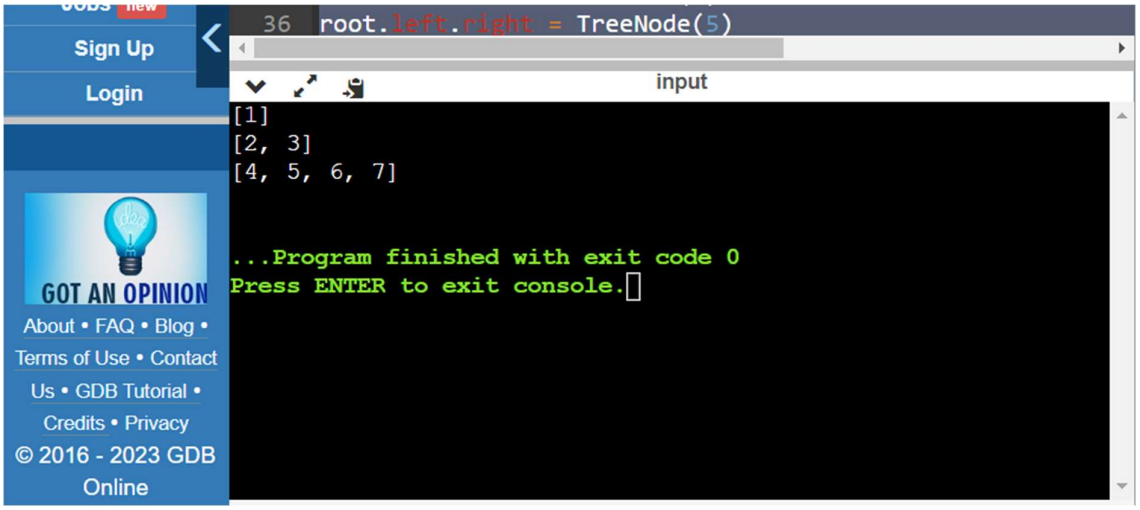
Problem 1: Level order traversal of a binary tree. Given the root node of the tree and you have to print the value of the level of the node by level.

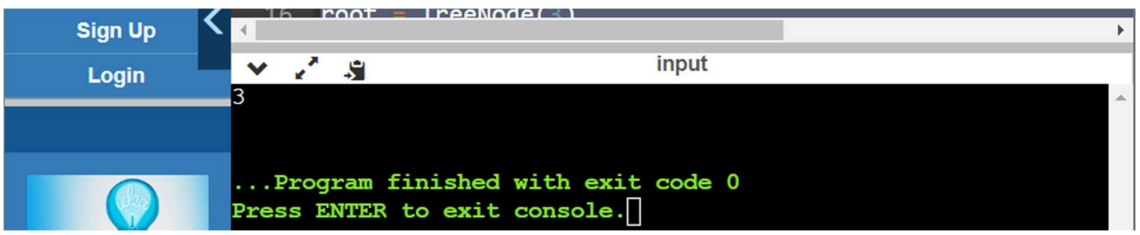
```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right


def levelOrder(root):
    if not root:
        return []

    result = []
    queue = [root]

    while queue:
        level = []
        level_size = len(queue)

        for _ in range(level_size):
            node = queue.pop(0)
            level.append(node.val)

            if node.left:
                queue.append(node.left)

            if node.right:
                queue.append(node.right)

        result.append(level)
```

```
    return result


root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)
root.right.left = TreeNode(6)
root.right.right = TreeNode(7)


result = levelOrder(root)
for level in result:
    print(level)
```



Problem 2: Find the **Maximum Depth** of Binary Tree. Maximum Depth is the **count of nodes of the longest path** from the root node to the leaf node.

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


def maxDepth(root):
    if root is None:
        return 0

    left_depth = maxDepth(root.left)
    right_depth = maxDepth(root.right)

    return max(left_depth, right_depth) + 1

root = TreeNode(3)
root.left = TreeNode(9)
root.right = TreeNode(20)
root.right.left = TreeNode(15)
root.right.right = TreeNode(7)

depth = maxDepth(root)
print(depth)
```



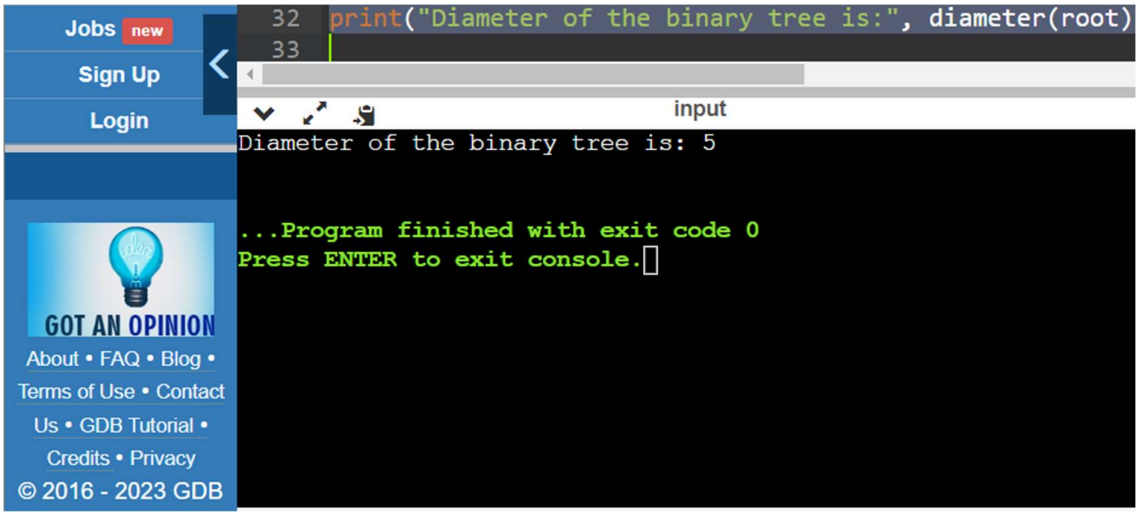
Problem 3: Find the Diameter of a Binary Tree. **Diameter** is the length of the longest path between any 2 nodes in the tree and this path may or may not pass from the root.

```
class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None


def height(node):
    if node is None:
        return 0
    return max(height(node.left), height(node.right)) + 1


def diameter(node):
    if node is None:
        return 0

    left_height = height(node.left)
    right_height = height(node.right)

    left_diameter = diameter(node.left)
    right_diameter = diameter(node.right)

    return max(left_height + right_height + 1, max(left_diameter, right_diameter))

root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
```

root.right.left = Node(6)

root.right.right = Node(7)

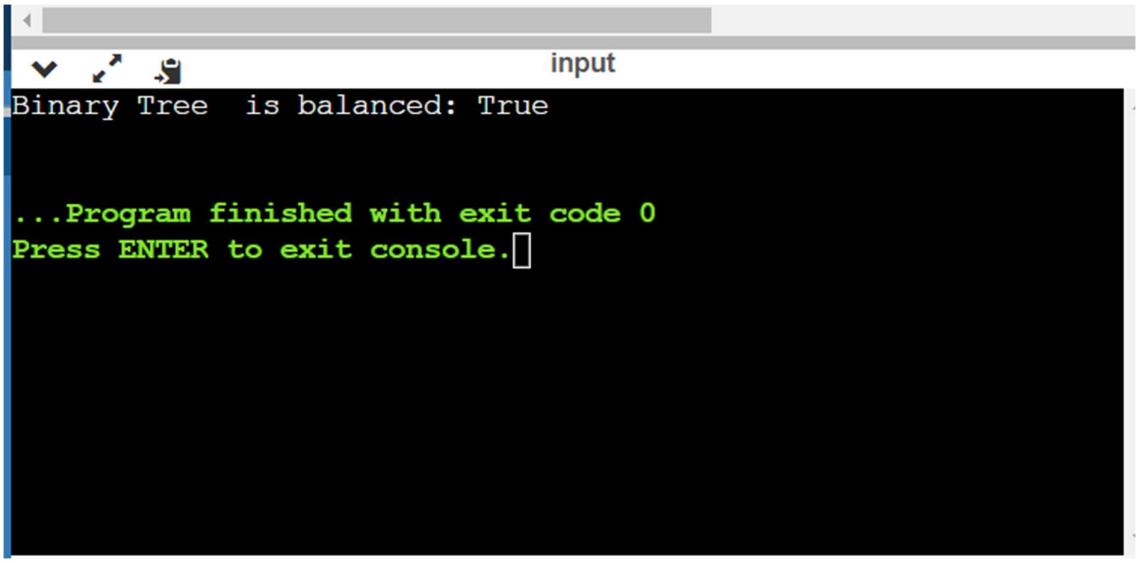
print("Diameter of the binary tree is:", diameter(root))



Problem 4: Check whether the given Binary Tree is a **Balanced Binary Tree** or not. A binary tree is balanced if, for all nodes in the tree, the difference between left and right subtree height is not more than 1.

```
class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None



def height(node):
    if node is None:
        return 0
    return max(height(node.left), height(node.right)) + 1



def is_balanced(root):
    if root is None:
        return True

    left_height = height(root.left)
    right_height = height(root.right)

    if (
        abs(left_height - right_height) <= 1
        and is_balanced(root.left)
        and is_balanced(root.right)
    ):
        return True

    return False
```

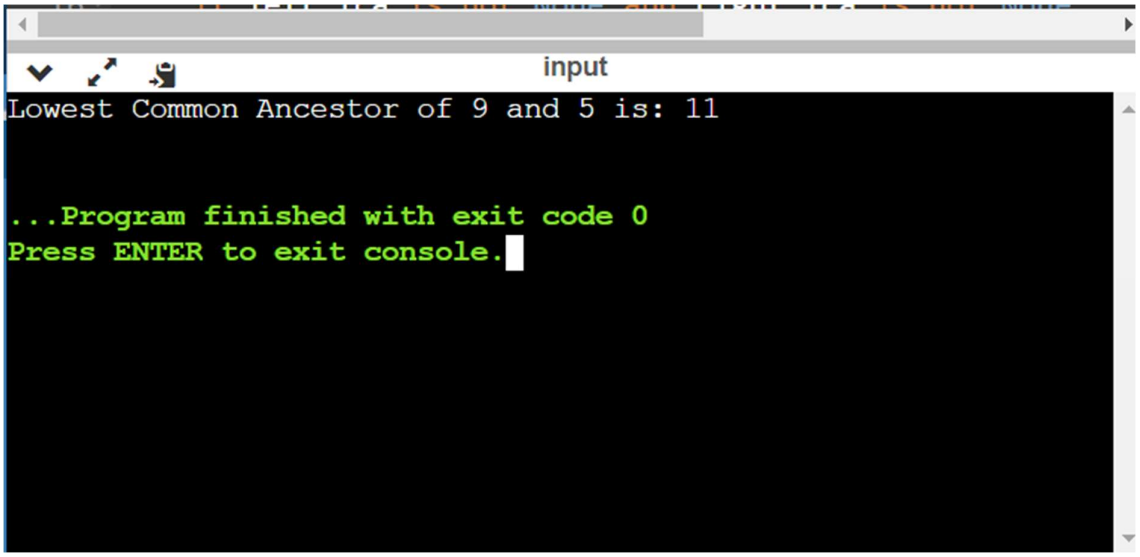
```
root = Node(1)

root.left = Node(2)

root.right = Node(3)

root.left.left = Node(4)

root.left.right = Node(5)

print("Binary Tree  is balanced:", is_balanced(root))
```



Problem 5: Given a binary tree, Find the Lowest Common Ancestor for two given Nodes (x,y).

```
class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None


def find_lowest_common_ancestor(root, x, y):
    if root is None:
        return None

    if root.value == x or root.value == y:
        return root.value

    left_lca = find_lowest_common_ancestor(root.left, x, y)
    right_lca = find_lowest_common_ancestor(root.right, x, y)
    if left_lca is not None and right_lca is not None:
        return root.value

    return left_lca if left_lca is not None else right_lca

root = Node(3)
root.left = Node(6)
root.right = Node(8)
root.left.left = Node(2)
root.left.right = Node(11)
root.left.right.left = Node(9)
root.left.right.right = Node(5)
root.right.right = Node(13)
root.right.right.left = Node(7)
```

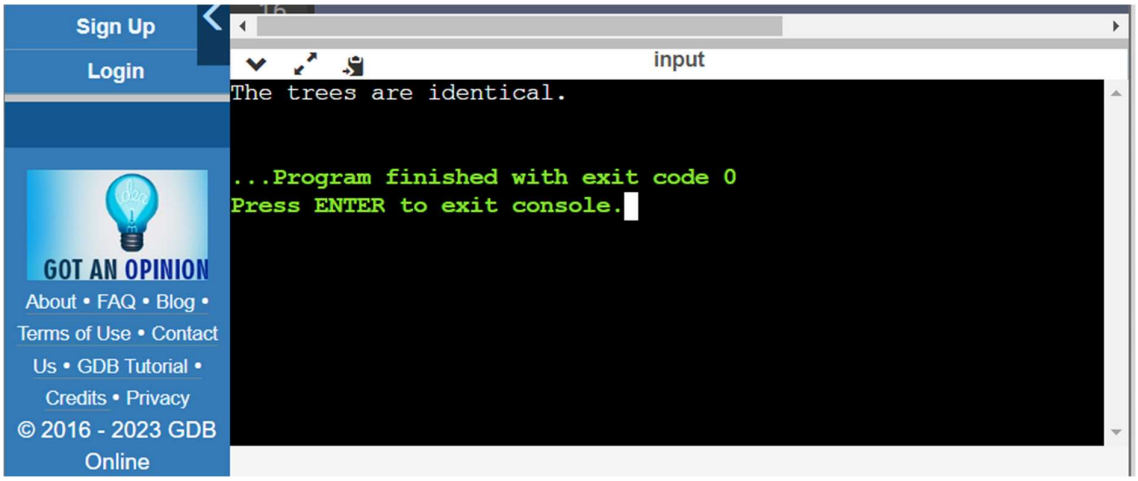
```
x = 9

y = 5

lca = find_lowest_common_ancestor(root, x, y)

print("Lowest Common Ancestor of", x, "and", y, "is:", lca)
```



Problem 6: Given two Binary Tree. Write a program to check if two trees are identical or not.

```
class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None


def are_identical(root1, root2):
    if root1 is None and root2 is None:
        return True
    if root1 is None or root2 is None:
        return False

    if root1.value != root2.value:
        return False
    return are_identical(root1.left, root2.left) and are_identical(root1.right, root2.right)



tree1 = Node(1)
tree1.left = Node(2)
tree1.right = Node(3)
```

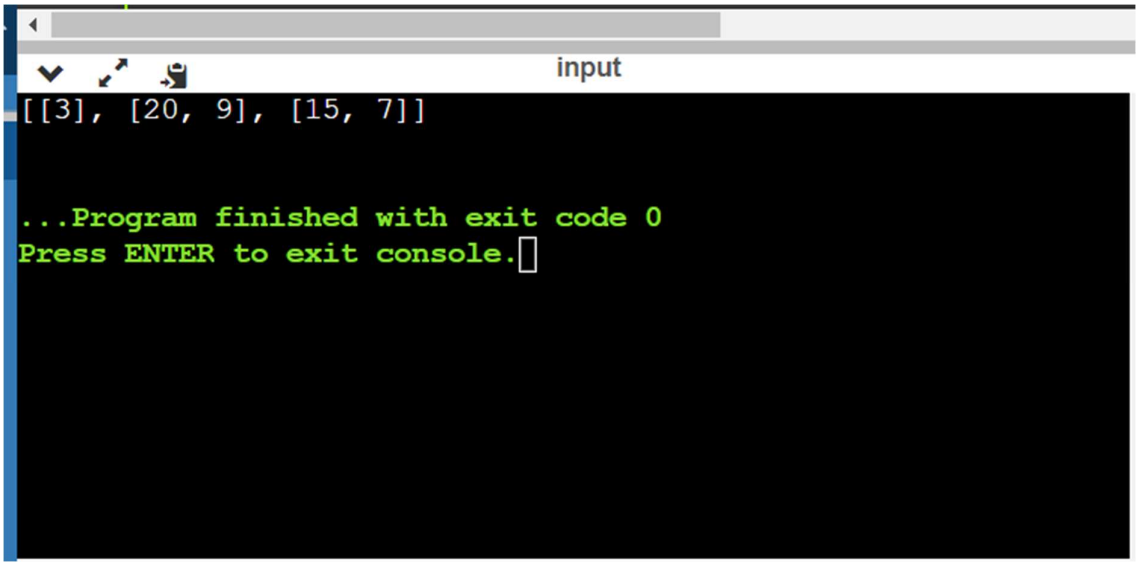
```
tree1.left.left = Node(4)
tree1.left.right = Node(5)

tree2 = Node(1)
tree2.left = Node(2)
tree2.right = Node(3)
tree2.left.left = Node(4)
tree2.left.right = Node(5)

if are_identical(tree1, tree2):
    print("The trees are identical.")
else:
    print("The trees are not identical.")
```



Problem Statement: Given the root of a binary tree, return the zigzag level order traversal of Binary Tree. (i.e., from left to right, then right to left for the next level and alternate between).

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



def zigzagLevelOrder(root):
    if not root:
        return []

    result = []
    queue = [root]
    level = 0

    while queue:
        level_values = []
        level_size = len(queue)

        for _ in range(level_size):
```

```
            node = queue.pop(0)
            level_values.append(node.val)

            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)

        if level % 2 == 1:
            level_values.reverse()

        result.append(level_values)
        level += 1

    return result

root = TreeNode(3)
root.left = TreeNode(9)
root.right = TreeNode(20)
root.right.left = TreeNode(15)
root.right.right = TreeNode(7)
result = zigzagLevelOrder(root)
print(result)
```


input
[[3], [20, 9], [15, 7]]
...Program finished with exit code 0
Press ENTER to exit console.

Problem Statement: BoundaryTraversal of a binary tree. Write a program for the Anti-Clockwise Boundary traversal of a binary tree.

```
class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None


def boundary_traversal(root):
    if not root:
        return

    def print_leaves(node):
        if node:
            print(node.data, end=" ")
            print_leaves(node.left)
            print_leaves(node.right)

    def print_left_boundary(node):
        if node:
            if node.left:
                print(node.data, end=" ")
                print_left_boundary(node.left)
```

```
            elif node.right:
                print(node.data, end=" ")
                print_left_boundary(node.right)

    def print_right_boundary(node):
        if node:
            if node.right:
                print_right_boundary(node.right)
                print(node.data, end=" ")
            elif node.left:
                print_right_boundary(node.left)
                print(node.data, end=" ")
    print(root.data, end=" ")
    print_left_boundary(root.left)
    print_leaves(root.left)
    print_leaves(root.right)
    print_right_boundary(root.right)

root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
```

root.left.right.left = Node(8)

root.left.right.left.left = Node(10)

root.right.left = Node(6)

root.right.right = Node(7)

root.right.right.left = Node(9)

boundary_traversal(root)