Code Jam 2013 - Round 1C

Analysis: The Great Wall

The small input

Despite the very long statement, solving the small input wasn't actually that hard. Still, the long statement scared many contestants off, which is probably why we saw the first submission only after half an hour of the contest, and relatively few submissions to the problem in general. With at most 10 tribes, at most 10 attacks and all the attacks happening on a short section of the Wall, we can just simulate all that happens. Let's look at it a bit more carefully.

Since **delta_p** is limited by 10, a tribe attacks at most 10 times, and the initial attack is between -100 and 100, all the attacks will occur between -200 and 200. Thus, we can afford to remember the height of the wall at each interesting point. This brings us to the first trick of this problem — what are the points we should be interested in?

Note that since the edges of attacked areas are always integers, the height of the wall in each open interval (*x, x+1*) for integral *x* is always constant. Moreover, the height at the integral points is never lower than at any of the two neighboring open intervals, since any attack that affects any of these intervals will also affect the integral point next to it. As $\mathbf{w_i} < \mathbf{e_i}$, any attack always affects at least one whole interval, and so the success of the attack depends only on the height of the wall in the intervals, and not on the edges. Thus, it is enough to keep information about the height of the wall in points of the form **x** + 0.5 for integral **x**. There are 400 such points to consider in the small input, and the height of each is initially zero.

There are a 100 attacks to consider. We can begin by generating all of them explicitly (noting the beginning and end point, day and strength for each of them), and sorting them by time of occurrence. For each day on which at least one attack occurs, we first check for each attack whether it succeeds (by examining the wall height at each attacked interval). Afterwards, for all attacks we go over all affected intervals and increase the height of the wall if necessary. Note that it is important to increase the wall height only after checking *all* the attacks that occur on a given day.

The large input

The numbers are much bigger for the large input. We can have 10^6 attacks, and they can range over an interval of length over 10^8. Let's analyse which parts of the previous approach will work, and which will not.

We can still generate all the attacks explicitly, and sort them by time. We probably need a more concise way to represent the Wall, though, and we surely need a faster way to check whether an attack succeeds and updating wall heights.

The problem of concise representation can be solved by noticing that since we have only 10^6 attacks, we will have around 10^6 interesting points. A sample way to take advantage of this it to "compress" all attack coordinates — sort all the coordinates that are beginnings or ends of attacks, and consider as interesting only the points in the middles of intervals of adjacent endpoints. We will end up with at most 2×10^6 points, and each will represent an interval such that the height of the wall on this interval is always the same. Using this tric to compress the attack coordinates, we can assume all attacks happen in a space of at most 2×10^6 points. We can rename these points to be consecutive for convenience.

To attack the problem of checking attack success and updating the wall, we will need some variant of an interval tree. We will present two interval-tree based approaches below.

An interval tree is a tree, in which each node represents an interval [**m** x $2^{\mathbf{k}}$, **m** x $2^{\mathbf{k}}$] for some **m,k**. The parent of a node containing an interval **I** will be the node representing a twice longer interval containing **I** (so, if **I** is [**m** x $2^{\mathbf{k}}$, **m** x $2^{\mathbf{k}}$], the parent is [(**m** / 2) x $2^{\mathbf{k+1}}$, (**m**/2 + 1) x $2^{\mathbf{k+1}}$]). This is the common pattern for interval trees, the trick is in what to store in nodes.

High and low

In the first approach, we will try to answer the questions directly by the means of using a modified interval tree. We will store two values in each node — hi and lo. The "hi" value will be pretty standard, and will be defined so that the height of the wall at any given point is the maximum "hi" value of all the intervals containing this point. This can be updated in logarithmic time when any interval of the wall is attacked - we can split any interval into a logarithmic number of intervals represented by nodes, and update the hi value in each of them. This will allow us to update the wall height, and to figure out what the height of the wall at a given point is, each in logarithmic time. We still need a way to figure out whether an attack will succeed in logarithmic time, though.

We will use the "lo" values for that. For a given node **X** and a path to a leaf from **X** we can define the maximum "hi" value on this path as the "partial height" of the leaf node. This is what would be the height, if we disregarded all the nodes above **X** (in particular, "partial heights" measured from the root node are simply wall heights). We now define the "lo" value of **X** as the smallest partial height of a descendant of **X**. We need to see how this is useful, and how to update it in logarithmic time when updating the "hi" values.

Note that if we have a "lo" value for a node calculated, we can easily figure out the height of the lowest wall point in this interval - it's the maximum of the "lo" value of this node and the "hi" values of all the ancestors of this node. Thus, to figure out whether an attack will succeed on a general interval we split it into a logarithmic number of intervals represented by nodes, and figure out the lowest wall segment in each of these sub-intervals. If any of these is lower than the strength of the attack, it will succeed. This is logarithmic-squared as described, but it's easy to implement it to actually be logarithmic.

Now note that the "lo" values have a simple recursive definition - take the minimum of the "lo" values of the children, or the "hi" value of the node itself, whichever is higher. This means that when updating the "hi" value for a node, we only need to update the "lo" values for this node and its ancestors - meaning we can update "lo" values in logarithmic-squared time for each attack (and, again, it's simple to update them in logarithmic time).

Order by strength

Another approach that allows us to solve this problem with an interval tree is to order the attacks by strength, descending, and not by chronology. In this approach, for each point we know what was the earliest time at which it was attacked. Note that since we process from the strongest attack, any section of the wall that was attacked earlier by an attack we already processed is immune to attacks that come later and are processed later. Thus, to learn whether the attack is successful we need to find what's the latest attack time in the whole interval this attack covers; and subsequently we need to update the attack times to the minimum of the time that was stored so far, and the time of the currently processed attack.

This is called a min-max interval tree (we update with the minimum, and we query for the maximum). We encourage you to figure out what to store in the nodes to make this work in logarithmic time!