数据结构第九周笔记——排序(上)(慕课浙大版本--XiaoYu)

9.1 简单排序(冒泡、插入)

9.1.1 概述

```
void X_Sort(ElementType A[],int N)//sort就是排序的意思，X是排序算法的名称
    //统一默认输入的参数有两个(一个是待排的元素放在一个数组里，数据类型为ElementType任意类型。另外一个是正整数N，表示的是我们要排的元素到底有多少个，默认讨论整数(从小到达)的排序)
    1.N是正整数
    2.只讨论基于比较的排序(>=< 有定义)
    3.只讨论内部排序(假设我们内存空间足够大，所有数据可以一次性导入内存空间里，然后所有的排序是在内存里面一次性完成的)
    //外部排序(假设我有的内存空间有2GB，但是要求我们对10TB的数据进行排序，这个时候内部排序就不行了)
    4.稳定性：任意两个相等的数据，排序先后的相对位置不发生改变
    5.没有一种排序是任何情况下都表现最好
```

9.1.2 冒泡排序

```
void Bubble_Sort(ElementType A[],int N)
{
    for(i = 0; i < P;P--){
        flag = 0;//表示还没有执行果任何一次交换
        for(i = 0;i < P;i++ ){//一趟冒泡
            if(A[i] > A[i+1] ){
                Swap(A[i],A[i+1]);
                flag = 1;//要交换的时候标识变为1
            }
    }
        if( flag == 0 ) break;//全程无交换
    }
}

最好情况：顺序T = O(N) 全程无交换
最坏情况：逆序T = O(N²)
```

冒泡排序优点：

1. 所有的待排元素是放在一个单向链表里的(冒泡排序可以对数组，对单项链表都可以实现，其他排序不好实现)
2. 算法稳定

9.1.3 插入排序

```
void Insertion_Sort( ElementType A[],int N )
{
    for( P = 1;P < N; P++ ){
        Tmp = A[P];//摸下一张牌，Tmp为临时存放的位置
        for( i = P; i > 0 && A[i - 1] > Tmp;i--)//旧牌大
            A[i] = A[i - 1];//移除空位
        A[i] = Tmp;//新牌落位
    }
}
//最好情况：顺序T = O(N)
//最坏情况：逆序T = O(N²)
```

插入排序好处：

1. 程序短，简单
2. 比冒泡排序好在：冒泡排序是两两交换，两两元素互换的时候他要涉及到第三步。而插入排序则是每个元素向后错，最后他一次性放到他那个空位里面去(不是插入排序最主要的)
3. 稳定
4.

给定初始序列{34, 8, 64, 51, 32, 21}，冒泡排序和插入排序分别需要多少次元素交换才能完成?

冒泡9次，插入9次

对一组包含10个元素的非递减有序序列，采用插入排序排成非递增序列，其可能的比较次数和移动次数分别是?

45, 44

9.1.4 时间复杂度下界

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

问题：序列{34，8，64，51，32，21}中有多少逆序对？9对

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

逆序对的数量跟交换元素次数是一样的，也就说明了交换两个相邻元素正好消去一个逆序对！

插入排序：T(N,I) = O(N+I)

1. 如果序列**基本有序**，则插入排序简单且非常高效

- 定理：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。

逆序对平均个数：大O(N2)数量级的，不管是冒泡排序还是插入排序，他们的平均时间复杂度是跟逆序对的个数有关系的

- 定理：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 $\Omega(N^2)$。

Ω：指的是下界

这意味着：要提高算法效率，我们必须

1. 每次消去不止一个逆序对
2. 每次交换相隔较远的2个元素，这样一次性就消掉了不止一个逆序对

9.2 希尔排序(by Donald Shell)

基本思路：利用了插入排序的简单，同时克服插入排序每次只交换相邻两个元素的缺点

到最后使用1-间隔的排序来保证序列有序(彻底的插入排序)。但此时这个序列已经基本有序了，大多数的逆序对已经在前面两趟5-间隔和3-间隔里面被消除掉了

- 定义增量序列 $D_M > D_{M-1} > \dots > D_1 = 1$
- 对每个 D_k 进行“D_k-间隔”排序$(k = M, M-1, \dots 1)$

重要性质：3-间隔有序的序列还保持了前面5-间隔有序的这个性质(没有把上一步的结果变坏)

- 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

希尔增量序列

1. 原始希尔排序 $D_M = \lfloor N/2 \rfloor$，$D_k = \lfloor D_{k+1}/2 \rfloor$

2.
```
void Shell_Sort( ElementType A[],int N )
{
    for(D = M/2; D > 0; D /= 2 ){//希尔增量序列
        for( P = D; P < N; P++ ){//插入排序，D是距离(第0张牌在我手里，下一张牌从第D张牌开始摸)
            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: 最坏情况: $T = \Theta(N^2)$

O是一个上界(可能达不到)

而Θ既是上界又是下界

增长速度跟N^2一样快

坏例子:

举个坏例子

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

增量元素不互质，则小增量可能根本不起作用

更多的增量序列

- **Hibbard 增量序列**
  - $D_k = 2^k - 1$ —— 相邻元素互质
  - 最坏情况： $T = \Theta(N^{3/2})$
  - 猜想： $T_{avg} = O(N^{5/4})$
- **Sedgewick增量序列**
  - $\{1, 5, 19, 41, 109, \ldots\}$
    —— $9\times4^i - 9\times2^i + 1$ 或 $4^i - 3\times2^i + 1$
  - 猜想： $T_{avg} = O(N^{7/6})$， $T_{worst} = O(N^{4/3})$

```
用Sedgewick增量序列
void ShellSort( ElementType A[], int N )
{ /* 希尔排序 - 用Sedgewick增量序列 */
     int Si, D, P, i;
     ElementType Tmp;
     /* 这里只列出一小部分增量 */
     int Sedgewick[] = {929, 505, 209, 109, 41, 19, 5, 1, 0};

     for ( Si=0; Sedgewick[Si]>=N; Si++ )
         ; /* 初始的增量Sedgewick[Si]不能超过待排序列长度 */

     for ( D=Sedgewick[Si]; D>0; D=Sedgewick[++Si] )
         for ( P=D; P<N; P++ ) { /* 插入排序*/
             Tmp = A[P];
             for ( i=P; i>=D && A[i-D]>Tmp; i-=D )
                 A[i] = A[i-D];
             A[i] = Tmp;
         }
}
```

9.3 堆排序

概念

```
大顶堆：每个节点的值都大于或者等于它的左右子节点的值。

堆排序的基本思想是：
1、将带排序的序列构造成一个大顶堆，根据大顶堆的性质，当前堆的根节点（堆顶）就是序列中最大的元素；
2、将堆顶元素和最后一个元素交换，然后将剩下的节点重新构造成一个大顶堆；
3、重复步骤2，如此反复，从第一次构建大顶堆开始，每一次构建，我们都能获得一个序列的最大值，然后把
它放到大顶堆的尾部。最后，就得到一个有序的序列了。
```

9.3.1 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]);//这两个元素通常情况下不是挨着的，可能跳了很远的距离做一个交换，一下子就消除掉很多逆序对
        //将未排序部分的最小元换到有序部分的最后位置
        //最坏情况就是每次都必须换一下，最多需要换N-1次
    }
}
//想要得到更快的算法取决于这个ScanForMin( A,i,N-1)，也就是如何快速找到最小元
```

$$\text{无论如何：}\quad T = \Theta(N^2)$$

最小堆的特点就是他的根结点一定存的是最小元

9.3.2 堆排序

算法1：

```
void Heap_Sort(ElementType A[],int N)
{
    BuildHeap(A);//O(N)
    for( i = 0;i < N;i++ )
        TmpA[i] = DeleteMin(A);//把根结点弹出来，依次存到这个临时数组里面。O(logN)
    for( i = 0;i < N;i++ )//O(N)
        A[i] = TmpA[i];//将TmpA里面所有的元素导回A里面
}
//缺点：需要额外O(N)空间，并且复制元素需要时间
```

算法2：

```
void Heap_Sort(ElementType A[],int N )
{
    for(i = N/2;i >= 0;i-- ){//BuildHeap，i对应的是根节点所在的位置，N对应的是当前这个堆里一共有多少个元素
        PercDown(A,i,N);
    for( i = N-1;i > 0;i--){//堆循环
        Swap(&A[0],&A[i]);//DeleteMax，A[0]根节点里面存的是最大的元素，i是当前最后一个元素的下标，把根节点换到当前这个堆的最后一个元素的位置上去
        PercDown(A,0,i);//调整的时候是以0为根节点，i是当前这个最大堆的元素个数
    }
    }
}
```

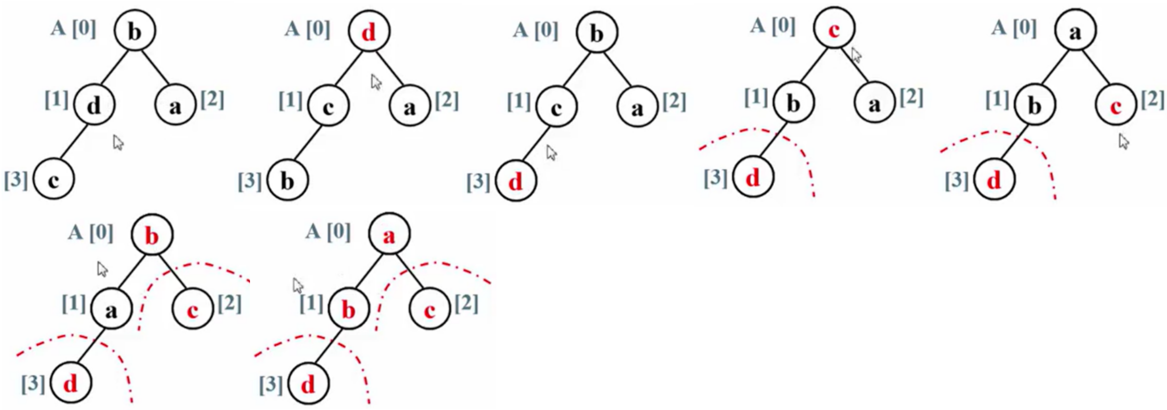
在堆排序中，元素下标从0开始。则对于下标为i的元素，其左、右孩子的下标分别为：2i+1, 2i+2

- 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

$$2N\log N - O(N\log\log N)$$ 。

- 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。

算法2的动态变化：




```
堆排序
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 );
```

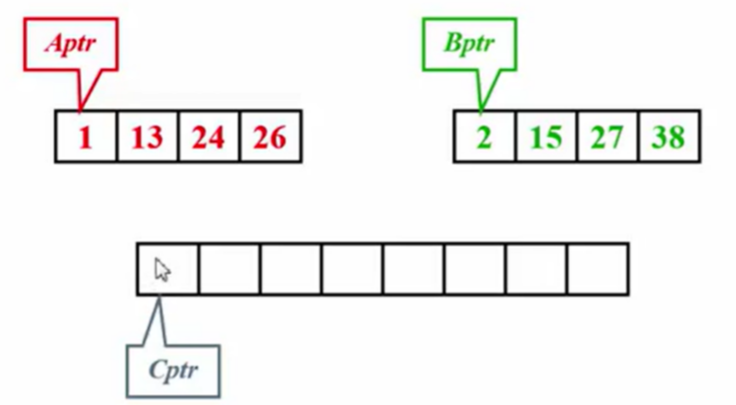
```
    for ( i=N-1; i>0; i-- ) {
        /* 删除最大堆顶 */
        Swap( &A[0], &A[i] ); /* 见代码7.1 */
        PercDown( A, 0, i );
    }
```

9.4 归并排序

9.4.1 有序子列的归并

需要3个指针(这个指针不一定是C语言里面说的那个语法上的指针)

指针：本质上他存的是位置



假设我们讨论的是数组(那位置由下标决定，那图中的指针就可以是整数，整数存的是这个元素的下标)

```
上方图中红色跟绿色的指针指向的位置进行比大小，小的填入下方的空位置中，红绿色其他一方填入数值后指针就往后挪一位，然后继续红绿色指针所指位置对比大小，直到下方空位置填满
```

如果两个子列一共有N个元素，则归并的时间复杂度是？**T(N) = O(N)**

有序子列归并的伪代码

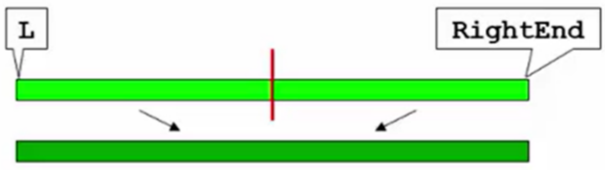
```
//L = 左边起始位置,R = 右边起始位置,RightEnd = 右边终点位置
void Merge(ElementType A[],ElementType TmpA[],int L,int R,int RightEnd)//Merge就是归并的意思
{//参数意思从左到右分别是：原始的待排的序列，临时存放的数组，归并左边的起始位置(也就是上图的Aptr)，归并右边的起始位置(也就是上图的Bptr)，右边终点的位置

    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，相当于上图的Cptr
    NumElements = RightEnd - L + 1;//元素的总个数
    //上方是准备工作，下方开始归并
    while( L <= LeftEnd && R <= RightEnd ){//一直走到左右两边其中一方不满足之后跳出(意味着其中一个子序列已经空了，没有元素了，另一方剩下的元素直接全部导入后面就可以了)
        if(A[L] <= A[R] ) TmpA[Tmp++] = A[L++];//左边小，将Aptr放入
        else              TmpA[Tmp++] = A[R++];//右边小，将Bptr放入
    }
    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++];//TmpA只是临时存放的地方，还需要导回去
```

```
    for( i = 0;i < NumElements;i++,RightEnd-- )//从后面开始才能知道终点的位置具体是哪个，因为RightEnd具体多少是不固定的
        A[RightEnd] = TmpA[RightEnd];
}
```

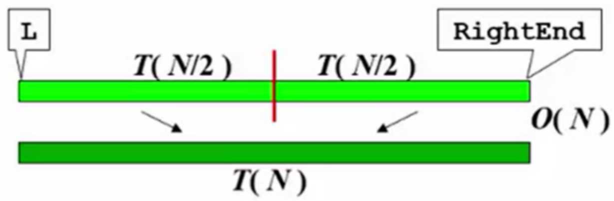
9.4.2递归算法

1. **分而治之**

   1. 先把整个一分为二，然后递归的去考虑问题，递归的去把左边排好序，再递归的把右边排好序。这样得到两个有序的子序列，而且肩并肩的放在一起，最后调用我们归并的算法，把他们归并到一个完整的数组里

   2. 算法的伪代码实现 


```
void MSort(ElementType A[],ElementType TmpA[],int L,int RightEnd )
{//上述参数：原始待排的数组，临时的数组，L指待排序列开头的位置，RightEnd则是待排序列结尾的位置
    int Center;//中间的位置
    if( L < RightEnd ){
        Center = (L + RightEnd ) / 2;
        MSort( A,TmpA,L,Center );//左边的递归排序
        MSort( A,TmpA,Center+1, RightEnd );//右边的递归
        Merge( A,TmpA,L,Center+1,RightEnd );//归并，传入的参数分别是原始数组A,临时数组TmpA，左边的起始点，右边的起始点吗，右边的终点。结果存在原来这个数组A里面
    }
}
//T(N) = T(N/2)+T(N/2)+O(N) => T(N) = O(NlogN)
NlogN：没有最坏时间复杂度也没有最好时间复杂度，更没有平均时间复杂度，任何情况下都是NlogN，非常稳定
```



统一函数接口

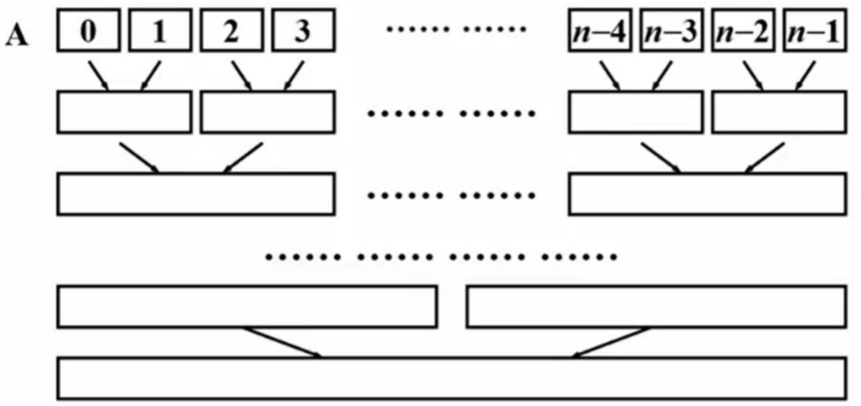
```
void Merge_sort( ElementType A[],int N )//参数：原始的数组A,元素的个数N
{
    ElementType *TmpA;
    TmpA = malloc(N * sizeof( ElementType ));//TmpA空间在这里临时申请
    if( TmpA != NULL ){//检查申请的空间是否还有位置
        MSort(A,TmpA,0,N-1);//TmpA在这里只是一个递归的调用，真正用到TmpA的地方是在Merge(核心的那个归并函数里)
        free( TmpA );//把临时空间给释放掉
    }
    else Error("空间不足")
}
```

```
如果只在Merge中声明临时数组TmpA
1.void Merge( ElementType A[],int L,int R,int RightEnd )
2.void MSort( ElementType A[],int L,int RightEnd)
```

白色砖块一样的东西是申请的空间，要不停的申请空间再释放掉，这样做实际上是不合算的(太麻烦了，申请一个释放掉在申请下一个不停循环)

最合算的做法：一开始就声明一个数组，每次只把数组的指针传进去，只在这个数组的某一段上面做操作，就不需要重复的malloc跟free

9.4.3非递归算法(归并排序)



上图的深度为logN

非递归算法的额外空间复杂度是？O(N)

```
只需要开一个临时数组就够了，没有必要每次合并都开一个
第一次我们把A给归并到临时数组里面
第二次把临时数组里面的东西归并回A里面去，然后再把A导到临时数组里，再把临时数组导回到A
最后一步运气好的话就是A，运气不好的话这最后一步可能是那个临时数组他不是A(需要再加一步导回到A里面去)
```

一趟归并伪代码

```
void Merge_pass( ElementType A[],ElementType TmpA[],int N,int length)//length = 当前有序子列长度(一开始为1，之后每次加倍)
{//参数：原始数组，临时数组，N为待排序列长度
    for(i = 0; i < N-2*length;i += 2*length )//i += 2*length就是跳过两段然后去找下一对。最后尾巴可能是单个的所以先把前面成对的那一部分处理完，终止条件就是处理到倒数第二对(这个处理完了再看尾巴)
        Merge1( A, TempA, i, i+length, i+2*length-1 );//不做Merge最后一步导入A中，在这里意味着把A中的元素归并到TmpA里面去，最好有序的内容是放在TmpA里面
    if( i+length < N )//归并最后两个子列，最后如果加上一段以后还是小于N的，那就说明我最后是不止一个子列，是有两个子列的
        //如果这个if条件不成立意味着当前i这个位置加上一个length之后他就跳到N外面去了，也就意味着我最后只剩下一个子列
        Merge1(A,TmpA,i,i+length,N-1);
    else//最后剩下一个子列
        for(j = i;i < N;j++ ) TmpA[j] = A[j];
}
```

原始统一接口

```
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);//传进来的length长度是2。前面这个TmpA是初始状态，后面A是归并以后的状态
            length *= 2;//这里length再次double(翻倍)变成了4
            //最后跳出while循环，结果都是存在A里面的，哪怕最后一步执行到Merge_pass(A,TmpA,N,length);就已经有序了，也会多执行一步Merge_pass，将TmpA原封不动的导到A里面然后自然跳出
        }
        free(TmpA);
    }
    else Error("空间不足");
}

//优点：稳定
//缺点：需要一个额外的空间，并且需要在数组跟数组之间来回来去的复制 导这个元素。所以实际运用中基本上不做内排序(在外排序的时候是非常有用的)
```