Kick Start 2021 - Round H

Analysis: Dependent Events

For this problem, we are given a graphical model in the form of a directed, rooted tree, where each vertex is an event. For any vertex v on this tree, let $v = 0$ and $v = 1$ denote the non-occurrence and occurrence of event v, respectively. Let l_j be the lowest common ancestor (LCA) of $\mathbf{u_j}$ and $\mathbf{v_j}$. By the total probability rule, $P[\mathbf{u_j} = 1, \mathbf{v_j} = 1] = \sum_{i=0}^{1} P[\mathbf{u_j} = 1, \mathbf{v_j} = 1, l_j = i]$. We know $\mathbf{u_j}$ and $\mathbf{v_j}$ are conditionally independent given l_j, because the paths $l_j \to \mathbf{u_j}$ and $l_j \to \mathbf{v_j}$ are edge-disjoint due to the definition of LCA. This allows us to simplify the earlier sum into $\sum_{i=0}^{1} P[\mathbf{u_j} = 1 | l_j = i] P[\mathbf{v_j} = 1 | l_j = i] P[l_j = i]$. This formula forms the basis of our solution, and the approaches for the two test sets differ only in how to compute the required probabilities efficiently.

To deal with the output format of this problem, we store all results as fractions and take the numerator and denominator modulo $10^9 + 7$ after each operation to avoid overflow. If our final result is $\frac{p}{q}$, we output $pq^{-1} \pmod{10^9 + 7}$. Due to the nature of the problem, q will always be a power of ten, so the inverse is guaranteed to exist, and can be efficiently computed via extended Euclidean algorithm. Alternatively, due to Fermat's little theorem, we can equivalently raise q to the $(10^9 + 5)$-th power to find its inverse; this can be done efficiently with exponentiation by squaring.

Test set 1

For this test set it suffices to naively compute l_j. We can then compute $P[\mathbf{u_j} = 1 | l_j]$ and $P[\mathbf{v_j} = 1 | l_j]$ by walking back down the tree from l_j. To do this, we can use the total probability rule similar to before; if $p(x)$ denotes the parent of x, then $P[x = 1 | l_j = i] = \sum_{k=0}^{1} P[x = 1, p(x) = k | l_j = i] = \sum_{k=0}^{1} P[x = 1 | p(x) = k] P[p(x) = k | l_j = i]$; note that the first term in this product comes from the input, and the second term comes from the previous step of our walk. We can use the same formula to compute $P[l_j = i]$, since $P[l_j = i] = \sum_{k=0}^{1} P[l_j = i | r = k] P[r = k]$, where r is the root of the tree.

Each vertex is visited at most $O(1)$ times per query using this technique, so our overall time complexity is $O(\mathbf{NQ})$ per test case, which is sufficient.

Test set 2

There are both more queries and a larger tree in this test set, so the naive solution is too slow. This test set requires computing l_j, the LCA of $\mathbf{u_j}$ and $\mathbf{v_j}$, in $O(\log \mathbf{N})$ time anyways, so we can think of ways to augment an LCA algorithm to also keep track of the necessary probabilities for us. For example, the binary lifting LCA algorithm pre-computes $p^i(v)$ for all vertices v, where $p^i(\cdot)$ gives the 2^i-th parent of a given vertex. We can extend the binary lifting algorithm to not only store the id of this parent, but to also store $P[v = 1 | p^i(v)]$. These probabilities can then be multiplied such that only $O(\log \mathbf{N})$ hops are needed to get from $\mathbf{u_j}$ or $\mathbf{v_j}$ up to l_j. The unconditional probability $P[l_j]$ can simply be pre-computed for the entire tree via DFS in $O(\mathbf{N})$ time. This allows us to answer each query in $O(\log \mathbf{N})$ time, so our overall time complexity is $O(\mathbf{N} \log \mathbf{N} + \mathbf{Q} \log \mathbf{N})$ per test case.