Code Jam 2021 - World Finals

Analysis: Infinitree

View problem and solution walkthrough video

The first thing to do to solve this problem is seeing it as a graph theoretical problem. We can consider the directed graph G where each node represents a color and there is an edge $c_1 \to c_2$ if $\mathbf{L_{c_1}} = c_2$ or $\mathbf{R_{c_1}} = c_2$ (if $\mathbf{L_{c_1}} = \mathbf{R_{c_1}} = c_2$, then there are two edges $c_1 \to c_2$). Nodes at level K in the tree correspond to the last node of paths in G of length K starting at the black node (the full paths correspond to the branches that go into those nodes). Therefore, if M is the adjacency matrix of G, ${M^K}_{i,j}$ is the number of descendants of color j that are K levels below a given node of color i. In particular, $\sum_j {M^K}_{\text{black},j}$ is the number of nodes at level K, and we can sum that over all $K \leq K'$ to get the number of nodes at levels up to K'.

Matrix powers

We can calculate M^K with only $O(\log K)$ matrix multiplications. Moreover, we need only values $K \leq \mathbf{B}$, and since we will do this for the same M and lots of different values of K, we can memoize M^{2^p} for each integer $p \leq \log_2 \mathbf{B}$) by starting with $M^1 = M^{2^0}$ and then using $M^{2^{p+1}} = (M^{2^p})^2$ to calculate each subsequent power. This requires $O(\log \mathbf{B})$ matrix multiplications to initialize, or $O(\mathbf{N}^3 \log \mathbf{B})$ time. Since K can be expressed uniquely as a sum of powers of 2, we can calculate M^K as the product of those powers. While this still requires the same worst-case time, it is much faster in practice. More importantly, if we want to calculate VM^K for a vector V we can do that by multiplying V by each matrix, requiring only $O(\log \mathbf{B})$ vector-matrix multiplications, which is $O(\mathbf{N}^2 \log \mathbf{B})$ time.

Summation of powers of matrices

Let x be a tree node of color i. We can calculate the number of descendants of x up to level K' as $\sum_{K \leq K'} \sum_j {M^K}_{i,j} = \sum_j (\sum_{K \leq K'} M^K)_{i,j}$. We can calculate $\sum_{K \leq K'} M^K$ efficiently with a divide and conquer approach, using $M^0 + M^1 + M^2 + \cdots + M^{2K-1} = (I + M^K)(M^0 + M^1 + \cdots + M^{K-1})$, where I is the identity matrix. This would require $O(\log K')$ matrix powers, or $O(\log^2 K')$ matrix multiplications. By overlapping the divide and conquer needed to calculate the matrix powers and the summation, we can get that down to $O(\log K')$ overall.

Notice that the values in the matrices can be really big. However, any value above $\mathbf{B}$ is equivalent for us, so we can do all math by capping the results at anything larger than $\mathbf{B}$. This allows to implement this with regular 64 bit integers and without adding a large-integer arithmetic factor to the overall time complexity.

Test Set 1

With the setup done in the previous paragraphs, solving Test Set 1 is straightforward. Since node $\mathbf{A}$ is always the root in this case, the answer is the level of node $\mathbf{B}$. We can get $\mathbf{B}$'s level by using bisection on the function "number of nodes up to a certain level". We saw that we can implement that function efficiently above with $O(\log \mathbf{B}')$ matrix multiplications, making this algorithm run in $O(\mathbf{N}^3 \log^2 \mathbf{B})$. Since the divide and conquer of the bisection and the matrix

multiplication can be overlapped, this can be reduced to $O(\mathbf{N}^3 \log \mathbf{B})$ overall, which allows for slower implementations/languages.

Test Set 2

The solution for Test Set 2 uses the same framing and graph theory as the solution for Test Set 1, but it requires a lot of additional work.

First, we will define a new naming convention for nodes in the tree. We will uniquely identify a node in the tree with a pair of numbers (h, x). The pair (h, x) represents the node at level h that has exactly X other nodes at level h to its left. This is similar to a coordinate system. We can convert from a node index to one of these pairs as follows. First, we can obtain the level of a node given its index as we did in the Test Set 1 solution. Then, if the node index is D and the level is h, we can find X as the difference between $D - 1$ and the number of nodes up to level $h - 1$, which we saw how to calculate as well. Now, we turn our attention to solving the problem for two nodes identified as pairs.

A slow solution

We will work our way down the tree, while always maintaining both of the input nodes $\mathbf{A}$ and $\mathbf{B}$ inside the current subtree. As state we will keep the color of the root of the current subtree, and the pairs representing $\mathbf{A}$ and $\mathbf{B}$ relative to the current subtree. Initially, the color of the current root is black, and the pairs that represent $\mathbf{A}$ and $\mathbf{B}$ are calculated as mentioned in the previous paragraph.

Given the color of a current root C and a pair (h, x) representing a non-root node, we can check whether (h, x) is in the left or right subtree as follows: The number of nodes at level h in the left subtree is exactly the number of descendants of L_C at level $h - 1$. Since this is simply $e_{L_C} M^{h-1}$, where e_i is the vector that has a 1 in position i and 0 in all other positions, we can calculate it in $O(\mathbf{N}^2 \log \mathbf{B})$ time, as we saw in the opening paragraphs. Therefore, we can simply compare that number with x to make our decision.

The observation above leads to an algorithm: Given the two target nodes as pairs (h_1, x_1) and (h_2, x_2) and current root color c, if $\min(h_1, h_2) = 0$, then the answer is $h_1 + h_2$. Otherwise, we check in which subtree each of the nodes is. If both are in different subtrees, the answer is also $h_1 + h_2$. If not, we move into the subtree. If we move into the left subtree, the root switches to L_c and the node represented by the pair (h, x) is now represented by the pair $(h - 1, x)$. If we move into the right subtree, the new root color is R_c and the node represented by the pair (h, x) is now represented by the pair $(h - 1, x - t)$ where t is the number of descendants at level $h - 1$ of L_c (the same amount we needed to calculate to decide on which subtree (h, x) belonged).

This algorithm would require $min(h_1, h_2)$ steps, and can be too slow if that is a large amount, which can happen.

Speeding up the solution

Instead of speeding up all cases, we focus only on the ones that definitely need the speed up. If the graph implied by $\mathbf{L}$ and $\mathbf{R}$ has any reachable color belonging to more than one cycle, then the total number of nodes of the tree grows exponentially. In this case, the h values in the pair representation of the input nodes are necessarily small (logarithmic on the node indices) and the algorithm above just works. So, we need a faster algorithm only for the case in which every reachable color belongs to at most one cycle.

In the case in which the current root is a color that does not belong to a cycle, we proceed as in the slow algorithm and move one step down. After this, the color left behind will not be a root color again. If the root color c_1 belongs to a cycle $c_1, c_2, \ldots, c_h$, then we want to do multiple steps at once. Consider the branch of the tree that is obtained by going through the cycle p times, with $p \times h < \min(h_1, h_2)$. The branch splits the tree in 3 parts: descendants of the last node of the branch, nodes that are to the left of that subtree, and nodes that are to the right of that subtree. Our strategy is to find out where in that partition are our target nodes. If they are both in the middle (right below the branch), we go into it. Otherwise, we cannot do that many passes without leaving one of the target nodes outside of our current subtree. We can try decreasing powers of 2 as values for p, which requires each value to be tried only once. After we try $p = 1$ (the smallest integer power of 2), we fall back to moving one step at a time as in the slow solution until we reach a root color outside of the cycle (this requires at most h single move steps).

To do the fast moves, we calculate the vector representing the number of nodes of each color that lie exactly $p \times h$ levels below our current root on the left side. We can calculate it for $p = 1$ as $\sum_j e_{c_1} M^{h-j+1}$ where the summation is only over the values j for which the step $c_j \to c_{j+1}$ (or $c_h \to c_1$ if $j = h$) is to the right. For larger values of p, we take the value for $p = 1$ and multiply it by $M^0 + M^h + \cdots + M^{(p-1)h}$. This summation can be calculated similarly to the summation over consecutive powers, and it has to be done only once per value of h, of which there are $O(\sqrt{\mathbf{N}})$ different ones. We can calculate the right side analogously.

From the vectors at level $p \times h$ we can find out how many nodes at level h_i are on each side by multiplying by $M^{h_i - p \times h}$. With those values, we can decide similarly to how we did in the slow solution whether both nodes are in the middle subtree. If we were to move into the middle subtree, we simply subtract $p \times h$ from both h_i values and the total number of nodes from the level that were left behind on the left side from the x_i values, as we did in the slow solution.

With the details above, this leads to an algorithm that takes $O(\mathbf{N}^{3.5} \log \mathbf{N} + \mathbf{N}^2 \log \mathbf{B})$ time to run. If less care is taken with how matrix powers and summations of matrix powers are calculated, larger complexities may arise. The time limit is purposefully not tight, so algorithms with larger complexities that are logarithmic in $\mathbf{B}$ can also get the problem correct.