

# 第十一次直播 排序

- 交换排序有：冒泡排序，选择排序

## 冒泡排序

冒泡排序的基本思想是：从后往前（或从前往后）两两比较相邻元素的值，（若  $A[j-1] > A[j]$ ），则交换它们，直到序列比较完。我们称它为第一趟冒泡，结果是将最小的元素交换到待排序列的第一个位置。关键字最小的元素如气泡一般逐渐往上“漂浮”直至“水面”。下一趟冒泡时前一趟确定的最小元素不再参与比较，每趟冒泡的结果是把序列中的最小元素放到了序列的最终位置。∴.....这样最多做  $n-1$  趟冒泡就能把所有元素排好序

冒泡过程

数组开头	49	13	13	13	13	13	13
	38	49	27	27	27	27	27
	65	38	49	38	38	38	38
	97	65	38	49	49	49	49
	76	97	65	49	49	49	49
	13	76	97	65	65	65	65
	27	27	76	97	76	76	76
数组末尾	49	49	49	76	97	97	97
	初始状态	第一趟后	第二趟后	第三趟后	第四趟后	第五趟后	第六趟后

CSDN @QuantumYou

冒泡排序分析

小知识点

- 内存copy接口，当你copy整型数组，或者浮点型时，要用 `memcpy`
- `strcpy` 是遇到0 停止拷贝
- 添加 `flag` 的原因在于减少排序次数，发现没有交换代码

总体排序代码

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4  #include <string.h>
5  typedef int ElemType;
6  typedef struct{
7      ElemType *elem; //存储元素的起始地址
8      int TableLen; //元素个数
9  }SSTable;
10
11 void ST_Init(SSTable &ST, int len)
12 {

```

```

13     ST.TableLen=len;
14     ST.elem=(ElemType *)malloc(sizeof(ElemType)*ST.TableLen);
15     int i;
16     srand(time(NULL)); //随机数生成, 每一次执行代码就会得到随机的10个元素
17     for(i=0;i<ST.TableLen;i++)
18     {
19         ST.elem[i]=rand()%100; //生成的是0-99之间
20     }
21 }
22 void ST_print(SSTable ST)
23 {
24     for(int i=0;i<ST.TableLen;i++)
25     {
26         printf("%3d",ST.elem[i]);
27     }
28     printf("\n");
29 }
30 void swap(ElemType &a,ElemType &b)
31 {
32     ElemType tmp;
33     tmp=a;
34     a=b;
35     b=tmp;
36 }
37 // 64 94 95 79 69 84 18 22 12 78
38 // 12 64 94 95 79 69 84 18 22 78
39 void BubbleSort(ElemType A[],int n)
40 {
41     int i,j;
42     bool flag;
43     for(i=0;i<n-1;i++) //i最多访问到8
44     {
45         flag=false;
46         for(j=n-1;j>i;j--) //把最小值就放在最前面
47         {
48             if(A[j-1]>A[j])
49             {
50                 swap(A[j-1],A[j]);
51                 flag=true;
52             }
53         }
54         if(false==flag)
55             return;
56     }
57 }
58
59 void BubbleSort1(ElemType A[], int n)
60 {
61     int i, j,flag;
62     for (i=0;i<n-1;i++) //i是控制有多少个有序了
63     {
64         flag = 0;
65         for (j = n-1; j>i;j--) //内层控制比较, 交换
66         {
67             if (A[j - 1] > A[j])
68             {
69                 swap(A[j - 1], A[j]);
70                 flag = 1;

```

```

71         }
72     }
73     if (0 == flag)
74     {
75         break;
76     }
77 }
78 }
79 // 64 94 95 79 69 84 18 22 12 78
80 //比64小的放在左边，比64大的放在右边
81 //int Partition(ElemType A[],int low,int high)
82 //{
83 //    ElemType pivot=A[low];
84 //    while(low<high)
85 //    {
86 //        while(low<high&&A[high]>=pivot)
87 //            --high;
88 //        A[low]=A[high];
89 //        while(low<high&&A[low]<=pivot)
90 //            ++low;
91 //        A[high]=A[low];
92 //    }
93 //    A[low]=pivot;
94 //    return low;
95 //}
96
97
98
99 int Partition(int* arr, int left, int right)
100 {
101     int k, i;
102     for (k = i = left; i < right; i++)
103     {
104         if (arr[i] < arr[right])
105         {
106             swap(arr[i], arr[k]);
107             k++;
108         }
109     }
110     swap(arr[k], arr[right]);
111     return k;
112 }
113 //递归实现
114 void QuickSort(ElemType A[],int low,int high)
115 {
116     if(low<high)
117     {
118         int pivotpos=Partition(A,low,high);//分割点左边的元素都比分割点要小，右边的
119         //比分割点大
120         QuickSort(A,low,pivotpos-1);
121         QuickSort(A,pivotpos+1,high);
122     }
123 }
124 //冒泡排序与快速排序
125 int main()
126 {
127     SSTable ST;

```

```
128     ElemType A[10]={ 64, 94, 95, 79, 69, 84, 18, 22, 12 ,78};
129     ST_Init(ST,10);//初始化
130     //memcpy(ST.elem,A,sizeof(A));//内存copy接口，当你copy整型数组，或者浮点型
    时，要用memcpy
131     ST_print(ST);
132     //BubbleSort1(ST.elem,10);//冒泡排序
133     QuickSort(ST.elem,0,9);
134     ST_print(ST);
135     system("pause");
136 }
```

## 快速排序

- 分治思想
- 快速排序的核心是分治思想：假设我们的目标依然是按从小到大的顺序排列，我们找到数组中的一个分割值，把比分割值小的数都放在数组的左边，把比分割值大的数都放在数组的右边，这样分割值的位置就被确定。数组一分为二，我们只需排前一半数组和后一半数组，复杂度直接减半采用这种思想，不断地进行递归，最终分割得只剩一个元素时，整个序列自然就是有序的

快速排序分析：

1↵	最初↵	k↵ i↵ 3 87 2 93 78 56 61 38 12 40↵
2↵	第 1 次比较后↵	k↵ i↵ 3 87 2 93 78 56 61 38 12 40↵
3↵	第 2 次比较后↵	k↵ i↵ 3 87 2 93 78 56 61 38 12 40↵
4↵	第 3 次比较发生交换后效果↵	k↵ i↵ 3 2 87 93 78 56 61 38 12 40↵
5↵	第 3 次将要发生交换↵	k↵ i↵ 3 2 87 93 78 56 61 38 12 40↵
6↵	第 3 次发生交换后↵	k↵ i↵ 3 2 38 93 78 56 61 87 12 40↵
7↵	第 4 次发生交换后↵	k↵ i↵ 3 2 38 12 78 56 61 87 93 40↵
8↵	最后一次交换↵	k↵ i↵ 3 2 38 12 78 56 61 87 93 40↵

CSDN @QuantumYou

i 用来遍历数组的下标

k 用来比40小的元素将要存的下标

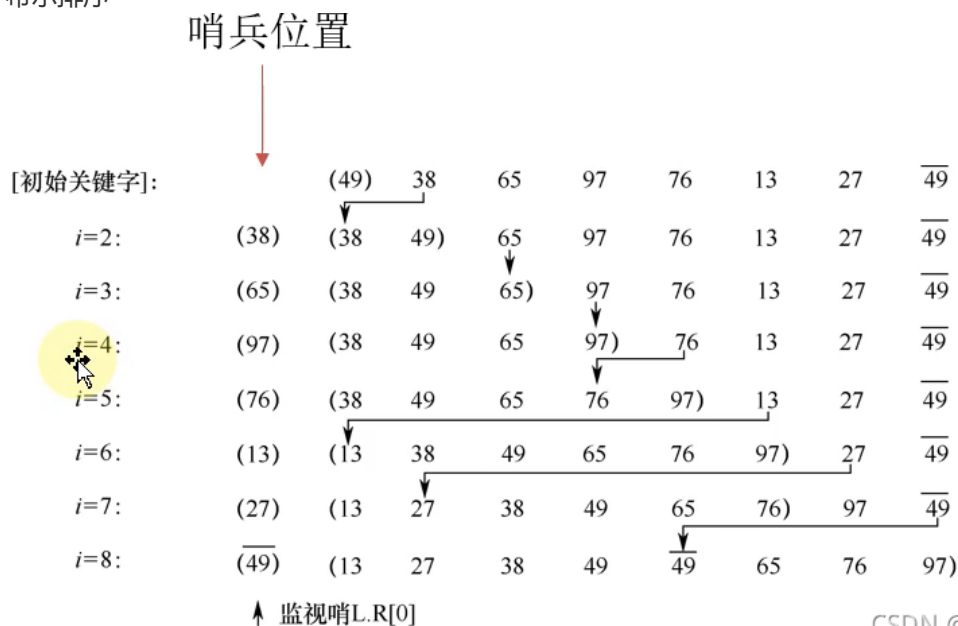
```

1  int Partition1(int* arr, int left, int right)
2  {
3      int k, i; //k记录要放入比分割值小的数据的位置
4      for (i = left, k = left; i < right; i++)
5      {
6          if (arr[i] < arr[right])
7          {
8              swap(arr[k], arr[i]);
9              k++;
10         }
11     }
12     swap(arr[k], arr[right]);
13     return k;
14 }
```

# 直接插入排序

插入排序分为

- 1、直接插入排序
- 2、折半插入排序
- 3、希尔排序



CSDN @QuantumYou

插入排序的总体代码如下

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4 typedef int ElemType;
5 typedef struct{
6     ElemType *elem; // 整型指针
7     int TableLen;
8 }SSTable;
9
10 void ST_Init(SSTable &ST, int len)
11 {
12     ST.TableLen = len + 1; // 实际申请11个元素的空间
13     ST.elem = (ElemType *) malloc(sizeof(ElemType) * ST.TableLen);
14     int i;
15     srand(time(NULL));
16     for(i = 0; i < ST.TableLen; i++)
17     {
18         ST.elem[i] = rand() % 100; // 随机了11个数，但是第一个元素是没有用到的
19     }
20 }
21 void ST_print(SSTable ST)
22 {
23     for(int i = 0; i < ST.TableLen; i++)
24     {
25         printf("%3d", ST.elem[i]);
26     }
27     printf("\n");
28 }
29 // 插入排序，从小到大排序，升序
30 void InsertSort(ElemType A[], int n)
31 {
```

```

32     int i,j;
33     //24 66 94  2 15 74 28 51 22 18  2
34     for(i=2;i<=n;i++)//第零个元素是哨兵，从第二个元素开始拿，往前面插入
35     {
36         if(A[i]<A[i-1])
37         {
38             A[0]=A[i];//放到暂存位置，A[0]即是暂存，也是哨兵
39             for(j=i-1;A[0]<A[j];--j)//移动元素，内层循环控制有序序列中的每一个元素和
要插入的元素比较
40                 A[j+1]=A[j];
41             A[j+1]=A[0];//把暂存元素插入到对应位置
42         }
43     }
44 }
45 //折半查找 插入排序
46 void MidInsertSort(ElementType A[],int n)
47 {
48     int i,j,low,high,mid;
49     for(i=2;i<=n;i++)
50     {
51         A[0]=A[i];
52         low=1;high=i-1;
53         while(low<=high)//先通过二分查找找到待插入位置
54         {
55             mid=(low+high)/2;
56             if(A[mid]>A[0])
57                 high=mid-1;
58             else
59                 low=mid+1;
60         }
61         for(j=i-1;j>=high+1;--j)
62             A[j+1]=A[j];
63         A[high+1]=A[0];
64     }
65 }
66 //希尔排序
67 //多轮插入排序，考的概率很低，因为编写起来复杂，同时效率并不如快排，堆排
68 void ShellSort(ElementType A[],int n)
69 {
70     int dk,i,j;
71     // 73 29 74 51 29 90 37 48 72 54 83
72     for(dk=n/2;dk>=1;dk=dk/2)//步长变化
73     {
74         for(i=dk+1;i<=n;++i)//以dk为步长进行插入排序
75         {
76             if(A[i]<A[i-dk])
77             {
78                 A[0]=A[i];
79                 for(j=i-dk;j>0&&A[0]<A[j];j=j-dk)
80                     A[j+dk]=A[j];
81                 A[j+dk]=A[0];
82             }
83         }
84     }
85 }
86
87 int main()
88 {

```

```

89     SSTable ST;
90     ST_Init(ST,10); //实际申请了11个元素空间
91     ST_print(ST);
92     InsertSort(ST.elem,10);
93     //MidInsertSort(ST.elem,10);
94     //ShellSort(ST.elem,10);
95     ST_print(ST);
96     system("pause");
97 }

```

## 第十二次直播 排序

### 快速排序

- 挖坑法, 第一次学习的方法

```

1 // 挖坑法, 王道书上使用的方法, 最左边作为分割值
2 int Partition(ElemType A[],int low,int high)
3 {
4     ElemType pivot=A[low]; //把最左边的值暂存起来
5     while(low<high)
6     {
7         while(low<high&&A[high]>=pivot) //让high从最右边找, 找到比分割值小, 循环结
束
8             --high;
9         A[low]=A[high];
10        while(low<high&&A[low]<=pivot) //让low从最左边开始找, 找到比分割值大, 就结束
11            ++low;
12        A[high]=A[low];
13    }
14    A[low]=pivot;
15    return low;
16 }

```

### 折半排序

- 不难看出折半插入排序仅减少了比较元素的次数, 约为  $O(m \log_2 n)$ , 该比较次数与待排序表的初始状态无关, 仅取决于表中的元素个数  $n$ ; 而元素的移动次数并未改变, 它依赖于待排序表的初始状态。因此, 折半插入排序的时间复杂度仍为  $O(n^2)$ , 但对于数据量不很大的排序表, 折半插入排序往往能表现出很好的性能

折半查找-->二分查找

```

1 //折半查找 插入排序, 考的很少
2 void MidInsertSort(ElemType A[],int n)
3 {
4     int i,j,low,high,mid;
5     for(i=2;i<=n;i++)
6     {
7         A[0]=A[i];
8         low=1;high=i-1; //low有序序列的开始, high有序序列的最后
9         while(low<=high) //先通过二分查找找到待插入位置
10        {
11            mid=(low+high)/2;
12            if(A[mid]>A[0])

```



```

13         high=mid-1;
14     else
15         low=mid+1;
16     }
17     for(j=i-1;j>=high+1;--j)
18         A[j+1]=A[j];
19     A[high+1]=A[0];
20 }
21 }

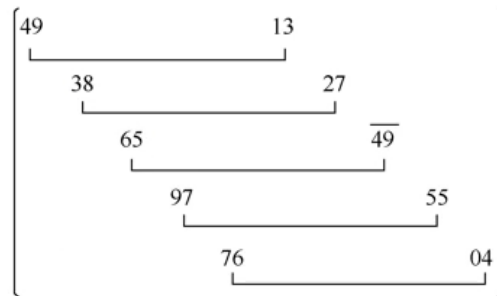
```

## 希尔排序

步长不同，通过多趟来实现，相当于减少了元素移动

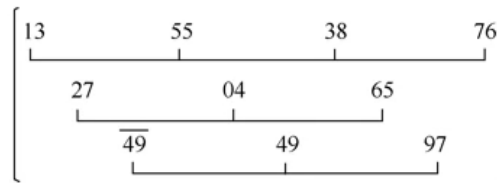
[初始关键字]:

49 38 65 97 76 13 27 49 55 04



一趟排序结果:

13 27 49 55 04 49 38 65 97 76



二趟排序结果:

13 04 49 38 27 49 55 65 97 76

三趟排序结果:

04 13 27 38 49 49 55 65 76 97

CSDN@QuantumYou

- 注意上述的第二个的步长为2，图的步长为3，绘画错误。

## 选择排序

### 简单选择排序

- 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，每一趟排序可以确定一个元素的最终位置，这样经过  $n-1$  趟排序就可使得整个排序表有序
- 选择排序每一次选择最小的值

```

1 void selectSort(ElemType A[],int n)
2 {
3     int i,j,min;//min记录最小的元素的下标
4     for(i=0;i<n-1;i++)//最多可以为8
5     {
6         min=i;
7         for(j=i+1;j<n;j++)//j最多可以为9
8         {
9             if(A[j]<A[min])
10                 min=j;
11         }
12         if(min!=i)

```

```
13     {
14         swap(A[i],A[min]);
15     }
16 }
17 }
```

堆排序（重点）

- 堆就是用数组实现的二叉树，所以它没有使用父指针或者子指针。堆根据“堆属性”来排序，“堆属性”决定了树中节点的位置。

1	数组元素情况	3 87 2 93 78 56 61 38 12 40
	数组元素对应的二叉树	
2	数组元素情况	3 87 2 93 78 56 61 38 12 40
	数组元素对应的二叉树，arr[4]与 arr[9]进行比较	
3	数组元素情况	3 87 2 93 78 56 61 38 12 40
	数组元素对应的二叉树，arr[3]与 arr[7]进行比较	
4	数组元素情况	3 87 61 93 78 56 2 38 12 40
	数组元素对应的二叉树，arr[2]与 arr[6]进行比较，发生交换	
5	数组元素情况	3 93 61 87 78 56 2 38 12 40
	数组元素对应的二叉树，arr[1]与 arr[3]进行比较，发生交换，发生交换后 arr[3]重新作为父结点，与孩子结点比较	
6	数组元素情况	93 87 61 38 78 56 2 3 12 40
	数组元素对应的二叉树，arr[0]与 arr[1]进行比较，发生交换，发生交换后 arr[1]重新作为父结点，与孩子结点比较	

```
1 //调整某个父亲节点
2 void AdjustDown(ElementType A[],int k,int len)
3 {
4     int i;
```

```

5     A[0]=A[k];
6     for(i=2*k;i<=len;i*=2)
7     {
8         if(i<len&&A[i]<A[i+1])//左子节点与右子节点比较大小
9             i++;
10        if(A[0]>=A[i])
11            break;
12        else{
13            A[k]=A[i];
14            k=i;
15        }
16    }
17    A[k]=A[0];
18 }
19 //用数组去表示树    层次建树
20 void BuildMaxHeap(ElemType A[],int len)
21 {
22     for(int i=len/2;i>0;i--)
23     {
24         AdjustDown(A,i,len);
25     }
26 }
27 void HeapSort(ElemType A[],int len)
28 {
29     int i;
30     BuildMaxHeap(A,len);//建立大顶堆
31     for(i=len;i>1;i--)
32     {
33         swap(A[i],A[1]);
34         AdjustDown(A,1,i-1);
35     }
36 }
37 //调整子树
38 void AdjustDown1(ElemType A[], int k, int len)
39 {
40     int dad = k;
41     int son = 2 * dad + 1; //左孩子下标
42     while (son<=len)
43     {
44         if (son + 1 <= len && A[son] < A[son + 1])//看下有没有右孩子，比较左右孩子选大的
45         {
46             son++;
47         }
48         if (A[son] > A[dad])//比较孩子和父亲
49         {
50             swap(A[son], A[dad]);
51             dad = son;
52             son = 2 * dad + 1;
53         }
54         else {
55             break;
56         }
57     }
58 }
59 void HeapSort1(ElemType A[], int len)
60 {
61     int i;

```

```

62 //建立大顶堆
63 for (i = len / 2; i >= 0; i--)
64 {
65     AdjustDown1(A, i, len);
66 }
67 swap(A[0], A[len]); //交换顶部和数组最后一个元素
68 for (i = len - 1; i > 0; i--)
69 {
70     AdjustDown1(A, 0, i); //剩下元素调整为大根堆
71     swap(A[0], A[i]);
72 }
73 }

```

## 各大排序时间复杂度

排序方式↵	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↵	↵	↵	↵
插入排序↵	$O(n^2)$ ↵	$O(n^2)$ ↵	$O(n)$ ↵	$O(1)$ ↵	稳定↵	简单↵
希尔排序↵	$O(n^{1.3})$ ↵	↵	↵	$O(1)$ ↵	不稳定↵	较复杂↵
冒泡排序↵	$O(n^2)$ ↵	$O(n^2)$ ↵	$O(n)$ ↵	$O(1)$ ↵	稳定↵	简单↵
快速排序↵	$O(n\log_2 n)$ ↵	$O(n^2)$ ↵	$O(n\log_2 n)$ ↵	$O(\log_2 n)$ ↵	不稳定↵	较复杂↵
选择排序↵	$O(n^2)$ ↵	$O(n^2)$ ↵	$O(n^2)$ ↵	$O(1)$ ↵	不稳定↵	简单↵
堆排序↵	 $O(n\log_2 n)$ ↵	$O(n\log_2 n)$ ↵	$O(n\log_2 n)$ ↵	$O(1)$ ↵	不稳定↵	较复杂↵
归并排序↵	$O(n\log_2 n)$ ↵	$O(n\log_2 n)$ ↵	$O(n\log_2 n)$ ↵	$O(n)$ ↵	稳定↵	较复杂↵
基数排序↵	$O(d(n+r))$ ↵	$O(d(n+r))$ ↵	$O(d(n+r))$ ↵	$O(r)$ ↵	稳定↵	较复杂↵

CSDN @QuantumYou