2313. Minimum Flips in Binary Tree to Get Result **Depth-First Search Dynamic Programming Binary Tree** Hard Tree

Problem Description

either 0 or 1, which stand for the boolean values false and true, respectively. The non-leaf nodes can have one of four values: 2 (OR operation), 3 (AND operation), 4 (XOR operation), or 5 (NOT operation). The goal is to perform a minimum number of flips on the leaf nodes, where a flip changes a 0 to a 1 or a 1 to a 0, so that when the expression represented by the whole tree is evaluated, it yields the boolean value given by the variable result. It's interesting to note that for the NOT operation, which is represented by the value 5, the tree node will have only one child. For all other operations, there will be two children.

In this problem, we're given the root of a binary tree that represents a boolean expression. The leaves of this tree have values of

Leetcode Link

The problem assures us that it is always possible to achieve the desired result through a series of flips. However, the challenge is to find the minimum number of flips needed to do so.

Intuition

To solve the problem, we must carefully navigate the tree and consider the cost (in terms of flips) of getting to the desired result.

We'll need to perform a depth-first traversal of the tree. For each node, we need to consider two scenarios: The number of flips required if the current node's evaluation needs to be true

• The number of flips required if the current node's evaluation needs to be false Because we are dealing with a boolean operation, we must consider the operation represented by each non-leaf node when

• If we encounter a leaf node, the cost to make it true or false is straightforward: if the leaf node's current value is 0, it would take

costs associated with flipping each of its children to make it either true or false.

O flips to make it false and 1 flip to make it true, and vice versa for a leaf with a value of 1.

• cost_to_false is the minimum number of flips to make the current subtree evaluate to false.

The core of the solution is a recursive depth-first search that, for each node, returns a tuple (cost_to_true, cost_to_false) where:

• For non-leaf nodes, the situation is more complex, and we need to consider the operation represented by the node and the

cost_to_true is the minimum number of flips to make the current subtree evaluate to true.

Here's how we implement the solution:

computing the number of flips.

Here's the intuition behind the solution:

- Depending on the node's operation, we combine the costs of its children according to the boolean logic of OR, AND, XOR, and NOT.
- Finally, the answer will be the cost associated with the root node to make its evaluation equal to the result. In the provided solution, this is implemented as dfs(root)[int(result)].

The solution implements a recursive approach to traverse the tree and calculate the minimum number of flips required to achieve the

• We define a recursive function called dfs which takes a node as input and returns a tuple (cost_to_true, cost_to_false). This

tuple represents the minimum number of flips required to make the subtree rooted at the given node evaluate to true and false,

desired result for each subtree. The algorithm uses Depth-First Search (DFS) which is a common tree traversal technique.

• OR (2): The minimum number of flips required to make the result true is simply the sum of flips for both children to be true. To make the result false, we need to have at least one child as false, so we take the minimum of either child being false or both being false.

 After calculating the costs for a node's children, we take the appropriate combination of costs as per the node's operation. Importantly, the solution ensures that results are mathematically minimized, taking advantage of the boolean nature of the problem to calculate the minimum number of steps.

• The base case for the recursion is when we encounter a None type for a child node; we return (inf, inf) since we don't perform

• NOT (5): This operation flips the outcome, so we swap the costs for true and false of the single child node.

- Let's illustrate the solution approach with a small example. Consider the following binary tree representing a boolean expression:
- In this tree: The root node represents an OR operation (2).

To make the NOT node true, since it's currently true, it would need to be flipped to false first, then NOT applied. That's 1 flip.

• To make the AND node true, we need both children to be true. In this case, we need 1 flip for the child 0 to become 1.

By following the approach outlined in the solution, we recursively calculate the minimum number of flips and choose the optimal path

o To make it false, we can choose either of the children to be false, which in this case requires 0 flips since the right child is

• To make it false, we do nothing since NOT 1 is false. That's 0 flips. 3. The right subtree is an AND operation with two leaf children 1 and 0:

already 0.

Python Solution

2 class TreeNode:

class Solution:

1 # Definition for a binary tree node.

at each step to achieve the desired boolean value with the least number of flips.

def minimumFlips(self, root: Optional[TreeNode], result: bool) -> int:

left_flips, right_flips = dfs(node.left), dfs(node.right)

return (left_flips[0] + right_flips[0], # 0 AND 0

return (min(left_flips[1], right_flips[1]), # 0 NAND 0

Call the dfs helper function starting with the root of the tree.

min(left_flips[0], right_flips[0])) # 1 NAND 1

return new int[] {Integer.MAX_VALUE / 2, Integer.MAX_VALUE / 2};

Convert the boolean result to an integer (0 or 1) to get the corresponding flips.

// Base case: if the node is null, we return large numbers because there is nothing to flip

// Leaf node condition, where value directly determines the number of flips (0 for no change, 1 for a flip)

Calculate the minimum number of flips depending on the operation

represented by the current node's value (AND, OR, XOR, NAND).

return node.val, node.val ^ 1

if node.val == 2: # AND

if node.val == 3: # OR

return dfs(root)[int(result)]

return dfs(root)[result ? 1 : 0];

if (node == null) {

2. The left subtree is a NOT operation with a child leaf node of 1:

4. Now, back at the root OR node, to make it true: • We can make either the left or the right subtree true. As per the calculated costs, making the left subtree true costs 1 flip, and making the right subtree true costs 1 flip as well. So, we would choose the minimum, which is 1 flip in this case.

def __init__(self, val=0, left=None, right=None): self.val = val self.left = left self.right = right

return float('inf'), float('inf') # No flips can be done on a non-existent node

31 return (min(left_flips[0] + right_flips[0], left_flips[0] + right_flips[1], left_flips[1] + right_flips[0]), # 0 0 32 left_flips[1] + right_flips[1]) # 1 OR 1 if node.val == 4: # XOR 33 34 return (min(left_flips[0] + right_flips[0], left_flips[1] + right_flips[1]), # 0 XOR 0 35 min(left_flips[0] + right_flips[1], left_flips[1] + right_flips[0])) # 1 XOR 1 36 # NAND operation (not standard, but assuming it's the opposite of AND)

min(left_flips[0] + right_flips[1], left_flips[1] + right_flips[0], left_flips[1] + right_flips[1])) # 1 A

- 1 class Solution { // Function to calculate the minimum flips to make the tree evaluate to the desired result public int minimumFlips(TreeNode root, boolean result) { // Call the DFS traversal and return the flips needed for the desired result (false: 0, true: 1)
- int value = node.val; **if** (value < 2) { return new int[] {value, value ^ 1}; 19 20 21 // Recurse on the left and right subtrees int[] leftFlips = dfs(node.left); 22 23 int[] rightFlips = dfs(node.right); 24 25 // Initialize the minimum flips for making the current subtree true or false 26 int minFlipsFalse, minFlipsTrue; 27 28 // Check the type of operator at the current node and calculate flips accordingly 29 if (value == 2) { // AND operator 30 minFlipsFalse = leftFlips[0] + rightFlips[0]; // Both subtrees should be false 31 minFlipsTrue = Math.min(Math.min(leftFlips[1] + rightFlips[0], leftFlips[0] + rightFlips[1]), leftFlips[1] + rightFlips 32 } else if (value == 3) { // OR operator 33 minFlipsFalse = Math.min(Math.min(leftFlips[0] + rightFlips[1], leftFlips[1] + rightFlips[0]), leftFlips[0] + rightFlip minFlipsTrue = leftFlips[1] + rightFlips[1]; // Both subtrees should be true 34 35 } else if (value == 4) { // XOR operator 36 minFlipsFalse = Math.min(leftFlips[0] + rightFlips[0], leftFlips[1] + rightFlips[1]); // Subtrees should have same trut 37 minFlipsTrue = Math.min(leftFlips[0] + rightFlips[1], leftFlips[1] + rightFlips[0]); // Subtrees should have opposite t 38 } else { // NOT operator (Assuming it's a unary operator on left subtree only) 39 minFlipsFalse = Math.min(leftFlips[1], rightFlips[1]); // The subtree should be true to propagate false upwards minFlipsTrue = Math.min(leftFlips[0], rightFlips[0]); // The subtree should be false to propagate true upwards 40 41 42 // Return the minimum flips required to make the subtree true and false return new int[] {minFlipsFalse, minFlipsTrue}; C++ Solution 1 /** * 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) {} 9 TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {} 10 11 }; 12 13 class Solution { 14 public: 15 int minimumFlips(TreeNode* root, bool result) { 16 // Utility lambda function to perform a depth-first search on the tree. // It returns a pair of integers representing the minimum flips needed to 17 18 // make the subtree rooted at 'node' evaluate to false (first) and true (second).

function<pair<int, int>(TreeNode*)> dfs = [&](TreeNode* node) -> pair<int, int> {

// Base case: if the node is null, return maximum possible values

// If the value of the node is 0 or 1, it's a leaf node and we return

// Recursively get the minimum flips for left and right subtrees.

auto [leftFlipsFalse, leftFlipsTrue] = dfs(node->left);

auto [rightFlipsFalse, rightFlipsTrue] = dfs(node->right);

flipsToFalse = leftFlipsFalse + rightFlipsFalse;

flipsToTrue = leftFlipsTrue + rightFlipsTrue;

flipsToFalse = min(leftFlipsTrue, rightFlipsTrue);

// Return the corresponding flips based on the 'result' parameter.

flipsToTrue = min(leftFlipsFalse, rightFlipsFalse);

// Call the DFS function and get the minimum flips needed for false and true.

// the cost to flip it to 0 (first) and to 1 (second), which is just one flip.

// Based on the value of the node (operation type), combine the results from left

} else { // NOT operation (we assume that 'value' is either 0, 1, 2, 3, 4, or 5)

flipsToFalse = min(leftFlipsFalse + rightFlipsFalse, leftFlipsTrue + rightFlipsTrue);

flipsToTrue = min(leftFlipsFalse + rightFlipsTrue, leftFlipsTrue + rightFlipsFalse);

flipsToTrue = min({leftFlipsFalse + rightFlipsTrue, leftFlipsTrue + rightFlipsFalse, leftFlipsTrue + rightFlipsTrue

flipsToFalse = min({leftFlipsFalse + rightFlipsFalse, leftFlipsFalse + rightFlipsTrue, leftFlipsTrue + rightFlipsFa

// and right subtrees to determine the minimum flips for current subtree.

// since we can't flip a null node.

return {INT_MAX, INT_MAX};

return {value, value ^ 1};

int flipsToFalse = 0, flipsToTrue = 0;

} else if (value == 3) { // OR operation

} else if (value == 4) { // XOR operation

// Return the pair of flips for false and true.

if (value == 2) { // AND operation

return {flipsToFalse, flipsToTrue};

auto [flipsForFalse, flipsForTrue] = dfs(root);

return result ? flipsForTrue : flipsForFalse;

// The TreeNode class represents each node in the binary tree.

if (!node) {

int value = node->val;

if (value < 2) {

9 10 11 } 12

```
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)
 13 // The minimumFlips function calculates the minimum number of flips required
 14 // to make the output of the binary tree equal to the result boolean.
 15 // @param root - The root of the binary tree.
 16 // @param result - The expected result of the binary tree after performing flips.
 17 // @returns — the minimum number of flips needed.
 18 function minimumFlips(root: TreeNode | null, result: boolean): number {
         // The dfs (Depth-First Search) function computes two values for each node:
 19
         // The minimum flips required to make the value of the subtree rooted at this node 0 and 1, respectively.
 20
 21
         const dfs = (node: TreeNode | null): [number, number] => {
 22
             if (!node) {
 23
                 // If the node is null, return maximum values as it contributes
 24
                 // nothing to the count of flips.
 25
                 return [Number.MAX_SAFE_INTEGER, Number.MAX_SAFE_INTEGER];
 26
 27
 28
             const value = node.val;
 29
 30
             // If the node is a leaf node or holds a value of 0 or 1, return the values
 31
             // itself and flip required to change it (0 becomes 1 and 1 becomes 0).
 32
             if (value < 2) {
                 return [value, value ^ 1];
 33
 34
 35
 36
             // Otherwise, compute the flips for left and right subtrees.
             const [leftFlipsToZero, leftFlipsToOne] = dfs(node.left);
 37
             const [rightFlipsToZero, rightFlipsToOne] = dfs(node.right);
 38
 39
 40
             // Calculate the flips based on the logical operation represented by the current node.
 41
             if (value === 2) {
 42
                 // Logical AND
 43
                 return [leftFlipsToZero + rightFlipsToZero, Math.min(leftFlipsToZero + rightFlipsToOne, leftFlipsToOne + rightFlipsToZe
 44
 45
 46
             if (value === 3) {
 47
                 // Logical OR
 48
                 return [Math.min(leftFlipsToZero + rightFlipsToZero, leftFlipsToZero + rightFlipsToOne, leftFlipsToOne + rightFlipsToZe
 49
 50
             if (value === 4) {
 51
 52
                 // Logical XOR
 53
                 return [Math.min(leftFlipsToZero + rightFlipsToZero, leftFlipsToOne + rightFlipsToOne), Math.min(leftFlipsToZero + right
 54
 55
 56
             // Logical NAND
             return [Math.min(leftFlipsToOne, rightFlipsToOne), Math.min(leftFlipsToZero, rightFlipsToZero)];
 57
 58
         };
 59
 60
         // Initial call to dfs function starting at the root,
 61
         // and return the required minimum flips based on the desired result.
         return dfs(root)[result ? 1 : 0];
 62
 63 }
Time and Space Complexity
The given code implements an algorithm to calculate the minimum number of flips necessary in a binary tree such that the value of
```

Time Complexity: The time complexity of the algorithm is O(N), where N is the number of nodes in the binary tree. This is because we visit each node exactly once in a depth-first search (DFS) manner. For each node, we perform a constant-time operation regardless of its value,

which either directly returns the values for leaves or calculates the minimum based on the values returned from the children for internal nodes. **Space Complexity:**

The space complexity of the algorithm can be analyzed by considering the call stack for the recursive function dfs. In the worst-case scenario, the recursion depth is equal to the height of the tree. For a balanced binary tree, the height is 0(log N), which would make the space complexity O(log N) due to the call stack. However, for a completely unbalanced tree (e.g., a linked list), the recursion could be as deep as O(N). Thus, the space complexity of the algorithm is O(H), where H is the height of the tree, and in the worst

Solution Approach

• For a leaf node with value 0 or 1, the cost to make it true is its value (since 0 means false and 1 means true), and the cost to make it false is its complement (which is calculated by $x ^1$, where x is the value of the node). For a non-leaf node, we recursively get the costs for its left and right children. The combined cost depends on the operation the non-leaf node represents:

respectively.

o AND (3): The flips to make the result true are the minimum required to get one or both children true. To make the result false, both children must be false, so we add both children's cost_to_false. XOR (4): To maintain the true result, either one child should be true and the other false, so we take the two minimums accordingly. To make the result false, both children need to have the same evaluation, so we consider either both true or

both false.

operations on non-existent children. • The function dfs(root)[int(result)] is used to extract the cost corresponding to the desired boolean result. The solution leverages tuple unpacking, bitwise operations, and the recursive structure of the DFS algorithm, which makes it efficient and covers all edge cases due to its minimization logic at each step of the recursion. **Example Walkthrough**

• The left child of the root is a NOT operation (5), with a leaf child of 1 (true). • The right child of the root is an AND operation (3), with two leaf children 1 (true) and 0 (false). Let's say we want the final evaluated result of the tree to be true (1). Here's how we walk through the solution approach: 1. At the root (OR operation), we need to evaluate both the left and right subtrees.

• We don't need to calculate the cost to make it false since the desired result is true. 5. The minimum number of flips to get to the result is hence 1 which corresponds to flipping the leaf node from 0 to 1 in the right AND subtree.

Helper function to perform a depth-first search on the binary tree. 10 # This function will return a tuple with the minimum number of flips required 11 # to get the current subtree to evaluate to 0 and 1, respectively. 12 13 def dfs(node: Optional[TreeNode]) -> (int, int): if node is None: 14 15 16 17 # If node value is 0 or 1, no need to flip to obtain the same value, 18 # but one flip is needed to change it to the opposite value. if node.val in (0, 1): 19 20 21 22 # Recurse on the left and right subtrees.

23

24

25

26

27

28

29

30

37

38

39

40

41

42

43

Note: It assumes that Optional[TreeNode] and float('inf') are already valid as per the Python 3 syntax. 45 Java Solution 6 8 // DFS traversal to calculate minimum flips at each subtree private int[] dfs(TreeNode node) { 9 10 11 12 13 14 15 16 17 18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

51

52

53

54

55

56

57

58

59

61

60 };

};

Typescript Solution

val: number

left: TreeNode | null

right: TreeNode | null

2 class TreeNode {

the root node is equal to the desired result.

case, it is O(N).