第八章 排序

1. 排序的基本概念

1） **排序**：见算法导论

2） **算法的稳定性**：关键字相同的元素在排序之后相对位置不变；

3） **内部排序和外部排序**：①内部排序是指排序期间元素全部放在内存中的排序（关注算法的时空复杂度）；②外部排序是指无法全部同时在内存中，排序时需要换入换出（还要使读写磁盘的次数最少）；一般情况下，内部排序算法在执行过程中都要进行两种操作，比较和移动（复制？）。

2. 插入排序

1） **直接插入排序**

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1..j - 1].
4      i = j - 1
5      while i > 0 and A[i] > key
6          A[i+1] = A[i]
7          i = i - 1
8      A[i + 1] = key
```

```
void InsertSort(ElemType A[],int n){
    int i,j;
    for(i=2;i<=n;i++)              //依次将A[2]~A[n]插入到前面已排序序列
        if(A[i]<A[i-1]){           //若A[i]关键码小于其前驱，将A[i]插入有序表
            A[0]=A[i];             //复制为哨兵，A[0]不存放元素
            for(j=i-1;A[0]<A[j];--j)//从后往前查找待插入位置
                A[j+1]=A[j];       //向后挪位
            A[j+1]=A[0];           //复制到插入位置
        }
}
```

①**操作步骤**：查找出 L(i)在 L[1...i-1]中的插入位置 k；将 L[k...i-1]中所有元素依次后移一个位置；将 L(i)复制到 K(k)；

②**空间复杂度**：采用就地排序，仅使用常数个辅助单元，空间复杂度为 O(1)；

③**时间复杂度**：插入元素共进行 n-1 趟，每趟分为比较关键字和移动元素；最好情况下为顺序排列，每插入一个元素只需比较 1 次（左/右）而不需要移动元素（右）/需要移动 3 次（左），时间复杂度为 O(n)；最坏情况下为逆序排列，总比较次数为$\sum_{i=2}^{n} i$（右），总的移动次数为$\sum_{i=2}^{n} i+1$（右）；

④**稳定性**：每次总是从后向前比较再移动，因此相同元素相对位置不会改变，稳定；

⑤**适用性**：顺序存储、链式存储（从前向后）

2） **折半插入排序**

①**操作步骤**：直接插入排序中，比较和移动是一起的，而折半插入排序则把比较和移动分离，先查找元素待插入位置，再统一移动元素；

②**空间复杂度**：O(1)；

③**时间复杂度**：折半插入排序减少了比较元素的次数，约为 O($n\log_2 n$)比较次数仅取决于输入规模 n;元素移动次数仍为 O(n^2)；因此折半插入时间复杂度为 O(n^2)；

```
void InsertSort(ElemType A[],int n){
    int i,j,low,high,mid;
    for(i=2;i<=n;i++){             //依次将A[2]~A[n]插入前面的已排序序列
        A[0]=A[i];                 //将A[i]暂存到A[0]
        low=1;high=i-1;            //设置折半查找的范围
        while(low<=high){          //折半查找(默认递增有序)
            mid=(low+high)/2;      //取中间点
            if(A[mid]>A[0]) high=mid-1; //查找左半子表
            else low=mid+1;        //查找右半子表
        }
        for(j=i-1;j>=high+1;--j)
            A[j+1]=A[j];           //统一后移元素，空出插入位置
        A[high+1]=A[0];            //插入操作
    }
}
```

④**稳定性**：一直到 low>high 时才停止折半查找，应将[low,i-1]内元素全部右移，并将 A[0]复制到 low 所指位置；当 mid 所指元素等于当前元素时，应继续令 low=mid+1 查找右部元素，以保证稳定性；最终将当前元素插入到 low 所指的位置（即 high+1）；

⑤**适用性**：只适用于顺序存储，不适用于链表；

3） **希尔排序（缩小增量排序）**

①**操作步骤**：直接插入排序算法适合基本有序的排序表和数据量不大的排序表；先将待排序表分割成 L[i,i+d,i+2d,...,i+kd]的特殊子表，即把相隔某个增量的记录组成一个子表；对各个子表进行直接插入排序；缩小增量 d，重复上述过程直到 d=1 为止；程序中对各个子表交替排序；

```
void ShellSort(ElemType A[],int n){
//A[0]只是暂存单元，不是哨兵，当j<=0时，插入位置已到
    for(dk=n/2;dk>=1;dk=dk/2)          //步长变化
        for(i=dk+1;i<=n;++i)
            if(A[i]<A[i-dk]){   //需将A[i]插入有序增量子表
                A[0]=A[i];              //暂存在A[0]
                for(j=i-dk;j>0&&A[0]<A[j];j-=dk)
                    A[j+dk]=A[j];       //记录后移，查找插入的位置
                A[j+dk]=A[0];           //插入
            }//if
}
```

②**空间复杂度**：O(1)；

③**时间复杂度**：O($n^{1.3}$)；最坏情况下时间复杂度为 O(n^2)；

④**稳定性**：当相同关键字被划分到不同子表时可能会改变相对次序，因此不稳定；

⑤**适用性**：只适用于顺序存储，不适用于链表；

3. 交换排序

1） **冒泡排序**

```
void BubbleSort(ElemType A[],int n){
    for(i=0;i<n-1;i++){
        flag=false;                 //表示本趟冒泡是否发生交换的标志
        for(j=n-1;j>i;j--)          //一趟冒泡过程
            if(A[j-1]>A[j]){        //若为逆序
                swap(A[j-1],A[j]); //交换
                flag=true;
            }
        if(flag==false)
            return;                 //本趟遍历后没有发生交换，说明表已经有序
    }
}
```

①**操作步骤**：从后向前（或相反）两两比较相邻元素的值，若为逆序（即 A[i-1]>A[i]），则交换，直到序列比较完成；称为第一趟冒泡，结果是将最小的元素交换到第一个位置；没有交换发生则说明有序；

②**空间复杂度**：O(1)；

③**时间复杂度**：当初始序列有序时，比较次序为 n-1，移动次数为 0，最好情况时间复杂度为 O(n)；最坏情况下为逆序排列，需要进行 n-1 趟排序，第 i 趟排序要进行 n-i 次关键字比较和交换，每次交换移动 3 次元素，最坏情况下时间复杂度为 $O(n^2)$；

④**稳定性**：当 i>j 且 A[i]=A[j]时不会交换，因此稳定；

⑤**适用性**：适用于链表；

2） 快速排序

```
void QuickSort(ElemType A[],int low,int high){
    if(low<high){                                   //递归跳出的条件
    //Partition()就是划分操作，将表A[low…high]划分为满足上述条件的两个子表
            int pivotpos=Partition(A,low,high);     //划分
            QuickSort(A,low,pivotpos-1);    //依次对两个子表进行递归排序
            QuickSort(A,pivotpos+1,high);
    }
}
int Partition(ElemType A[],int low,int high){  //一趟划分
    ElemType pivot=A[low]; //将当前表中第一个元素设为枢轴，对表进行划分
    while(low<high){        //循环跳出条件
        while(low<high&&A[high]>=pivot) --high;
        A[low]=A[high];     //将比枢轴小的元素移动到左端
        while(low<high&&A[low]<=pivot) ++low;
        A[high]=A[low];     //将比枢轴大的元素移动到右端
    }
    A[low]=pivot;           //枢轴元素存放到最终位置
    return  low;            //返回存放枢轴的最终位置
}
```

```
QUICKSORT(A, p, r)
1  if  p < r
2       q = PARTITION(A, p, r)
3       QUICKSORT(A, p, q−1)
4       QUICKSORT(A, q+ 1, r)
PARTITION(A, p, r)
1  x = A[r]
2  i = p−1
3  for j = p to r−1
4       if A[j]≤x
5            i = i + 1
6            exchange A[i] with A[j]
7  exchange A[i+ 1] with A[r]
8  return i + 1
```

1. 若 $p \leqslant k \leqslant i$，则 $A[k] \leqslant x$。
2. 若 $i+1 \leqslant k \leqslant j-1$，则 $A[k] > x$。
3. 若 $k=r$，则 $A[k]=x$。

①**操作步骤**：在待排序表 L[1...n]中任取一个元素 pivot 作为枢轴，通过一趟排序将待排序表划分为独立的两部分 L[1...k-1]和 L[k+1...n]，使得 L[1...k-1]中的所有元素小于 pivot，L[k+1...n]中的所有元素大于等于 pivot，则 pivot 放在最终位置 L(k)上，这个过程称为一趟快速排序；一趟排序和一次划分不等价，一趟排序可能确定多个元素的位置，即进行多次划分；

②**空间复杂度**：快速排序是递归的，需要借助一个递归工作栈来保存每层递归调用的必要信息，最好情况下为 $O(\log_2 n)$，最坏情况下要进行 n-1 次递归调用，栈深为 O(n)；平均情况下，栈的深度为 $O(\log_2 n)$；

③**时间复杂度**：快速排序的最坏情况发生在两个区域分别包含 n-1 个元素和 0 个元素时，对应于初始排序表基本有序或基本逆序时，最坏情况的时间复杂度为 $O(n^2)$；最理想情况发生在平衡地划分，时间复杂度为 $O(n\log_2 n)$；快速排序是所有内部排序算法中平均性能最优的排序算法；

④**稳定性**：若右端区间有两个关键字相同，且均小于基准值，交换到左区间后相对位置会发生变化，因此不稳定；

4. 选择排序

1） 简单选择排序

```
void SelectSort(ElemType A[],int n){
    for(i=0;i<n-1;i++){                 //一共进行 n-1 趟
        min=i;                          //记录最小元素位置
        for(j=i+1;j<n;j++)              //在A[i…n-1]中选择最小的元素
            if(A[j]<A[min]) min=j;      //更新最小元素位置
        if(min!=i) swap(A[i],A[min]);   //封装的swap()函数共移动元素 3 次
    }
}
```

①**操作步骤**：假设排序表为 L[1...n]，第 i 趟排序即从 L[i...n]中选择关键字最小的元素与 L(i)交换，每趟排序可以确定一个元素的最终位置；

②**空间复杂度**：O(1)；

③**时间复杂度**：元素移动次数少，最多为 3(n-1)次，最好为 0 次；元素比较次数与初始状态无关，始终是 n(n-1)/2 次，因此时间复杂度是 $O(n^2)$；

④**稳定性**：在第 i 趟找到最小元素后，和第 i 个元素交换，可能导致第 i 个元素与其含有相同关键字元素的相对位置发生改变，因此不稳定。

⑤**适用性**：适用于链表；

2） 堆排序