1339. Maximum Product of Splitted Binary Tree Medium Tree Depth-First Search Binary Tree

Leetcode Link

The problem presents us with a binary tree and asks us to perform an operation where we remove one edge from this tree,

Problem Description

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

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.

operation rather than taking the modulo of the sums during your calculations.

- To calculate the candidate product, we consider the current subtree sum t and take the product with (total sum t), since
- each node of the tree since any node's removal could potentially maximize the product. The nonlocal variables and and 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

removing the edge leading to the subtree would leave a remaining subtree whose sum is total sum - t. This process is applied at

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**

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.

The solution uses the recursive depth-first search (DFS) algorithm to explore the binary tree. The key data structure used here is the

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

subtree and the right subtree, recursively applied until leaf nodes are reached.

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

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.

sum s using the sum function.

worst case where the tree is essentially a linked list).

Here are the steps following the solution approach outlined above:

def maxProduct(self, root: Optional[TreeNode]) -> int:

if node is None:

return 0

if current_sum < total_sum:</pre>

public int maxProduct(TreeNode root) {

private long calculateSum(TreeNode root) {

if (root == null) { // Base case

return 0;

TreeNode *left;

class Solution {

};

};

public:

8

9

11 };

10

12

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

TreeNode *right;

int maxProduct(TreeNode* root) {

// Initialize answer to 0.

const int MOD = 1e9 + 7;

return 0;

ll totalSum = calculateSum(root);

// Calculate subtree sum.

if (subtreeSum < totalSum) {</pre>

if (!node) {

if (!node) {

return 0;

return subtreeSum;

using ll = long long;

ll answer = 0;

TreeNode(): val(0), left(nullptr), right(nullptr) {}

TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}

// Use long long type to avoid integer overflow.

// Define the modulo as specified in the problem.

// Calculate the total sum of the tree values.

// Recursive lambda function to find the max product.

TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}

// Recursive lambda function to calculate the sum of values in the tree.

return node->val + calculateSum(node->left) + calculateSum(node->right);

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.

answer = std::max(answer, subtreeSum * (totalSum - subtreeSum));

function<ll(TreeNode*)> calculateSum = [&](TreeNode* node) -> ll {

function<ll(TreeNode*)> findMaxProduct = [&](TreeNode* node) -> ll {

// Return the sum of the subtree rooted at the current node.

// Start the depth-first-search traversal from the root.

return current_sum

modulo base = 10**9 + 7

max_product = 0

def calculate_tree_sum(node: Optional[TreeNode]) -> int:

Helper function to calculate the sum of all nodes' values in the tree

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)

Check if we can maximize the product using the sum of the current subtree

total_sum = calculate_tree_sum(root) # Getting the sum of all nodes in the tree

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

final int MODULO = (int) 1e9 + 7; // The modulo value as per the problem

// Helper function to calculate the sum of all nodes in the subtree rooted at 'root'

return (int) (maxProduct % MODULO); // Return the result under modulo

calculateMaxProduct(root); // Calculate the maximum product

totalSum = calculateSum(root); // Calculate the total sum of all nodes in the tree

// Calculate the sum recursively for left, right subtrees and add the node's value

max_product = max(max_product, current_sum * (total_sum - current_sum))

nonlocal max_product # Allows us to modify the outer scope's max_product variable

divide-and-conquer approach.

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

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

In Python syntax, code referencing values like ans from an outer scope within a nested function requires the use of the nonlocal

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

3. The crucial part of the dfs function is where it updates the current answer ans with the maximum product found so far. It

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

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

+ 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:

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)

At the root (value 1), calculate the sum of its subtree (1 + sum(left subtree) + sum(right subtree)) which would be 15.

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 =

product for these subtrees would be 4 * 11 = 44. But this product does not exceed our current maximum (also 44), so we

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

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 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

4 (the value of the right subtree, Node 3). Calculate the product candidate 11 * 4 = 44.

 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.

Python Solution

class Solution:

answer.

9

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

8

9

10

11

12

13

14

15

16

17

18

19

don't update.

Start the DFS from the root.

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

3. Applying Modulo Operation: After conducting the DFS, the highest product calculated is 50. We would then take this number

product would be 3 * 12 = 36, which is less than our current maximum. We don't update the answer.

Helper function to find the maximum product of splitting the tree 10 def find_max_product(node: Optional[TreeNode]) -> int: 11 if node is None: 12 13 return 0 14 15 # Calculate the sum of the current subtree current_sum = node.val + find_max_product(node.left) + find_max_product(node.right) 16

// Global variables to store the maximum product and total sum of the tree private long maxProduct; private long totalSum; 5 // Computes the maximum product of the sum of two subtrees of any node 6

class Solution {

Java Solution

```
return root.val + calculateSum(root.left) + calculateSum(root.right);
 20
 21
 22
 23
         // Helper function to update the maximum product
 24
         private long calculateMaxProduct(TreeNode root) {
 25
             if (root == null) { // Base case
 26
                 return 0;
 27
 28
 29
             // Recursively calculate the sum of the current subtree
 30
             long currentSum = root.val + calculateMaxProduct(root.left) + calculateMaxProduct(root.right);
 31
 32
             // If currentSum is less than the total sum, consider the product of the sum of
 33
             // this subtree and the sum of the rest of the tree and update maxProduct accordingly
 34
             if (currentSum < totalSum) {</pre>
 35
                 maxProduct = Math.max(maxProduct, currentSum * (totalSum - currentSum));
 36
 37
 38
             // Return the sum of nodes of the current subtree
 39
             return currentSum;
 40
 41
 42
 43 /**
     * Definition for a binary tree node provided by the problem statement.
 45
     */
     class TreeNode {
 47
         int val;
 48
         TreeNode left;
 49
         TreeNode right;
 50
         TreeNode() {}
 51
 52
 53
         TreeNode(int val) { this.val = val; }
 54
 55
         TreeNode(int val, TreeNode left, TreeNode right) {
 56
             this.val = val;
             this.left = left;
 57
 58
             this.right = right;
 59
 60 }
 61
C++ Solution
  1 #include <functional>
    // Definition for a binary tree node provided in the problem statement.
  4 struct TreeNode {
         int val;
```

56 57 58 59 };

```
findMaxProduct(root);
 54
 55
             // Return the answer modulo by 1e9+7 as required by problem statement.
             return answer % MOD;
 60
Typescript Solution
  1 // Definition for a binary tree node.
  2 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;
  8
             this.right = right === undefined ? null : right;
  9
 10
 11 }
 12
    // 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.
 15
         const calculateSum = (node: TreeNode | null): number => {
 16
 17
             if (!node) {
 18
                 return 0;
 19
 20
             return node.val + calculateSum(node.left) + calculateSum(node.right);
         };
 21
 22
 23
         // Store the total sum of the tree.
 24
         const totalSum = calculateSum(root);
 25
 26
         // Variable to keep track of the maximum product found.
 27
         let maximumProduct = 0;
 28
 29
         // The modulo to prevent overflow issues, mathematically equivalent to 10^9 + 7.
         const mod = 1e9 + 7;
 30
 31
 32
         // DFS function to explore the tree and find the maximum product possible.
         const dfs = (node: TreeNode | null): number => {
 33
             if (!node) {
 34
 35
                 return 0;
 36
 37
 38
             // Calculate the subtree sum including the current node.
 39
             const subtreeSum = node.val + dfs(node.left) + dfs(node.right);
 40
             // If the subtree sum is less than the total sum, update the maximum product.
 41
 42
             if (subtreeSum < totalSum) {</pre>
                 maximumProduct = Math.max(maximumProduct, subtreeSum * (totalSum - subtreeSum));
 43
 44
 45
 46
             // Return the subtree sum.
 47
             return subtreeSum;
         };
 48
 49
 50
         // Start DFS from the root of the tree.
 51
         dfs(root);
 52
 53
         // Return the maximum product found, modulo the specified value.
 54
         return maximumProduct % mod;
 55 }
```

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

Time and Space Complexity

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).

O(log(N)).

56

Space Complexity The space complexity of the maxProduct function includes the space used by the recursive call stack and any additional space used

and the maximum product at each step. Since it also visits each node exactly once, its time complexity is O(N).

contributing a time complexity of O(N), where N is the total number of nodes in the tree.

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

1. The sum function computes the sum of values of all the nodes in the binary tree. This function visits each node exactly once,

2. The dfs function traverses the binary tree in a similar way to the sum function, visiting each node to calculate the subtree sums

on the size of the input and thus is 0(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 $O(\log(N))$.

2. The maxProduct function uses a few nonlocal variables (ans, s, and mod) that contribute constant space, which does not depend