Recommendations

Calvin Fung

September 10, 2025

1 Problem

Suppose you are working in some audio streaming service. The service has n active users and 10^9 tracks users can listen to. Users can like tracks and, based on likes, the service should recommend them new tracks.

Tracks are numbered from 1 to 10^9 . It turned out that tracks the i-th user likes form a segment $[l_i, r_i]$.

Let's say that the user j is a predictor for user i ($j \neq i$) if user j likes all tracks the i-th user likes (and, possibly, some other tracks too).

Also, let's say that a track is strongly recommended for user i if the track is not liked by the i-th user yet, but it is liked by every predictor for the i-th user.

Calculate the number of strongly recommended tracks for each user i . If a user doesn't have any predictors, then print 0 for that user.

2 Solution

We present an $O(n \lg n)$ solution.

2.1 analysis

Observation 1. Consider user i and let P be the set of all its predictors. Clearly the set of strongly recommended tracks for i is $(\cup_{j \in P}[l_j, r_j]) \setminus [l_i, r_i]$. If P is empty then clearly the number of strongly recommended tracks is 0. Otherwise let $l^* = \max_{j \in P} l_j$ and $r^* = \min_{j \in P} r_j$, then the set of strongly recommended tracks is $(\cup_{j \in P}[l_j, r_j]) \setminus [l_i, r_i] = [l^*, r^*] \setminus [l_i, r_i]$ and its size is $r^* - l^* + 1 - (r_i - l_i + 1) = r^* - l^* - r_i + l_i$ because $l^* \leq l_i, r_i \leq r^*$ (recall property of a predictor).

We'll sort the intervals with the criteria : $[l_i, r_i]$ is consider to go before $[l_j, r_j]$ if (i) it starts earlier $l_i < l_j$ or (ii) in the case that $l_i = l_j$, it's longer $r_i - l_i + 1 > r_j - l_j + 1$. We'll then process the intervals in this order.

Let T be an ordered-set (implemented using AVL tree) storing all end times of processed intervals. S be a stack storing some of the processed intervals in strictly increasing end time ($S.top()$ is the interval with smallest end time in S), consider two consecutive intervals $[l_i, r_i], [l_j, r_j]$ in S where i is closer to $S.top()$, we maintain the property that j is the interval with the largest start time (or equivalently the last interval processed) among all processed intervals with end time $> r_i$. The special case is that $S.top()$ is the interval with the largest start time (or equivalently the last interval processed) among all processed intervals.

Suppose we're now processing an interval $[l, r]$ and assume it's unique among the set of all intervals (we'll deal with the other case later). Notice all predictors for $[l, r]$ must have already been processed because they either start strictly before l or they start at l and is longer. Consider the smallest $t \in T$ such that $t \geq r$, notice all predictors for $[l, r]$ is exactly all processed intervals with end time $\geq t$ because all processed intervals start no later than l (if t doesn't exist then there's no predictor). We now keep popping intervals from S until we see an interval $[l_j, r_j]$ where $r_j \geq r$, since j is a processed interval we must have $r_j \geq t$. If some interval was popped, then there was an interval i right infront of j in S (that was popped), noticing $r_i < r \leq t \leq r_j$ (and recall the property of S) we conclude that j is the interval with the largest start time among all processed intervals with end time $\geq t$. Similar (actually easier) argument show this holds when no interval is popped. Now applying observation 1 here, we have $l^* = l_j$ and $r^* = t$ so the number of strongly recommended tracks for $[l, r]$ is $t - l_j - r + l$. Finally, we insert r to T and keeping popping S until either it's empty or that $S.top()$ becomes some $[l', r']$ such that $r < r'$, then we push $[l, r]$ into S.

For the edge case where interval $[l, r]$ is not unique, first note that this can be easy detected because all such intervals will be consecutive in our sorted order. In such case we will have 0 strongly recommended tracks for all such intervals so we only need to insert r to T and keeping popping S until either it's empty or that $S.top()$ becomes some $[l', r']$ such that $r < r'$, then we push $[l, r]$ into S.

Since we need to sort and in each iteration we're doing at most 3 AVL tree operations, moreover, each interval is inserted and popped at most twice in S, the total run time is $O(n \lg n)$.

2.2 implementation

```
#include<iostream>
#include<set>
#include<stack>
#include <algorithm>
#define N 200001
using namespace std;

int t,n,A[N],L[N],R[N],res[N];

bool cmpl(int i,int j) {
    if(L[i] == L[j])
        return R[i]-L[i] > R[j]-L[j];
    else
        return L[i]<L[j];
}

bool cmpr(int i,int j) {return R[i]<R[j];}

void slv(){
    scanf("%d",&n);
    for(int i=1;i<=n;++i)    scanf("%d %d",&L[i],&R[i]), A[i]=i;
    sort(A+1,A+n+1,cmpl);

    bool(*fn_pt)(int,int) = cmpr;
    set<int,bool(*)(int,int)> T (fn_pt);
    stack<int> S;

    //base case
    int i = 2;
    T.insert(A[1]);
    S.push(A[1]);
    res[A[1]] = 0;
    if(1 < n && L[A[1]] == L[A[2]] && R[A[1]] == R[A[2]]){
        while(i <= n && L[A[i]] == L[A[i-1]] && R[A[i]] == R[A[i-1]]){
            res[A[i]] = 0;
            i++;
        }
    }

    while(i <= n){
        if(i < n && L[A[i]] == L[A[i+1]] && R[A[i]] == R[A[i+1]]){
            T.insert(A[i]);
```

```
            while(!S.empty() && R[S.top()] <= R[A[i]]){
                S.pop();
            }
            S.push(A[i]);
            res[A[i]] = 0;
            i++;
            while(i <= n && L[A[i]] == L[A[i-1]] && R[A[i]] == R[A[i-1]]){
                res[A[i]] = 0;
                i++;
            }
        }else{
            set<int>::iterator j = T.find(A[i]);
            if(j == T.end()){
                j = T.upper_bound(A[i]);
            }
            if(j != T.end()){
                while(R[S.top()] < R[A[i]]){
                    S.pop();
                }
                res[A[i]] = R[*j] - L[S.top()] - R[A[i]] + L[A[i]];
                if(R[S.top()] == R[A[i]])   S.pop();
                S.push(A[i]);
            }else{
                while(!S.empty()){
                    S.pop();
                }
                S.push(A[i]);
                res[A[i]] = 0;
            }
            T.insert(A[i]);
            i++;
        }
    }

    for(int i=1;i<=n;++i)    printf("%d\n",res[i]);

}

int main(){

    scanf("%d",&t);
    while(t--) slv();return 0;

}
```