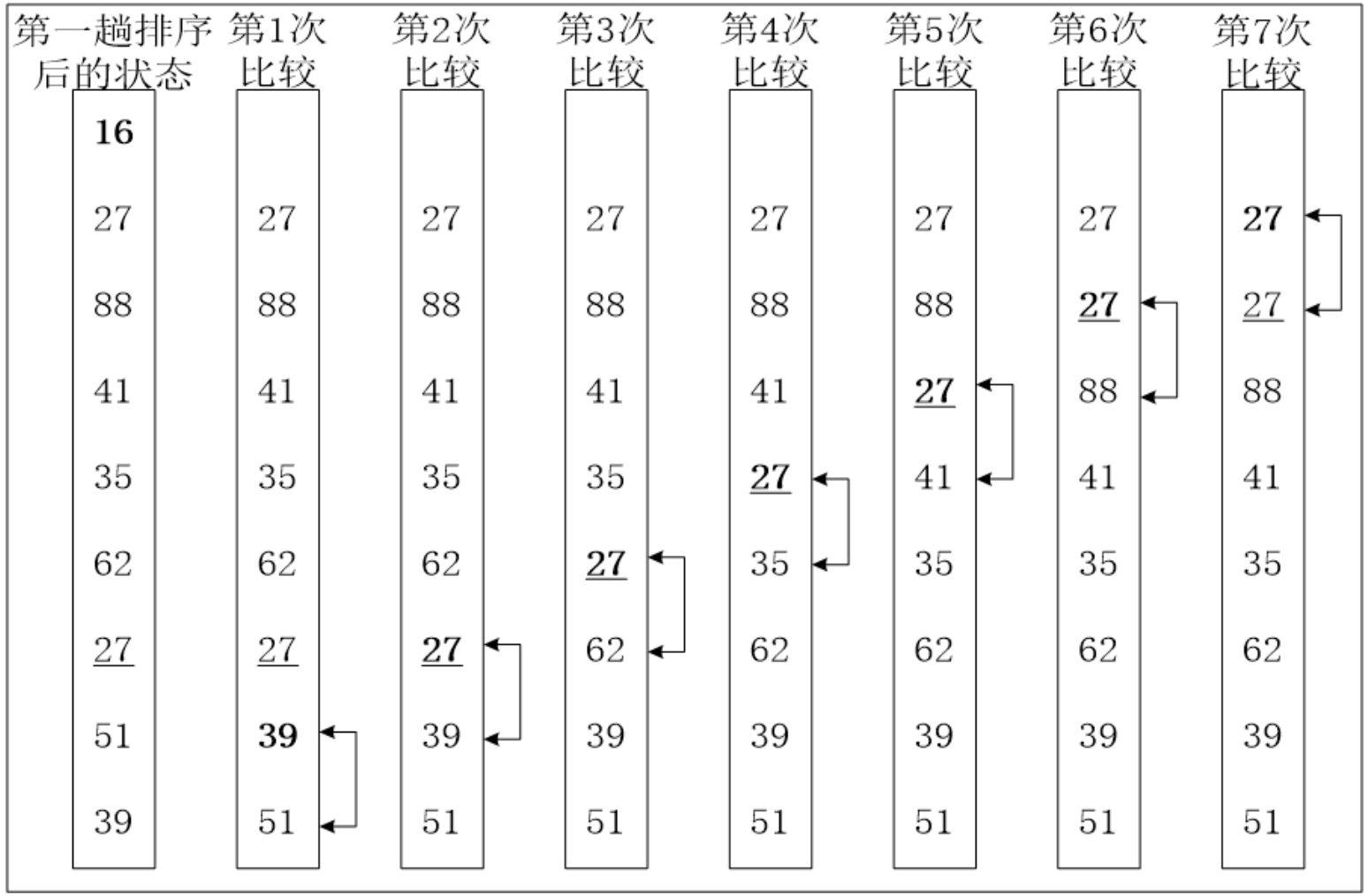
7.2.2 冒泡排序-例

【例7.2】将关键字分别为27，88，41，35，62，16，51，39，27的待排序记录进行冒泡排序的过程，如图所示。其中“_”是为了区分关键字值相同的记录。（黑体数字表示每趟冒泡排序所选出来的“最小”关键字）。

初始序列	第1趟	第2趟	第3趟	第4趟	第5趟	第6趟	第7趟	第8趟	结果
27	**16**								16
88	27	**27**							27
41	88	27	**27**						27
35	41	88	35	**35**					35
62	35	41	88	39	**39**				39
16	62	35	41	88	41	**41**			41
51	27	62	39	41	88	51	**51**		51
39	51	39	62	51	51	88	62	**62**	62
27	39	51	51	62	62	62	88	88	88

7.2.2 冒泡排序-例

以第2趟排序为例解释冒泡排序算法的具体执行过程：



7.2.2 冒泡排序-代码

算法7.3：冒泡排序的算法

```
1.  void recordList::BubbleSort( ){
2.      //对R[0]到R[n-1]进行逐趟比较，逆序就交换。n表示待排序的记录个数
3.      int pass=1;
4.      bool flag=false;     //当flag为true时则停止排序
5.      while(pass<CurrentSize && !flag){        //冒泡排序趟数不会超过n-1次
6.          flag=true;                    //交换标识设置为true，假定未交换
7.          for(int j=CurrentSize-1; j>=pass; j--){
8.              if(R[j].key<R[j-1].key){   //逆序
9.                  RecType temp;
10.                 temp=R[j];
11.                 R[j]=R[j-1];
12.                 R[j-1]=temp;
13.                 flag=false;     //交换标识设置为false,标识有交换
14.             }
15.         }
16.         pass++;
17.     }
18. }
```

7.2.2 冒泡排序

- 冒泡排序法是自底向上相邻元素两两比较，逆序则交换，使最小元素上移。
- 还有另一种方法是自顶向下相邻元素两两比较，逆序交换，最大元素下移，这种方法称为吊桶法，吊桶法与冒泡法的排序过程类似。

7.2.2 冒泡排序-算法评价

- 对冒泡排序算法过程中的关键字比较次数和移动次数进行分析

（1）最好情况（正序）：

- 待排序记录序列关键字已经按从小到大的顺序排列。此时算法只执行一趟冒泡，做n-1次关键字比较，不移动记录，即移动次数为0。

（2）最坏情况（逆序）：

- 待排序记录序列的关键字按从大到小的顺序排列。此时算法执行了n-1趟冒泡，第i趟（1≤i<n）做了n-i次关键字比较，执行了n-i次记录交换，移动次数是3倍的n-i。
- 关键字的比较次数：$\sum_{i=1}^{n-1}(n-i)$ = n(n-1)/2
- 移动次数：$3\sum_{i=1}^{n-1}(n-i)$ = 3n(n-1)/2

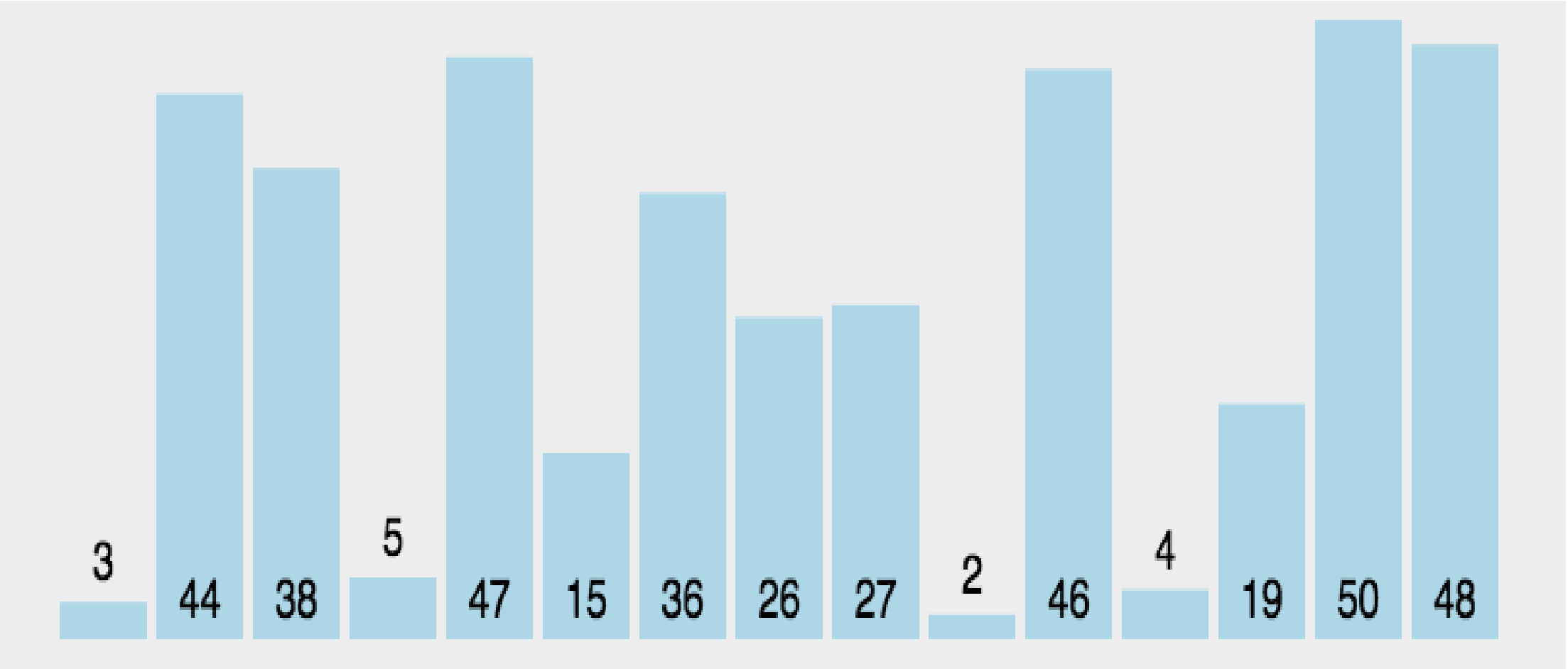
7.2.2 冒泡排序-算法评价

（3）平均情况：

- 考虑冒泡排序的范围为j到n，对于每一趟冒泡排序有n-j次比较。
- 关键字的比较次数 = $\sum_{j=1}^{n-1}[n-j]=\frac{1}{2}n(n-1)$

- 冒泡排序需要一个附加记录以实现记录值的交换，空间代价为O(1)。
- 冒泡排序是一个稳定的排序方法。
- 结论：排序算法中相邻元素的交换，是稳定的。
  - 元素出现大范围交换都是不稳定的。

7.2.3 简单选择排序



7.2.3 简单选择排序-基本思想

简单选择排序的基本思想：

- 从无序子序列中“选择”关键字最小或最大的记录，并将它加入到有序子序列中，以此方法增加记录的有序子序列的长度。
  - 每一趟排序都是在后面n-i(i=0，1，…，n-1)个待排序记录中选出关键字最小的记录，作为有序记录序列的第i个记录。
  - 选择排序的关键是如何从剩余的待排序记录中找出最小（或者最大）的那个记录。

7.2.3 简单选择排序-基本

- 简单选择排序是一种比较简单的排序方法，它的基本过程为：

  ① 在一组未排序记录R[i..n-1]中选择具有最小关键字的记录。

  ② 如果它不是第i个数据元素，则将最小的元素与第i个元素交换。

  ③ 然后在剩下的记录R[i+1..n-1]中重复执行第(1)、(2)步，直至只有一个剩余记录为止。

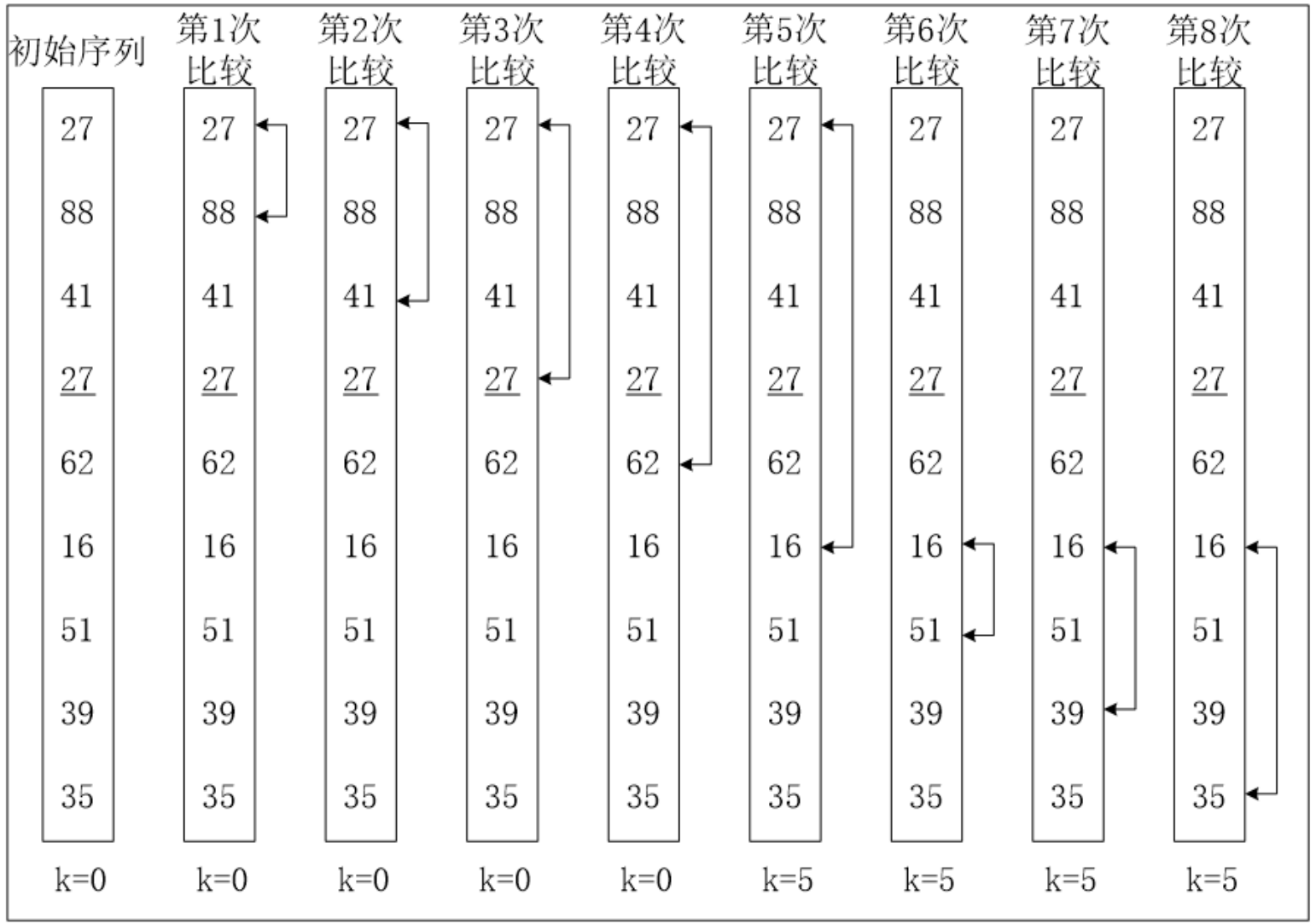
7.2.3 简单选择排序-例

【例7.3】对关键字序列{27，88，41，27，62，16，51，39，35}进行简单选择排序。

初始序列	第1趟	第2趟	第3趟	第4趟	第5趟	第6趟	第7趟	第8趟	第9趟	结果
27	**16**	16	16	16	16	16	16	16	16	16
88	88	**27**	27	27	27	27	27	27	27	27
41	41	41	**27**	27	27	27	27	27	27	27
27	27	**88**	88	**35**	35	35	35	35	35	35
62	62	62	62	62	**39**	39	39	39	39	39
16	**27**	27	**41**	41	41	**41**	41	41	41	41
51	51	51	51	51	51	51	**51**	51	51	51
39	39	39	39	39	**62**	62	62	**62**	62	62
35	35	35	35	**88**	88	88	88	88	**88**	88

7.2.3 简单选择排序

以第1趟选择排序为例解释简单选择排序算法的具体执行过程：



7.2.3 简单选择排序-代码

算法7.3：简单选择排序的算法

```
1.  void recordList::SelectSort(Element rec[], int n){
2.      //对待排序记录序列R[0]..R[n-1]进行排序,n表示当前的长度
3.      Element  it;
4.      for(int i=0; i<CurrentSize-1; i++){
5.          int k=i;   //k用来标识R[i..n-1]之间具有最小关键字的记录的位置
6.          for(int j=i+1; j<CurrentSize; j++)
7.              if(R[j].key<R[k].key)
8.                  k=j;
9.          if(k!=i){   //对换到第i个位置
10.             it = rec[k];
11.             rec[k] = rec[i];
12.             rec[i] = it;
13.         }
14.     }
15. }
```

★注：简单选择排序的关键字比较次数与待排序记录的初始排列无关。

7.2.3 简单选择排序-算法评价

- 假定待排序记录的个数为n，第i（0~n-2）趟选择具有最小关键字记录所需的比较次数总是n-i-1次。所以，总的关键字比较次数为：
  - 关键字比较次数= $\sum_{i=0}^{n-2}(n-i-1)=\frac{n(n-1)}{2}$
- 记录的移动次数与待排序记录序列的初始状态有关。
  - 最好情况，如果待排序记录的初始序列是按照其关键字从小到大有序排列，记录不需要移动，即移动次数为0。
  - 最坏情况，每一趟排序都要进行交换，总共有n-1趟排序，最后总的记录移动次数为3×(n-1)。
  - 该算法用到了交换操作，需要设置一个临时记录，因此该算法的空间复杂度为O(1)。

7.2.3 简单选择排序-算法评价

- 简单选择排序的时间复杂度为$O(n^2)$。
  - 该算法不依赖于原始数据的输入顺序，故最大和最小的平均时间代价都为$O(n^2)$。
  - 选择排序实质上就是冒泡排序，但交换的次数要比冒泡排序少得多。
  - 对于处理那些做一次交换花费时间较多的问题，选择排序是很有效的。
- 简单选择排序是一种**不稳定**的排序算法。
  - 初始数据：3，3，1，4
  - 第一趟排序：1，3，3，4

单选题 1分

设置

请对下列数据13，24，7，1，8，9，11，56，34，51，2，77，5，进行简单插入排序，冒泡排序（每趟大数冒到后面）和选择排序，2趟后的排序结果为：（　　）

A
2趟后的简单插入排序序列为：7,13，1，24，8，9，11，56，34，51，2，77，5
2趟后的冒泡排序为：1，7，8，9，11，13，24，34，2，51，5，56，77
2趟后的选择排序为：1，2，5，7，13，8，9，11，56，34，51，24，77

B
2趟后的简单插入排序序列为：7,13，1，24，8，9，11，56，34，51，2，77，5
2趟后的冒泡排序为：1，7，8，9，11，13，24，34，2，5，51，56，77
2趟后的选择排序为：1，2，7，13，8，9，11，56，34，51，24，77，5

C
2趟后的简单插入排序序列为：7,13，24，1，8，9，11，56，34，51，2，77，5
2趟后的冒泡排序为：7，1，8，9，11，13，24，34，2，51，5，56，77
2趟后的选择排序为：1，2，7，13，8，9，11，56，34，51，24，77，5

D
2趟后的简单插入排序序列为：7,13，1，24，8，9，11，56，34，51，2，77，5
2趟后的冒泡排序为：1，7，8，9，11，13，24，34，2，51，5，56，77
2趟后的选择排序为：1，2，7，13，8，9，11，56，34，51，24，77，5

提交

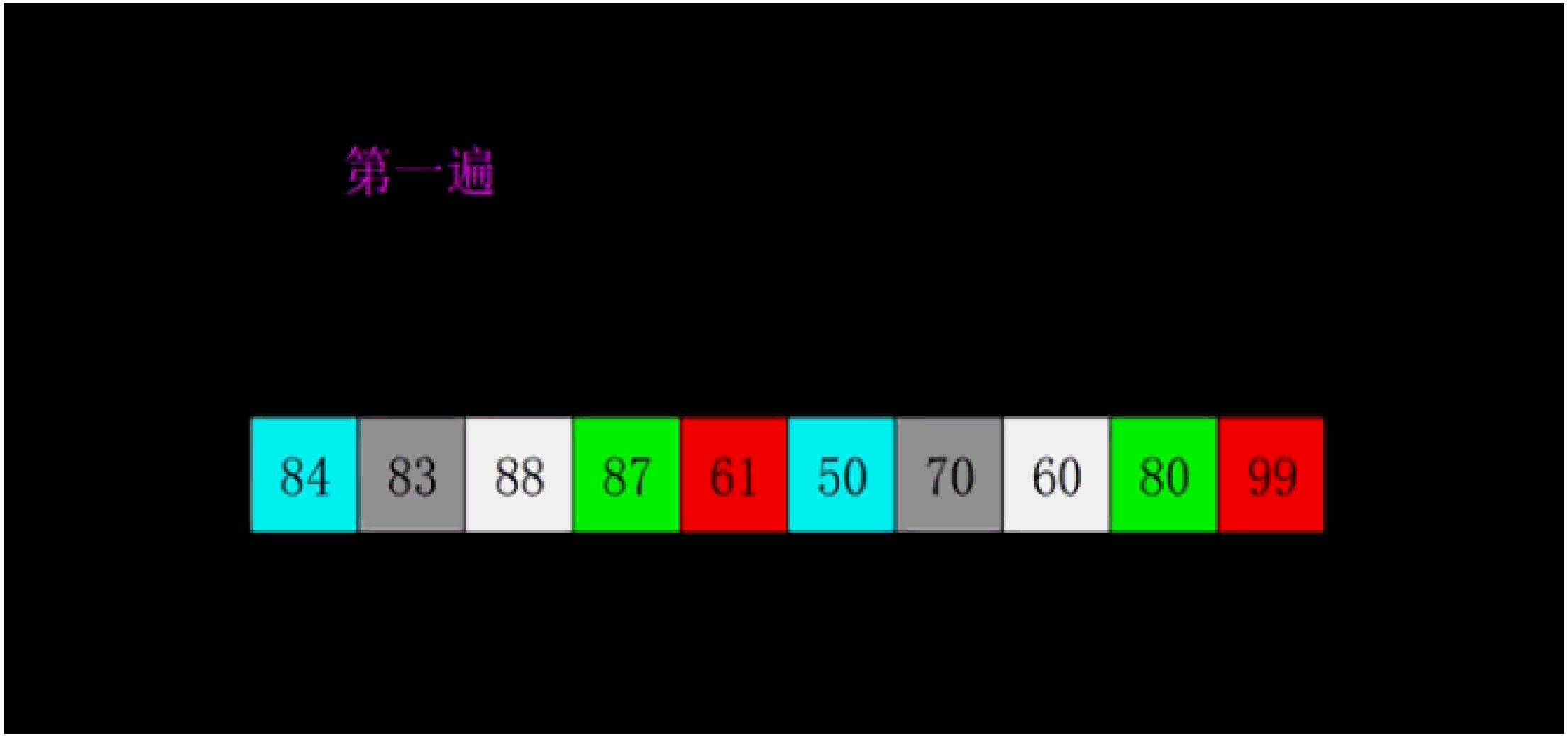
大作业

- 从简单排序和高级排序里分别任选1种排序算法，理解该排序的思想，参考一些排序视频和flash动图做一个相应排序算法的展示，形式不限，可以是录制小视频、制作flash动图、文本文件等。
  - 简单排序：冒泡、简单插入、简单选择
  - 高级排序：希尔、快速、归并、堆、锦标赛、基数排序等
  - 截止时间：14周前

7.3 高级排序

- 高级排序方法的时间复杂度基本都是O(nlogn)。

7.3.1 希尔排序



7.3.1 希尔排序

- 希尔（Shell）排序，又称为“缩小增量排序”，它是对简单插入排序算法的一种改进。
  - 基本思想：分割成若干个较小的子文件，对各个子文件分别进行直接插入排序，当文件达到基本有序时，再对整个文件进行一次直接插入排序。
- 对待排记录序列先作“宏观”调整，再作“微观”调整。
  - “宏观”调整，指的是，“跳跃式”的插入排序。
- ★注：希尔排序中的间隔从大到小逐渐缩小，一定要保证最后的步长为1。
  - 间隔通常选奇数。

7.3.1 希尔排序

- 希尔排序是变间隔子序列插入，具体过程为：

① 假设待排序记录序列有n个记录，首先取一个整数gap作为间隔（gap<n），将整个待排记录序列划分为gap个子序列，并使各个子序列的元素在整个数组中的间距相同；

② 对每个子序列分别进行简单插入排序；

③ 缩小间隔gap，再按新的gap划分的子序列分别进行简单插入排序；

④ 重复上述子序列划分和排序过程，直到最后取gap=1，将所有待排序记录放在同一个序列中，通过直接插入排序得到有序序列为止。

7.3.1 希尔排序-例

【例7.4】将关键字分别为27，88，41，35，62，16，51，39，27，的记录序列进行希尔排序的过程.

初始序列	27	88	41	35	62	16	51	39	27
gap=3	27			35			51		
		88			62			39	
			41			16			27
第一趟排序结果	27	39	16	35	62	27	51	88	41
gap=2	27		16		62		51		41
		39		35		27		88	
第二趟排序结果	16	27	27	35	41	39	51	88	62
gap=1	16	27	27	35	39	41	51	62	88

7.3.1 希尔排序-代码

希尔排序的算法如下，算法中缩小增量（间隔）的方式是gap=⌊gap/2⌋。

算法7.5：希尔排序的算法

```
1.  static void insertSort2(Element rec[],int n, int start, int incr ){  //间隔直接插入
2.      for (i = start + incr;  i < n; i = i + incr ) {  // 控制插入元素
3.          Element it = rec[i];
4.          for (  j = i; j >= incr; j = j - incr)  //找到合适位置
5.              if ( it < rec[j - incr])  rec[j] = rec[j - incr];
6.              else break;
7.          if  ( i != j ) rec[j] = it;
8.      }
9.  }
10. static void Shellshort(Element rec[], int n){  //Shell排序的主程序
11.     for ( gap = n / 2;  gap > 2; gap = gap / 2 )
12.             for ( j = 0; j < gap; j++)  insertSort2(rec, n, j, gap); //控制变步长各子序列
13.      insertSort2( rec, n, 0, 1);
14. }
```

7.3.1 希尔排序

希尔排序特点：

- 子序列的构成不是简单的“逐段分割”，而是将相隔某个增量的记录组成一个子序列
- 希尔排序可提高排序速度，因为
  - 分组后n值减小，n更小，而$T(n)=O(n^2)$,所以T(n)从总体上看是减小了
  - 每经过一次希尔排序以后，关键字较小的记录跳跃式前移，每个关键字更加接近它的最终位置；
  - 在最后一趟排序之前，待排序记录序列已经基本有序。
- 增量序列取法
  - 应该尽量避免序列中的值(尤其是相邻的值)互为倍数的情况。
  - 最后一个增量值必须为1

7.3.1 希尔排序

- 最坏复杂度分析：
- 【定理】使用希尔增量的最坏时间复杂度是$O(n^2)$

8-sort	1	9	2	10	3	11	4	12	5	13	6	14	7	15	8	16
4-sort	1	9	2	10	3	11	4	12	5	13	6	14	7	15	8	16
2-sort	1	9	2	10	3	11	4	12	5	13	6	14	7	15	8	16
1-sort	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

P增量对不互质（relatively prime）. 因此小的增量可能没有效果

7.3.1 希尔排序-更多增量序列

- Hibbard增量序列
  - $D_k = 2^k - 1$ —— 相邻元素互质
  - 最坏情况：$T = O(N^{3/2})$
  - 猜想：$T_{avg} = O(N^{5/4})$
- Sedgewick增量序列
  - $\{1, 5, 19, 41, 109, \cdots\}$ —— $9x4^i - 9x2^i + 1$ 或 $4^i - 3x2^i + 1$
  - 猜想：$T_{avg} = O(N^{7/6})$，$T_{worst} = O(N^{4/3})$

7.3.1 希尔排序

- 一般情况，当增量取值合理，n在一定范围内，则希尔排序中所需的比较和移动次数约为$O(n^{1.5})$。
- 从所需的附加空间来看，需要一个临时变量来存放待插入的记录，所以其空间复杂度为O(1)。

时间复杂度			空间复杂度	稳定性	复杂性
平均情况	最坏情况	最好情况			
$O(nlogn)$~$O(n^2)$	$O(nlogn)$~$O(n^2)$		O(1)	不稳定	较复杂

单选题 1分

设置

已知输入数据13，24，7，1，8，9，11，56，34，51，2，77，5，增量序列d=5,3,1,请采用希尔排序算法进行排序，2趟排序后的结果为（　　）

A 1，7，8，9，13，24，11，34，51，2，5，56，77

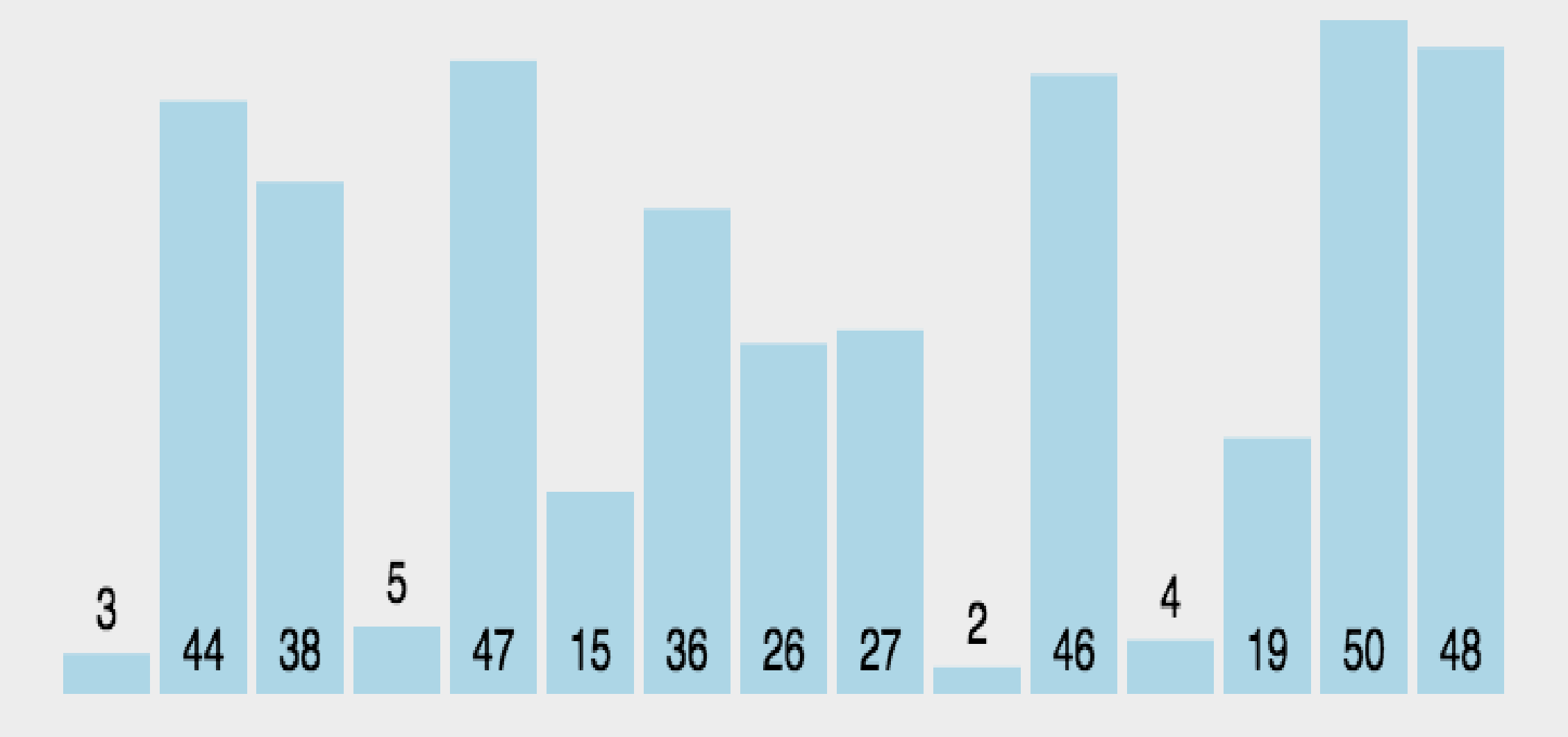
B 2，5，11，1，8，9，7，24，34，13，51，77，56

C 2，11，5，1，8，9，24，7，34，51，13，77，56

D 1，7，5，2，8，9，24，11，34，51，13，77，56

提交

7.3.2 快速排序



7.3.2 快速排序-思想

- 快速排序的基本思想：
- 1）任选待排序记录序列中的一个记录（通常选取第一个记录）作为枢轴（pivot），按照该记录的关键字大小，将整个待排序记录序列分为左右两个子序列。
  - 将记录的关键字小于或等于枢轴记录的关键字的待排序记录放在左序列中，将记录的关键字大于枢轴记录的关键字的待排序记录放在右序列中，枢轴记录排在这两个子序列的“中间”位置（这也是该记录的最终位置），这个过程称作一趟快速排序。
- 2）然后分别对这左右两个子序列重复执行上述方法，直到所有的记录都排放在相应的位置上为止。

7.3.2 快速排序-例

【例7.5】对关键字序列{49，37，66，97，37，68}进行快速排序。快速排序可以递归地进行。进行一趟快速排序后再分别对分割所得到的两个子序列递归进行快速排序。如果待排序记录序列中只有一个记录，显然已经有序，递归结束。

初始状态	{49	37	66	97	37	68}
一次划分以后	{37	37}	**49**	{97	66	68}
分别快速排序	**37**	{37}	**49**	{68	66}	**97**
		结束		{66}	**68**	
				结束		
有序序列	37	37	49	66	68	97

7.3.2 快速排序-步骤

- 一趟快速排序的算法过程如下：

①设置两个变量low和high，排序开始的时候low=0，high=n-1；

②以第一个数组元素作为枢轴记录的关键字，赋值给pivot，即pivot= R[low]；

③从high开始向前搜索，即由后开始向前搜索（high=high-1），找到第一个小于pivot的值，两者交换；

④从low开始向后搜索，即由前开始向后搜索（low=low+1），找到第一个大于pivot的值，两者交换；

⑤重复③和④，直到low=high

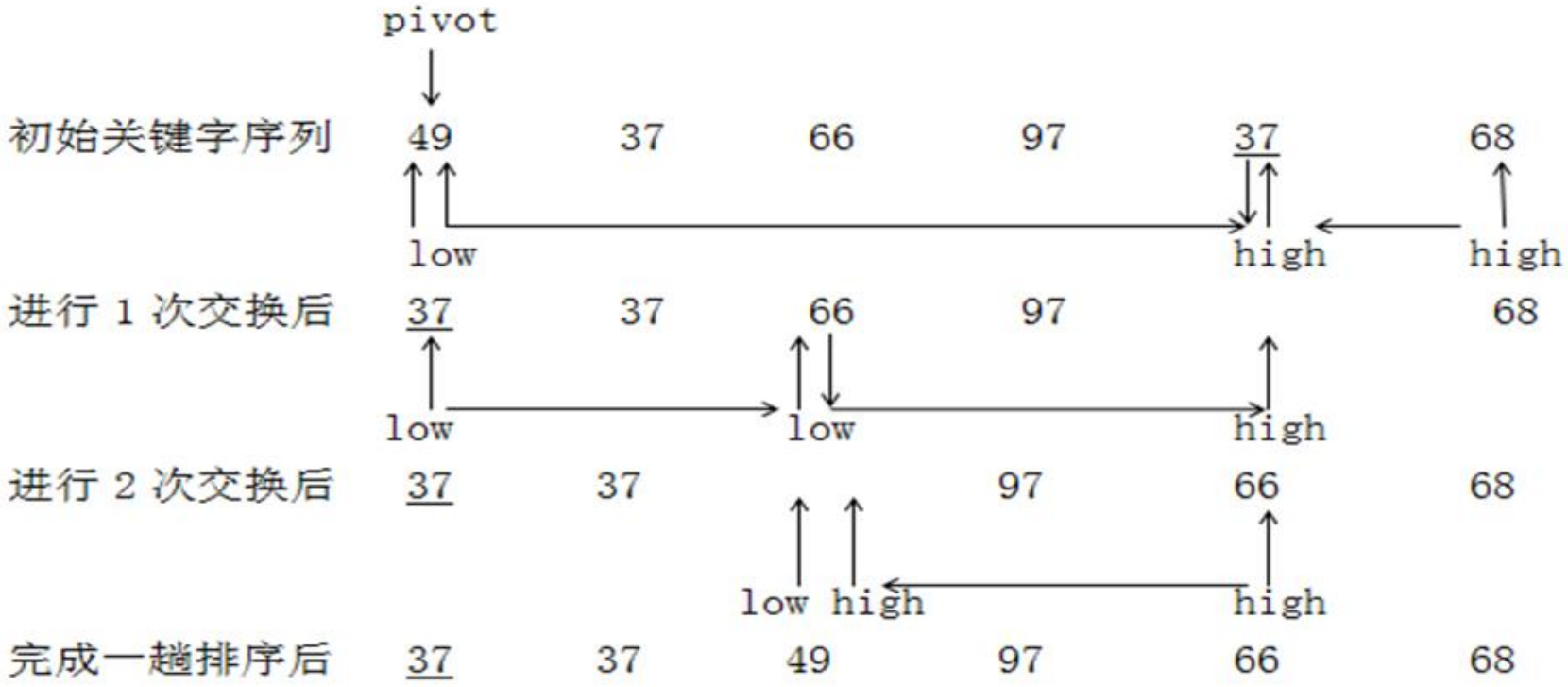
7.3.2 快速排序-代码

算法7.7：快速排序的算法（划分过程）

```
1.  int recordList::Partition(Element R[], int low, int high){ //一次划分
2.    //将子序列中的第一个待排序记录作为枢轴记录
3.    Element pivot=R[low];       //枢轴记录的关键字
4.    while(low<high){            //从子序列中的两端交替地向中间扫描
5.      while(low<high&&R[high].key>pivot.key) --high;
6.      R[low]=R[high]; //将小于或等于枢轴记录的关键字移动到低位置
7.      while(low<high&&R[low].key<=pivot.key) ++low;
8.      R[high]=R[low];  //将大于枢轴记录的关键字移动到高位置
9.    }
10.   R[low]=pivot;              //枢轴记录应该在的位置
11.   return low;                //返回枢轴记录所在的位置
12. }//Partition
```

7.3.2 快速排序-例

- 在一趟快速排序中其具体过程如下：
  - 将第一个数组元素49作为pivot，low指针指向序列的第一个元素，high指针指向序列的最后一个元素，此时，low=0，high=5。



7.3.2 快速排序

- 快排算法中枢轴选取问题讨论：
- 枢轴的选择对快速排序的性能有比较大的影响。
  - 最理想的枢轴选取是，每次选择的枢轴可以将待排序记录序列分为两个长度均等的部分；
  - 最坏情况是，每次划分完待排序记录后，有一个空子序列。
- 一般对于枢轴的选择有以下几种方法，它们都是O(1)级的：

  (1) 取固定位置的值，比如第一个元素，中间元素，最后一个元素；

  (2) 取第一个元素、中间元素和最后一个元素的中值元素（常用）；

  (3) 在待排序记录序列中寻找3个不同的元素，取它们的中间值。

7.3.2 快速排序

- 快速排序的性能：
  - 快速排序可以看作是由一系列相似步骤执行完成的，每一步都包含枢轴的选择和划分待排序记录序列。每一步枢轴选择的时间复杂度是O(1)，每一步划分时的时间复杂度不会超过O(N)。
- 为了分析的简单，假定对1到N进行划分，C(N)代表快速排序对N个待排序记录进行排序总的比较次数，S(N)代表总的交换次数。假设每次划分产生一个长度为r的子序列，另一个长度为N-r-1的子序列。则有如下递推式：
  - $C(N)=N-1+C(r)+C(N-r-1)$
  - 快速排序在最坏情况下：时间复杂度为$O(N^2)$。交换次数的分析与比较次数类似，可以得到$S(N) = 0.5N^2 - 0.5N$，由于一次交换有3次元素的赋值，故快速排序总的移动次数为$O(1.5N^2)$。
  - 快速排序在一般情况下：比较次数和交换次数都是O(nlogn)

7.3.2 快速排序

结论：

① 快速排序的平均时间复杂度为O(nlogn)。

② 快速排序的性能跟初始序列中关键字的排列和选取的枢纽有关。

③ 快速排序被公认为在所有同数量级O(nlogn)的排序方法中，平均性能最好，但如果待排序序列有序（正序或逆序）时，快速排序将蜕化为一般冒泡排序(最坏情况)，其时间复杂度变为$O(n^2)$。

④ 快速排序是一种不稳定的排序算法。

比如：待排序序列：49 49 38 65

快速排序结果：38 49 49 65

7.3.2 快速排序

表3 快速排序算法的性能

时间复杂度			空间复杂度	稳定性	复杂性
平均情况	最坏情况	最好情况			
O(nlogn)	$O(n^2)$	O(nlogn)	O(logn)	不稳定	较复杂

单选题 1分

设置

已知输入数据13，24，7，1，8，9，11，56，34，51，2，77，5，请采用快速归排序算法进行排序，其中枢纽采用3者取中法，2趟排序后的结果为（　　）

A （（2，5，1），7，（8，9）），11，（（13，34，24），51，（77，56））

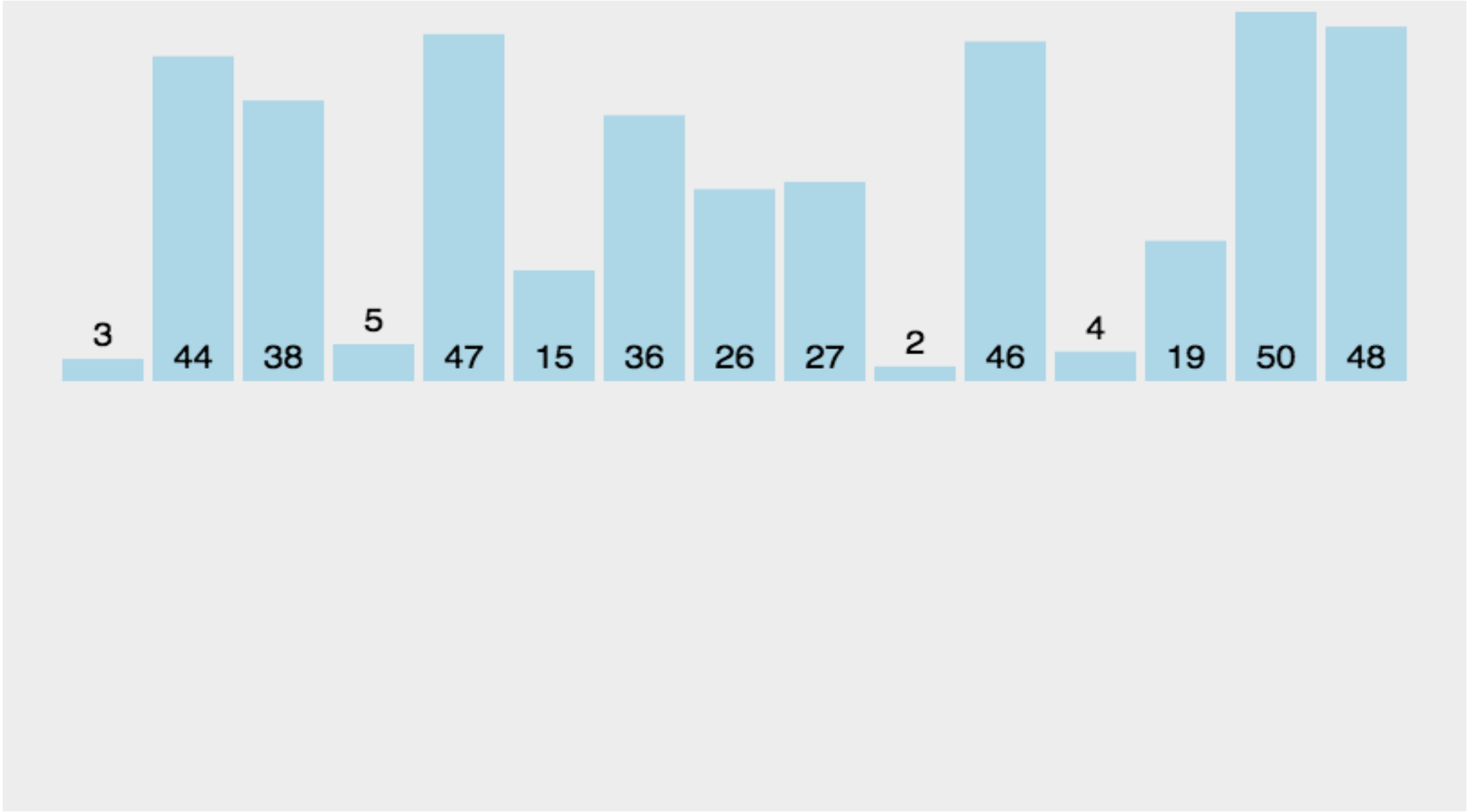
B （（2，5，1），（7，8），9）），11，（（13，34，24），51，（77，56））

C （（1，2），5，（7，8，9，11）），13，（（24），34，（56，77，51））

D （（1，2），5，（13，8，9，7）），11，（（24），34，（51，77，56））

提交

7.3.3 归并排序



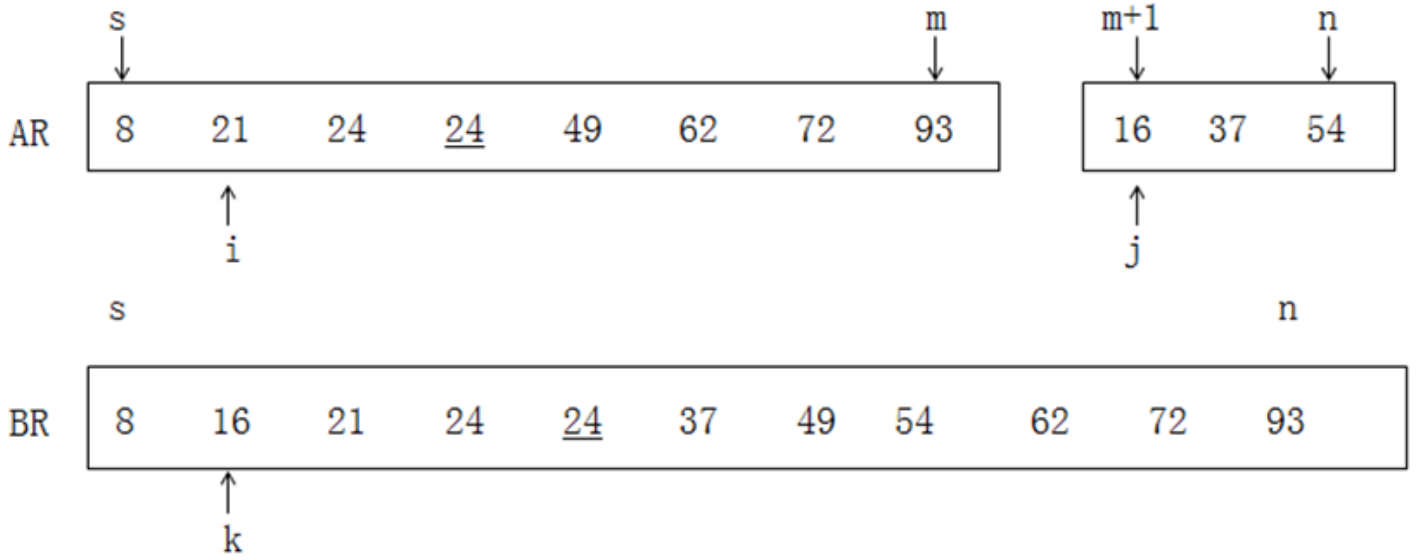
7.3.3 归并排序-思想

- 归并：将两个或两个以上的有序表组合成一个新的有序表，叫归并排序。
- 归并排序（Merging Sort）是一种高级排序方法，它同快速排序一样，也是基于分治法的。
- 归并排序的思想：将一个序列分成两个（或多个）长度几乎相等的子序列，对每个子序列排序，然后再将它们“归并”为一个有序序列。而对子序列又是递归的“归并”过程。
  - 在内部排序中，通常采用2-路归并排序，它是建立在2-路归并的基础上，即将两个位置相邻的有序子序列R[s..m]和R[m+1..t]归并成一个有序序列R[s..t]。

7.3.3 归并排序- 2-路归并的过程

- 2-路归并的过程：记录序列AR中有两个已经排好序的有序子序列AR[s..m]，AR[m+1..n]，将它们归并为一个有序表，并存放于另一个记录序列BR[s..n]中。
  - 设初始序列含有n个记录，则可看成n个有序的子序列，每个子序列长度为1
  - 两两合并，得到[n/2]个长度为2或1的有序子序列，再两两合并，……如此重复，直至得到一个长度为n的有序序列为止

7.3.3 归并排序



- 2-路归并的基本思想：

（1）假设有两个有序表A和B的记录个数（即表长）分别为al和bl，变量i和j分别是表A和表B的当前检测指针。假设表C是归并后的新有序表，变量k是它的当前存放指针；

（2）当i和j都在两个表的表长内变化时，根据A[i]和B[j]的关键字大小，依次把关键字小的记录存放到新表C[k]中；

（3）若i与j有一个已经超出表长时，则将另一个表中的剩余部分复制到新表C[k]中。

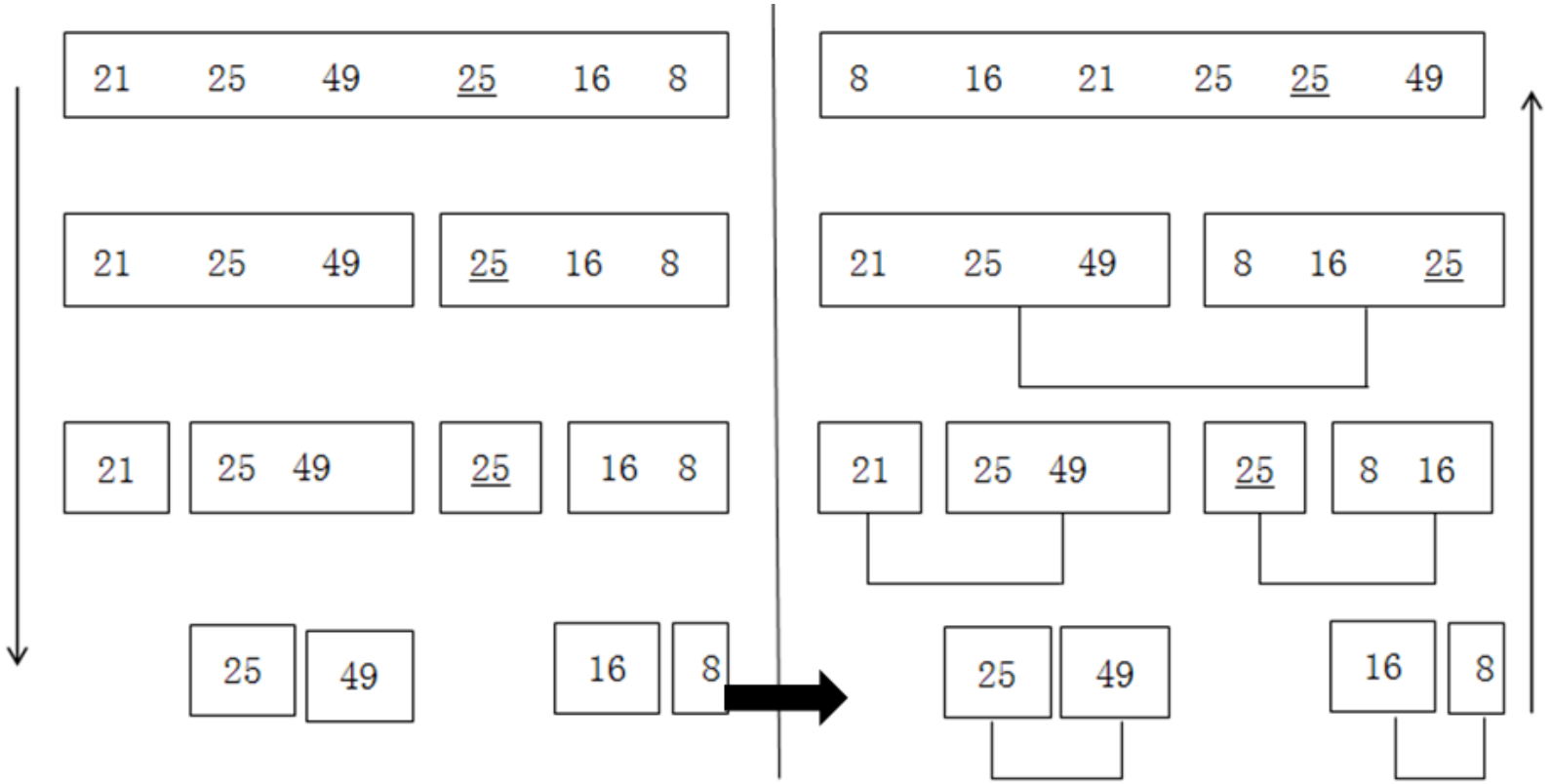
7.3.3 归并排序-代码

算法7.8：2-路归并的算法

```
1.  void  recordList::merge(recordList A, recordList &B, int s, int m, int n){
2.    //将两个有序表A[s..m]和A[m+1..n]归并成一个有序表B[s..n]
3.        int i=s;
4.        int j=m+1;
5.        int k=s;      //i,j是两个表的检测指针，k是存放指针
6.        while(i<=m && j<=n){
7.        if(A[i].key<=A[j].key){  //两两进行比较
8.                                  B[k]=A[i];
9.                                  i++;  k++;
10.       } else{
11.                                 B[k]=A[j];
12.                                 j++;  k++;
13.       }
14.       }
15.       while(i<=m)  {B[k]=A[i];i++;k++;} //将剩余的A[i..m]复制到B
16.       while(j<=n)  {B[k]=A[j];j++;k++;} //将剩余的A[j..n]复制到B
17. }//merge
```

7.3.3 归并排序-例

- 【例7.6】将关键字序列{21，25，49，25，16，8}进行归并排序



（a）递归过程　　（b）回退过程

7.3.3 归并排序-代码

算法7.2：2-路归并排序算法

```
1.  void recordList::MSort(recordList A, recordList &B, int s, int t){
2.      //将A[s..t]归并排序为B[s..t]
3.      int middle;
4.      recordList C;
5.      if(s==t) B[s]=A[s];          //区间内只有一个元素
6.      else{
7.          middle=(s+t)/2;      //将A[s..t]平分为A[s..m]和A[m+1..t]
8.          MSort(A, C, s, middle);  //递归地将A[s..middle]归并为有序的C[s..middle]
9.          MSort(A, C, middle+1, t);//递归地将A[middle+1..t]归并为有序的C[middle+1..t]
10.         merge(C, B, s, middle, t); //将C[s..middle]和C[middle+1..t]归并到B[s..t]
11.     }
12. }//Msort
13.
14. void recordList::MergeSort(recordList &L){
15.     MSort(L, L, 0, L.currentSize-1);
16. }
```

7.3.3 归并排序-复杂度分析

- 归并排序的运行时间并不依赖于输入数组中元素的组合方式，所以它避免了快速排序中的最差情况。
- 归并排序算法所花费的时间主要包括子序列划分时间，两个子序列的排序时间以及归并时间。
  - 两个子序列的排序时间与它们的序列长度之和有关，也就是每一趟的归并时间，复杂度为O(n)。总共需要归并[logn]趟。故对于一个长度为n的数组进行归并排序的时间代价为：
  - $T(n)=2T(n/2)+c*n$
- 归并排序的最大、最小以及平均时间代价为O(nlogn)。
- 归并排序算法需要一个与原待排序记录数组相同大小的辅助数组，因此归并排序占用较多的存储空间O(n)。
- 归并排序是一个稳定的排序方法。

7.3.3 归并排序

表4 归并排序算法的性能

时间复杂度			空间复杂度	稳定性	复杂性
平均情况	最坏情况	最好情况			
O(nlogn)	O(nlogn)	O(nlogn)	O(n)	稳定	较复杂

单选题 1分

设置

将两个各有n个元素的有序表归并成一个有序表，其最少的比较次数是（）

A n-1

B n

C 2n

D n/2

提交

单选题 1分

设置

已知输入数据13，24，7，1，8，9，11，56，34，51，2，77，5，请采用2路归并递归排序算法进行排序，2趟排序后的结果为（　　）

A 13，7，24，1，8，9，11，56，34，51，2，5，77

B 13，24，1，7，8，9，11，34，56，51，2，77，5

C 1，7，13，24，8，9，11，34，51，56，2，5，77

D 7，13，24，1，8，9，11，34，51，56，2，5，77

提交

7.3.4 树形选择排序-锦标赛排序

- 锦标赛选择排序，也称为树形选择排序（Tree Selection Sort），是一种根据锦标赛的思想来进行选择排序的方法。
- 锦标赛选择排序的基本思想：

（1）对n个待排序记录的关键字进行两两比较，并且记下它们之间比较后的优胜者；

（2）然后在上述优胜者之间再进行两两比较，如此重复进行；

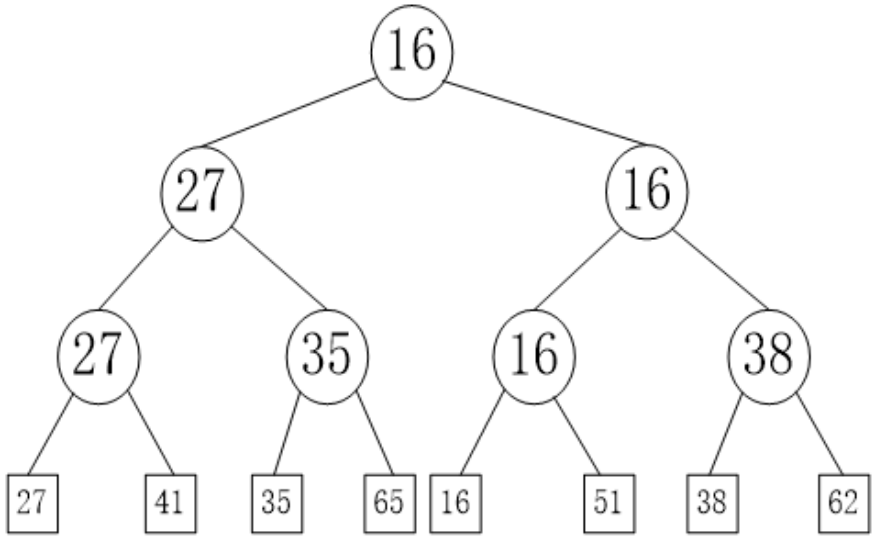
（3）进行完一次锦标赛选择排序后，最小的记录将被筛选出来，从而输出；

（4）将底层最小记录的关键字置为∞，然后从该叶子结点开始，和其左（右）兄弟的关键字进行比较，修改从叶子结点到根的路径上各结点的关键字，则根结点的关键字即为次小关键字，将之输出；

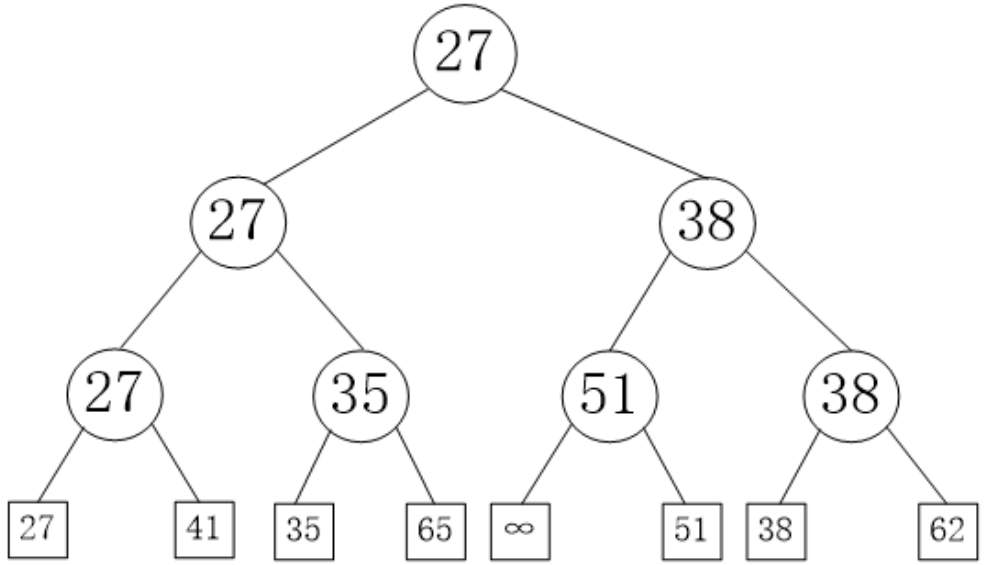
（5）重复执行，直到最后全部数据都已经输出。

7.3.4 树形选择排序-锦标赛排序

- ★注：使用锦标赛选择排序的时候，待排序记录的关键字不能相同，如果相同的话，将无法判断谁是优胜者。
- 【例7.7】对关键字序列{27，41，35，65，16，51，38，62}进行锦标赛选择排序，各趟排序过程如下图所示。

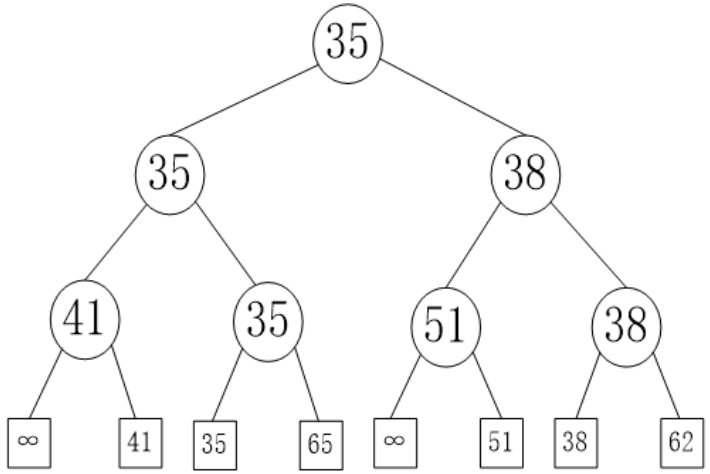


(a)第一趟排序，输出 16

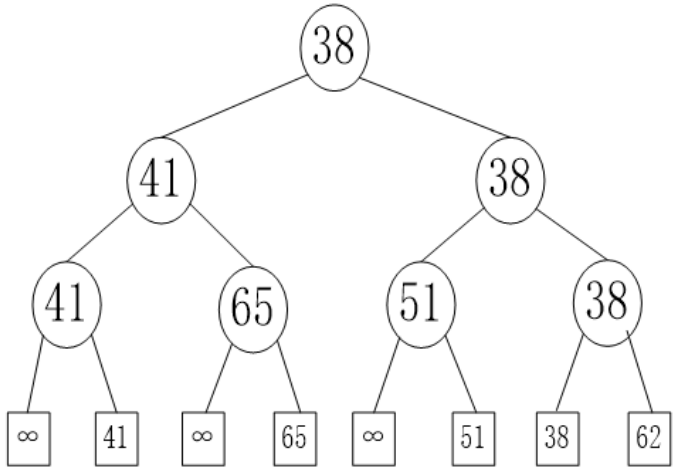


（b）把 16 置为∞，进行第二趟排序，输出 27

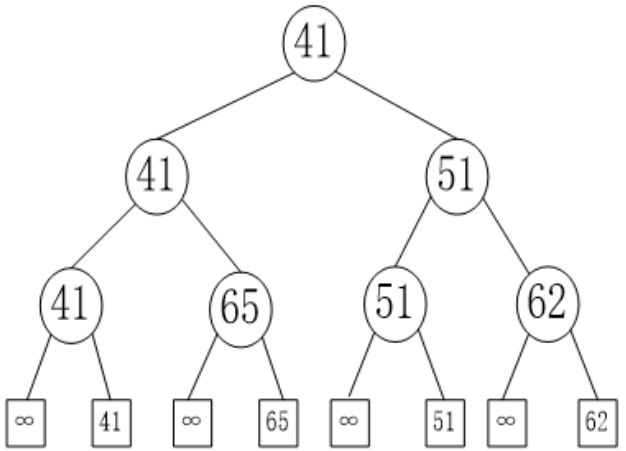
7.3.4 树形选择排序-锦标赛排序



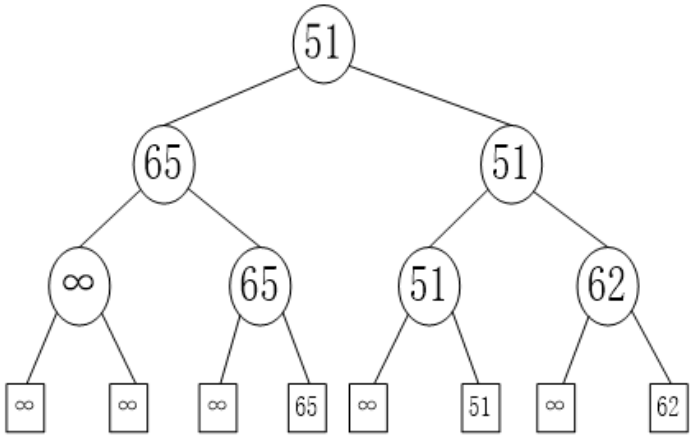
（c）把 27 置为∞,排序输出 35



（d）把 35 置为∞，排序输出 38

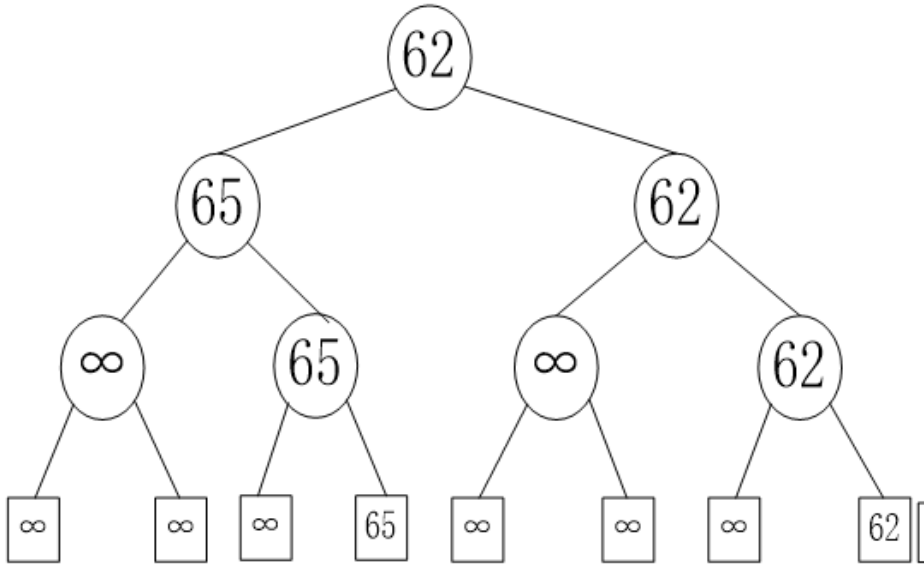


(e)把 38 置为∞，排序输出 41

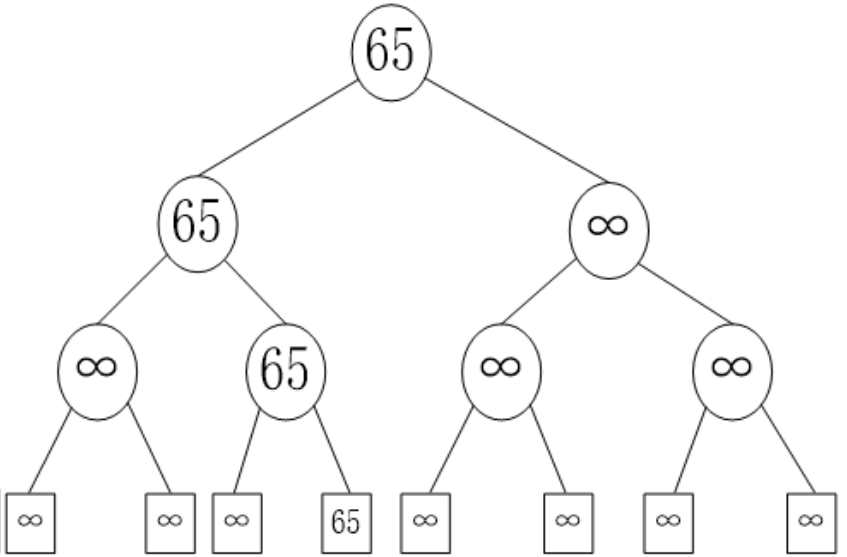


（f）把 41 置为∞，排序输出 51

7.3.4 树形选择排序-锦标赛排序



（g）把 51 置为∞，排序输出 62

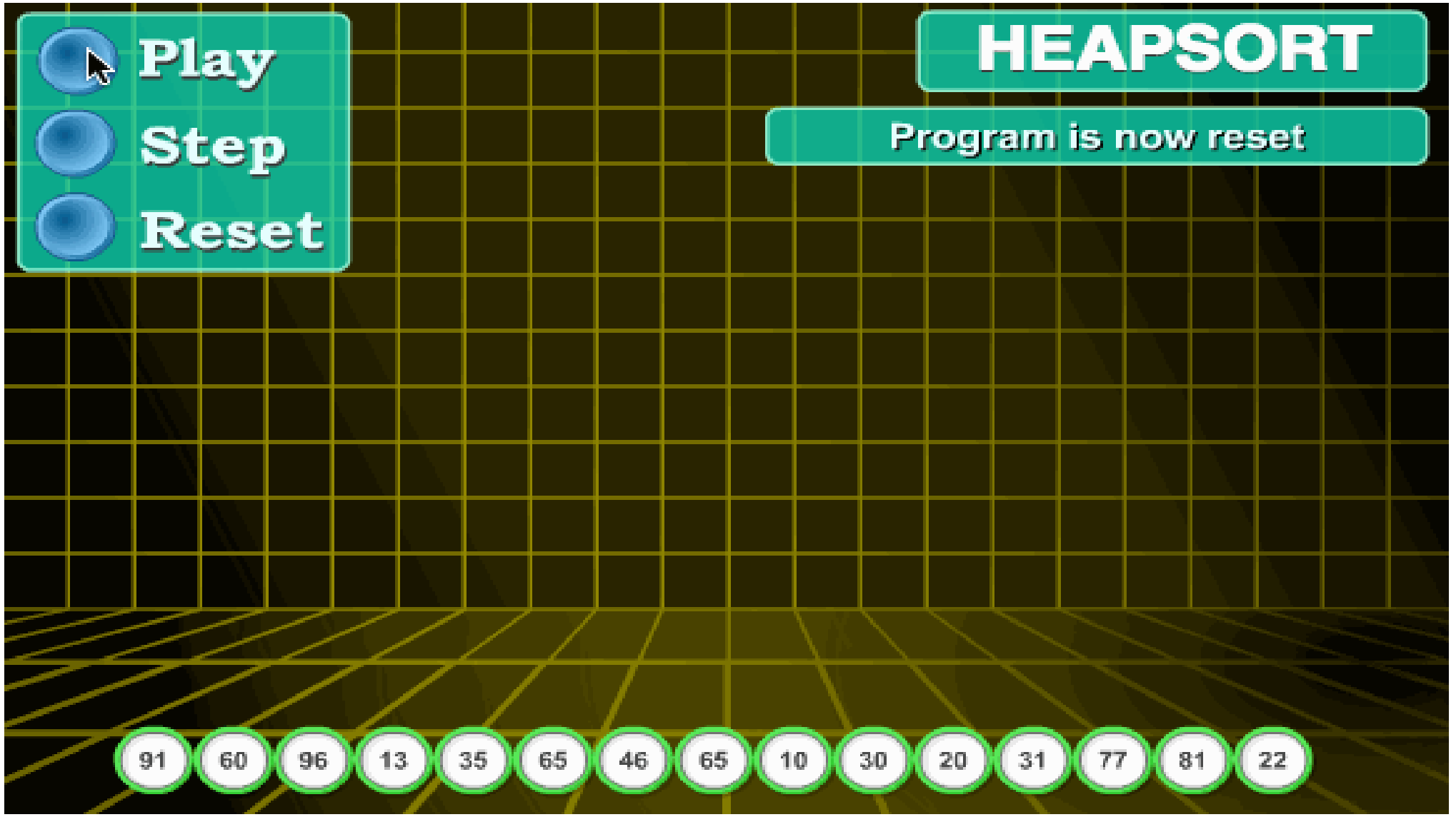


（h）把 62 置为∞，排序输出 65

7.3.4 树形选择排序-锦标赛排序

- 锦标赛选择排序可以在O(nlogn)时间内对n个元素进行排序。
  - 因为每趟锦标赛选择排序的时间复杂度为O(logn)，
  - 总共需要执行n-1次，因此整个排序过程所需要的时间为O(nlogn)。
- 由于它的时间复杂度不受待排序记录序列的影响，故它的最坏、平均和最好时间复杂度都为O(nlogn)。
- 锦标赛选择排序对n个元素进行排序时，需要n-1的额外存储空间，故它的空间复杂度为O(n)。

7.3.4 树形选择排序-堆排序



7.3.4 树形选择排序-堆排序-步骤

- 堆排序(Heap Sort)利用堆数据结构来保存待排记录的信息，因此是更加有效的选择排序。
- 堆排序分为两个步骤：
  - 第一步：根据初始输入数据，利用堆的调整算法FilterDown( )形成初始堆；
  - 第二步：通过一系列的记录交换和重新调整堆进行排序。
- 为了实现待排序记录按关键字从小到大排序，需要建立大顶堆。建立大顶堆的调整算法FilterDown()如下所示：

7.3.4 树形选择排序-堆排序

算法7.10：大顶堆的调整算法

```
1.  void MaxHeap::FilterDown(int i,int EndOfHeap){
2.      //调整使以i下标为根节点的子树为大顶堆
3.      int current=i;
4.      int child=2*i+1;  //i节点的左孩子节点的下标
5.      Type temp=heap[i];
6.      while(child<=EndOfHeap){
7.          if(child+1<=EndOfHeap&&heap[child].key<heap[child+1].key)
8.              child=child+1; //表明右孩子值比左孩子值大，孩子节点的下标值加一
9.          if(temp.key>=heap[child].key)
10.             break;    //说明已经符合大根堆条件，跳出循环
11.         else{
12.             heap[current]=heap[child];
13.             current=child;
14.             child=2*child+1;
15.         }
16.     }
17.     heap[current]=temp;  //找到temp应该放的最终位置
18. }
```

7.3.4 树形选择排序-堆排序

- 堆排序算法的具体过程为：

（1）若建立的堆满足大顶堆的条件，则堆的第一个记录R[0]具有最大的关键字，将R[0]与R[n-1]进行交换，把具有最大关键字的记录交换到最后，然后再对前面的n-1个记录，使用堆的调整算法FilterDown(0,n-2)，重新建立大顶堆。

（2）具有次最大关键字的记录又浮到了堆顶的位置，即R[0]的位置，再将R[0]和R[n-2]进行交换，并调用FilterDown(0,n-3)，对前面n-2个记录重新调整；

（3）如此重复执行，最后得到全部排好序的记录序列。

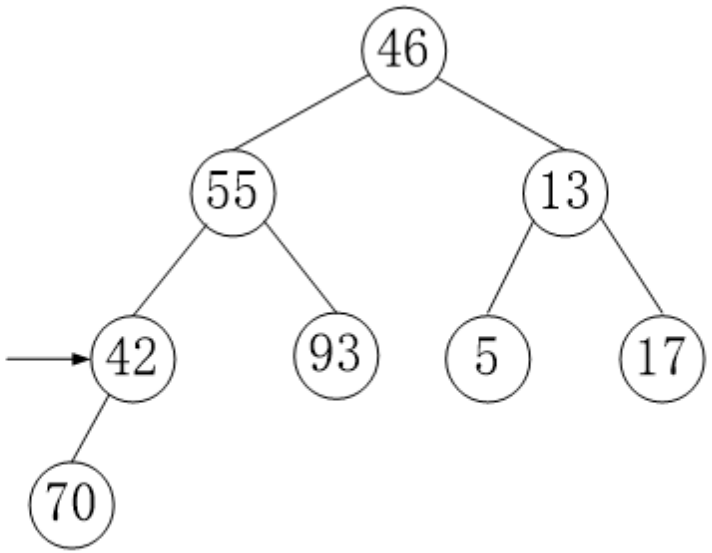
7.3.4 树形选择排序-堆排序

算法7.11：堆排序算法

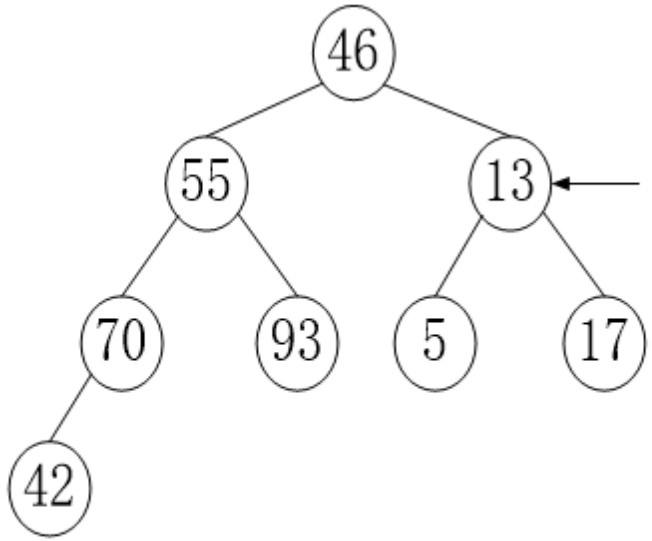
```
1.  void MaxHeap::HeapSort(){
2.    //对表heap[0]到heap[n-1]进行排序，使得表中各个待排序记录按其关键字，非
      递减排序
3.       for(int i=(CurrentSize-2)/2;i>=0;i--)
4.              FilterDown(i,CurrentSize-1); //建立初始大顶堆
5.       for(int i=CurrentSize-1;i>=1;i--){
6.              swap(heap[0],heap[i]);    //交换元素
7.              FilterDown(0,i-1);       //重新构建最大堆
8.       }
9.   }
```

7.3.4 树形选择排序-堆排序-例

【例7.8】，给定关键字序列{46,55,13,42,93,5,17,70}，建立初始堆的过程如图所示。建成如图所示的大顶堆后，堆排序的过程如图所示。

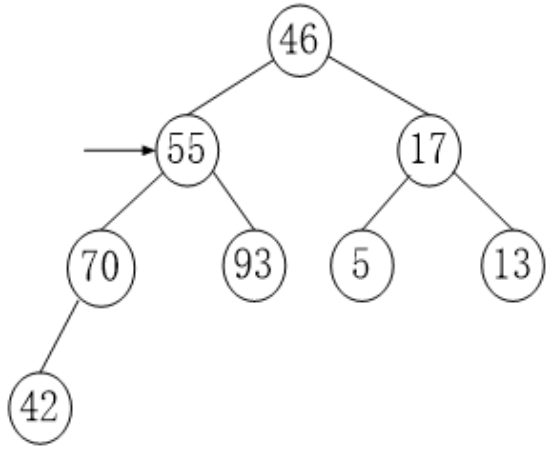


（a）初始无序的节点，从 42 开始调整

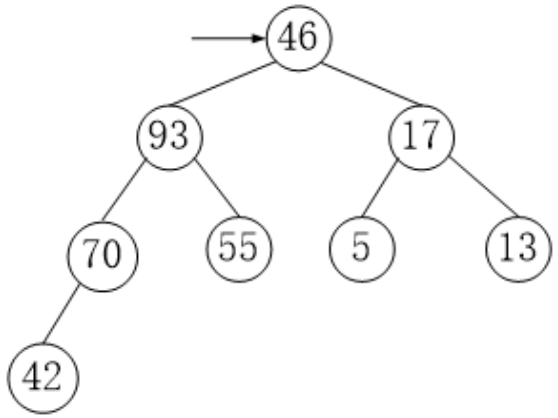


（b）将以 13 为根的子树调整成堆

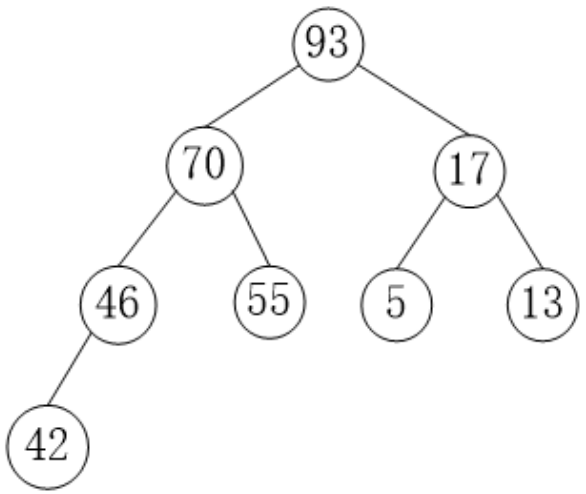
7.3.4 树形选择排序-堆排序



（c）将以 55 为根的子树调整成堆



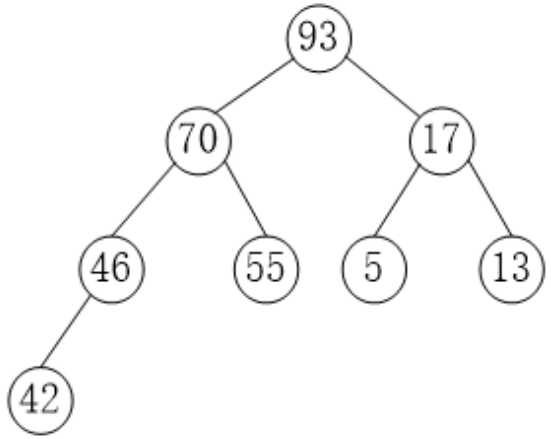
（d）将以 46 为根的子树调整为堆



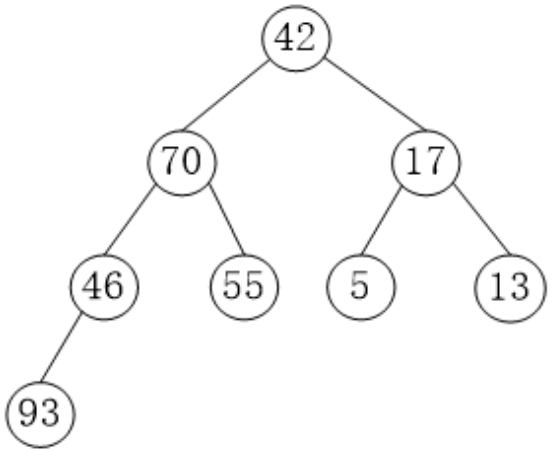
（e）成堆

7.3.4 树形选择排序-堆排序

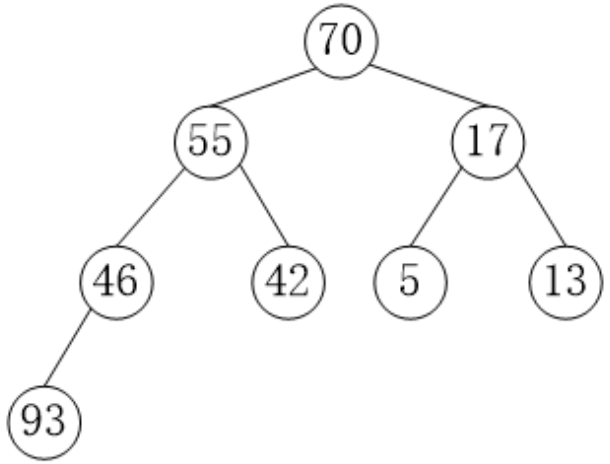
堆排序过程



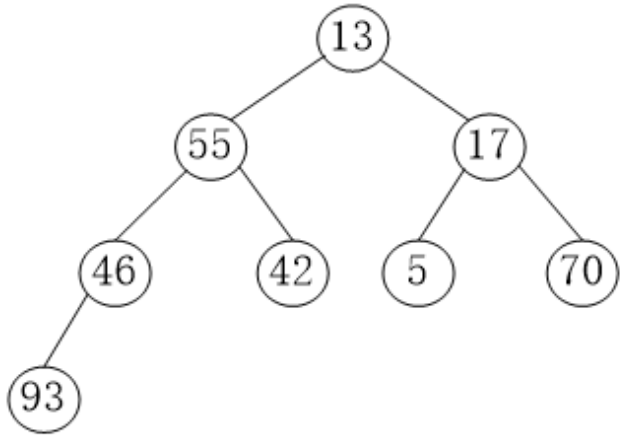
（a）初始堆



（b）将 93 与 42 交换



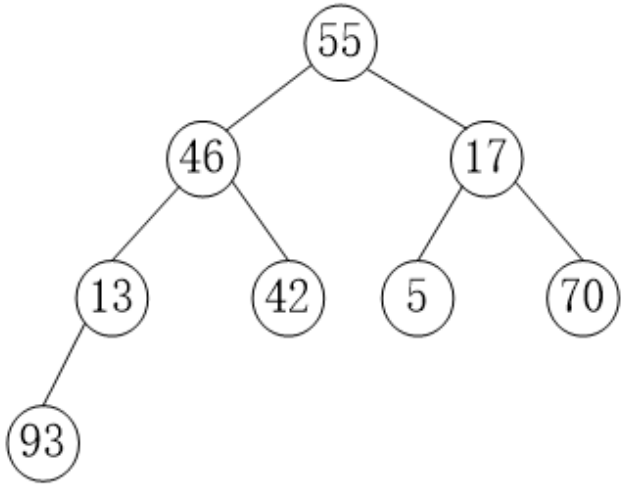
（c）前 7 个排序码重新建成堆



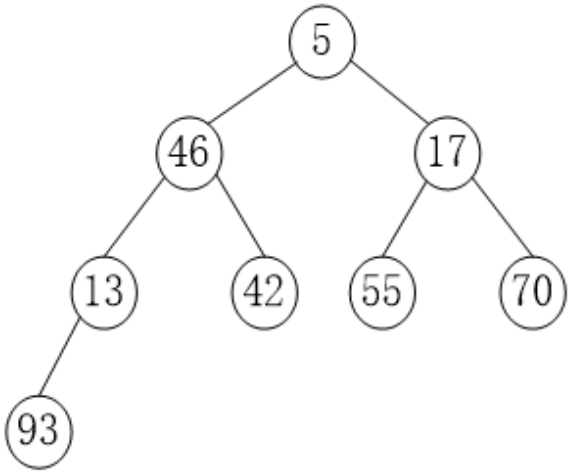
（d）将 70 和 13 交换

7.3.4 树形选择排序-堆排序

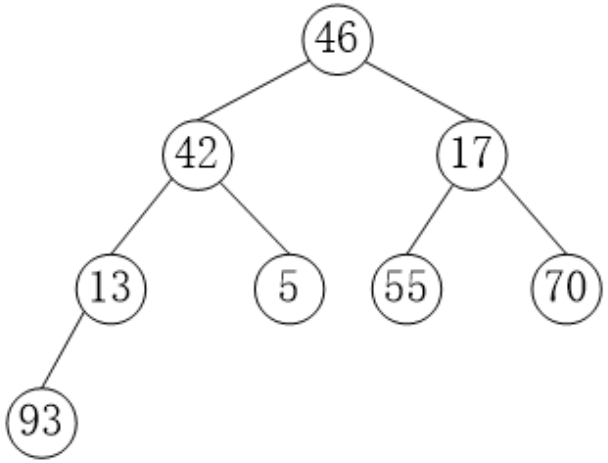
堆排序过程



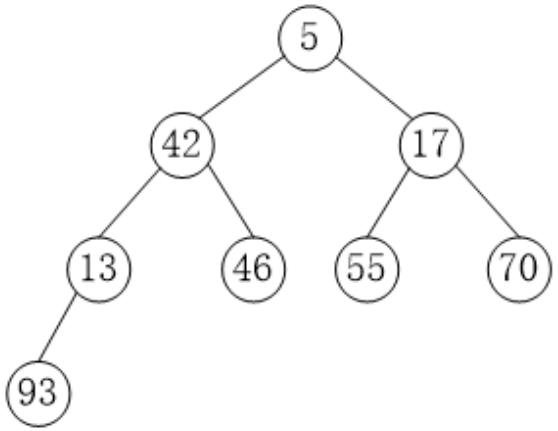
（e）前 6 个排序码重新建成堆



（f）55 和 5 交换



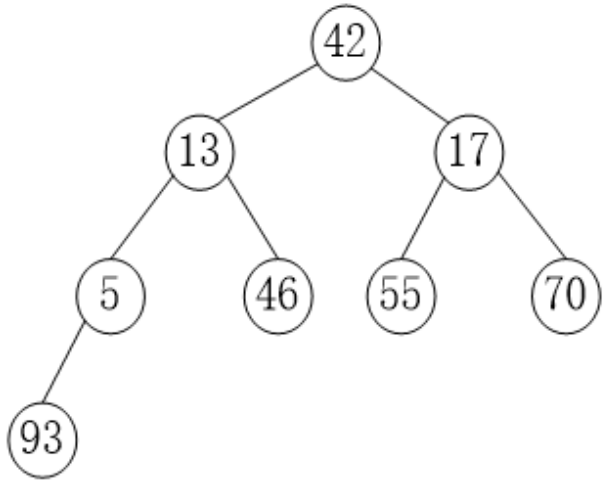
（g）前 5 个排序码重新建成堆



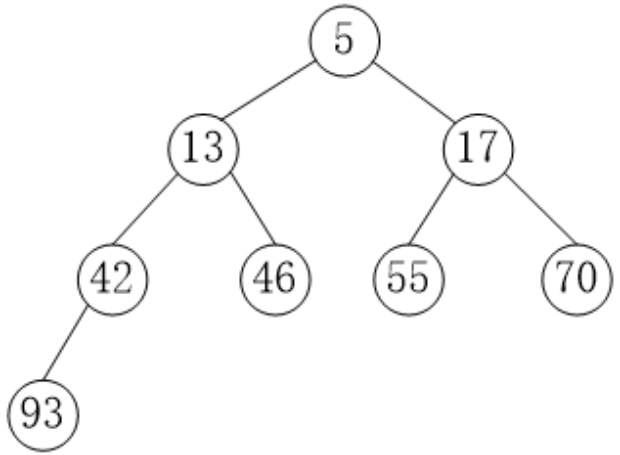
（h）46 和 5 交换

7.3.4 树形选择排序-堆排序

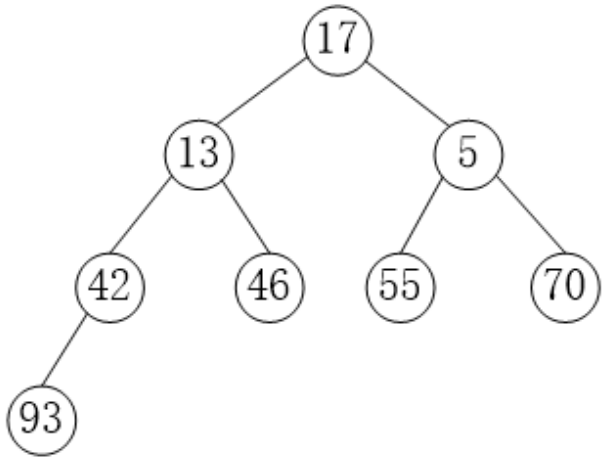
堆排序过程



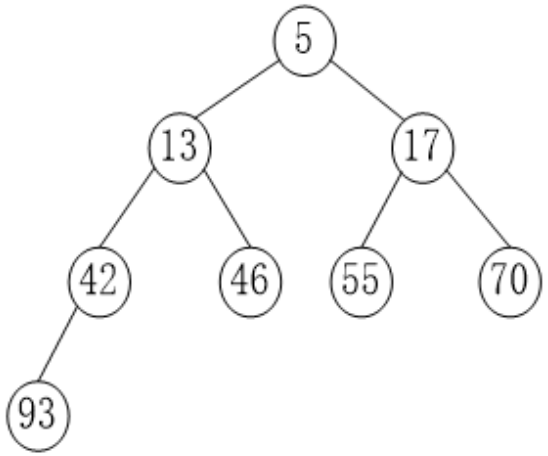
（i）前 4 个排序码重新建成堆



（j）42 和 5 交换

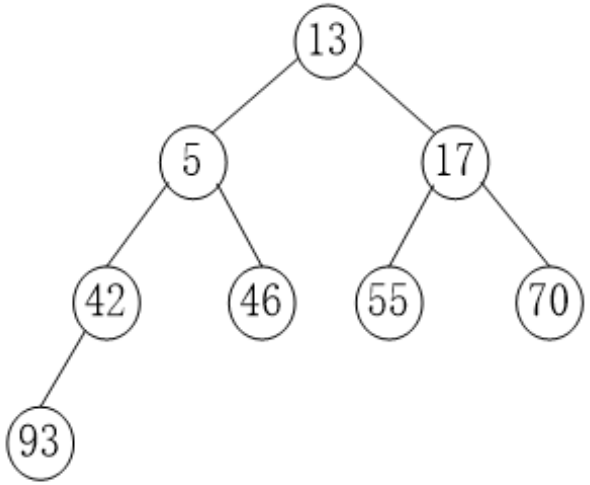


（k）前 3 个排序码重新建成堆

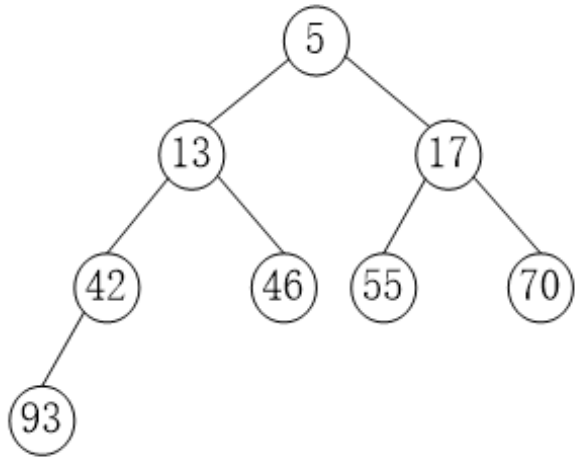


（l）17 和 5 交换

7.3.4 树形选择排序-堆排序



（m）前 2 个排序码重新建成堆



（n）13 和 5 交换

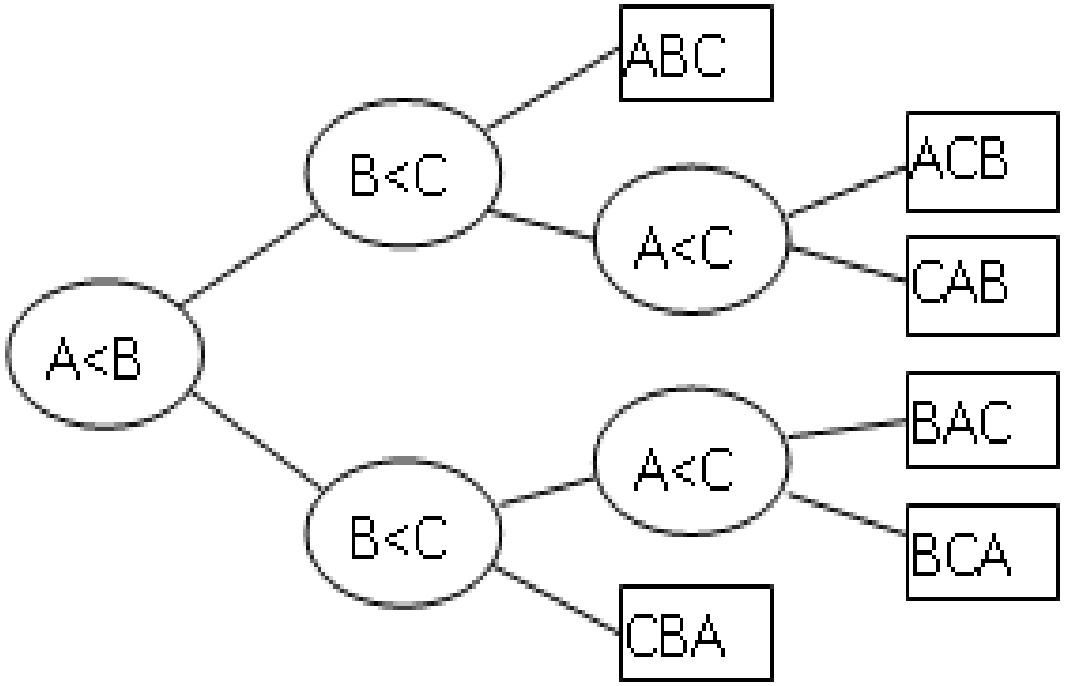
堆排序的时间复杂度为O(nlogn)。

堆排序算法过程中，用到了一个临时记录的变量，故其空间复杂度为O(1)。

堆排序是一种不稳定的排序算法。

7.4 关键字比较排序下界问题

- n个元素关键字比较判定树有n!叶子结点，判定树高度为$\log_2(n!)$，可以推导证明$\log_2(n!)\approx n\log_2 n$。基于关键字比较的排序算法都需要$O(n\log n)$次关键字的比较。故关键字比较排序的时间下界是$O(n\log n)$。
- 从理论上说，基于非比较的排序算法可以对现有的$O(n\log n)$排序算法进行改进。



7.5 非关键字比较的排序

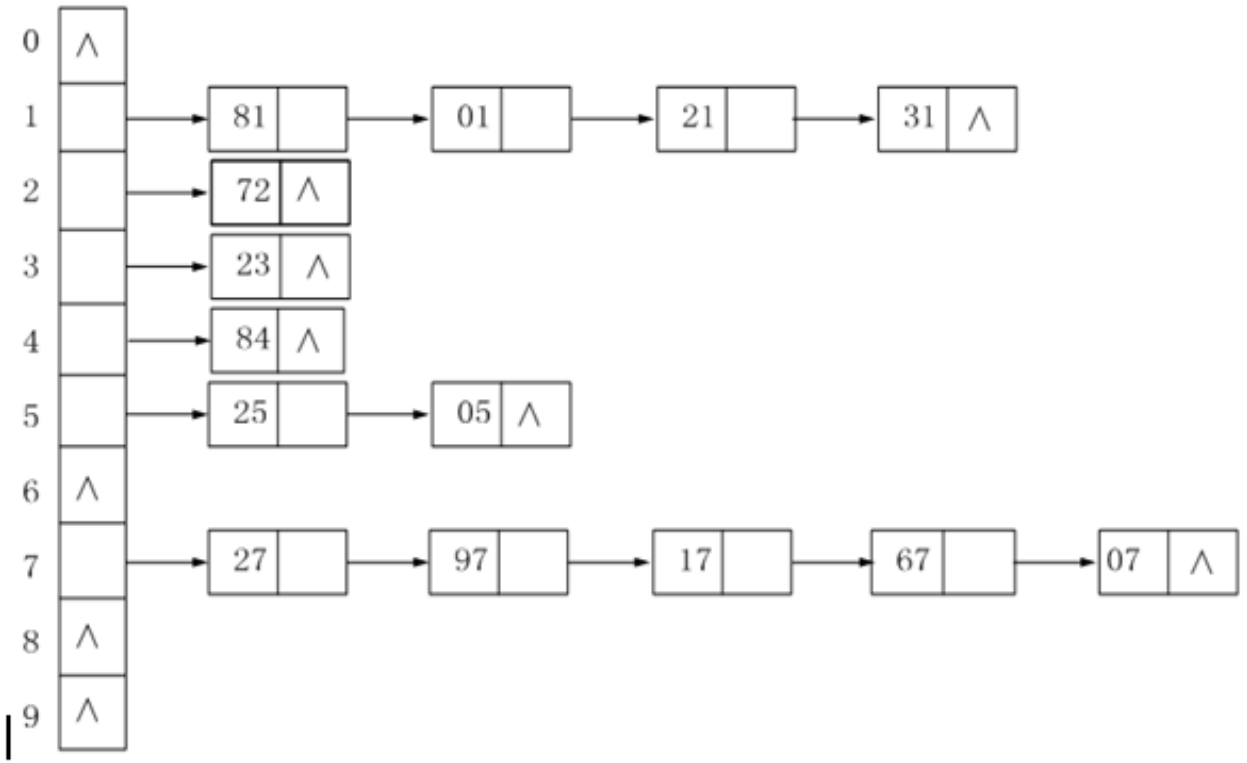
7.5.1 桶排序（基数排序）

- 基数排序的本质是借助于“分配”和“收集”算法对单关键字进行排序，基数排序是将关键字K_i在逻辑上看成是d个关键字（K_i^0，K_i^1，…，K_i^{d-1}），如果K_i^j（$0 \leqslant j \leqslant d-1$）有radix种可能的取值，称radix为基数。

7.5.1 基数排序-动画

7.5.1 基数排序-例

【例7.9】对关键字序列（27，81，01，97，17，23，72，25，05，67，84，07，21，31）进行基数排序的过程如图所示。

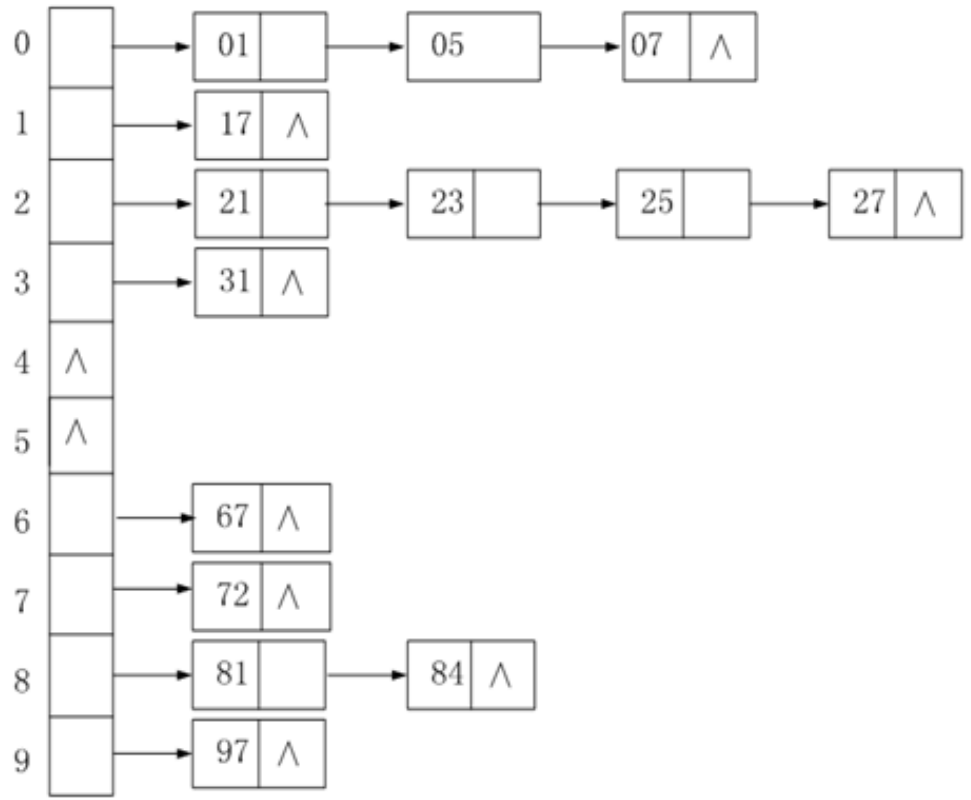


第 1 趟收集的结果为：81,01,21,31,72,23,84,25,05,27,97,17,67,07。

（a）按个位数进行分配（第 1 趟分配）

7.5.1 基数排序-例

【例7.9】对关键字序列（27，81，01，97，17，23，72，25，05，67，84，07，21，31）进行基数排序的过程如图所示。



第 2 趟收集的结果为：01,05,07,17,21,31,67,72,81,84,97

7.5.1 基数排序

算法7.12：基数排序算法

```
1.  //求出第i趟时，某一个元素的第i位数。
2.  int GetNDigit(int nNumber, int nIdx) {
3.  for (int i = nIdx-1; i > 0; i--) {
4.          nNumber /= 10;
5.      }
6.      return nNumber % 10;
7.  }
```

```
1.  //分配过程
2.  void Distribute(Element rec[],int n,int r,int d,int i,
    LinkList<Element>list[]){
3.  // 初始条件：r为基数，d为关键字位数，list[0..r-1]为
    被分配的线性表数组，i是第几位数
4.  // 操作结果：进行第i趟分配
5.          for(int j=0; j<n; j++)
6.          {  //进行第i趟分配
7.                  int index=GetNDigit((rec[j]),i);
8.          list[index].Insert(list[index].length()+1,rec[j]);
9.          }
10. }
```

7.5.1 基数排序

```
1.  //收集过程
2.  void Colect(Element rec[],int n,int r,int d,int
    i,LinkList<Element>list[]){
3.  //初始条件：r为基数，d为关键字的位数，
    list[0..r-1]为被分配的线性表数组
4.  //操作结果：进行第i趟收集
5.      for(int k=0,j=0; j<r; j++)
6.      {  //进行第i趟收集
7.              Element tmpRec;
8.              while(!list[j].Empty())
9.              {  //收集list[j]
10.                     list[j].Delete(1,tmpRec);
11.                     rec[k++]=tmpRec;
12.             }
13.     }
14. }
```

```
1.  //整个基数排序过程
2.  void RadixSort(Element rec[], int n, int r, int d){
3.  //初始条件：r为基数，d为关键字位数
4.  //操作结果：对rec进行基数排序
5.      LinkList<Element> *list;        //用于存储
    被分配的线性表数组
6.      list=new LinkList<Element>[r];
7.      for(int i=1;i<=d;i++)
8.      {  //第i趟分配与收集
9.              Distribute（rec, n, r, d, i, list); //分配
10.             Colect(rec, n, r, d, i, list);      //收集
11.     }
12.     delete []list;
13. }
```

7.5.1 基数排序-复杂度分析

- 假设数组的长度为n，基数为r，关键字位数为d，则每趟分配的时间为O(n)，每趟收集的时间为O(n+r)，共需进行d趟分配与收集，因此总的时间代价为O(d(2n+r))。
- 基数排序是稳定的排序方法，比较适合于关键字位数d和基数r较小的数组。

7.5.2 多关键字排序

- 假设有n个待排序记录序列：
  - {R[0]，R[1]，……，R[n-1]}
- 记录R[i]含有d个关键字（K_i^0，K_i^1，…，K_i^{d-1}），其中K_i^0称为**最主位关键字**，K_i^{d-1}称为**最次位关键字**，如果对任意的两个记录R[i]和R[j](0≤i<j≤n-1)都满足：
  - （K_i^0，K_i^1，…，K_i^{d-1}）≤（K_j^0，K_j^1，…，K_j^{d-1}）
- 则称记录序列按关键字（K^0，K^1，…，K^{d-1}）有序。

7.5.2 多关键字排序

- 多关键字序列的排序通常有两种方法：
  - 最低位优先法LSF（Least Significant First）
  - 最高位优先法MSF（Most Significant First）。
- 本节只介绍经常使用的最低位优先法。
- 最低位优先法的排序过程为：
  - 首先对最低位关键字K^{d-1}进行排序，
  - 然后再对高一位关键字K^{d-2}进行排序，
  - 依次类推，直到对K^0进行排序为止。
- 多关键字排序可以不比较关键字的大小，而是通过“分配”和“收集”来实现的。

7.5.2 多关键字排序

- 以扑克牌排序为例解释多关键字排序思想。
  - 每张扑克牌有两个“关键字”：花色和面值，花色是高位关键字，面值是低位关键字，假设有如下次序关系：
  - 花色：♥<♣<♠<♦
  - 面值：2<3<4<5<6<7<8<9<10<J<Q<K<A

7.5.2 多关键字排序

- 可以通过采用如下的“分配”和“收集”来对扑克牌进行排序：
  - 第1趟“分配”：按扑克牌的面值将扑克牌分配成不同面值的13堆。
  - 第1趟“收集”：将这13堆扑克牌按面值自小到大收集起来（3收集在2的上面，4收集在3的上面，……，A收集在“K”的上面）。
  - 第2趟“分配”：按扑克牌的花色分配成不同花色的4堆。
  - 第2趟“收集”：将这4堆扑克牌按花色自小至大收集起来（♣收集在♥的上面，♠收集在♣的上面，♦收集在♠的上面）。
- 这样经过两趟“分配”和“收集”后便可得到如上次序关系的扑克牌。
- 使用这种排序方法对每一个关键字进行排序时，不需要再分组，而是整个记录组都参加排序。

7.6 各种排序算法的比较

表 7-1 各种排序方法的比较

排序方法	时间复杂度			辅助存储	稳定性
	最好情况	平均情况	最坏情况		
直接插入排序	O(n)	O(n2)	O(n2)	O(1)	√
希尔排序	O(n1.5)			O(1)	×
冒泡排序	O(n)	O(n2)	O(n2)	O(1)	√
快速排序	O(nlog2n)	O(nlog2n)	O(n2)	O(log2n)	×
简单选择排序	O(n2)	O(n2)	O(n2)	O(1)	×
堆排序	O(nlog2n)	O(nlog2n)	O(nlog2n)	O(1)	×
归并排序	O(nlog2n)	O(nlog2n)	O(nlog2n)	O(n)	×
基数排序	O(d(n+rd))	O(d(n+rd))	O(d(n+rd))	O(rd)	√

单选题 1分

设置

已知输入数据1，2，8，5，9，11，34，56，51，77，请分析数据，采用（　　）排序算法效率最低

A 简单插入排序

B 选择排序

C 冒泡排序

D 三者取中法作为枢纽的快速排序

提交

End