2015 级算法第一次上机解题报告（助教版）

目 录

2015 级算法第一次上机解题报告

（马国瑞）

一、引言

本次上机考察分治、递推等算法知识点，对《算法导论》课本前四章的内容进行了考察。本次上机有相当一部分题目的思想来源于课本，其中有一道题直接引用了课本介绍最大子数组的例子。

关于题目难度，本次上机前五道题的整体难度较后续上机题目难度而言偏低，但比去年 2014 级算法第一次上机难度稍大，C、D、E 三题通过细节等方面还是能体现出其区分度；最后一题需要在普通汉诺塔知识深入理解的基础之上进行进一步的分析，需要对递归的知识有深入的理解，难度较前五题而言大了许多。

二、解题报告

A 多项式计算器

思路分析

本题主要是希望大家解决这样一个问题：如何能使用尽量少的乘法次数来解决一元 n 次多项式结果的计算问题？

计算机当中乘法的计算机计算时间远大于加法，这在《深入理解计算机系统》中会讲到。我们能做的就是优化代码降低乘法次数。

对于计算一元 n 次多项式的结果，我们有下述几种解法：

方法一 直接计算法

此方法的思路是这样的：存储 x 值和一元 n 次多项式的系数数组，计算的时候针对每一项分别计算，相加，最后得到答案。实现代码如下：

```
long long solve(long long a[], long long x, int n){
    /*此处 a 数组为一元 n 次多项式系数，
    n 为一元 n 次多项式的次数*/
    long long sum = 0;
    long long buf = 1;
    for(int i=0;i<=n;i++){
        buf=1;
        for(int j=0;j<i;j++)
            buf = (buf * x) % 1000007; //此处计算各项 x 的 i 次方
```

```
        sum+=(buf*a[i])%1000007;
        sum%=1000007;
    }
    return sum;
}
```

不难发现，所需的乘法次数为 T(n)=n*(n-1)/2=O(n^2)，大大浪费了时间，也是本题卡时间主要卡的代码。

方法二

此方法是对方法一的一种优化，即计算 x 的 k 次项的之前，通过一个变量存储前一项时候计算得到的 x 的 k-1 次幂。实现代码如下：

```
long long solve(long long a[], long long x, int n){
    /*此处 a 数组为一元 n 次多项式系数，
    n 为一元 n 次多项式的次数*/
    long long sum = 0;
    long long buf = 1;      //通过 buf 的值存储 x 的幂
    for(int i=0;i<=n;i++){
        if(i>0)
            buf = (buf * x) % 1000007; //计算 x 的 i 次幂
        sum+=(buf*a[i])%1000007;
        sum%=1000007;
    }
    return sum;
}
```

和方法一相比，方法二的乘法次数大大降低。针对每次查询，所需要的乘法次数为 2n 次；而针对每组数据的 k 次查询，所需要的乘法次数为 2kn 次。

方法三

方法三则是对方法二的进一步优化。针对本题对一个 x 值有多次查询，我们可以对所有 x 的 n 次幂存储成一个数组，针对多组系数使用同样的 n 个 x 的 n 次幂值。针对每组数据的 k 次查询，所需要的乘法次数为(k+1)n 次，平均每次查询所需乘法次数为$\frac{k+1}{k}n$次。

具体代码操作详见参考代码一。

方法四 秦九韶算法（霍纳法则）

这是大家的解题报告中提到最多的算法，《算法导论》中文第三版第 23 页也有提到。伪代码片段如下：

```
y = 0;
for i = n downto 0
```

$y = a_i + x * y$

思路是：把一个 n 次多项式

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

改写成如下形式：

$$f(x) = ((\ldots(a_n x + a_{n-1})x + a_{n-2})x + \cdots + a_1)x + a_0$$

求多项式的值时，首先计算最内层括号内一次多项式的值，然后由内向外逐层计算一次多项式的值。这样，求 n 次多项式 f(x)的值就转化为求 n 个一次多项式的值。

针对每次查询，所需的乘法次数为 n；针对每组数据的 k 次查询，所需的乘法次数为 kn。

具体代码操作详见参考代码二。

参考代码一

```
#include<iostream>
#include<cstdio>
using namespace std;
#define MO 1000007
long long bufx[10010];
int main()
{
    int n,x,t;
    int ans;
    int cnt = 1;
    bufx[0]=1;
    while(cin>>n>>x>>t){
        cout<<"Case #"<<cnt<<":\n";
        cnt++;
        //下述代码对 x 的 n 次幂进行存储操作，即数据预处理
        for(int i=1;i<=n;i++)
            bufx[i]=((bufx[i-1]%MO)*(x%MO)+MO)%MO;
        while(t--){
            ans=0;     //注意各次查询前对答案初始化
            for(int i=0;i<=n;i++){
                int buf;
                cin>>buf;
                //下述代码计算各项的值并进行加法操作
                ans+=((buf%MO)*(bufx[i]%MO)+MO)%MO;
                ans%=MO;
```

```
            }
            cout<<ans<<endl;
        }
    }
}
```

参考代码二

```
#include<iostream>
#include<cstdio>
using namespace std;
#define MO 1000007
long long buf[10010];   //buf 数组用于存储系数
long long gans[10010];  //gans 数组用于存储各次查询的计算结果
int main()
{
    int n,t;
    long long x;
    long long ans;
    int cnt = 1;
    while(cin>>n>>x>>t){
        for(int j=0;j<t;j++){
            ans=0;
            for(int i=0;i<=n;i++)
                cin>>buf[i];
            //下述代码运用秦九韶算法计算一元 n 次多项式的值
            ans+=buf[n];
            for(int i=n-1;i>=0;i--){
                ans*=x;
                ans+=MO;
                ans%=MO;
                ans+=buf[i];
                ans%=MO;
            }
            gans[j]=ans;
        }
        printf("Case #%d:\n",cnt++);
        for(int i=0;i<t;i++)
            cout<<gans[i]<<endl;
    }
}
```

Hint

本题方法二、三、四均能 AC。

参考资料

1.《算法导论》中文第三版，机械工业出版社

2. http://baike.baidu.com/view/1431260.htm

B 怠惰的园丁王木木

思路分析

本题目可以认为是一道数学递推问题。

除题目中提到的样例解释外，再举一些例子：

n=4 时，草的高度分别为 1, 2, 3, 4. 先以长度 3 剪去后两棵草，剩余高度为 1, 2, 0, 1；再以长度 1 减去三棵草，剩余高度为 0, 1, 0, 0；最后以长度 1 剪去第二棵草，需要 3 步。

n=5 时，草的高度为 1, 2, 3, 4, 5，先以高度 3 减去后三棵草，剩余高度为 1, 2, 0, 1, 2，此后再两次以长度 1 剪去剩余的草，同样需要 3 步。

以此类推，发现每次除草最多能把不同高度数 n 降低至[n/2]。和上例一样，通过高度为[n/2]+1 的一次除草可将高度在[n/2]以上草，变成[n/2]及其以下的草，这样不同高度数 n 可降低到[n/2]。这样消耗的体力值最小，为[$\log_2 n$]。

（上述文段的中括号均为取整符号）

实际计算这个最小体力值的最佳方案是通过判断 n 最多能算术右移多少位使 n 大于 0（或者是最多能除 2 多少次使 n 大于 0）。

此处思路分析部分参考了 **优秀解题报告 15101061 林克廉** 同学的分析，在此致谢。同时，这个问题严格的数学证明欢迎大家思考。

具体操作详见参考代码。

参考代码

```
#include <cstdio>
int main() {
    int n;
    while(~scanf("%d", &n)) {
        int ans = 0;
        while(n) {
            n >>= 1;  //此处与 n/=2 意义等同，但位运算效率更高
            ans++;
        }
        printf("%d\n", ans);
    }
}
```

C jhljx 学位运算

思路分析

拿出这道题，大家最容易想到的是针对每次查询，对每次查询区间范围内的数一个一个进行异或和的计算。伪代码片段如下：

```
ans = 0;
for x = i to j
    ans ^= ax
```

这样的方法，针对每次查询时间复杂度为 O(n)，而这并不是我们想要的。那么问题来了：如何才能高效的得到结果呢？

我们可以根据异或的性质，和求数组区间和的思路相类似，每次输入的时候，输入第 i 个数，数组 a 的第 i 个元素存储前 i 个数的异或和，然后针对每次查询，输入 x 与 y，在确认 x 不大于 y 的情况下，只需计算 a[y]^a[x-1]的值，相当于将前面部分清零。这样，我们就可以高效地得出结果。

至于此做法的正确性，说明如下：

设 a[n]为给出的整数数列，s[n]中的第 i 项表示数列 a[n]的前 i 项异或和（1≤i≤n）。由于异或的性质有 a^a=0, a^0=a, 对于给出的 m，n，**在 m 不大于 n 的情况下**有：

$$a[m] \oplus a[m+1] \oplus \dots \oplus a[n]$$

$$= (a[0] \oplus a[1] \oplus \dots \oplus a[m-1]) \oplus (a[0] \oplus a[1] \oplus \dots \oplus a[m-1] \oplus a[m] \oplus a[m+1] \oplus \dots \oplus a[n])$$

$$= s[m-1] \oplus s[n]$$

算法分析

对于上文提到的第二种算法，数据预处理时间复杂度为 O(n)，每次查询所需要的计算的时间复杂度为 O(1)。

参考代码

```
#include<cstdio>
int buf[1000010];        //此数组存储前 i 项异或和
int a[1000010];          //此数组存储输入数据，可不需要
void MySwap(int &i, int &j){
    if(i>j){
        int temp=j;
        j=i;
        i=temp;
    }
```

```
}
void run(){
    int n;
    while(~scanf("%d",&n)){
        //边输入边进行数据预处理
        for(int l=1;l<=n;l++){
            scanf("%d",&a[l]);
            //数据预处理过程
            if(l==1) buf[l]=a[l];
            else buf[l]=buf[l-1]^a[l];
        }
        int t;
        scanf("%d",&t);
        while(t--){
            int i,j;
            scanf("%d%d",&i,&j);
            MySwap(i,j); //题目中 i, j 大小不确定，因此需要交换操作
            if(i==j) printf("%d\n",a[i]);
            else{
                int ans=buf[i-1]^buf[j];      //计算结果
                printf("%d\n",ans);
            }
        }
    }
}
int main()
{
    buf[0]=0; //注意如果查询有一个数为 1 需要对这个数进行操作
    run();
}
```

D 股票交易

思路分析

方法一 暴力求解法

这道题我们很容易地设计出一个暴力方法来求解本问题：简单地尝试每对可能的买进和卖出日期组合，只要卖出日期在买入日期之后即可然后求得最大收益。显然，日期组合有$\Theta(n^2)$种，而处理每对日期所花费的实践至少也是常量。因此，这种方法的运行时间是$\Omega(n^2)$。这并不是我们需要的。

方法二 分治算法

针对这样的问题，我们可以求各个时间点之间的价格变化。如：

天	0	1	2	3	4
价格	10	11	7	10	6
变化		1	-4	3	-4

针对长度比较小的数组，求出来当然很快；然而对于很长的数组而言，暴力求解上述变化数组中和最大数组的和的运行时间是$\Omega(n^2)$，也是行不通的。因此，我们可以递归地二分子问题如下：

设价格变化数组为 A[low…high]，子数组中央位置为 mid，则数组 A 的子数组 A[i…j]只可能是下述三种情况之一：

1.完全位于 A[low…mid]，即 low≤i≤j≤mid

2.完全位于 A[mid+1…high]，即 mid＜i≤j≤high

3.跨越了中点，即 low≤i≤mid＜j≤high

因此，其最大子数组必然是上述三种情况之一。因此我们可以递归地求解左右两个子数组的最大子数组，剩下的就是寻找跨越中点的最大子数组，然后在这三种情况中选和最大的。

伪代码实现过程参见《算法导论》中文第三版第 40 至 41 页 FIND-MAX-CROSSING-SUBARRAY 方法和 FIND-MAXIMUM-SUBARRAY 方法（英文第三版为第 71 至 73 页，宋老师给大家的电子版教材输入第 92 页即可找到）；C++代码实现参见参考代码一。

方法三 线性时间求解法

事实上，一般我们求解这类问题使用的是线性时间的算法。

针对方法一带来的重复计算，我们可以从前到后扫一遍代码，一边扫描一边记录数组前 i 项的最小值，一边记录当前值与记录到的最小值的差，迭代得到的最大值即为所求。即：扫描到第 i 项时，设 a[n]是给出的股票价格数组，x 是前 i-1 项的最小值，则第 0 到第 i 天的最大收益 ans=max(ans, a[i]-x).

C++代码实现参见参考代码二。

算法分析

对于方法二，FIND-MAX-CROSSING-SUBARRAY 方法花费的时间是$\Theta(n)$，FIND-MAXIMUM-SUBARRAY 方法运行时间 T(n)可以以下述递归式表示：

$$T(n)=\begin{cases}1 & (n=1)\\ 2T\left(\frac{n}{2}\right)+\Theta(n) & (n>1)\end{cases}$$

由主定理可知，a=2，b=2，$log_b a=1$，$f(n)=\Theta(n)=\Theta(n^{log_b a})$

因此，$T(n)=\Theta(n^{log_b a}lgn)=\Theta(nlgn)$

对于方法三，针对每个数组，只需执行从头到尾一遍扫描即可，时间复杂度为$\Theta(n)$

参考代码一

```
#include<cstdio>
#define INF 0x80000000  //int 类型能表示的最小负数
//数组 a 存储给出的股票价格数组，数组 b 存储股票价格变化数组
int a[1000010],b[1000010];
//此函数用于分治法求解最大子数组问题
int max_sub_array(int arr[],int l,int r)
{
    if(l<r){
        int mid=(l+r)/2;
        int suml=max_sub_array(arr,l,mid);   //求左边子数组的最大子数组
        int sumr=max_sub_array(arr,mid+1,r); //求右边子数组的最大子数组
        int sum_both=0;

        //寻找左半部分数组的最大和
        int max_left=INF;
        for(int i=mid;i>=l;i--)
        {
            sum_both+=arr[i];
            if(sum_both>max_left)
                max_left=sum_both;
        }

        //寻找右半部分数组的最大和
        int max_right=INF;
        sum_both=0;
```

```
        for(int i=mid+1;i<=r;i++)
        {
            sum_both+=arr[i];
            if(sum_both>max_right)
                max_right=sum_both;
        }

        //计算跨越中点子数组的最大和
        sum_both=max_left+max_right;

        //判断三种情况哪种情况求得的子数组最大
        if(sumr<sum_both && suml<sum_both)
            return sum_both;
        else if(suml<sumr)
            return sumr;
        else
            return suml;
    }
    else
        return arr[l];   //处理 l==r 的情形
}

int main()
{
    int n,t;
    while(~scanf("%d",&n)){
        for(int i=0;i<n;i++){
            scanf("%d",&a[i]);
            if(i>0) b[i]=a[i]-a[i-1];  //处理得到价格变化数组
        }
        int sum = max_sub_array(b,1,n-1);
        if(sum<=0) printf("No solution\n");
        else printf("%d\n",sum);
    }
}
```

参考代码二

```
#include<cstdio>
int main(){
    int n;
    while(~scanf("%d",&n)){
        int res,ans=0,x;     //ans 记录最终结果，res 记录数组 a[n]前 i-1 项的最小值
```

```
        int inp1=scanf("%d",&x);
        res=x;
        for(int i=2;i<=n;i++){
            scanf("%d",&x);  //x 相当于记录 a[i]的当前值
            //迭代计算最大收益的计算过程
            if(ans<x-res)
                ans=x-res;

            //迭代计算前 i-1 项最小值的计算过程
            if(x<res)
                res=x;
        }
        if(ans==0) printf("No solution\n");
        else printf("%d\n",ans);
    }
}
```

参考资料

1.《算法导论》中文第三版，机械工业出版社

E 模式寻对

思路分析

我们最容易想到通过直接暴力搜索可以得到逆序对数。代码实现如下：

```
long long slot(int a[], int head, int tail)
{
    long long ans=0;
    if(head>=tail)
        return 0;
    for(int i=head;i<tail;i++){
        for(int j=i+1;j<=tail;j++){
            if(a[j]<a[i]) ans++;
        }
    }
    return ans;
}
```

不难看出，上述代码标记为红色的部分，即暴力搜索求逆序对数的过程时间复杂度为$\Theta(n^2)$，效率非常低。因此，我们需要高效一些的算法来解决这个问题。归并排序就是其中一种上佳的选择。

简单来说，在归并排序操作中，当把 rightSubArray（右半部分有序数组）中的元素复制到原 Array 的时候，**统计这个数字一下子跨过了 leftSubArray（左半部分有序数组）中的多少个数字，这个数字就是这次排序中的逆序数。**

和 D 题分治算法相同，逆序数同样需要考虑以下三种情形：

① 完全在左子数组中；

② 完全在右子数组中；

③ 合并数组时跨越中间的数的情况。

最终结果即为上述三种情况所得逆序数之和。

具体计算上，两个子区间已经完成升序排序，我们不断从两个子区间的左端取出最小的元素，从左到右放置在一个临时数组上。如果我们发现当前最小元素（记为 A）在右子区间上，说明左子区间中剩下的所有元素和 A，都是逆序对（①A<左子区间最小元素<左子区间剩下的其他元素 ②左子区间所有元素和 A 的位置关系保持不变）。

对于本题而言，使用归并排序算法求逆序数的时候需要注意，同一个数组的每次查询都会让原本乱序的数组排好序，因此在每次查询前或者查询后需要复制原数组到排序数组中，否则从第二次查询开始结果

均为 0。不过此过程相比排序而言所花费的开销可以不计。

举个例子：

已知

A[5] = {1,6,8,2,7}; leftSubArray ={1,6,8}; rightSubArray = {2,7};

void Merge(int A[], int low, intmid, int right);

//left, right, mid 都是物理位置，而非逻辑位置

我们现在进行到了 Merge(A, 0, 2, 5)这一步，也就是说最后一步

我们采用算法导论上的算法，从而 A 的变化过程如下：

S1: A[5] = {1,6,8,2,7};

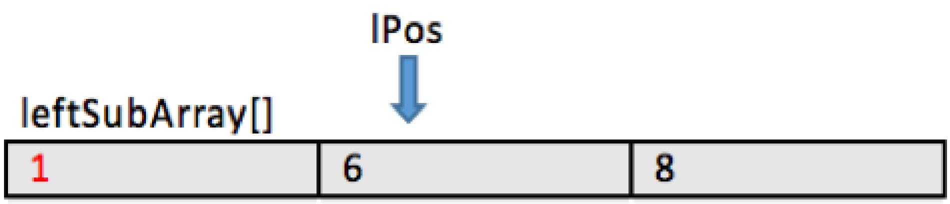
S2: A[5] = {1,2,8,2,7};

/* s1->s2 这一步中，数字 2 一下跳到了 **position = 1 的位置**。

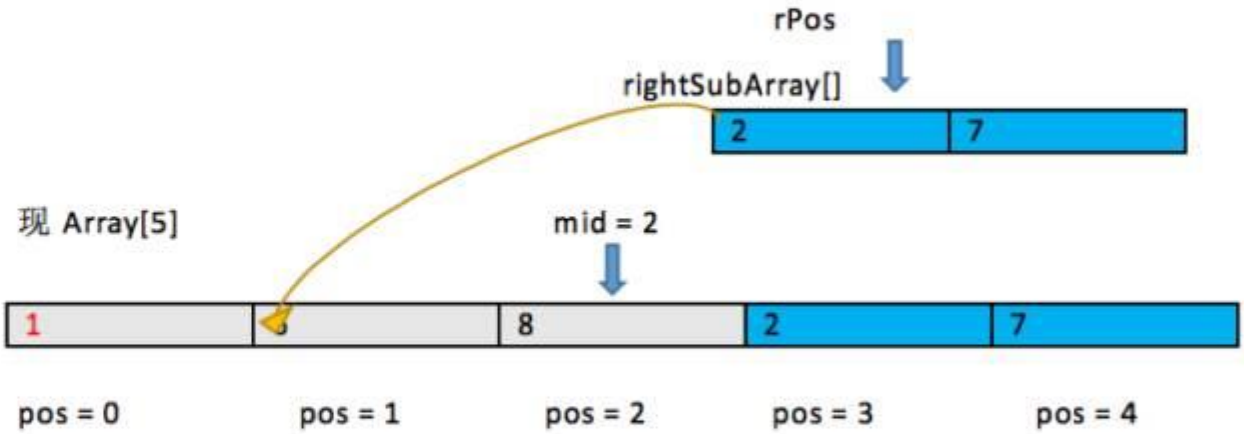
现在我们有一个简单的结论：当我们把 leftSubArray 中任意一个数字复制到数组 A 左边位置的时候，不会改变逆序数；而对于 rightSubArray 中的数字，它的移动会改变逆序数。

来一组 s1->s2 的直观感受

排序



现 Array[5]



可见，Array[3] = 2 从 pos = 3 左移到了 pos = 1 的位置，这时的相当于跨过了 Array[lPos] = 6 到 Array[mid] = 8 之间的所有数字，这些数字的个数是 mid – lPos + 1 = 2，这就是这次移动改变的逆序数。

原因很简单，由于 leftSubArray 本身就位于数组 A 的左边，而且它本身已经排好序，所以**对于 leftSubArray 中，即将左移复制进入数组 A 的数字，它左边的数字必定小于它，所以逆序数不会变！**

而对于 rightSubArray，即将被左移复制的元素是未复制的元素中最小的一个，所以它左边所有未被复制进入数组 A 的 leftSubArray 中的数字都会大于它，所以**这次左移复制改变的逆序数就是 leftSubArray 中未被复制的元素个数，即 mid– leftPos + 1** (这里的 leftSubArray 中，未被复制的第一个数字所在的位置的下标，即逻辑位置)。*/

S3: A[5] = {1,2,6,2,7};

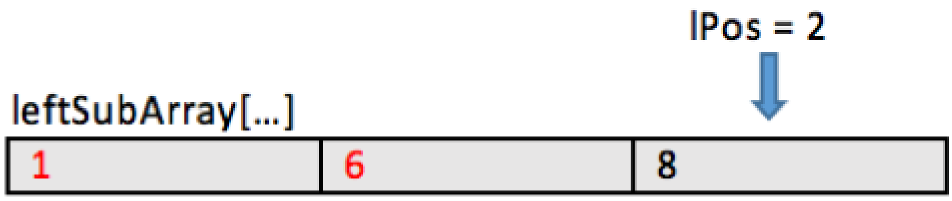
S4: A[5] = {1,2,6,7,7};

/*

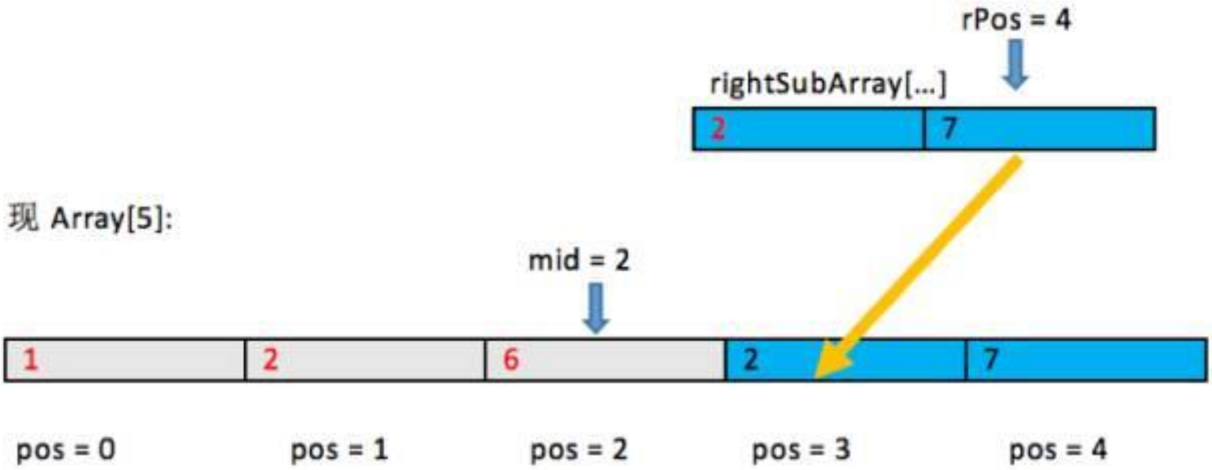
再来一组 s3->s4 的变化的直观感受：

未排序的 Array[5]

排序



现 Array[5]



Array[4]的左移复制跨过了 Array[lPos] = 8，所以改变的逆序数: mid – lPos + 1 = 1。

*/

S5: A[5] = {1,2,6,7,8};

算法分析

和归并排序相同，将数组递归二分子问题分别求解，将两个长度之和为 n 的有序子序列合并成一个有序序列过程至多进行 n-1 次比较，时间复杂度为$\Theta(n)$. 对于两路归并排序算法，递归式如下：

$$T(n)=\begin{cases}\Theta(1) & (n=1)\\ 2T\left(\frac{n}{2}\right)+\Theta(n) & (n>1)\end{cases}$$

由主定理可知，a=2，b=2，$log_b a=1$，$f(n)=\Theta(n)=\Theta(n^{log_b a})$

因此，$T(n)=\Theta(n^{log_b a}lgn)=\Theta(nlgn)$

故算法的时间复杂度为$\Theta(nlgn)$

参考代码

```
#include<cstring>
#include<iostream>
```

```
using namespace std;
int a[10010],b[10010],c[10010];
//c 存储原数组，a 将原数组复制之后存储排序结果，b 为辅助数组

//归并排序算法
long long slot(int head, int tail)
{
    long long ans=0;
    if(head>=tail)   //需要处理这样的情形
        return 0;
    int mid=(head+tail)/2;
    int i=head,j=mid+1,k=i;
    //统计左右两个子数组的逆序数并进行子数组的排序
    ans=slot(head,mid)+slot(mid+1,tail);

    //合并子数组过程
    //操作过程中累加逆序对的数量
    while((i<=mid) && (j<=tail))
    {
        if(a[i]<=a[j])
        {
            b[k++]=a[i++];
            ans+=(j-mid-1);
        }
        else
            b[k++]=a[j++];
    }

    //必要的补充
    while(i<=mid)
    {
        b[k++]=a[i++];
        ans+=(j-mid-1);
    }
    while(j<=tail)
        b[k++]=a[j++];
    for(i=head; i<=tail; ++i)
        a[i]=b[i];
    return ans;
}
int main()
```

```
{
    int n,t,x,y;
    while(cin>>n)
    {
        for(int i=0; i<n; i++)
            cin>>c[i];
        cin>>t;
        while(t--){
            cin>>x>>y;
            memset(b,0,sizeof(b));
            //复制数组操作
            for(int i=0;i<n;i++)
                a[i]=c[i];
            long long ans=slot(x,y);
            cout<<ans<<endl;
        }
    }
}
```

参考资料

1.2014 级算法第一次上机解题报告 E 题 Inverse number：Reborn, 提交人：徐硕

http://mp.weixin.qq.com/s?__biz=MjM5NjA3OTYxMg==&mid=400373696&idx=6&sn=68c2c6a73b979f5e472e3ba741384548&mpshare=1&scene=1&srcid=#rd

2.15101061_林克廉_算法第一次上机解题报告 E 题

3.《算法设计与分析——C++语言描述》，电子工业出版社，2006.5

问题思考

针对这类题，还有陈丹琦分治等算法能解决此类问题，并且效率更高，欢迎感兴趣的同学思考并交流。

F 究极汉诺塔

思路分析 Author: 15101061 林克廉

记法：第 i 小的盘 Plate[i]，初始柱为 Origin[i]，目标柱为 Final[i]

问题的分解

1. 找出需要移动的最大的盘 Plate[m]。显然比它大的盘 Plate[m+1..n]可以无视。
2. 如果 Plate[m]需要从柱 Origin[m]移动至柱 Final[m]，显然比它小的 m-1 个盘 Plate[1..m-1]都需要移动到第三根柱子（记为柱 Medium，Medium=6-Origin[m]-Final[m]）上。
3. 因此，移动方法分为三步走：把 Plate[1..m-1]集中到柱 Medium，然后移动 Plate[m]，然后再把 Plate[1..m-1]移动到各自的 Final[1..m-1]上。
4. “把 Plate[1..m-1]从 Medium 移到各自的 Final[1..m-1]上”，和“把他们从 Final[1..m-1]集中到 Medium 上”这两个问题是完全对称的，移动步数因而也是一样的。
5. 这样，待解决的问题就只剩一个了。给定 k 个盘的初始柱 Origin[1..k]；把 k 个盘集中于柱 Target；求移动步数。记该问题为 F(k, Target, Origin[])
6. 最终答案 ANS = F(m-1, Medium, Origin[]) + F(m-1, Medium, Final[]) + 1

分解化归出的问题：F(k, Target, Origin[])=?

1. 找出 k 个盘中需要移动的最大的盘 Plate[j]（1≤j≤k，如果都不需要移动，则得解 **F=0**）。由于 Plate[j+1..k]都已经移到柱 Target 上了，因此可以不考虑。
2. 要想把 Plate[1..j]移至 Target，必须先把 Plate[1..j-1]移到第三根柱子上（设为柱 NextTarget=6-Origin[j]-Target）。我们发现这个问题正是 F(j-1, NextTarget, Origin[])
3. 然后我们再移动 Plate[j]至 Target；最后把 Plate[1..j-1]从 NextTarget 上全部移到 Target 上，这是一个标准 Hanoi 问题，移动步数为 2^{j-1}-1。这两步移动步数共计 2^{j-1}
4. 因此 F(k, Target, Origin[]) = F(j-1, 6-Target-Origin[j], Origin[]) + 2^{j-1}。
5. 边界条件 1≤j，否则 F=0

综上，问题得解。

参考代码

```
#include<iostream>
using namespace std;

const int SUM = 6;  //此处 6 原因为 1+2+3
```

```
long long slot(int p[], int i, int res)
{
    if(i==0) return 0;
    else if(p[i]==res) return slot(p,i-1,res);
    else{
        int other = SUM - p[i] - res;
        long long buf = 1LL<<(i-1);
        return slot(p,i-1,other)+buf;
    }
}

int main()
{
    int n;
    int st[65],ed[65];
    while(cin>>n){
        if(n==0) break;
        for(int i=1;i<=n;i++) cin>>st[i];
        for(int i=1;i<=n;i++) cin>>ed[i];

        long long ans = 0;
        int k = n;
        //寻找最大的一个需要移动的盘子
        while(st[k]==ed[k]&&k>0) k--;
        if(k>0){
            int other = SUM - st[k] - ed[k];  /*求第 k 根柱子除起始、终止柱子外的
                                                那根辅助柱子*/
            ans = slot(st,k-1,other)+slot(ed,k-1,other)+1;
        }
        else ans=0;
        cout<<ans<<endl;
    }
}
```

参考资料

http://www.cnblogs.com/SeaSky0606/p/4569690.html 关于 UVa 10795 问题的解答