1339. Maximum Product of Splitted Binary Tree

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

Problem Description

The problem presents us with a binary tree and asks us to perform an operation where we remove one edge from this tree, effectively splitting it into two separate subtrees. Our goal is to find which edge to remove in order to maximize the product of the sums of the resultant two subtrees' node values. Once we have this maximum product, we must return it modulo `10^9 + 7`, as the number can be quite large.

The complexity here is that we're not just trying to find the sums of any two subtrees - we are specifically looking for an edge whose removal results in two subtrees whose sum-product is the greatest possible among all such removable edges. We have to take care to maximize the product before applying the modulo operation.

It's important to note the subtlety in the problem statement that you must maximize the product and then perform the modulo operation rather than taking the modulo of the sums during your calculations.

Intuition

The intuition behind solving this problem involves two main steps:

1. Calculate the total sum of all values in the tree. This will help us determine the total value that we're working to divide in the manner that gives the largest possible product.

2. Perform a depth-first search (DFS) on the tree. During this search, we'll consider each subtree's sum and use it to calculate a candidate product - which is the product of the current subtree's sum and the sum of the rest of the tree. This concept is similar to partitioning an array into subarrays and maximizing the product of their sums.

To calculate the candidate product, we consider the current subtree sum `t` and take the product with `(total sum - t)`, since removing the edge leading to the subtree would leave a remaining subtree whose sum is `total sum - t`. This process is applied at each node of the tree since any node's removal could potentially maximize the product.

The nonlocal variables `ans` and `s` are used to keep track of the maximum product found so far and the total sum of the tree, respectively. As we recursively calculate the sum of subtrees, we update the product if we find a partition that leads to a larger product.

By performing this exhaustive search, where every node is considered as a potential split point, we ensure that we find the configuration that gives us the maximum possible product. Then we apply the modulo as the last step before returning our answer.

The overall approach is quite efficient because we traverse each node once for the total sum calculation and once more during the DFS, resulting in a time complexity that is linear in the number of nodes in the tree.

Solution Approach

The solution uses the recursive depth-first search (DFS) algorithm to explore the binary tree. The key data structure used here is the binary tree itself, which consists of nodes defined by the class `TreeNode`. Each `TreeNode` contains a value and pointers to its left and right child nodes.

Here is a step-by-step breakdown of the solution approach:

1. The `sum` function is a recursive helper function that computes the sum of all node values in the tree. When called with the root of the entire tree, it returns the total sum `s` of all values. It adds the value of the current `root` node to the sum of values in the left subtree and the right subtree, recursively applied until leaf nodes are reached.

2. The `dfs` function is the main recursive function that performs the depth-first traversal of the tree. For each node, it computes a temporary sum `t`, which is the sum of the current node's value and the sums of its left and right subtrees. Every call to `dfs` returns the sum of the subtree rooted at that node.

3. The crucial part of the `dfs` function is where it updates the current answer `ans` with the maximum product found so far. It calculates the product of the current subtree sum `t` and the rest of the tree `(s - t)` if `t` is less than `s`, and updates `ans` if this product is larger than the current value of `ans`.

4. The `mod` variable is used to store the modulo value `10^9 + 7`. After the entire tree has been traversed and the maximum product is found, the `dfs` function simply returns the answer `ans` modulo `mod`.

5. The overall recursion starts by calling the `dfs` function from the root of the tree after initializing `ans` to 0 and computing the total sum `s` using the `sum` function.

In terms of algorithms and patterns, the solution employs a tree traversal and divide-and-conquer strategy. By splitting the tree at different points and comparing the product of the sums, we're effectively trying to find an optimal partition of the tree — a classic divide-and-conquer approach.

In Python syntax, code referencing values like `ans` from an outer scope within a nested function requires the use of the `nonlocal` keyword to indicate that the variable is not local to the nested function and should be looked up in the nearest enclosing scope where it is defined.

The DFS traversal and sum calculation both have a time complexity of $O(n)$, where n is the number of nodes in the tree. The space complexity is also $O(n)$ due to the recursion stack depth in a skewed tree, which can be as deep as the number of nodes (in the worst case where the tree is essentially a linked list).

Example Walkthrough

Let's take a small binary tree as an example to illustrate the solution approach:

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

Here are the steps following the solution approach outlined above:

1. **Total Sum Calculation:** Invoke the `sum` function to compute the total sum `s` of the tree. The `sum` of the tree would be 1 (root value) + 2 (left child value) + 4 (left grandchild value) + 5 (right grandchild value) + 3 (right child value) which equals 15.

2. **Depth-First Search (DFS) Traversal and Maximizing Product:**
   - Start the DFS from the root.
   - At the root (value 1), calculate the sum of its subtree `(1 + sum(left subtree) + sum(right subtree))` which would be 15. Since we're at the root, the other subtree doesn't exist; we are not removing any edge here. Move to the left child.
   - At Node 2, calculate its subtree sum, which is `2 + 4 + 5 = 11`. The rest of the tree's sum if we split here would be `15 - 11 = 4` (the value of the right subtree, Node 3). Calculate the product candidate `11 * 4 = 44`.
   - At Node 4, its sum is just 4 since it's a leaf node. Removing Node 4 creates subtrees with sums `4` and `15 - 4 = 11`. The product for these subtrees would be `4 * 11 = 44`. But this product does not exceed our current maximum (also 44), so we don't update.
   - At Node 5, also a leaf node, the sum is just 5. Removing Node 5 creates subtrees with sums `5` and `15 - 5 = 10`. The product would be `5 * 10 = 50`. This is greater than our current maximum, so we update our answer `ans` to 50.
   - At Node 3, its sum is just 3 since it's also a leaf node. Removing Node 3 creates subtrees with sums `3` and `15 - 3 = 12`. The product would be `3 * 12 = 36`, which is less than our current maximum. We don't update the answer.
3. **Applying Modulo Operation:** After conducting the DFS, the highest product calculated is 50. We would then take this number and perform a modulo operation with `10^9 + 7` to get the result that should be returned by the algorithm. Since 50 is much less than `10^9 + 7`, our final answer remains 50.

The crucial aspect to realize is that the DFS traversal explores each node, calculating the product for all possible splits and always retains the maximum product found so far. Finally, we apply the modulo operation to the maximum product and return this as our answer.

Python Solution

```python
class Solution:
    def maxProduct(self, root: Optional[TreeNode]) -> int:
        # Helper function to calculate the sum of all nodes' values in the tree
        def calculate_tree_sum(node: Optional[TreeNode]) -> int:
            if node is None:
                return 0
            # Sum the node's value and the sum of left and right subtrees
            return node.val + calculate_tree_sum(node.left) + calculate_tree_sum(node.right)

        # Helper function to find the maximum product of splitting the tree
        def find_max_product(node: Optional[TreeNode]) -> int:
            if node is None:
                return 0

            # Calculate the sum of the current subtree
            current_sum = node.val + find_max_product(node.left) + find_max_product(node.right)

            # Check if we can maximize the product using the sum of the current subtree
            if current_sum < total_sum:
                nonlocal max_product  # Allows us to modify the outer scope's max_product variable
                max_product = max(max_product, current_sum * (total_sum - current_sum))

            return current_sum

        modulo_base = 10**9 + 7
        total_sum = calculate_tree_sum(root)  # Getting the sum of all nodes in the tree
        max_product = 0
        find_max_product(root)  # Running the DFS function to find the maximum product

        return max_product % modulo_base   # Return the result modulo 10**9 + 7

```

Java Solution

```java
class Solution {
    // Global variables to store the maximum product and total sum of the tree
    private long maxProduct;
    private long totalSum;

    // Computes the maximum product of the sum of two subtrees of any node
    public int maxProduct(TreeNode root) {
        final int MODULO = (int) 1e9 + 7; // The modulo value as per the problem
        totalSum = calculateSum(root); // Calculate the total sum of all nodes in the tree
        calculateMaxProduct(root); // Calculate the maximum product
        return (int) (maxProduct % MODULO); // Return the result under modulo
    }

    // Helper function to calculate the sum of all nodes in the subtree rooted at 'root'
    private long calculateSum(TreeNode root) {
        if (root == null) { // Base case
            return 0;
        }
        // Calculate the sum recursively for left, right subtrees and add the node's value
        return root.val + calculateSum(root.left) + calculateSum(root.right);
    }

    // Helper function to update the maximum product
    private long calculateMaxProduct(TreeNode root) {
        if (root == null) { // Base case
            return 0;
        }

        // Recursively calculate the sum of the current subtree
        long currentSum = root.val + calculateMaxProduct(root.left) + calculateMaxProduct(root.right);

        // If currentSum is less than the total sum, consider the product of the sum of
        // this subtree and the sum of the rest of the tree and update maxProduct accordingly
        if (currentSum < totalSum) {
            maxProduct = Math.max(maxProduct, currentSum * (totalSum - currentSum));
        }

        // Return the sum of nodes of the current subtree
        return currentSum;
    }
}

/**
 * Definition for a binary tree node provided by the problem statement.
 */
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;

    TreeNode() {}

    TreeNode(int val) { this.val = val; }

    TreeNode(int val, TreeNode left, TreeNode right) {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}

```

C++ Solution

```cpp
#include <functional>

// Definition for a binary tree node provided in the problem statement.
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 maxProduct(TreeNode* root) {
        // Use long long type to avoid integer overflow.
        using ll = long long;

        // Initialize answer to 0.
        ll answer = 0;
        // Define the modulo as specified in the problem.
        const int MOD = 1e9 + 7;

        // Recursive lambda function to calculate the sum of values in the tree.
        function<ll(TreeNode*)> calculateSum = [&](TreeNode* node) -> ll {
            if (!node) {
                return 0;
            }
            return node->val + calculateSum(node->left) + calculateSum(node->right);
        };

        // Calculate the total sum of the tree values.
        ll totalSum = calculateSum(root);

        // Recursive lambda function to find the max product.
        function<ll(TreeNode*)> findMaxProduct = [&](TreeNode* node) -> ll {
            if (!node) {
                return 0;
            }

            // Calculate subtree sum.
            ll subtreeSum = node->val + findMaxProduct(node->left) + findMaxProduct(node->right);

            // If subtree sum is less than total sum, check if the product is the maximum.
            if (subtreeSum < totalSum) {
                answer = std::max(answer, subtreeSum * (totalSum - subtreeSum));
            }

            // Return the sum of the subtree rooted at the current node.
            return subtreeSum;
        };

        // Start the depth-first-search traversal from the root.
        findMaxProduct(root);

        // Return the answer modulo by 1e9+7 as required by problem statement.
        return answer % MOD;
    }
};

```

Typescript Solution

```typescript
// Definition for a binary tree node.
class TreeNode {
    val: number;
    left: TreeNode | null;
    right: TreeNode | null;
    constructor(val?: number, left?: TreeNode | null, right?: TreeNode | null) {
        this.val = val === undefined ? 0 : val;
        this.left = left === undefined ? null : left;
        this.right = right === undefined ? null : right;
    }
}

// Function to calculate the maximum product of splits of a binary tree.
function maxProduct(root: TreeNode | null): number {
    // Calculate the total sum of values in the binary tree.
    const calculateSum = (node: TreeNode | null): number => {
        if (!node) {
            return 0;
        }
        return node.val + calculateSum(node.left) + calculateSum(node.right);
    };

    // Store the total sum of the tree.
    const totalSum = calculateSum(root);

    // Variable to keep track of the maximum product found.
    let maximumProduct = 0;

    // The modulo to prevent overflow issues, mathematically equivalent to 10^9 + 7.
    const mod = 1e9 + 7;

    // DFS function to explore the tree and find the maximum product possible.
    const dfs = (node: TreeNode | null): number => {
        if (!node) {
            return 0;
        }

        // Calculate the subtree sum including the current node.
        const subtreeSum = node.val + dfs(node.left) + dfs(node.right);

        // If the subtree sum is less than the total sum, update the maximum product.
        if (subtreeSum < totalSum) {
            maximumProduct = Math.max(maximumProduct, subtreeSum * (totalSum - subtreeSum));
        }

        // Return the subtree sum.
        return subtreeSum;
    };

    // Start DFS from the root of the tree.
    dfs(root);

    // Return the maximum product found, modulo the specified value.
    return maximumProduct % mod;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the `maxProduct` function primarily depends on two recursive functions: `sum` and `dfs`.

1. The `sum` function computes the sum of values of all the nodes in the binary tree. This function visits each node exactly once, contributing a time complexity of `O(N)`, where `N` is the total number of nodes in the tree.

2. The `dfs` function traverses the binary tree in a similar way to the `sum` function, visiting each node to calculate the subtree sums and the maximum product at each step. Since it also visits each node exactly once, its time complexity is `O(N)`.

Since we need to run both `sum` and `dfs` once, and both have the same time complexity `O(N)`, the overall time complexity of the `maxProduct` function is `O(N) + O(N)`, which simplifies to `O(N)`.

Space Complexity

The space complexity of the `maxProduct` function includes the space used by the recursive call stack and any additional space used by variables and data structures within the function.

1. Both `sum` and `dfs` are recursive functions that can go as deep as the height of the tree. Therefore, the space complexity due to the recursive call stack is `O(H)`, where `H` is the height of the tree. In the worst case of a skewed tree where the tree behaves like a linked list, the height `H` can be `N`, leading to a space complexity of `O(N)`. In the best case, for a balanced tree, this would be `O(log(N))`.

2. The `maxProduct` function uses a few nonlocal variables (`ans`, `s`, and `mod`) that contribute constant space, which does not depend on the size of the input and thus is `O(1)`.

When combined, the overall space complexity of the `maxProduct` function is dominated by the space used by the recursive call stack, which is `O(H)`. In general terms, considering the worst case, this would be `O(N)`, and in the best case, it is `O(log(N))`.