2509. Cycle Length Queries in a Tree

Hard | Tree | Binary Tree | Leetcode Link

Problem Description

You are given a virtual complete binary tree that contains `2^n - 1` nodes, where `n` is an integer. Each node in this tree is assigned a unique value, and the root node starts with value `1`. For any node in the tree that has a value `val`, you can always find its children by following these rules:

- The left child will have the value `2 * val`.
- The right child will have the value `2 * val + 1`.

In addition to the binary tree, you are presented with a list of queries. Each query is a pair of integers `[a_i, b_i]` representing nodes in the tree. For each query, you are expected to:

1. Add an edge directly connecting the nodes labeled `a_i` and `b_i`.
2. Determine the length of the cycle that includes this new edge.
3. Remove the edge you added between `a_i` and `b_i`.

Note that a *cycle* in a graph is defined as a closed path where the starting and ending nodes are identical and no edge is traversed more than once. The length of this cycle is considered the total number of edges within that cycle.

Your task is to process all the queries and return an array of integers, each representing the length of the cycle that would be formed by adding the edge specified in the corresponding query.

Intuition

Since we are working with a complete binary tree, we can take advantage of its properties to calculate the distance between two nodes. The properties of a binary tree let us know that any node `val` in the tree has its descendants determined by multiplication by 2 for the left child or multiplication by 2 and adding 1 for the right child.

The intuition behind the solution for determining the cycle length for a pair of nodes `(a, b)` is based on finding their lowest common ancestor (LCA). The lowest common ancestor of two nodes in a binary tree is the deepest (or highest-level) node that has both `a` and `b` as descendants (where we allow a node to be a descendant of itself).

Since we are adding an edge between nodes `a` and `b`, the cycle length would be equal to the distance from `a` to the LCA plus the distance from `b` to the LCA plus 1 (the added edge itself).

The algorithm starts by setting a counter `t` to 1, representing the added edge. Next, we iterate while `a` does not equal `b` - effectively, we are moving `a` and `b` up the tree towards the root until they meet, which would be at their LCA:

- If `a > b`, we move `a` up to its parent, which is calculated by `a >>= 1` (this is a bitwise right shift which is equivalent to dividing by 2).
- If `b > a`, we do the same for `b`.
- After each move, we increment the counter `t`.

Once `a` equals `b`, we have found the LCA and thus the cycle. The length of the cycle is the value stored in `t` at this point. We append the value of `t` to our `ans` list, which will eventually contain the answer for each query.

Solution Approach

The solution provided is straightforward and leverages the binary nature of the tree to efficiently find the lengths of cycles for each query. The data structure used is simply the tree modeled by the rules of the complete binary tree, and no additional complex data structures are required. The algorithm utilizes a while loop and bitwise operations to navigate the tree. Let's break down the steps and the patterns used:

1. **Initialization**: The solution initializes an empty list `ans` which will store the lengths of cycles for each query.
2. **Processing Queries**: The solution iterates over each query `(a, b)` within the `queries` list.
3. **Moving Up the Tree**: Within a while loop, where the condition is `a != b`, the algorithm compares the values of `a` and `b`.
   - If `a > b`, it means that `a` is further down the tree. To move `a` up the tree (towards the root), the operation `a >>= 1` is applied, which is a right bitwise shift equivalent to integer division by 2 — this moves `a` to its parent node.
   - If `b > a`, the same process is applied to `b` using a right bitwise shift.
4. **Counting Edges**: Each time either `a` or `b` moves up the tree towards their lowest common ancestor (LCA), the counter `t` is incremented, since this represents traversing one edge towards the LCA for each node.
5. **Calculating Cycle Length**: Once `a` and `b` are equal—meaning the LCA has been reached—the value in `t` now represents the total number of edges traversed, plus 1 for the edge that was added between `a` and `b` initially.
6. **Recording Results**: The current length of the cycle is appended to the `ans` list.
7. **Returning Results**: After all queries have been processed, the `ans` list is returned, which contains the calculated cycle length for each query.

By using bitwise shifts and a simple counter, the algorithm avoids any complicated traversal or search methods. This takes advantage of the inherent properties of a binary tree and results in an efficient O(log n) time complexity for finding the LCA and cycle length per query, where `n` is the total number of nodes in the tree. Hence, the overall time complexity of the solution depends on the number of queries `m` and is O(m log n).

Example Walkthrough

Let's consider a small example with a binary tree of `2^3 - 1 = 7` nodes and a single query `[4, 5]`. The nodes in this binary tree would be numbered from `1` to `7` with the following structure:

```
      1
    /   \
   2     3
  / \   / \
 4   5 6   7
```

Here's the step-by-step breakdown using our solution approach for the query `[4, 5]`:

1. **Initialization**: Start with an empty list `ans` to store the lengths of cycles for the queries.
2. **Processing the Query `[4, 5]`**: We consider the pair `(a, b)` where `a = 4` and `b = 5`.
3. **Moving Up the Tree**:
   - Initially, `a = 4` and `b = 5`. Since `a < b`, we move `b` up the tree. Applying `b >>= 1` changes `b` to `5 >> 1`, which equals `2`. We increase our counter `t` by `1`.
   - Now `a = 4` and `b = 2`. Since `a > b`, we move `a` up the tree. Applying `a >>= 1` changes `a` to `4 >> 1`, which equals `2`. We increase our counter `t` by `1`.
4. **Counting Edges**:
   - At this point, after moving both `a` and `b` up the tree by one step each, our counter `t` is `3` because we started with `1` for the added edge, plus one increment for each of `a` and `b`.
5. **Calculating Cycle Length**:
   - Now, since `a` equals `b` (`a = 2` and `b = 2`), we have found the lowest common ancestor (LCA), which is node `2`.
6. **Recording Results**:
   - We append the current counter `t` value, which is `3`, to our `ans` list. It signifies the cycle length for the query `[4, 5]` because it includes one step from `4` to `2`, one step from `5` to `2`, and one for the direct edge we added between `4` and `5`.
7. **Returning Results**:
   - After processing the query, the `ans` list contains `[3]`, which represents the cycle length created by adding an edge between the nodes `4` and `5`.

So for the query `[4, 5]`, the algorithm would output `[3]`, indicating that the cycle formed by adding an edge between node `4` and `5` has a length of three edges.

Python Solution

```python
from typing import List

class Solution:
    def cycleLengthQueries(self, n: int, queries: List[List[int]]) -> List[int]:
        # Initialize an empty list to store the answer for each query
        results = []

        # Iterate through each query in the list of queries
        for query in queries:
            # Extract the starting and ending nodes from the query
            start_node, end_node = query

            # Initialize the cycle length variable to 1 (since the start and end nodes are counted as a step)
            cycle_length = 1

            # Continue looping until the start node and end node are the same
            while start_node != end_node:
                # If the start node has a greater value, divide it by 2 (essentially moving up one level in the binary tree)
                if start_node > end_node:
                    start_node >>= 1
                # Otherwise, if the end node has a greater value, divide it by 2
                else:
                    end_node >>= 1

                # Increment the cycle length with each step taken
                cycle_length += 1

            # Append the calculated cycle length to the results list
            results.append(cycle_length)

        # Return the list of results after all queries have been processed
        return results
```

Java Solution

```java
class Solution {
    public int[] cycleLengthQueries(int n, int[][] queries) {
        // m represents the length of the queries array
        int m = queries.length;

        // Initialize the array to store answers for the queries
        int[] answers = new int[m];

        // Loop through every query
        for (int i = 0; i < m; ++i) {
            // Extract the start and end points (a and b) from the current query
            int start = queries[i][0], end = queries[i][1];

            // Initialize the cycle length as 1 for the current query
            int cycleLength = 1;

            // While the start point is not equal to the end point
            while (start != end) {
                // If start is greater than end, right shift start (equivalent to dividing by 2)
                if (start > end) {
                    start >>= 1;
                } else {
                    // If start is not greater than end, right shift end
                    end >>= 1;
                }
                // Increase the cycle length counter since we've made a move
                ++cycleLength;
            }

            // Store the computed cycle length in the answers array
            answers[i] = cycleLength;
        }

        // Return the array containing answers for all queries
        return answers;
    }
}
```

C++ Solution

```cpp
#include <vector>

class Solution {
public:
    // Function to find the cycle lengths in a binary tree for given queries
    // @param n : an integer representing the number of nodes in a full binary tree
    // @param queries : a vector of queries where each query is a vector of two integers
    // @return : a vector of integers representing the cycle lengths for each query
    vector<int> cycleLengthQueries(int n, vector<vector<int>>& queries) {
        // Initialize a vector to store the answers for each query
        vector<int> answers;

        // Iterate over each query in the list of queries
        for (auto& query : queries) {
            // Extract the two nodes being queried from the current query
            int nodeA = query[0], nodeB = query[1];

            // Initialize a variable to keep track of the number of steps in the cycle
            int steps = 1; // Starting from 1 because the first node is also counted as a step

            // Keep adjusting the nodes until they are equal, which means they have met at their common ancestor
            while (nodeA != nodeB) {
                // If nodeA is greater than nodeB, move it one level up towards the root
                if (nodeA > nodeB) {
                    nodeA >>= 1; // Equivalent to nodeA = nodeA / 2;
                } else {
                    // Otherwise, move nodeB one level up towards the root
                    nodeB >>= 1; // Equivalent to nodeB = nodeB / 2;
                }

                // Increase the step count with each move
                ++steps;
            }

            // Once the common ancestor is reached, add the step count to the answers list
            answers.emplace_back(steps);
        }

        // Return the answers to the queries
        return answers;
    }
};
```

Typescript Solution

```typescript
function cycleLengthQueries(numberOfNodes: number, queries: number[][]): number[] {
    // Initialize an array to store the answers for each query
    let answers: number[] = [];

    // Iterate over each query in the list of queries
    for (let query of queries) {
        // Extract the two nodes being queried from the current query
        let nodeA: number = query[0];
        let nodeB: number = query[1];

        // Initialize a variable to keep track of the number of steps in the cycle
        // Starting from 1 because the first node is also counted as a step
        let steps: number = 1;

        // Keep adjusting the nodes until they are equal, which means they have met at their common ancestor
        while (nodeA !== nodeB) {
            // If nodeA is greater than nodeB, move it one level up towards the root
            if (nodeA > nodeB) {
                nodeA >>= 1; // This is a bitwise operation equivalent to dividing nodeA by 2 and flooring the result
            } else {
                // Otherwise, move nodeB one level up towards the root
                nodeB >>= 1; // This is a bitwise operation equivalent to dividing nodeB by 2 and flooring the result
            }

            // Increase the step count with each move
            steps++;
        }

        // Once the common ancestor is reached, add the step count to the answers array
        answers.push(steps);
    }

    // Return the answers to the queries
    return answers;
}
```

Time and Space Complexity

Time Complexity

The given code executes a for loop over the `queries` list, which contains `q` pairs (a, b), where `q` is the number of queries. Within this loop, there's a while loop that runs until the values of `a` and `b` are the same. In the worst case, this while loop runs for the height of the binary tree which represents the number of divisions by 2 that can be done from the max value of `a` or `b` down to 1. Given that `a` and `b` are less than or equal to `n`, the height of such a tree would be `O(log(n))`.

Thus, for `q` queries, the while loop executes at most `O(log(n))` for each pair. Therefore, the time complexity for the entire function is `O(q * log(n))` where `q` is the length of the queries list and `n` is the maximum value of `a` or `b`.

Space Complexity

The space complexity of the code is relatively simpler to assess. It uses an array `ans` to store the results for each query. The size of this array is directly proportional to the number of queries `q` which is the length of the input `queries` list. No additional significant space is required; thus, the space complexity is `O(q)`.

No complex data structures or recursive calls that would require additional space are used in this solution.