SPOJ LITE(USACO 08 NOV) Solution

Write by Gestalt Lur
2012-06-11

題目大意

第一行輸入 N，M 兩個整數，表示有 N（1<=N<=10^5）個燈排成一行，每個燈有兩種狀態，關閉或者開啟。初始的時候所有的燈都處于關閉狀態。接下來 M(1<=M<=10^5)行，每行格式如下：一種操作,Si,Ei(1<=Si<=Ei<=N)。操作用 0 或者 1 來表示。為 0 表示切換 Si 至 Ei 的所有燈的狀態。為 1 則輸出這個區間內有多少亮著的燈。

算法分析

通過題目描述顯然可以聯想到區間值相關的問題，因而可以考慮使用線段樹。線段樹的節點中記錄當前區間亮著的燈的個數即可,這裏將節點 s 的亮著的燈的個數記作 sum[s]。實現的時候對于要切換燈的狀態的節點則有 sum[s] = r - l + 1 - sum[s].但是可以想到，在切換了節點 s 上的燈的狀態之後，它的孩子節點以及這其子樹上的所有節點也需要轉換狀態，如果同樣要轉換其孩子節點的 sum 值，那麼每次維護的時間復雜度約是 O(2^[N-log(2)])，顯然不能滿足題目的要求。這時我們就考慮在切換（更新）操作時，如何不去更新節點 s 的孩子。

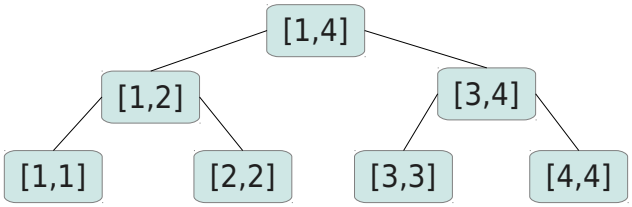
一個簡便的方法就是在每個節點上做一個標記，表示**以該節點為根的子樹是否需要轉換**。就是說，當我們在更新的時候找到一個需要轉換狀態的節點 s 時，只更新 s 的 sum，而不去管它的子樹的狀態，只做一個標記 mark[s]:=true,表示它的子樹的狀態需要轉換。

在之後的操作中，碰到查找時，**如果沒有找到要找的節點，當前的節點 mark[s]若為 true,則轉換 s 的孩子們的 sum 值，并將孩子的 mark 值异或 1（因為以某個孩子為根的子樹如果需要轉換，這裏就恰好抵消了）。最後將 mark[s]賦為 false（這個節點已經轉換完了）。**然後繼續查找過程。最後當找到相應的節點後，需要向上更新查找路徑上節點的 sum 值（可以用 遞歸的方式完成）。

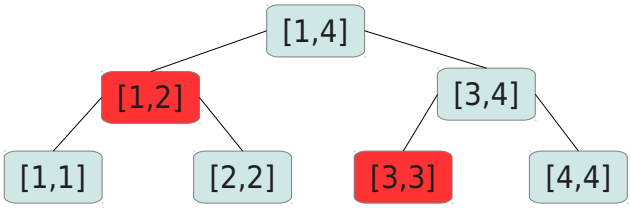
在轉換（更新）操作中，如果我們沒有找到要更改的節點，則也需要進行上一段黑體字的操作。

這個方法的大體思想就是在更新的時候只做標記，再次訪問這個做過標記節點的時候才真正去更新這個節點的值。

下面舉個例子:

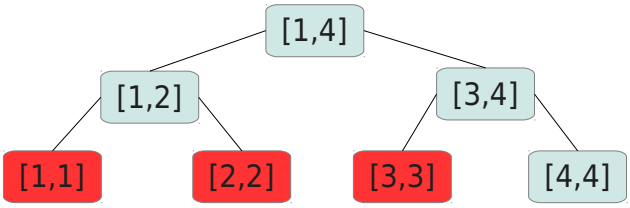


如果在這顆線段樹上轉換[1,3]的燈的狀態，顯然找到的節點是[1,2]和[3,3]，所以只要把這兩個節點的值轉換之後給這兩個節點做標記（紅色表示）向上更新查找經過的節點的 sum 值就可以了，這個操作結束之後線段樹的狀態應該是這樣的：



node	[1,4]	[1,2]	[3,4]	[1,1]	[2,2]	[3,3]	[4,4]
sum	3	2	1	0	0	1	0
mark	FALSE	TRUE	FALSE	FALSE	FALSE	TRUE	FALSE

接下來的操作如果是查找[2,4]，那麼對應找到的節點應該是[2,2]和[3,4]。在查找[2,2]節點時經過了有標記的[1,2]，則轉換[1,1]和[2,2]的值并向下繼續查找，這時線段樹的狀態如下：



node	[1,4]	[1,2]	[3,4]	[1,1]	[2,2]	[3,3]	[4,4]
sum	3	2	1	1	1	1	0
mark	FALSE	FALSE	FALSE	TRUE	TRUE	TRUE	FALSE

所以當返回[2,2]和[3,4]的時候這兩個節點的 sum 值就是正確的了。

可以看出以有標記的節點為根的子樹實際的意義就是這些點上的 sum 值不是最新的，需要在訪問的時候更新，在訪問的過程中也只對訪問到的標記做最少的改變（把標記賦給它的孩子）。這樣就可以做到遞歸實現線段樹的一個比較快速的方法了。所以這個標記方法有人也稱之為 lazy 標記[1]。

1　這個名稱的出處？

參考代碼

```
#include <cstdio>
#define MAXN 2000002

int sum[ MAXN << 2 ];//節點的 sum 值,由于節點的左右端點可以在查找時推出，所以無需記錄
bool st[ MAXN << 2 ];//標記

//更新標記的操作
void update_child( int rt , int l , int r )
{
 if( st[ rt ] )
   {
    int mid = l + r >> 1;
    st[ rt ] = 0;
//更新節點孩子的標記
    st[ rt << 1 ] ^= 1;
    st[ rt << 1 | 1 ] ^= 1;
   //更新孩子的 sum 值
    sum[ rt << 1 ] = ( mid - l + 1 ) - sum[ rt << 1 ];
    sum[ rt << 1 | 1 ] = ( r - mid ) - sum[ rt << 1 | 1 ];
   }
 return ;
}

 //the interval that to be found is [ L , R ]
void update( int L , int R , int l , int r , int rt )
{
 if ( L <= l and r <= R )//查找的区间不变，判断当前的区间是否在要找的区间中
   {
    st[ rt ] ^= 1;
    sum[ rt ] = ( r - l + 1 ) - sum[ rt ];
    return ;
   }

 //update children
 update_child( rt , l , r );

 int m = l + r >> 1;
 if ( L <= m ) update( L , R , l , m , rt << 1 );
 if( R > m ) update( L , R , m + 1 , r , rt << 1 | 1 ) ;

 //update precursors，递归地更新查找路径上的节点的 sum 值
```

```
 sum[ rt ] = sum[ rt << 1 ] + sum[ rt << 1 | 1 ];
 return ;
}

int query( int L , int R , int l , int r , int rt )
{

 if (L <= l and r <= R )
   return sum[ rt ];

 //update children
 update_child( rt , l , r );

 int m = l + r >> 1;
 int ret = 0;

 if (L <= m) ret += query( L , R , l , m , rt << 1 );
 if (R > m) ret += query( L , R , m + 1 , r , rt << 1 | 1 );

 return ret;
}

int main()
{
 int m , n , a , b;
 int tl , tr , ins;

 scanf( "%d%d" , &n , &m );

 for( int i = 1 ; i <= m ; ++ i )
   {
    scanf( "%d%d%d" , &ins , &tl , &tr );
    if( !ins )
      update( tl , tr , 1 , n , 1 );
    else
      printf( "%d\n" , query( tl , tr , 1 , n , 1 ) );
   }
 return 0;
}
```