**线段树基础知识**

从简单说起，线段树其实可以理解成一种特殊的二叉树。但是这种二叉树较为平衡，和静态二叉树一样，都是提前已经建立好的树形结构。针对性强，所以效率要高。这里又想到了一句题外话：动态和静态的差别。动态结构较为灵活，但是速度较慢；静态结构节省内存，速度较快。

接着回到线段树上来，线段树是建立在线段的基础上，每个结点都代表了一条线段[a , b]。长度为1的线段成为元线段。非元线段都有两个子结点，左结点代表的线段为[a , (a + b ) / 2]，右结点代表的线段为[( a + b ) / 2 , b]。

图一就是一棵长度范围为[1 , 10]的线段树。

长度范围为[1 , L] 的一棵线段树的深度为log ( L - 1 ) + 1。这个显然，而且存储一棵线段树的空间复杂度为O（L）。

线段树支持最基本的操作为插入和删除一条线段。下面已插入为例，详细叙述，删除类似。

将一条线段[a , b] 插入到代表线段[l , r]的结点p中，如果p不是元线段，那么令mid＝（l＋r）/2。如果a<mid，那么将线段[a , b] 也插入到p的左儿子结点中，如果b>mid，那么将线段[a , b] 也插入到p的右儿子结点中。

插入（删除）操作的时间复杂度为O （ Log n ）。

上面的都是些基本的线段树结构，但只有这些并不能做什么，就好比一个程序有输入没输出，根本没有任何用处。

最简单的应用就是记录线段有否被覆盖，并随时查询当前被覆盖线段的总长度。那么此时可以在结点结构中加入一个变量int count；代表当前结点代表的子树中被覆盖的线段长度和。这样就要在插入（删除）当中维护这个count值，于是当前的覆盖总值就是根节点的count值了。

另外也可以将count换成bool cover；支持查找一个结点或线段是否被覆盖。

**例题1**（ZJU1610 Count The Colors 线段树基本应用题目）

给出在线段[0,8000]上的若干次涂色，问最后能看见哪些颜色，并统计能看到多少段。

**解析**

就这个题目而言，方法很多，而且数据范围不大，但我们由线段树的角度来解决这个问题。

建立一棵代表线段[0,8000]的线段树，涂色操作就是将[a , b]涂成颜色c。最后做统计。

结构如下：

struct  TNode {

        int     left , right;

        int     col;

        TNode   \*LeftChild , \*RightChild;

};

col 有几种情况，如果col为－1，代表了尚未涂色，－2代表了是混和色，就是说这条线段并不是单一的颜色。其他情况，便是这条线段都是这个颜色的了。

全部程序见附录1。  
  
**线段树的第一种变化**

基本的线段树代表的是线段，如果我们把线段离散成点来看，那么线段树可以变化成一种离散类型线段树。

这里可以有两种理解。一种离散关系可以是连续的线段的点，比方说在一条直线上放置的连续小球的着色问题；另一种则是完全将线段离散化分成若干小段，对每一段小段做为元线段来建立线段树，这种线段树可以支持实数划分类型的线段。

**例题2**（ZJU2451 Minimizing maximizer ）

Andy想要得到一组数中的最大值。会有一系列的操作*Sorter*(i[1], j[1]), ..., *Sorter*(i[k], j[k])。作用是将数组中的第i[k]个数字到第j[k]个数字排序。按照输入给出的顺序，你可以选择要不要执行这个操作。问题是最少需要多少步操作，可以求出这个最大值。题目保证可以求出。

多组数据。

第一行为两个数字N，M。表示N个数，M个操作。

接下来M行，每行描述一个操作i[k] , j [k]。

对于每组数据，输出最少需要多少次操作分离得到最大值。

每组数据一行。

**解析**

由于要将最大的数字分离到最后的一位，如果我们考虑将数组看成一条[1,n]的线段，而每项操作也看成是从[i[k],j[k]]的线段，那么就是要求按照输入的顺序，将线段[1,n]从左到右依次覆盖掉，问题变成求最小的覆盖线段总数。

考虑最基本的规划方法，用Opt [k] 表示覆盖掉 [1,k]的线段最少需要的步数，那么状态转移方程为：

Opt [k] = min { Opt [d] + 1  | j [p] = k && d >= i [p] && d <= j [p] && k > 1 }

Opt [1] = 0;

最后的答案就是Opt [n]了，但是考虑时间复杂度，是O（m^2）级别的，m最大为500000，超时无疑。但是这里我们看到了规划的决策集合是一条连续的线段，是要在这条线段上面取得最小值，那么线段树的结构就正好适合在这里应用了。

由于这里最小的单位是一个点，所以我们采取线段树的第一种变化，把元线段设置为单位点，即[k,k]。在规划的时候维护线段树即可。

线段树结点结构中，需要加入的元素是int minstep 代表最少需要用到的覆盖线段数目可以覆盖到当前结点所代表的线段中。

全部程序见附录2。

**例题3**（PKU2104K-th Number）

给出一个大小为n的数组A[]，给出m个问题（1 <= n <= 100 000, 1 <= m <= 5 000）。问题格式为Q（i，j，k），询问从A[i]到A[j]第k大的元素是什么。A[]中的数各不相同。

**解析**

由于仍旧是离散的整数问题，我们依旧采取第一种变化。看到题目，最基本的想法就是排序然后求第k个数了，但是时限上不能满足要求。

线段树的最强大方面就是将一组数（一条线段）放到一起处理。每层树需要的线段数目不会超过4，而深度为logn，所以最后操作的复杂度会是O(logn)。

但是仅仅应用线段树还是不够的，即使我们知道了需要处理的线段是那些，但是由于线段过多，也无法准确求出第k个元素究竟是什么。这里二分策略就派上了用场。我们用二分枚举第k个数字P，然后再在所要的线段中找到枚举的P所在的位置，同样是用二分的策略。所以复杂度是O(mlognlognlogn)。

我们在找P所在的位置的时候需要用到二分策略，也就是说，我们需要将线段所代表的结点排序，这里可以将每一层的所有数放到一起，统一成一个数组SortArray[depth][]。其实也可以理解成将归并排序的每个步骤记录下来。

全部程序见附录3。

**线段树的第二种变化 （树状数组）**

在结构上对线段树进行改变，可以得到线段树的另一种变化。

用O（n）的一维数组构造出线段树，无其他附加空间。比方说，一棵从[0,L]的线段树表示为TNode Tree[L];

这里应用二进制将树进行划分。将整数R的二进制表示中的最后一个1换成0，得到数L。Tree[R]代表的线段就是[L,R]。例如：6的二进制表示为（110）2将最后一个1换成0即为（100）2＝4，所以Tree[6]代表的线段就是[4,6]。

析出数R的最后一位1的方法是：LowBit（R）＝R^~R。

包含点L的一系列数为x1,x2,……，这里x1=R，x2=x1+LowBit (x1)，x3=x2+LowBit(x2)，……

这种线段树的优点在于：

1．  节省空间。完全线段长度的空间，无需左右指针，无需左右范围。

2．  线段树查找严格log(R)，因为二进制的每位查找一遍。

3．  状态转移快，操作简单。

4．  扩展到多维较为容易。

缺点：

1．随意表示线段[a,b]较为困难。

这种线段树适用于：

1．  查找线段[0,L]的信息。

2．  求线段[a,b]的和（应用部分和做差技术）。

// problem zju 1610

// Segment Tree

#define NoColor -1

#define MulColor -2

＃i nclude <stdio.h>

＃i nclude <string.h>

int     Len;

struct  TNode {

        int     left , right;

        int     col;

        TNode   \*LeftChild , \*RightChild;

        void    Construct ( int , int );

        void    Insert ( int , int , int );

        void    Calculate ();

} Tree [16000] , \*Root = &Tree [0];

int     CalColor [8001] , Many [8001];

void    TNode :: Construct ( int l , int r )

{

        left = l; right = r;

        if ( l + 1 == r ) { LeftChild = NULL; RightChild = NULL; return; }

        int     mid = ( l + r ) >> 1;

        LeftChild = &Tree [Len ++];

        RightChild = &Tree [Len ++];

        LeftChild->Construct( l , mid );

        RightChild->Construct( mid , r );

}

void    TNode :: Insert ( int l , int r , int c )

{

        if ( col == c ) return;

        if ( l == left && r == right ) { col = c; return; }

        int     mid = ( left + right ) >> 1;

        if ( col != MulColor ) { LeftChild -> col = col; RightChild -> col = col; }

        col = MulColor;

        if ( r <= mid ) { LeftChild -> Insert ( l , r , c ); return; }

        if ( l >= mid ) { RightChild -> Insert ( l , r , c ); return; }

        LeftChild -> Insert ( l , mid , c );

        RightChild -> Insert ( mid , r , c );

}

void    TNode :: Calculate ()

{

        if ( col != MulColor && col != NoColor ) {

                int     i;

                for ( i = left; i < right; i ++ ) CalColor [i] = col;

        }

        if ( col == MulColor ) { LeftChild -> Calculate (); RightChild -> Calculate (); }

}

main ()

{

        int     Total , a , b , c , i , t;

        Len = 1; Tree [0].Construct( 0 , 8000 );

//        printf ( "After Construct the Tree , Len = %d\n" , Len );

        while ( scanf ( "%d" , &Total ) != EOF ) {

                Tree [0].col = NoColor;

                while ( Total ) {

                        scanf ( "%d %d %d" , &a , &b , &c );

                        Root -> Insert( a , b , c );

                        Total --;

                }

                memset ( CalColor , 0xff , sizeof ( CalColor ) );

                memset ( Many , 0 , sizeof ( Many ));

                Root -> Calculate ();

                t = -1;

                for ( i = 0; i <= 8000; i ++ ) {

                        if ( CalColor [i] == t ) continue;

                        t = CalColor [i];

                        if ( t != -1 ) Many [t] ++;

                }

                for ( i = 0; i <= 8000; i ++ ) if ( Many [i] )

                        printf ( "%d %d\n" , i , Many [i] );

                printf ( "\n" );

        }

}

// Problem zju2451

// DP with Segment Tree

＃i nclude <stdio.h>

#define MAX     50000

int     Len;

struct  TNode {

        int     left , right;

        int     minstep;

        TNode   \*LeftChild , \*RightChild;

        void    Construct ( int , int );

        void    Insert ( int , int );

        int     GetRank ( int , int );

}       STree [MAX \* 2 + 2] , \*Root = &STree [0];

int     N , M;

void    TNode :: Construct ( int l , int r )

{

        left = l; right = r; minstep = 999999;

        if ( l == r ) { LeftChild = NULL; RightChild = NULL; return; }

        int     mid = ( l + r ) >> 1;

        LeftChild = &STree [Len ++];

        RightChild = &STree [Len ++];

        LeftChild->Construct ( l , mid );

        RightChild->Construct( mid + 1 , r );

}

void    TNode :: Insert ( int p , int x )

{

        if ( x < minstep ) minstep = x;

        if ( left == right ) return;

        if ( p <= ( left + right ) >> 1 ) LeftChild->Insert( p , x );

                else RightChild->Insert( p , x );

}

int     TNode :: GetRank ( int l , int r )

{

        if ( l == left && r == right ) return minstep;

        int     mid = ( left + right ) >> 1;

        if ( r <= mid ) return LeftChild->GetRank( l , r );

        if ( l > mid ) return RightChild->GetRank( l , r );

        int     ret1 , ret2;

        ret1 = LeftChild->GetRank( l , mid );

        ret2 = RightChild->GetRank( mid + 1 , r );

        return ret1 < ret2 ? ret1 : ret2;

}

main ()

{

        int     i , a , b , p;

        while ( scanf ( "%d %d" , &N , &M ) != EOF ) {

                Len = 1; Root->Construct( 1 , N );

                Root->Insert ( 1 , 0 );

                for ( i = 0; i < M; i ++ ) {

                        scanf ( "%d%d" , &a , &b );

                        if ( a < b ) {

                                p = Root->GetRank ( a , b - 1 );

                                Root->Insert ( b , p + 1 );

                        }

                }

                printf ( "%d\n" , Root->GetRank( N , N ) );

        }

}

// PKU 2104

// Segment Tree && Binnary Search

＃i nclude <stdio.h>

#define MAX     100000

int     len;

struct  TNode {

        int     left , right;

        char    depth;

        TNode   \*LeftChild , \*RightChild;

        void    construct ( int , int , int );

        int     GetRank ();

}       Node [2 \* MAX + 2];

int     SortArray [18] [MAX + 2];

int     Key , ls , rs;

void    TNode :: construct ( int l , int r , int dep )

{

        left = l; right = r; depth = dep;

        if ( left == right ) {

                scanf ( "%d" , &SortArray [dep] [l] );

                return;

        }

        int     mid = ( l + r ) >> 1;

        LeftChild = &Node [len ++];

        LeftChild->construct( l , mid , dep + 1 );

        RightChild = &Node [len ++];

        RightChild->construct( mid + 1 , right , dep + 1 );

        int     i = left , j = mid + 1 , k = left;

        while ( i <= mid && j <= r ) {

                if ( SortArray [dep + 1] [i] < SortArray [dep + 1] [j] )

                        SortArray [dep] [k ++] = SortArray [dep + 1] [i ++];

                        else

                        SortArray [dep] [k ++] = SortArray [dep + 1] [j ++];

        }

        while ( i <= mid ) SortArray [dep] [k ++] = SortArray [dep + 1] [i ++];

        while ( j <= right ) SortArray [dep] [k ++] = SortArray [dep + 1] [j ++];

}

int     TNode :: GetRank ()

{

        if ( ls <= left && right <= rs ) {

                if ( SortArray [depth] [left] >= Key ) return 0;

                if ( SortArray [depth] [right] < Key ) return right - left + 1;

                if ( SortArray [depth] [right] == Key ) return right - left;

                int     low = left , high = right , mid;

                while ( low + 1 < high ) {

                        mid = ( low + high ) >> 1;

                        if ( SortArray [depth] [mid] < Key ) low = mid;

                                else high = mid;

                }

                return low - left + 1;

        }

        int     ret = 0;

        if ( ls <= LeftChild->right ) ret += LeftChild->GetRank();

        if ( RightChild->left <= rs ) ret += RightChild->GetRank();

        return ret;

}

main ()

{

        int     N , Q , i;

        int     low , high , mid , Index;

        scanf ( "%d%d" , &N , &Q );

        len = 1; Node [0].construct( 0 , N - 1 , 0 );

        for ( i = 0; i < Q; i ++ ) {

                scanf ( "%d%d%d" , &ls , &rs , &Index );

                ls --; rs --;

                low = 0; high = N;

                while ( low + 1 < high ) {

                        mid = ( low + high ) >> 1;

                        Key = SortArray [0] [mid];

                        if ( Node [0].GetRank() >= Index ) high = mid;

                                else low = mid;

                }

                printf ( "%d\n" , SortArray [0] [low] );

        }

}