971. Flip Binary Tree To Match Preorder Traversal

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

Problem Description

In this problem, we are working with a binary tree where each node has a unique integer value assigned from `1` to `n`. The `root` of this binary tree is provided to us. Additionally, we are given a sequence of `n` values named `voyage`. This sequence represents the desired pre-order traversal of the binary tree.

Pre-order traversal is a method where we visit the root node first, then recursively do a pre-order traversal of the left subtree, followed by a pre-order traversal of the right subtree.

One key aspect of the problem is that we can flip any node in the binary tree. Flipping a node means swapping its left and right subtrees. Our objective is to flip the smallest number of nodes in the tree so that the actual pre-order traversal matches the given `voyage` sequence.

If it is possible to achieve a match, we need to return a list of the values of all flipped nodes. If it's not possible, we simply return `[-1]`.

Intuition

To solve this problem, we will use a depth-first search (DFS) strategy that follows the pre-order traversal pattern. Since we're matching the `voyage` sequence in a pre-order fashion, we can keep track of where we are in the `voyage` sequence using a variable, let's say `i`.

We start with the root node and explore the tree in pre-order, checking at each step:

1. If the current node is `None`, we just return as there's nothing to process.
2. If the current node's value does not match the `voyage` sequence at index `i`, it means it's impossible to achieve the desired pre-order by flipping nodes. In this case, we set a flag (`ok`) to `False`.
3. If the current node has a left child and its value does not match the next value in the `voyage` sequence, this means we need to flip the current node. We add the current node's value to the answer list and continue the traversal with the right subtree first, followed by the left subtree.
4. If the left child's value matches the next value in the `voyage` or there is no left child, we traverse the left subtree first, followed by the right subtree without flipping.

Using this approach, we attempt to align our traversal with the `voyage` sequence. Whenever we find a mismatch with the left child, we flip the node. If flag `ok` remains `True` throughout the traversal, then the `voyage` sequence can be matched by flipping the nodes collected in the answer list. If `ok` turned `False` at any point, it means it is impossible to match the `voyage` sequence, and we return `[-1]`.

Solution Approach

The solution to this problem is based on the `Depth-First Search` (DFS) algorithm. This algorithm is a recursive approach to traverse a tree which fits perfectly with our need to explore each node in the context of the `voyage` sequence. Let's dive into the solution implementation using the DFS traversal pattern.

Firstly, a nested function named `dfs` is defined, which will be used to traverse each node of the tree. This function takes a single parameter `root`, referring to the current node being visited.

The algorithm uses a few external variables which are not part of the function parameters:

- `i`, which is an index keeping track of the current position in the `voyage` sequence.
- `ok`, which is a flag that tracks whether the pre-order traversal has been successful so far.
- `ans`, which is a list where the values of flipped nodes are stored.

Here are the main steps taken during the DFS algorithm:

1. **Base Condition**: If we reach a `None` node or if the `ok` is already `False` (meanwhile-found mismatch), we return immediately, as there's nothing left to traverse.
2. **Match Check**: We check if the current node's value matches the `voyage` sequence at the index `i`. If not, we set `ok` to `False` and return, since we cannot proceed further with a mismatch.
3. **Pre-order Traversal**:
   - We increment `i` to move to the next element in the `voyage` sequence after successful matching.
   - If there is no left child or the left child's value matches the next expected value in the `voyage` sequence, we first traverse left (`dfs(root.left)`) then right (`dfs(root.right)`).
   - If the left child does not match the next element in `voyage`, we must flip the current node to try and match the `voyage` sequence. We add the value of the current node to the `ans` list. Then we traverse right (`dfs(root.right)`) followed by left (`dfs(root.left)`) since we are considering the left and right children flipped.

Finally, after we initiate the `dfs` with `root` as the starting node, once the DFS completes, there are two possible outcomes based on the `ok` flag:

- If `ok` is `True`, we successfully followed the `voyage` sequence (potentially with some flips), and we return the `ans` list, which contains the values of all flipped nodes.
- If `ok` is `False`, we encountered a situation where no flipping could result in matching the `voyage` sequence, and therefore we return `[-1]`.

With the above implementation, we utilize the DFS algorithm to attempt to make the actual tree's pre-order traversal align with the `voyage` sequence while keeping track of any flipping needed along the way.

Example Walkthrough

Let's consider a binary tree example to illustrate the solution approach. Suppose we have a binary tree represented as:

```
    1
   / \
  2   3
```

and we're given a `voyage` sequence `[1, 3, 2]`. We want to determine if we can achieve this pre-order traversal sequence by flipping nodes in the tree.

1. We begin at the root with the value `1`. This matches the first element of `voyage`, so no action is needed. We set `i` to 0 and increment it after the match.

2. Next, we check the left child of the root. However, our `voyage` sequence expects the value `3` instead of the current left child value `2`. This indicates that we need to flip the root node to match the next element of `voyage`. We add the root's value `1` to the `ans` list.

3. We flip the children nodes of the root and proceed with the right child first (which used to be the left child). The right child's value `2` does not match the next `voyage` value `3`, so we set `ok` to `False`. However, we've just flipped the nodes, so we continue the traversal assuming the children are reversed.

4. We move to the left child (which used to be the right child) with the value `3`. This matches the next element in the `voyage` sequence, so we proceed and increment `i`.

5. With all nodes visited and matched after the flip, we check the `ok` flag. It remains `True` since we managed to match the `voyage` sequence with only one flip.

6. At this point, we've completed the DFS traversal, and since the `ok` flag is `True`, we return the list `[1]`, which represents the value of the node we flipped.

The outcome `[1]` indicates that by flipping the root node, we could achieve the given `voyage` sequence `[1, 3, 2]` in the pre-order traversal of the tree.

This simple example clearly illustrates how the DFS approach can be used to decide whether a binary tree's nodes could be flipped to match a given pre-order traversal sequence, and to record which nodes to flip if possible.

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 flipMatchVoyage(self, root: Optional[TreeNode], voyage: List[int]) -> List[int]:
        # Helper function to perform depth-first search traversal
        def dfs(node):
            # Use nonlocal to modify variables defined outside the nested function
            nonlocal index, is_voyage_matched
            # If we reach None or the voyage has already failed to match, return
            if node is None or not is_voyage_matched:
                return

            # If the current node's value doesn't match the voyage at the current index, the voyage doesn't match
            if node.val != voyage[index]:
                is_voyage_matched = False
                return

            index += 1  # Move to the next index in the voyage list

            # If the left child is present and its value matches the next value in the voyage list, explore left then right
            if node.left and node.left.val == voyage[index]:
                dfs(node.left)
                dfs(node.right)
            else:  # If there's a mismatch, we must flip the left and right children, record the node, and explore right then left
                flips.append(node.val)  # Record the flip
                dfs(node.right)
                dfs(node.left)

        flips = []  # Stores the list of flipped nodes
        index = 0   # Tracks the current index in the voyage list
        is_voyage_matched = True  # Flag to determine if the voyage matches the binary tree

        # Start DFS traversal from the root
        dfs(root)
        # Return the list of flips if voyage matched; otherwise, return [-1]
        return flips if is_voyage_matched else [-1]

```

Java Solution

```java
class Solution {
    private int currentIndex; // to keep track of current index in voyage
    private boolean isPossible; // flag to check if the flip is possible
    private int[] voyageArray; // the traversal array to match with tree traversal
    private List<Integer> flippedNodes = new ArrayList<>(); // list to keep track of flipped nodes

    public List<Integer> flipMatchVoyage(TreeNode root, int[] voyage) {
        this.voyageArray = voyage;
        isPossible = true; // initially assume flip is possible
        traverseTree(root);
        // if flip is not possible, return list containing only -1
        return isPossible ? flippedNodes : List.of(-1);
    }

    private void traverseTree(TreeNode node) {
        // if the current node is null or flip is already impossible, stop traversal
        if (node == null || !isPossible) {
            return;
        }
        // if current node's value does not match current voyage value, set flip as impossible
        if (node.val != voyageArray[currentIndex]) {
            isPossible = false;
            return;
        }
        currentIndex++; // move to the next element in voyage
        // check if left child exists and matches next voyage value, if not, flip is needed
        if (node.left == null || node.left.val == voyageArray[currentIndex]) {
            // if no flip needed or left child matches, continue with left subtree
            traverseTree(node.left);
            // then traverse right subtree
            traverseTree(node.right);
        } else {
            // flip needed, add current node value to flippedNodes list
            flippedNodes.add(node.val);
            // since we flip, we traverse right subtree before left subtree
            traverseTree(node.right);
            traverseTree(node.left);
        }
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <functional>

// Definition for a binary tree node.
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x = 0, TreeNode *left = nullptr, TreeNode *right = nullptr)
        : val(x), left(left), right(right) {}
};

class Solution {
public:
    // This function flips the nodes of the tree to match the given voyage and returns
    // the values of nodes flipped. If impossible, returns {-1}.
    std::vector<int> flipMatchVoyage(TreeNode* root, std::vector<int>& voyage) {
        bool isVoyagePossible = true;  // To keep track if the voyage is possible
        int currentIdx = 0;            // Index to keep track of the current position in the voyage vector
        std::vector<int> results;      // Vector that will contain the values of the flipped nodes

        // Lambda function for depth-first search
        std::function<void(TreeNode*)> dfs = [&](TreeNode* node) {
            if (!node || !isVoyagePossible) {  // If node is null or voyage so far is impossible, end DFS
                return;
            }
            if (node->val != voyage[currentIdx]) {  // If node's value doesn't match the current voyage value
                isVoyagePossible = false;           // It's not possible to achieve the voyage
                return;
            }
            ++currentIdx;  // Move to the next index in the voyage vector

            // Determine if we can continue with left child or need to flip
            if (node->left && node->left->val != voyage[currentIdx]) {
                results.push_back(node->val); // Flip the current node
                dfs(node->right);             // Attempt the right child next
                dfs(node->left);              // Then the left child last, since we flipped
            } else {
                dfs(node->left);              // Attempt the left child next
                dfs(node->right);             // Then the right child
            }
        };

        dfs(root);  // Start DFS with the root node

        // If the voyage is possible, return results; otherwise, return {-1}
        return isVoyagePossible ? results : std::vector<int>{-1};
    }
};

```

Typescript Solution

```typescript
// TypeScript definition for a binary tree node.
interface TreeNode {
  val: number;
  left: TreeNode | null;
  right: TreeNode | null;
}

/**
 * Returns a list of values in the order of flipped nodes required to match
 * the given voyage or [-1] if it is impossible.
 * @param root - The root of the binary tree.
 * @param voyage - The desired pre-order traversal (voyage) of the tree.
 * @returns A list of the values of the flipped nodes, or [-1] if impossible.
 */
function flipMatchVoyage(root: TreeNode | null, voyage: number[]): number[] {
  let isPossible = true; // Flag to track if a matching voyage is possible.
  let currentIndex = 0; // Index to track the current position in the voyage.
  const flippedNodes: number[] = []; // List to store flipped node values.

  /**
   * Depth-first search helper function to attempt flipping to match voyage.
   * @param node - The current node being visited in the tree.
   */
  function dfs(node: TreeNode | null): void {
    if (!node || !isPossible) {
      // Stop processing if we reach a null node or if it's already impossible.
      return;
    }

    if (node.val !== voyage[currentIndex++]) {
      // If the current node's value doesn't match the current voyage value, it's impossible.
      isPossible = false;
      return;
    }

    // Check if the current left child is the next in the voyage, or if we need to flip.
    if (node.left && node.left.val !== voyage[currentIndex]) {
      // If the left child's value doesn't match, we flip the node.
      flippedNodes.push(node.val);
      dfs(node.right); // Visit the right child first since we flipped.
      dfs(node.left); // Next, visit the left child.
    } else {
      // If the left child matches or is null, traverse normally.
      dfs(node.left);
      dfs(node.right);
    }
  }

  dfs(root); // Start the DFS traversal from the root.
  return isPossible ? flippedNodes : [-1]; // Return the result based on the isPossible flag.
}

// You can now use the flipMatchVoyage function by providing the root of a tree and the voyage array.

```

Time and Space Complexity

The code defines a recursive function `dfs` to traverse the binary tree. Given `n` as the number of nodes in the tree, the analysis is as follows:

Time Complexity:

- Each node in the binary tree is visited exactly once in the worst-case scenario.
- For each node, the algorithm performs a constant amount of work, checking node values and possibly appending to the `ans` list.
- Consequently, the time complexity of the `dfs` function is `O(n)`.

Thus, the overall time complexity of the `flipMatchVoyage` function is `O(n)`.

Space Complexity:

- The space complexity includes the space taken by the `ans` list and the implicit call stack due to recursion.
- In the worst case, we might need to flip all nodes, so the `ans` list could potentially grow to `O(n)`.
- The space complexity due to the recursion call stack is also `O(n)` in the worst case (this occurs when the tree is completely unbalanced, e.g., a linked list).

Combining both aspects, the total space complexity is `O(n)` due to the list and recursion stack.

Therefore, the space complexity of the `flipMatchVoyage` function is `O(n)`.