Analysis

The first subtask is only for 10 points. It is for the contestants that can't solve anything. We do brute force for every possibility for distribution of the players between the teams for each query. The complexity is $O(\boldsymbol{Q} * 2^{\boldsymbol{N}} * (\boldsymbol{N} + \boldsymbol{M}))$.

The second subtask is for 35 points. Here we begin the essential part of the task. We have to find a way for finding the value of the optimal distribution fast. It is clear that we have a connected undirected graph with vertices being the players and edges – the values of the friendships. Firstly, we will make easier what we try to find. Let we define the two-dimensional array $val[2][\boldsymbol{N}]$ – how much does every player contribute for the teams and with the matrix $a[\boldsymbol{N}][\boldsymbol{N}]$ – the values of the friendships. Let with $teams[\boldsymbol{N}]$ we define the distribution of the players between the teams. We want to maximize this: $\sum_{i=0}^{N-1} val[teams[i]][i] - \sum_{i,j=0}^{N-1} a[i][j],\ \ if\ teams[i] \neq teams[j]$. We want to make this expression more convenient, so we will subtract it from the constant sum $\sum_{i=0}^{N-1}(val[0][i] + val[1][i])$ and we will get that we have to minimize $\sum_{i=0}^{N-1} val[1 - teams[i]][i] + \sum_{i,j=0}^{N-1} a[i][j],\ \ if\ teams[i] \neq teams[j]$. The new sum is more convenient for minimizing because we don't have subtractions. Let we change the initial graph a little bit so it is more useful for finding the optimum. We add two new vertices for the two teams. The vertex for the team of the good guys is connected with every player and the weights of the edges are the contributions of every player for the team of the good guys. Analogously, we connect the vertex for the team of the bad guys with every player and the weights are the contributions for the team of the bad guys. Now in the new graph we have to find this: the distribution of the vertices into two connected components (in the first component is the vertex for the team of the good guys and in the second – the vertex for the team of the bad guys) in such a way that the sum of the edges, which connect the two components, is minimal. This is called $minimum - cut$ with fixed vertex for each component. Let we first think why the minimum cut gives us the optimal distribution. The vertices that are for the two teams will be in different components. So for every vertex (player) we will have only one of the edges, connecting it to the special vertices, in the cut. For the vertices (players) which are in different connected components (which correspond to different teams), the edges, connecting them, are in the cut. So we have that the cut corresponds to a sum, which we wanted to minimize. This task is more famous. There is a theorem which states that finding the minimal cut with fixed vertex for every component is equivalent to finding the maximum flow in the graph with source and sink – these vertices. This means that we just have to construct a flow network from the new graph and find the maximum flow in it. After that we have to subtract the flow from $\sum_{i=0}^{N-1}(val[0][i] + val[1][i])$. Here the complexity is the complexity of the maximum flow – $O(\boldsymbol{N}^2\boldsymbol{M})$ (for Dinic's algorithm), but in reality the algorithm runs much faster because the graphs aren't complete and the edges don't have big weights.

The third subtask is only for 10 points. It is for the contestants that can't solve the final big part of the task. Now we have queries that change the set of the players. Here we have to note that we only have queries for removing players and the queries from type 3 are very little in count. The queries "compatible" with the maximum flow are the ones for adding edges, vertices etc. But the task is offline for the queries, so we can save them. Then we will traverse them in reversed order. Now the removing of vertices will be adding vertices. Again the complexity is for the maximum flow as in the previous subtask.

The last subtask is for 45 points. Here we have to solve the whole task when we have queries for getting back all of the initial players. The naïve approach will be to remove the player for every query and to run the maximum flow in the new network from the start (from 0 flow) but it isn't very fast. Again, we will use that the queries are offline and that the adding of vertices is "compatible" operation with the maximum flow. Firstly, we read the queries and save them with additional information for the time of the query (the index). For each query

of the third or fourth type we add approximately $\boldsymbol{N}$ queries for adding or removing some of the players (the times are the same!). After that we sort the queries first by the numbers of the players and for the same players – we sort by the time. In such a way, we can have intervals of time in which every player is in the game. Let we represent this information as queries with intervals. We can have a segment tree (with leaves – the moments of time) and in its' vertices we can store these queries. When we go to the leaves, we will have added the needed vertices for that moment of time. But as we said earlier, the maximum flow isn't "compatible" with removing vertices and when we go back (up in the tree) we will have to do that. For this, we will use a stack in which we will store the changes. The changes, which we save, are the edges with changed flow (this is the only thing, which changes, as the algorithm progresses). One optimization, which we can use, is to add one edge at most once for one execution of the maximum flow, because we only need the first time in order to know what the old value of the flow was. Furthermore, every element in the stack will contain information for the index of the vertex in the segment tree when the maximum flow was executed. In this way, when we leave the vertex in the segment tree, we will know exactly what changes were made. For every vertex in the segment tree we will execute the flow only 1 time – after we have added all players for the vertex. One little detail for the correct work of the algorithm is to add one fictive query of type 1 with time 0 at the beginning (this way we add all players at the beginning). We can further optimize this algorithm. The part that we will optimize is the addition of the new vertices. We don't need to add the edges every time (and then to remove them). It is enough to include all vertices and edges in the beginning but to save which vertices are part of the current graph. The modification of the algorithm for maximum flow is only that we can't visit vertices which aren't in the current graph. The complexity here is harder for calculating because it heavily depends on the tests and the complexity of the flow but it is better that the naïve approach because we don't run empty flow every time and we don't do unneeded changes of the edges for every run. Sample complexity is: $O(\boldsymbol{QN} + \boldsymbol{QN^2M})$ but as we said the second addend is a very rough estimation for the time for the flow in general, and more rough in our case.

Author: Iliyan Yordanov