549. Binary Tree Longest Consecutive Sequence II

Medium | Tree | Depth-First Search | Binary Tree | Leetcode Link

Problem Description

This problem provides us with the root of a binary tree and asks us to find the length of the longest consecutive path in that tree. The consecutive path we are looking for can be either increasing or decreasing in terms of node values, and every two consecutive nodes in this path should have values that differ by exactly one. To clarify, paths such as `[1,2,3,4]` (increasing) and `[4,3,2,1]` (decreasing) are valid. However, a path like `[1,2,4,3]` is invalid as the values do not differ by one. Additionally, it's important to note that the path does not need to follow parent-child relationships and could include a 'bounce', going from child to parent to another child (child-parent-child pattern).

Intuition

The intuition behind this solution lies in using a recursive depth-first search (DFS) algorithm to traverse the tree and compute the increasing and decreasing consecutive paths. For every node, there can be four kinds of consecutive paths:

1. Paths that only include this node.
2. Paths that start from this node and extend to any node in the left subtree.
3. Paths that start from this node and extend to any node in the right subtree.
4. Paths that go through this node, meaning they start from the left subtree, include this node, and go to the right subtree.

To track these paths, for each node, we calculate two values: `incr` and `decr`, which represent the lengths of the longest increasing and decreasing paths ending at this node, respectively.

When we visit a node, we check its children. If either child's value is one more than the current node's value, we increment `incr`; if it's one less, we increment `decr`. We take these two values from both children and use them to update the `incr` and `decr` values of the current node.

The magic happens by considering not only the deepest path from this node to each child but also the possibility of continuing the path by "bouncing" from one child to the other through the current node, which effectively can increase the overall length of the consecutive path.

As we compute the `incr` and `decr` values for each node, we keep track of the global maximum length (`ans`) found so far. To get the maximum consecutive path that can pass through a node, we add the `incr` and `decr` values of this node but subtract 1 because the node itself is counted twice (once in each path). After the DFS traverses the whole tree, `ans` will store the length of the longest consecutive path.

Solution Approach

The solution uses a recursive depth-first search (DFS) to explore the binary tree and calculate the length of the longest consecutive path. Here's a step-by-step walkthrough of the implementation:

1. **Recursive Function Definition**: A function `dfs` is defined, which takes a node of the tree as an argument and returns a tuple `[incr, decr]`. `incr` holds the maximum length of an increasing consecutive path ending at that node, and `decr` is the same for decreasing paths.
2. **Base Case**: When `dfs` encounters a `None` (indicating the node being visited is non-existent or a leaf's child), it returns `[0, 0]` as there are no consecutive paths beyond this point.
3. **State Variables**: The solution introduces a nonlocal variable `ans` to track the maximum length found during traversal.
4. **Child Nodes Analysis**: Each call to `dfs` considers the current node's left and right child nodes. For each child, the function computes `i1/d1` and `i2/d2` which are tuples returned by the recursive `dfs` call on the left and right children, respectively.
5. **Update Increments/Decrements**:
   - It then checks the value of the left child (if it exists), updating `incr` or `decr` based on whether the left child's value is one less or one more than the current node's value, respectively.
   - Similarly, checks are performed on the right child, updating `incr` and `decr` by comparing the values of the right child and the current node.
6. **Global Maximum Update**: After calculating the `incr` and `decr` for both the left and right children, `ans` is updated by taking the sum of the current node's `incr` and `decr` minus one — as the current node is counted in both `incr` and `decr` and should only be counted once.
7. **Return Values**: Finally, the function returns a tuple `[incr, decr]` for the current node, which signifies the maximum lengths of consecutive paths ending at this node (both increasing and decreasing).
8. **Result**: After `dfs` is called on the root, `ans` contains the length of the longest consecutive path in the tree, which is then returned as the result of the `longestConsecutive` function.

The algorithm effectively scans the entire tree only once, ensuring an efficient solution with a time complexity of O(n), where n is the number of nodes in the tree. By considering each node and its potential paths (both increasing and decreasing), as well as potential "bounces" between children, we ensure no possible paths are missed.

Example Walkthrough

Let's walk through an example to illustrate the solution approach. Consider the following binary tree:

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

Here's how the recursive DFS would process this tree:

1. **Starting at the Root (Node 3)**: The initial call to DFS is made on the root (node 3). At this point, `ans` is initialized to 0.
2. **Recursive Calls**: The `dfs` function will make recursive calls to the left (node 2) and right (node 4) children:
   - **Left Child (Node 2)**:
     - The left child has the value 2, which is less by 1 than its parent (node 3), so for node 3, `decr = 2`.
     - It recursively calls `dfs` on node 2's left child (node 1).
     - Node 1 is also less by 1 than node 2, so now for node 2, `decr = 2`.
     - Node 1 has no children, so the recursive call would return `[0, 0]`.
     - The `decr` from node 2 is now combined with node 3's `decr` to update `ans`, if necessary, to `ans = decr_2 + decr_3 - 1`.
   - **Right Child (Node 4)**:
     - The right child has the value 4, which is more by 1 than its parent (node 3), so for node 3, `incr = 2`.
     - It recursively calls `dfs` on node 4's right child (node 5).
     - Node 5 is more by 1 than node 4, so for node 4, `incr = 2`.
     - The `incr` from node 4 is now combined with node 3's `incr` to update `ans`, if necessary, to `ans = incr_4 + incr_3 - 1`.
3. **Update Global Maximum (`ans`)**: Since both left and right children of node 3 form consecutive sequences, we calculate the maximum sum of `decr` and `incr`.
   - From left child's `decr` (2 from node 2) and right child's `incr` (2 from node 4), we get `2 + 2 - 1 = 3` for node 3, which means including node 3, there is a path of length 3 that goes from node 5 to node 3 to node 2.
4. **Finalizing the Result**: After the full traversal,
   - We have computed `incr` and `decr` for all nodes.
   - We've also computed `ans` at each node, which is the maximum value obtained by adding `incr` and `decr` and subtracting 1 (to account for the current node being included in both sequences).
   - Since the tree was recursively traversed, `ans` holds the maximum length of the longest consecutive path after taking all nodes into consideration.

Based on our example, the longest consecutive path is 3 (which is the path `[2, 3, 4]` or `[4, 3, 2]`), and this is output by the `longestConsecutive` function as the DFS is executed from the root of the tree to all its children and their subsequent descendants.

Python Solution

```python
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right


class Solution:
    def longestConsecutive(self, root: TreeNode) -> int:
        # Recursive depth-first search to find the longest consecutive path
        def dfs(node):
            if node is None:
                return [0, 0]  # Base case: return 0's for the length of increasing and decreasing sequences

            nonlocal max_length  # Use nonlocal keyword to modify the non-local max_length variable
            incr_length = decr_length = 1  # Initialize lengths of increasing and decreasing paths

            # Perform DFS on the left and right children
            left_incr, left_decr = dfs(node.left)
            right_incr, right_decr = dfs(node.right)

            # Check for consecutive increments or decrements on the left child
            if node.left:
                if node.left.val + 1 == node.val:
                    incr_length = left_incr + 1
                elif node.left.val - 1 == node.val:
                    decr_length = left_decr + 1

            # Check for consecutive increments or decrements on the right child
            if node.right:
                if node.right.val + 1 == node.val:
                    incr_length = max(incr_length, right_incr + 1)
                elif node.right.val - 1 == node.val:
                    decr_length = max(decr_length, right_decr + 1)

            # Update the max_length considering both increasing and decreasing paths
            max_length = max(max_length, incr_length + decr_length - 1)
            return [incr_length, decr_length]

        max_length = 0  # Initialize the maximum length of consecutive sequence
        dfs(root)  # Start DFS from the root
        return max_length  # Return the maximum length found

```

Java Solution

```java
class Solution {
    private int longestLength;

    // Function to start the longest consecutive sequence process
    public int longestConsecutive(TreeNode root) {
        longestLength = 0;
        dfs(root);
        return longestLength;
    }

    // Perform a Depth First Search on the tree
    private int[] dfs(TreeNode node) {
        if (node == null) {
            return new int[] {0, 0};
        }

        int incrementing = 1; // Length of incrementing sequence ending at this node
        int decrementing = 1; // Length of decrementing sequence ending at this node

        // Recurse left
        int[] leftSubtree = dfs(node.left);
        // Recurse right
        int[] rightSubtree = dfs(node.right);

        // Check left child
        if (node.left != null) {
            if (node.left.val + 1 == node.val) {
                incrementing = leftSubtree[0] + 1;
            }
            if (node.left.val - 1 == node.val) {
                decrementing = leftSubtree[1] + 1;
            }
        }

        // Check right child
        if (node.right != null) {
            if (node.right.val + 1 == node.val) {
                incrementing = Math.max(incrementing, rightSubtree[0] + 1);
            }
            if (node.right.val - 1 == node.val) {
                decrementing = Math.max(decrementing, rightSubtree[1] + 1);
            }
        }

        // Update longestLength if the current sequence is the longest
        // -1 is to not double count the current node in both incrementing and decrementing sequences
        longestLength = Math.max(longestLength, incrementing + decrementing - 1);

        // Return the length of the longest incrementing and decrementing sequence ending at this node
        return new int[] {incrementing, decrementing};
    }
}

```

C++ Solution

```cpp
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:
    int longestStreak;

    // Function that starts the process and returns the longest consecutive sequence length
    int longestConsecutive(TreeNode* root) {
        longestStreak = 0;
        dfs(root);
        return longestStreak;
    }

    // Helper function to perform DFS and calculate the consecutive sequence length
    vector<int> dfs(TreeNode* root) {
        // Base case: if the node is null, return {0, 0} as there is no sequence
        if (!root) return {0, 0};

        // Initialize the length of the increasing and decreasing sequences to 1 (the root itself)
        int increaseLength = 1, decreaseLength = 1;

        // Recursively call dfs for the left and right subtrees
        vector<int> leftSequence = dfs(root->left);
        vector<int> rightSequence = dfs(root->right);

        // Process left child
        if (root->left) {
            // Check if it's consecutively increasing
            if (root->left->val + 1 == root->val) increaseLength = leftSequence[0] + 1;
            // Check if it's consecutively decreasing
            if (root->left->val - 1 == root->val) decreaseLength = leftSequence[1] + 1;
        }

        // Process right child
        if (root->right) {
            // Check if it's consecutively increasing
            if (root->right->val + 1 == root->val) increaseLength = max(increaseLength, rightSequence[0] + 1);
            // Check if it's consecutively decreasing
            if (root->right->val - 1 == root->val) decreaseLength = max(decreaseLength, rightSequence[1] + 1);
        }

        // Update the longest streak result by taking the maximum sum of increasing and
        // decreasing lengths from the current node minus 1 (to avoid double-counting the node itself)
        longestStreak = max(longestStreak, increaseLength + decreaseLength - 1);

        // Return a pair of the longest increasing and decreasing sequences starting from the current node
        return {increaseLength, decreaseLength};
    }
};

```

Typescript Solution

```typescript
// Definition for a binary tree node.
class TreeNode {
    val: number;
    left: TreeNode | null;
    right: TreeNode | null;

    constructor(val: number = 0, left: TreeNode | null = null, right: TreeNode | null = null) {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}

let longestStreak: number;

// Function that starts the process and returns the longest consecutive sequence length
function longestConsecutive(root: TreeNode | null): number {
    longestStreak = 0;
    dfs(root);
    return longestStreak;
}

// Helper function to perform DFS and calculate the consecutive sequence length
function dfs(root: TreeNode | null): number[] {
    // Base case: if the node is null, return [0, 0] as there is no sequence
    if (!root) return [0, 0];

    // Initialize the length of the increasing and decreasing sequences to 1 (the root itself)
    let increaseLength: number = 1, decreaseLength: number = 1;

    // Recursively call dfs for the left and right subtrees
    const leftSequence: number[] = dfs(root.left);
    const rightSequence: number[] = dfs(root.right);

    // Process left child
    if (root.left) {
        // Check if it's consecutively increasing
        if (root.left.val + 1 === root.val) {
            increaseLength = leftSequence[0] + 1;
        }
        // Check if it's consecutively decreasing
        if (root.left.val - 1 === root.val) {
            decreaseLength = leftSequence[1] + 1;
        }
    }

    // Process right child
    if (root.right) {
        // Check if it's consecutively increasing
        if (root.right.val + 1 === root.val) {
            increaseLength = Math.max(increaseLength, rightSequence[0] + 1);
        }
        // Check if it's consecutively decreasing
        if (root.right.val - 1 === root.val) {
            decreaseLength = Math.max(decreaseLength, rightSequence[1] + 1);
        }
    }

    // Update the longest streak result by taking the maximum sum of increasing and
    // decreasing lengths from the current node minus 1 (to avoid double-counting the node)
    longestStreak = Math.max(longestStreak, increaseLength + decreaseLength - 1);

    // Return a pair of the longest increasing and decreasing sequences starting from the current node
    return [increaseLength, decreaseLength];
}

```

Time and Space Complexity

Time Complexity

The provided code executes a depth-first search (DFS) on a binary tree. For every node, it calculates the longest consecutive sequence that can be formed both increasing and decreasing. The decision to increment or decrement the consecutive count, or to reset it, is done at every node. This leads to each node being visited exactly once.

Therefore, the time complexity of the DFS is `O(N)`, where `N` is the number of nodes in the tree. This is because the function processes each node a single time without revisiting anything.

Space Complexity

The space complexity of the algorithm includes the space used by the recursive call stack during the DFS traversal as well as the space used for storing the variables in each recursive call.

In the worst case, the height of the binary tree may be `O(N)` (in case of a skewed tree where each node has only one child), which would imply `O(N)` recursive calls stack space would be used. For balanced trees, the average case height would be `O(log N)` leading to an average case space complexity of `O(log N)`.

Hence, the space complexity is `O(N)` in the worst case, but in the average case for a balanced tree, it would be `O(log N)`.

The `ans` variable used to store the maximum length of the consecutive sequence doesn't significantly affect the space complexity as it is a single integer value.