IZBORI

The first subtask (finding the largest number of parliament seats for a party) is solved by simulation. For each party, we assume it receives all outstanding votes and then use the D'Hondt method to calculate the number of seats won. The time complexity is $O(S^2 \cdot M)$.

The second subtask is more difficult. The solution is based on dynamic programming and binary search.

For each party we are asked to calculate the smallest number of seats it can win, assuming the least favourable distribution of remaining votes. Parties that initially have less than 5% of votes can win zero parliament seats so it is not necessary to perform further calculations for them.

If the remaining votes can be distributed so that a party wins X seats, then they can also be distributed so that the party gets X+1 votes (up to the maximum calculated in the first subtask). We can use binary search to find the smallest such X, if we can answer the question "Is it possible to distribute the outstanding votes to other parties so that this party wins X or fewer seats?"

That question can be answered by a dynamic programming algorithm. The function f(P, S) is the least number of votes we need to add to parties up to and including P so that all of them would win S seats in the parliament. In the end, all parties involved in the allocation of seats must have received at least 5% of all votes (meaning that at most 20 of them will be considered by the algorithm). The complexity of this DP check is about $20 \cdot M^2$ steps. There will be at most log(M) checks for at most 20 parties, so the overall complexity is about $20 \cdot \log(M) \cdot 20 \cdot M^2$.

OTOCI

Suppose that all bridges have already been built and that islands and bridges form not just a tree, but a chain.

Now the number of penguins on islands can be kept in an array of integers. If we build a Fenwick tree or interval tree over this array of integers, we can efficiently change the number of penguins on an island and calculate the sum of numbers in an interval. The commands 'penguins' and 'excursion' in this simpler variant can be processed in O(log N) time. The command 'bridge' does not appear in this variant so the overall complexity is O(Q log N).

Now let the graph be a tree. In order to efficiently process 'penguins' and 'excursion' commands, we need to decompose the tree into a set of chains. There are many ways to do that, one of which (called heavy-light decomposition) is:

Choose an arbitrary node as the root of the tree. Now calculate for each node the number of nodes in its subtree. From each node with children, the child node with the largest degree is chosen and the edge connecting the parent and child is dubbed heavy. These heavy edges form chains. All other edges are called light.

This decomposition can be found in O(N) time. An important property is that the number of heavy chains on the path from the root to any node is at most $\log_2 N$. This is true because, any time we step off a heavy chain (move down a light edge), we at least halve the number of nodes in the current subtree.

Let depth(A) be the depth of node A in the tree.

Let dad(A) be the parent of node A.

Let chain(A) be the chain node A is part of.

Let first(L) be the topmost node on chain L.

The following algorithm calculates the number of penguins between nodes A and B:

```
output = 0
while chain(A) ≠ chain(B):
        if depth(first(chain(A))) < depth(first(chain(B))):
                swap A and B
        output = output + chain(A).count(first(chain(A)), A)
        A = dad(first(chain(A)))
output = output + chain(A).count(A, B)
```

In other words, while A and B are not on the same chain, take the node on the deeper chain – call that node A and calculate the number of penguins on that chain (from the top to node A). Then move A up to the first node not on that chain. Finally, when A and B are on the same chain, count the penguins between them.

By decomposing the tree and building a data structure over the vertices on each chain (as in the previously described simpler variant of the problem), the complexity of the 'penguins' command remains O(log N), while the complexity of the 'excursion' command becomes $O(\log^2 N)$. The overall complexity is $O(Q \log^2 N)$.

Now reintroduce the 'bridge' command. For each component we need to maintain some sort of decomposition into chains.

Let A and B be the nodes we need to connect. Choose the smaller of the two components (suppose node A is in that component), make the component rooted in A and find the heavy-light decomposition of the tree. Now connect node A as the child of node B. This way in each component we still have some sort of decomposition. Unfortunately, this decomposition is not necessarily heavy-light because the sizes of subtrees in component B change, but the decomposition of the component does not change.

To fix this, we can rebuild the decomposition of a component when it becomes too unbalanced. One way to do this is to make the algorithm introspective – it can keep track of the number of steps needed to process 'penguins' commands. When this number goes over $\sqrt{N}$, we find the decomposition again. The overall complexity of the algorithm is $O(Q\sqrt{N}\log N)$.

PLAHTE

Consider one of the sheets and how the area of stained fabric in it changes over time. Analysing various cases we can see that, if X squares of fabric are initially stained when the oil front hits, then X+K squares will be stained after a second, X+2K after two seconds etc. where K can be zero, two or four. This continues until the entire width or height of the rectangle is stained, after which one additional row or column gets stained per second (we can also recalculate X at that point and change K to zero).

One efficient solution calculates the important time points (events), in which X and K for a sheet change. The time complexity of this part is O(N). Then we simulate second by second and, using the known data about X and K for every rectangle, we can calculate the change of area of stained fabric in O(1) per second.

An alternative approach is to first fold all rectangles into the first quadrant. Then we can represent every rectangle as a combination of four quarter-planes extending up and right (inclusion-exclusion principle) and, similar as before, calculate the change of area of stained fabric in each quarter-plane. Representing rectangles as quarter-planes reduces the number of special cases our implementation needs to handle.