Editorial

X

Naive Approach

One of the naive approaches to solve this problem could be that

1. Check if tree with current node as root is a valid BST and if it is then return its size.
2. If tree with current node as its root is not a valid BST, then make recursive calls to left and right sub-trees and return the maximum of values returned by these recursive calls. Condition for valid BST could be checked by creating inorder array and checking if it is sorted.

The worst case time complexity of above algorithm is O(n^2) which occurs when the binary tree is skewed one with each node having only right child which is less than its parent.

C++ Code for Naive Algorithm

```
bool isBST(node *root, int minV = INT_MIN, int maxV = INT_MAX)
{
    if (root == NULL)
    {
        return true;
    }
    if (root->data >= minV && root->data <= maxV && isBST(root->left, mi
    {
        return true;
    }
    return false;
}

int count(node *root)
{
    if (root == NULL)
    {
        return 0;
    }
    return 1 + count(root->left) + count(root->right);
}

int sizeOfMaxBST(node *root)
{
    int maxSize = 0;
    if (root == NULL)
    {
        return 0;
    }
    if (isBST(root))
    {
        maxSize = max(maxSize, count(root));
    }
    return max(maxSize, max(sizeOfMaxBST(root->left), sizeOfMaxBST(root-
}
```

Java Code for Naive Approach

```
static boolean isBST(TreeNode root, int minV, int maxV) {
    if (root == null) {
        return true;
    }
    if (root.data >= minV && root.data <= maxV && isBST(root.left, minV,
            && isBST(root.right, root.data, maxV)) {
        return true;
    }
    return false;
}

static int count(TreeNode root) {
    if (root == null) {
        return 0;
    }
    return 1 + count(root.left) + count(root.right);
}

static int sizeOfMaxBST(TreeNode root) {
    int maxSize = 0;
    if (root == null) {
        return 0;
    }
    if (isBST(root, Integer.MIN_VALUE, Integer.MAX_VALUE)) {
        maxSize = Math.max(maxSize, count(root));
    }
    return Math.max(maxSize, Math.max(sizeOfMaxBST(root.left), sizeOfMax
}
```

Optimised Approach

Now let's look at the algorithm which runs in O(n) time. The basic idea used here is that for a tree with root as currentNode to be a valid BST, it's left and right sub-trees have to be valid BSTs, value of currentNode has to be greater than the maximum valued node from its left sub-tree and value of currentNode has to be less than the minimum valued node from its right sub-tree.

To check for above conditions, we return value of maximum valued node, minimum valued node and boolean variable stating if the tree is valid BST or not from every sub-tree. We also return size of a sub-tree if it is a valid BST or return -1 if it is not a valid BST. Using these returned variables from left and right sub-trees, we can check at the currentNode if the tree with its root as currentNode is a validBST or not. If the tree is a valid BST, we calculate size of tree using (1+sizeOfLeftSubtree +sizeOfRightSubtree) and return that along with other variables stated above. If it is not a valid BST we return -1 along with other variables.

C++ Code for Optimised Approach

```
#include <bits/stdc++.h>
using namespace std;

class node
{
public:
    int data;
    node *left;
    node *right;

    node(int d)
    {
        data = d;
        left = NULL;
        right = NULL;
    }
};

node *createTreeFromTrav(int *in, int *pre, int s, int e)
{
    static int i = 0;
    //Base Case
    if (s > e)
    {
        return NULL;
    }
    //Rec Case
    node *root = new node(pre[i]);

    int index = -1;
    for (int j = s; j <= e; j++)
    {
        if (in[j] == pre[i])
        {
            index = j;
            break;
        }
    }

    i++;
    root->left = createTreeFromTrav(in, pre, s, index - 1);
    root->right = createTreeFromTrav(in, pre, index + 1, e);
    return root;
}
class Info
{
public:
    bool is_bst;
    int maxBSTsize;
    int leftmax;
    int rightmin;
    Info()
    {
        is_bst = true;
        maxBSTsize = 0;
        leftmax = INT_MIN;
        rightmin = INT_MAX;
    }
};

Info greatestBSTinBT(node *root)
{

    if (root == NULL)
    {
        return Info();
    }

    Info left = greatestBSTinBT(root->left);
    Info right = greatestBSTinBT(root->right);
    Info ans;
    if (root->data >= left.leftmax && root->data < right.rightmin &&
        left.is_bst && right.is_bst)
    {
        ans.maxBSTsize = left.maxBSTsize + right.maxBSTsize + 1;
        ans.is_bst = true;
        ans.leftmax = max(left.leftmax, root->data);
        ans.rightmin = min(right.rightmin, root->data);
    }
    else
    {
        ans.is_bst = false;
        ans.maxBSTsize = max(left.maxBSTsize, right.maxBSTsize);
    }
    return ans;
}

int main()
{
    int n;
    cin >> n;

    int preOrder[10000], inOrder[10000];
    for (int i = 0; i < n; i++)
    {
        cin >> preOrder[i];
    }
    for (int i = 0; i < n; i++)
    {
        cin >> inOrder[i];
    }
    node *root = createTreeFromTrav(inOrder, preOrder, 0, n - 1);

    cout << greatestBSTinBT(root).maxBSTsize;
    return 0;
}
```

Java Code for Optimised Approach

```
import java.util.LinkedList;
import java.util.Scanner;

public class BinaryTree {

    private class Node {
        int data;
        Node left;
        Node right;

        Node(int data, Node left, Node right) {
            this.data = data;
            this.left = left;
            this.right = right;
        }
    }

    private Node root;
    private int size;

    public int size() {
        return this.size;
    }

    public boolean isempty() {
        return this.size == 0;
    }

    public BinaryTree() {
        Scanner scn = new Scanner(System.in);
        this.root = this.takeinput(scn, null, false);
    }

    private Node takeinput(Scanner scn, Node parent, boolean leftorright
        if (parent == null) {
            System.out.println("Enter the data for root");
        } else {
            if (leftorright) {
                System.out.println("Enter the data for left child of" +
            } else {
                System.out.println("Enter the data for right child of" +
            }
        }
        int cdata = scn.nextInt();
        Node child = new Node(cdata, null, null);
        this.size++;
        boolean choice = false;
        System.out.println("Do you want have left child for" + child.dat
        choice = scn.nextBoolean();
        if (choice) {
            child.left = this.takeinput(scn, child, true);
        }
        System.out.println("Do you have a right child");
        choice = scn.nextBoolean();
        if (choice) {
            child.right = this.takeinput(scn, child, false);
        }
        return child;
    }

    public void display() {
        this.display(this.root);
    }

    private void display(Node node) {
        if (node.left != null) {
            System.out.print(node.left.data + " =>");
        } else {
            System.out.print("END =>");
        }
        System.out.print(node.data + "<= ");
        if (node.right != null) {
            System.out.print(node.right.data);
        } else {
            System.out.print("END");
        }
        System.out.println();
        if (node.left != null) {
            this.display(node.left);
        }
        if (node.right != null) {
            this.display(node.right);
        }
    }

    /////////////////// New Constructor//////////

    public BinaryTree(int[] pre, int[] in) {
        // this.root = this.construct(pre, 0, pre.length - 1, in, 0, in.
        // 1);//for preorder
        this.root = this.construct(pre, in, 0, in.length - 1);// for
                                                              // posto

    }

    private static int preIndex = 0;

    private Node construct(int[] pre, int[] in, int isi, int iei) {
        if (isi > iei) {
            return null;
        }
        Node tNode = new Node(pre[preIndex++], null, null);

        if (isi == iei) {
            return tNode;
        }

        int inIndex = search(in, isi, iei, tNode.data);
        tNode.left = construct(pre, in, isi, inIndex-1);
        tNode.right = construct(pre, in, inIndex+1, iei);
        return tNode;

    }

    private int search(int[] arr, int si, int ei, int data) {
        for (int i = si; i <= ei; i++) {
            if (arr[i] == data)
                return i;
        }
        return -1;
    }

    class Info {
        int size;
        int max;
        int min;
        int ans;
        boolean isBST;

        Info() {

        }

        Info(int s, int max, int min, int ans, boolean isBST) {
            this.size = s;
            this.max = max;
            this.min = min;
            this.ans = ans;
            this.isBST = isBST;
        }
    }

    public int largestBSTinBT() {
        return this.largestBSTinBT(this.root).ans;
    }

    private Info largestBSTinBT(Node root) {
        if (root == null) {
            return new Info(0, Integer.MIN_VALUE, Integer.MAX_VALUE, 0,
        }
        if (root.left == null && root.right == null) {
            return new Info(1, root.data, root.data, 1, true);
        }

        Info l = largestBSTinBT(root.left);
        Info r = largestBSTinBT(root.right);

        Info ret = new Info();
        ret.size = (1 + l.size + r.size);

        if (l.isBST && r.isBST && l.max < root.data && r.min > root.data
            ret.min = Math.min(l.min, Math.min(r.min, root.data));
            ret.max = Math.max(r.max, Math.max(l.max, root.data));
            ret.ans = ret.size;
            ret.isBST = true;
            return ret;
        }

        ret.ans = Math.max(l.ans, r.ans);
        ret.isBST = false;
        return ret;
    }

    public static void main(String[] args) {
        Scanner scn = new Scanner(System.in);
        int n = scn.nextInt();
        int[] pre = new int[n];
        int[] in = new int[n];
        for (int i = 0; i < n; i++) {
            pre[i]=scn.nextInt();
        }
        for (int i = 0; i < n; i++) {
            in[i]=scn.nextInt();
        }

        BinaryTree bt = new BinaryTree(pre, in);
//        bt.display();
        System.out.println(bt.largestBSTinBT());
    }
}
```