第九节：排序（上）

9.1 概述

对于之后应用到的一些说明：

（1）void X_Sort(ElementType A[], int N) X 为排序名称。

①大多数情况下，为了简单起见，讨论从小到大的整数排序。

②默认 N 为正整数。

③只讨论基于比较的排序（>=<都是有定义的）。

④只讨论内部排序（一次性可以写入内存，然后只在内存里面的数据排序）。

⑤稳定性：任意两个相等的数据，排序前后的相对位置不发生改变。

没有一种排序是任何情况下都表现最好的!!!

9.2 简单排序算法

9.2.1 冒泡排序

在一次排序完成后，最后面的一定是最大的，第二次排序的时候只需要对前 N-1 个排序即可，然后 N-2……一直到最后完成。

冒泡排序的伪码描述如下：

```
void Bubble_Sort(ElementType A[], int N)
{
    for(P=N-1;P>=0;P--)
    {
        flag = 0;
        for(i=0;i<P;i++)
        {/*一趟冒泡*/
            if(A[i]>A[i+1])
            {
                Swap(A[i],A[i+1]);]
                flag = 1;/*标志发生了交换*/
            }
        }
        if(flag==0)
            break;/*全程无交换，已经排好*/
    }
}
```

最好情况：顺序 $T=O(N)$；最差情况：逆序 $T=O(N^2)$

这个算法具有**稳定性**!

9.2.2 插入排序

类似于打扑克的时候拿到牌之后进行排序 。

源代码：

```
void InsertionSort( ElementType A[], int N )
{ /* 插入排序 */
     int P, i;
     ElementType Tmp;

     for ( P=1; P<N; P++ ) {
         Tmp = A[P]; /* 取出未排序序列中的第一个元素*/
         for ( i=P; i>0 && A[i-1]>Tmp; i-- )
             A[i] = A[i-1]; /*依次与已排序序列中元素比较并右移*/
         A[i] = Tmp; /* 放进合适的位置 */
     }
}
```

最好情况：顺序 T=O(N)

最坏情况：逆序 T=O (N^2)

此方法也具有**稳定性**。

举例，问序列{34,8,64,51,32,21}中用插入排序和冒泡排序分别需要交换多少次？

答：冒泡法：9 次；插入法，9 次。

它们的次数相等，是巧合还是必然？见下一节。

9.2.3 时间复杂度下界

1.逆序对

对于下标 i<j，如果 A[i]>A[j]，则称(i,j)是一对**逆序对**（inversion）。

我们来看上一节中最后的序列{34,8,64,51,32,21}，它里面有多少逆序对呢？

(34,8)(34,42)(34,21)(64,51)(64,21)(64,21)(51,32)(51,21)(32,21)

一共 9 个逆序对。这说明，前面两种算法，**交换 2 个相邻元素正好消去 1 个逆序对**！

插入排序时间复杂度：**T(N,I)=O(N+I)**。这里面 N 是元素个数，I 是逆序对个数。这个可以这样理解，时间复杂度最低为元素个数 N，即逆序对为 0，也就是顺序排列情况下；然后随着多一个逆序对，复杂度加 1。

如果序列**基本有序**，那么**插入排序**简单且高效。

2.定理 1：任意 N 个不同元素组成的序列平均具有 N(N-1)/4 个逆序对。

3.定理 2：任何仅以交换相邻两元素来排序的算法，其平均时间复杂度为Ω (N^2)。（*注意：Ω是下界，O 是上界*）

这意味着，要提高算法效率，我们必须：每次消去不止 1 个逆序对！每次交换相隔较远的 2 个元素！

9.3 希尔排序

1.简单举例

它利用了插入排序的简单，同时克服了交换每次只交换相邻两个元素的缺点。

用下面这个序列来进行希尔排序的举例：

81	94	11	96	12	35	17	95	28	58	41	75	15

（1）**首先以每 5 个元素取一个的规律进行排序**：这里取了 81、35、41。用插入排序对其进行排序：

35	94	11	96	12	**41**	17	95	28	58	**81**	75	15

然后取 94、17、75 进行插入排序，得到下面的结果：

35	17	11	96	12	**41**	75	95	28	58	**81**	94	15

然后考虑 11、95、15，得到下面结果：

35	17	11	96	12	**41**	75	15	28	58	**81**	94	95

然后考虑 96、28，得到下面结果：

35	17	11	28	12	**41**	75	15	96	58	**81**	94	95

然后考虑 12 和 58：

35	17	11	28	12	**41**	75	15	96	58	**81**	94	95

（2）**再用每 3 个元素取一个的规律**进行排序：这里取 35、28、75、58、95，结果如下：

28	**12**	**11**	**35**	**15**	**41**	**58**	**17**	**94**	**75**	**81**	**96**	**95**

（3）再做 1 间隔排序（即普通插入排序）。我们可以发现，此时整个序列已经基本有序，所以此时插入排序简单高效。

11	**12**	**15**	**17**	**28**	**35**	**41**	**58**	**75**	**81**	**94**	**95**	**96**

该算法的步骤：

①.定义增量序列 $D_M > D_{M-1} > \cdots > D_1 = 1$

②对每个 D_k 进行“D_k-间隔”排序。

注意：“D_k-间隔”有序的序列，在执行“D_{k-1}-间隔”排序后，仍然是“D_k-间隔”有序的。

2.希尔增量序列

（1）原始希尔排序：$D_M=[N/2]$，$D_k=[D_{k+1}/2]$。其中，[]代表取整。

伪代码如下：

```
void Shell_Sort(ElementType A[], int N)
{
    for(D=N/2;D>0;D/=2)
    {
        for(P=D;P<N;P++)
        {
            Tmp= A[P];
            for(i=P;i>=D&&A[i-D]>Tmp;i-=D)
                A[i]=A[i-D];
            A[i]=Tmp;
        }
    }
}
```

最坏情况：T=**$\theta(N^2)$**（**θ代表即使上界也是下界，即可以达到的值**）

这种情况即，增量元素不互质，则小增量可能根本不起作用。

3.其它增量序列

（1）Hibbard 增量序列：$D_k=2^k-1$（相邻元素互质）。最坏情况：$\mathbf{T=\theta(N^{3/2})}$，猜想 $\mathbf{T_{avg}=O(N^{5/4})}$。

（2）Sedgewick 增量序列：{1,5,19,41,109}（$9\times4^i-9\times2^i+1$ 或 $4^i-3\times2^i+1$），猜想 $\mathbf{T_{avg}=O(N^{7/6})}$，$\mathbf{T_{worst}=O(N^{4/3})}$。

4.稳定性：不稳定。

9.4 堆排序

9.4.1 选择排序

```
void Selection_Sort(ElementType A[], int N)
{
	for(i=0;i<N;i++)
	{
		MinPosition = ScanForMin(A, i N-1);
		/*从 A[i]到 A[N-1]中找最小元，并将其位置赋给 MinPosition*/
		Swap(A[i], A[MinPosition]);
		/*将未排序部分的最小元换到有序部份的最后位置*/
	}
}
```

无论如何，$\mathbf{T=\theta(N^2)}$。

如何快速寻找到最小元？见下一节。

9.4.2 堆排序（选择排序的改进）

1.算法 1：

```
void Heap_Sort ( ElementType A[], int N )
{
	BuildHeap(A); /*调整为最小堆，复杂度为 O(N) */
	for ( i=0; i<N; i++ )
		TmpA[i] = DeleteMin(A); /*以此把最小元弹出来放到这个临时数组中*/
/*复杂度为 O(logN) */
	for ( i=0; i<N; i++ )/* O(N) */
		A[i] = TmpA[i];/*把临时数组的数组返回原来的数组*/
}
```

时间复杂度：**T(N)=O(NlogN)**

缺点：需要额外 O(N)空间，并且复制元素需要时间。

2.算法 2：

注意，在堆排序里面的这个堆，元素是从 0 的位置开始的；原来学堆的时候，0 是用来放哨兵的。

因此，在堆排序中，元素下标从 0 开始。则对于下标为 i 的元素，其左、右孩子的下标

分别为：**2i+1**, **2i+2**。

伪码描述如下：

```
void Heap_Sort ( ElementType A[], int N )
{
    for ( i=N/2-1; i>=0; i-- )/* BuildHeap */
        PercDown( A, i, N );
    for ( i=N-1; i>0; i-- ) {
        Swap( &A[0], &A[i] ); /* DeleteMax */
        PercDown( A, 0, i );
    }
}
```

定理：堆排序处理 N 个不同元素的随机排列的平均比较次数是 2NlogN-O(Nlog logN)。

注意：虽然堆排序给出最佳平均时间复杂度，但实际效果不如用 Sedgewick 增量序列的希尔排序。

整体 C 语言源代码：

```
void Swap( ElementType *a, ElementType *b )
{
     ElementType t = *a; *a = *b; *b = t;
}

void PercDown( ElementType A[], int p, int N )
{ /* 改编代码 4.24 的 PercDown( MaxHeap H, int p )    */
  /* 将 N 个元素的数组中以 A[p]为根的子堆调整为最大堆 */
    int Parent, Child;
    ElementType X;

    X = A[p]; /* 取出根结点存放的值 */
    for( Parent=p; (Parent*2+1)<N; Parent=Child ) {
        Child = Parent * 2 + 1;
        if( (Child!=N-1) && (A[Child]<A[Child+1]) )
            Child++;  /* Child 指向左右子结点的较大者 */
        if( X >= A[Child] ) break; /* 找到了合适位置 */
        else  /* 下滤 X */
            A[Parent] = A[Child];
    }
    A[Parent] = X;
}

void HeapSort( ElementType A[], int N )
{ /* 堆排序 */
     int i;

     for ( i=N/2-1; i>=0; i-- )/* 建立最大堆 */
         PercDown( A, i, N );
```

```
    for ( i=N-1; i>0; i-- ) {
        /* 删除最大堆顶 */
        Swap( &A[0], &A[i] ); /* 见代码 7.1 */
        PercDown( A, 0, i );
    }
}
```

9.5 归并排序

9.5 1 有序子列的归并

1.举例：假设我们有两个有序序列如下：

1	13	24	26

2	15	27	38

我们要将它们合并成一个序列并按照顺序排序。我们需要设置三个指针，如图 1 所示。Aptr 指向 A 序列的第一个元素（1），Bptr 指向 B 序列的第一个元素（2），Cptr 指向合并后的第一个元素。这里的指针其实是三个整数，分别存储的三个元素的下标。首先比较 Aptr 和 Bptr 指向的元素那个比较小，选择比较小的放入 Cptr 所代表的下标的那个位置。然后将 Aptr 加 1，Cptr 加 1，用 Bptr 所代表的的下标的元素和 Aptr 所代表的的下标的元素比较，发现 Bptr 下标的元素（2）小，将 2 存入 Cptr 代表下标的元素，即 C[Cptr]。然后依次类推。

图 1

时间复杂度 T(N)=O(N)。

伪代码描述如下：

```
/* L = 左边起始位置, R = 右边起始位置, RightEnd = 右边终点位置 */
void Merge( ElementType A[], ElementType TmpA[],
            int L, int R, int RightEnd )
{
    LeftEnd = R - 1; /* 左边终点位置。假设左右两列挨着 */
    Tmp = L; /* 存放结果的数组的初始位置 */
    NumElements = RightEnd - L + 1;
    while( L <= LeftEnd && R <= RightEnd )
    {
        if( A[L] <= A[R] )
            TmpA[Tmp++] = A[L++];
        else TmpA[Tmp++] = A[R++];
    }
    while( L <= LeftEnd ) /* 直接复制左边剩下的 */
        TmpA[Tmp++] = A[L++];
    while( R <= RightEnd ) /*直接复制右边剩下的 */
        TmpA[Tmp++] = A[R++];
```

```
    for(i=0;i<NumElements;i++,RightEnd--)
        A[RightEnd] = TmpA[RightEnd];
}
```

9.5.2 递归算法

1.分而治之

如图 2 所示，该算法是**稳定的**。

图 2

伪码描述如下：

```
void MSort( ElementType A[], ElementType TmpA[], int L, int RightEnd )
{
    int Center;
    if ( L < RightEnd ) {
        Center = ( L + RightEnd ) / 2;
        MSort( A, TmpA, L, Center );
        MSort( A, TmpA, Center+1, RightEnd );
        Merge( A, TmpA, L, Center+1, RightEnd );
    }
}
```

时间复杂度为：**T(N)=T(N/2)+T(N/2)+O(N)，即 T(N)=O(NlogN)。**

2.统一函数接口

为了与前面的函数接口统一，因此我们需要再写一个函数来统一函数接口。其伪码描述如下：

```
void Merge_sort( ElementType A[], int N )
{
    ElementType *TmpA;
    TmpA = malloc( N * sizeof( ElementType ) );
    if ( TmpA != NULL )
    {
        MSort( A, TmpA, 0, N-1 );
        free( TmpA );
    }
    else Error( “空间不足" );
}
```

如果只在 Merge 中声明临时数组，那么两个子函数声明如下：

```
void Merge( ElementType A[], int L, int R, int RightEnd )
void MSort( ElementType A[], int L, int RightEnd )
```

这样也不是不行，但是这样的话，**每一次在子函数里面用一次释放一次，增加了时间复杂度，还不如直接在最外层的代码里面定义好。**

整体的源代码实现如下：

```
/* 归并排序 - 递归实现 */
```

```
/* L = 左边起始位置, R = 右边起始位置, RightEnd = 右边终点位置*/
void Merge( ElementType A[], ElementType TmpA[], int L, int R, int RightEnd )
{ /* 将有序的 A[L]~A[R-1]和 A[R]~A[RightEnd]归并成一个有序序列 */
     int LeftEnd, NumElements, Tmp;
     int i;

     LeftEnd = R - 1; /* 左边终点位置 */
     Tmp = L;             /* 有序序列的起始位置 */
     NumElements = RightEnd - L + 1;

     while( L <= LeftEnd && R <= RightEnd ) {
         if ( A[L] <= A[R] )
             TmpA[Tmp++] = A[L++]; /* 将左边元素复制到 TmpA */
         else
             TmpA[Tmp++] = A[R++]; /* 将右边元素复制到 TmpA */
     }

     while( L <= LeftEnd )
         TmpA[Tmp++] = A[L++]; /* 直接复制左边剩下的 */
     while( R <= RightEnd )
         TmpA[Tmp++] = A[R++]; /* 直接复制右边剩下的 */

     for( i = 0; i < NumElements; i++, RightEnd -- )
         A[RightEnd] = TmpA[RightEnd]; /* 将有序的 TmpA[]复制回 A[] */
}

void Msort( ElementType A[], ElementType TmpA[], int L, int RightEnd )
{ /* 核心递归排序函数 */
     int Center;

     if ( L < RightEnd ) {
          Center = (L+RightEnd) / 2;
          Msort( A, TmpA, L, Center );              /* 递归解决左边 */
          Msort( A, TmpA, Center+1, RightEnd );     /* 递归解决右边 */
          Merge( A, TmpA, L, Center+1, RightEnd );  /* 合并两段有序序列 */
     }
}

void MergeSort( ElementType A[], int N )
{ /* 归并排序 */
     ElementType *TmpA;
     TmpA = (ElementType *)malloc(N*sizeof(ElementType));

     if ( TmpA != NULL ) {
```

```
            Msort( A, TmpA, 0, N-1 );
            free( TmpA );
        }
        else printf( "空间不足" );
}
```

9.5.3 非递归算法

非递归算法的图示如图 3 所示。

图 3

额外空间复杂度是 **O(N)**。我们只需要开一个临时数组就可以了，没必要每次都开一个。

其伪码描述如下：

```
void Merge_pass( ElementType A[], ElementType TmpA[], int N,int length )
 /* length = 当前有序子列的长度*/
{
    for ( i=0; i <= N–2*length; i += 2*length )
        Merge1( A, TmpA, i, i+length, i+2*length–1 );
    if ( i+length < N ) /* 归并最后 2 个子列*/
        Merge1( A, TmpA, i, i+length, N–1);
    else /* 最后只剩 1 个子列*/
    for ( j = i; j < N; j++ ) TmpA[j] = A[j];
}
```

统一接口如下：

```
void Merge_sort( ElementType A[], int N )
{
    int length = 1; /* 初始化子序列长度*/
    ElementType *TmpA;
    TmpA = malloc( N * sizeof( ElementType ) );
    if ( TmpA != NULL ) {
        while( length < N ) {
            Merge_pass( A, TmpA, N, length );
            length *= 2;
            Merge_pass( TmpA, A, N, length );
            length *= 2;
        }
        free( TmpA );
    }
    else Error( “空间不足" );
}
```

具有稳定性！

缺点是需要额外空间之类的~此方法主要用在外排序。

C 语言实现源代码：

```
/* 归并排序 - 循环实现 */
```

```
/* 这里 Merge 函数在递归版本中给出 */

/* length = 当前有序子列的长度*/
void Merge_pass( ElementType A[], ElementType TmpA[], int N, int length )
{ /* 两两归并相邻有序子列 */
    int i, j;

    for ( i=0; i <= N-2*length; i += 2*length )
        Merge( A, TmpA, i, i+length, i+2*length-1 );
    if ( i+length < N ) /* 归并最后 2 个子列*/
        Merge( A, TmpA, i, i+length, N-1);
    else /* 最后只剩 1 个子列*/
        for ( j = i; j < N; j++ ) TmpA[j] = A[j];
}

void Merge_Sort( ElementType A[], int N )
{
    int length;
    ElementType *TmpA;

    length = 1; /* 初始化子序列长度*/
    TmpA = malloc( N * sizeof( ElementType ) );
    if ( TmpA != NULL ) {
        while( length < N ) {
            Merge_pass( A, TmpA, N, length );
            length *= 2;
            Merge_pass( TmpA, A, N, length );
            length *= 2;
        }
        free( TmpA );
    }
    else printf( "空间不足" );
}
```