DIFFICULTY:

medium-hard

PREREQUISITES:

sqrt decompositions, Fenwick Trees (or Interval Trees).

PROBLEM:

You're given an array, which can modify in time. You're given some functions which return sum of some contiguous subsequence (subarray). Answer queries, which ask sum of a contiguous subsequence of functions.

QUICK EXPLANATION (thanks to Sunny Aggarwal):

In the begining, we can find the value of every function and thus we can easily find the sum of functions of every chunk in the beginning easily. Now we need to take care for updating. For updating, we need in O(1) time for each chunk that how many time x comes in summation of the functions in that chunk. We can precompute this factor in O(N*sqrt(N)) space and time.

Thus, we can update this structure in O(sqrt(N)) by traversing through all the chunks and applying +(y*factor)-(prev*factor) at every chunk.

We will quickly travel between 2 points which has full chunks. Only thing to worry is with partial chunks. We can use BIT's to achieve this in O(sqrt(N)*log(N)).

EXPLANATION:

Complexity of the official solution

This looks like a weird start, isn't it? Well, actually the idea is that you *might* try to find a complexity better than needed. Our solution uses O(log(n) * sqrt(n)) per update and O(sqrt(n)) per query. If you're trying to find a better complexity, this might be a hint to look for this complexity, as it passes the systests. If you have a solution with better complexity, please share it with us in the comments!

Smart brute force

A brute force solution would be at following: do an update in O(1) and query by iterating over all functions from m to n and for each function iterate all elements from L to R. This gives a running time of O(n ^ 2) per query worst case... not good.

An optimization might be pretty obvious for those who know Segment Tree / Fenwick Tree. We keep iterating all functions from m to n. But, for a function i, we can avoid calculating it in O(n) worst case. A function f(i) asks you for a sum of a range. Also, updates change a single element. Both Fenwick Trees and Segment Trees allow changing an element and asking for a sum in O(log(n)) time. So, for an update, using this technique, we can get O(log(n)) time and for a query we can get O(n * log(n))... still not good enough.

No updates version

Suppose we have no updates - then we can precalculate some information. In this way, a query can be answered faster. A well know trick is used: sqrt decomposition. If you don't know it, please read it from here. We keep L "buckets" of length L (L = sqrt(n)). First bucket will contain sum of 1st, 2nd, ..., Lth functions. Second bucket will contain sum of (L+1)th, (L+2)th, ..., (2*L)th functions and so on. As showed in the link, when we're given a query, we can iterate over "full blocks". These values are already calculated. Also, there can be at most 2 "incomplete blocks". For functions in incomplete blocks, but which are in query range, we need to calculate the values of the functions using the Fenwick tree. This way we get O(log(n)) per update and O(log(n) * sqrt(n)) per query.

Full version

What happens when a position gets a new value? Firstly, we need to update Fenwick tree (or Segment tree). This will help for having the correct values when iterating elements in incomplete blocks. Suppose previous value was prev and my new value is val. We need to iterate each "complete" bucket. Suppose for a bucket we know how many functions from it contain position pos. Formally, we need to count functions f(i) from a bucket such as L[i] <= pos <= R[i]. For all those functions, we'll have to erase prev and add val. Let's denote number of functions from bucket which contain position pos by x. Then, it can be showed that bucket's value is modified by adding x * (val - prev) (for each function which contains position pos, we remove prev and add val, in total whole bucket modifies by x * (val - prev)).

It remains a single detail to handle: we need to know, for each position and for each bucket, how many functions from that bucket contain that position (value x from above). Let's define matrix x[bucket][pos] = how many functions from the bucket contain position pos. How to do this in a reasonable time?

Let's iterate over all buckets. Suppose we're solving now bucket B. It will contain some functions which say: f(x0, y0) = all positions between x0 and y0 appear one more time in the bucket B. So, we need to increase x[B][x0], x[B][x0 + 1], ..., x[B][y0 - 1], x[B][y0] by 1. If the increase is done in O(1) time plus O(n) at final, complexity becomes O(n * sqrt(n)). How do to this?

The problem reduced to: given M queries of form (x0, y0), increase all elements from all positions from x0 to y0 by 1. The solution is well known. We keep two arrays: A and B. When you get an operation x0, y0, you do:

A[x0] += 1; A[y0 + 1] -= 1;

Then, you do B[i] = A[0] + A 1 + ... + A[i] = A[i] + B[i - 1].

The correct answers will be in array B. We leave as an exercise to proof why this works.

This problem allows multiple solutions. If you have alternative solutions, please share them with us in the comment section!