CodeForces Educational Round 178 G

omeganot

July 22, 2025

First we try to solve the general problem without modulo and without updates. Our answer is the product of the answers for each connected component. For any arbitrary connected component, nodes that are in the same strongly connected component must be the same, and otherwise they may be different. In a connected component of a functional graph, there is exactly one cycle. Thus, the answer for any arbitrary connected component is k^{v+1}, where v is the number of nodes not in the cycle.

Now we may consider adding the modulo. If $k \equiv 0 \pmod 3$ or $k \equiv 1 \pmod 3$, then $k^{v+1} \equiv k \pmod 3$. Thus, we only need to worry about the case where $k \equiv 2 \pmod 3$.

It seems like we need to know the number of connected components as well as how many nodes are in a cycle. However, if we recall we only need our answer mod3, we can abuse the fact that the answer is either 1 or 2. In fact, our final answer takes the form of k^x, and if $x \equiv 0 \bmod 2$, our answer is 1. Otherwise, our answer is 2. So really, we only need to count the number of connected components where $v + 1$ is odd (if $v + 1$ is even, then that connected component's answer is 1).

This still seems difficult, but we can notice that, though we may not know how many nodes are in the cycle of a connected component, we can know the parity, by checking whether that component is bipartite or not. If we know both the number if nodes in a connected component and whether it's bipartite or not, we know the parity of $v + 1$ since $v + c = n$, where c is the number of nodes in the cycle. Thus, $v \equiv n - c \pmod 2$, and $c \bmod 2$ is 1 if the component is not bipartite, and 0 otherwise.

Thus, for each query, we must keep track of the sizes of the connected components as well as whether or not it's bipartite. Then, as the graph changes, we manually keep track of the parity of x in our answer k^x. Since we will be adding and deleting edges to a graph, we are motivated to use a DSU with offline deletion. Inside the DSU structure we can incorporate a boolean variable for each connected component determining whether it's bipartite or not. We must then update the number of components with odd $v + 1$. These operations are easy to rollback.

When adding an edge between two nodes in the same set, we only need to change the bipartite status of the component. To check if the edge connects two nodes of the same color, for each node, we store whether its parent in the DSU is a different color than it or not. Then, in $O(\log n)$ time we can find the color by just recursively calculating the parent's color. When adding an edge between a and b, we will add an edge between the root of set a to the root of set b. Then, we can check if the root of set a and the root of set b are the same color, does our find color function correctly determine that a and b are different colors? If not, we say that the root of set b and the root of set a must have differnet colors and update accordingly.

Since both rollbacks and our find color function require direct links to parents and no amortization, our final algorithm is $O((n + q) \log q \log n)$.