**Code Challenge #7 Node Depths (Easy)**

**Graphical user interface, text

Description automatically generated**

**Solution #1**

1. function nodeDepths(root) {
2. let sumOfDepths = 0
3. const stack = [{node: root, depth: 0}];
4. while (stack.length > 0) {
5. const {node, depth} = stack.pop();
7. if (node === null) continue;
8. sumOfDepths += depth;
9. stack.push({node: node.left, depth: depth + 1});
10. stack.push({node: node.right, depth: depth + 1})
11. }
12. return sumOfDepths;
13. }
15. // This is the class of the input binary tree.
16. class BinaryTree {
17. constructor(value) {
18. this.value = value;
19. this.left = null;
20. this.right = null;
21. }
22. }

**Explanation**

This code challenge requires you to add the total number of nodes based on depth of a binary tree. For example, the depth of this binary tree is 16. The depth of a binary tree is the number of levels a node is from the root. The first node is the root node which has a depth of 0 (it is 0 depth from the root since it is the root itself). The second level (depth of 1) contains two items, so we say it is 2 times 1 which is 2. The third level (depth of 2) contains 4 times, so we say it is 4 times 2 which is 8. The fourth level (depth of 3) contains 2 items so we it is 2 times 3 which is 6.

Shape, arrow

Description automatically generated

In order to solve this iteratively we will first create a let variable called sumOfDepths = 0. We will then create a const called stack which is an array which initially contains an object where the key value pairs are node: root and depth: 0. We will then create a while loop that runs as long as the stack.length is greater than 0 aka there are items in the stack. Then we will create a const with variables called node and depth using object destructuring and using stack.pop which removes the last item from the stack. If the node === null aka no children node we will continue. We will add depth to the SumOfDepths using +=. We will then push onto stack an object {node: node.left, depth: depth + 1}) . Followed by pushing onto the stack {node: node.right, depth: depth + 1}). We finally return the sumOfDepths. The code runs in O(n) time.

**Solution #2**

1. function nodeDepths(root, depth = 0) {
2. if (root === null) return 0;
3. return depth + nodeDepths(root.left, depth + 1) + nodeDepths(root.right, depth + 1)
5. }
7. // This is the class of the input binary tree.
8. class BinaryTree {
9. constructor(value) {
10. this.value = value;
11. this.left = null;
12. this.right = null;
13. }
14. }

The recursive solution is much simpler. It starts off with two arguments which are root and depth. First we check to see if the root is equal to null. If it is we return 0. If it isn’t we return depth plus recursive call on root.left, depth + 1 plus recursive call on root.right, depth + 1. The code runs in O(n) times.