<https://leetcode.com/problems/amount-of-time-for-binary-tree-to-be-infected/description/>

You are given the root of a binary tree with **unique** values, and an integer start. At minute 0, an **infection** starts from the node with value start.

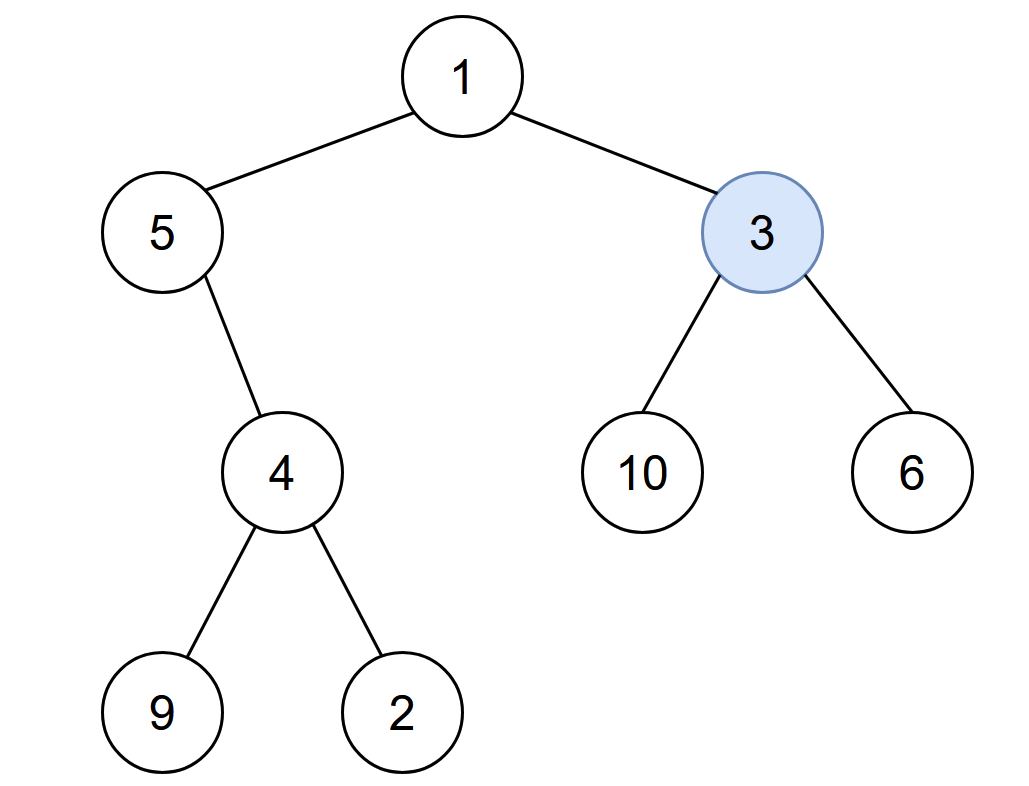
Each minute, a node becomes infected if:

The node is currently uninfected.

The node is adjacent to an infected node.

Return *the number of minutes needed for the entire tree to be infected.*

**Example 1:**



**Input:** root = [1,5,3,null,4,10,6,9,2], start = 3

**Output:** 4

**Explanation:** The following nodes are infected during:

- Minute 0: Node 3

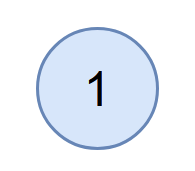
- Minute 1: Nodes 1, 10 and 6

- Minute 2: Node 5

- Minute 3: Node 4

- Minute 4: Nodes 9 and 2It takes 4 minutes for the whole tree to be infected so we return 4.

**Example 2:**



**Input:** root = [1], start = 1

**Output:** 0

**Explanation:** At minute 0, the only node in the tree is infected so we return 0.

**Constraints:**

The number of nodes in the tree is in the range [1, 105].

1 <= Node.val <= 105

Each node has a **unique** value.

A node with a value of start exists in the tree.

**Attempt 1: 2025-08-10**

**Solution 1: DFS + Hash Table (30 min)**

/\*\*

 \* Definition for a binary tree node.

 \* public class TreeNode {

 \*     int val;

 \*     TreeNode left;

 \*     TreeNode right;

 \*     TreeNode() {}

 \*     TreeNode(int val) { this.val = val; }

 \*     TreeNode(int val, TreeNode left, TreeNode right) {

 \*         this.val = val;

 \*         this.left = left;

 \*         this.right = right;

 \*     }

 \* }

 \*/

class Solution {

    int maxDepth = 0;

    public int amountOfTime(TreeNode root, int start) {

        Map<TreeNode, TreeNode> parentMap = new HashMap<>();

        buildParentMap(root, null, parentMap);

        Set<TreeNode> visited = new HashSet<>();

        TreeNode startNode = findStartNode(root, start);

        helper(startNode, 0, parentMap, visited);

        return maxDepth;

    }

    private void helper(TreeNode node, int curDepth, Map<TreeNode, TreeNode> parentMap,

        Set<TreeNode> visited) {

        if(node == null || visited.contains(node)) {

            return;

        }

        visited.add(node);

        maxDepth = Math.max(maxDepth, curDepth);

        // Recursively infect adjacent nodes with increased time

        helper(node.left, curDepth + 1, parentMap, visited);

        helper(node.right, curDepth + 1, parentMap, visited);

        helper(parentMap.get(node), curDepth + 1, parentMap, visited);

    }

    private TreeNode findStartNode(TreeNode node, int start) {

        if(node == null) {

            return null;

        }

        if(node.val == start) {

            return node;

        }

        TreeNode left = findStartNode(node.left, start);

        if(left != null) {

            return left;

        }

        return findStartNode(node.right, start);

    }

    private void buildParentMap(TreeNode node, TreeNode parent,

        Map<TreeNode, TreeNode> parentMap) {

        if(node == null) {

            return;

        }

        parentMap.put(node, parent);

        buildParentMap(node.left, node, parentMap);

        buildParentMap(node.right, node, parentMap);

    }

}

Time Complexity: O(n)

Space Complexity: O(n)

**Solution 2: One Pass DFS (60 min)**

/\*\*

 \* Definition for a binary tree node.

 \* public class TreeNode {

 \*     int val;

 \*     TreeNode left;

 \*     TreeNode right;

 \*     TreeNode() {}

 \*     TreeNode(int val) { this.val = val; }

 \*     TreeNode(int val, TreeNode left, TreeNode right) {

 \*         this.val = val;

 \*         this.left = left;

 \*         this.right = right;

 \*     }

 \* }

 \*/

class Solution {

    int maxTime = 0;

    public int amountOfTime(TreeNode root, int start) {

        helper(root, start);

        return maxTime;

    }

    /\*\*

    \*           3

    \*      /         \

    \*     2           1

    \*                  \

    \*                   4

    \*/

    private int helper(TreeNode node, int start) {

        if(node == null) {

            return 0;

        }

        int leftDepth = helper(node.left, start);

        int rightDepth = helper(node.right, start);

        // Case 1: Current node is the infection source (e.g node 4)

        if(node.val == start) {

            // Now we need the temporary 'maxTime' calculated

            // till root as 'infected' node (collected from

            // its left and right child branch)

            // e.g For node 4 already leave node, its left and

            // right subtree not exist, so the maxTime for node 4

            // is Math.max(0, 0) = 0

            maxTime = Math.max(leftDepth, rightDepth);

            // Mark as infected node

            return -1;

        }

        // Case 2: Neither subtree contains the infected node (e.g node 2)

        if(leftDepth >= 0 && rightDepth >= 0) {

            // Return as normal tree depth calculation

            return Math.max(leftDepth, rightDepth) + 1;

        }

        // Case 3: Now the remain case is current node has two

        // branches, one is infected branch, one is healthy branch

        // (e.g node 3), we deliberately assign negative depth for

        // infected branch earlier, now the distance between maximum

        // depth node on healthy branch node and infected node on

        // infected branch is the maximum time need to travel

        int infectedDepth = Math.min(leftDepth, rightDepth);

        int healthyDepth = Math.max(leftDepth, rightDepth);

        maxTime = Math.max(maxTime, healthyDepth - infectedDepth);

        // Propagate the infection depth upwards

        return infectedDepth - 1;

    }

}

Time Complexity: O(n)

Space Complexity: O(n)

**Solution 3: BFS + Hash Table (30 min)**

/\*\*

 \* Definition for a binary tree node.

 \* public class TreeNode {

 \*     int val;

 \*     TreeNode left;

 \*     TreeNode right;

 \*     TreeNode() {}

 \*     TreeNode(int val) { this.val = val; }

 \*     TreeNode(int val, TreeNode left, TreeNode right) {

 \*         this.val = val;

 \*         this.left = left;

 \*         this.right = right;

 \*     }

 \* }

 \*/

class Solution {

    public int amountOfTime(TreeNode root, int start) {

        Map<TreeNode, TreeNode> parentMap = new HashMap<>();

        buildParentMap(root, null, parentMap);

        Set<TreeNode> visited = new HashSet<>();

        Queue<TreeNode> queue = new LinkedList<>();

        TreeNode startNode = findStartNode(root, start);

        queue.offer(startNode);

        visited.add(startNode);

        // Initialize to -1 to account for level counting

        int depth = -1;

        while(!queue.isEmpty()) {

            int size = queue.size();

            for(int i = 0; i < size; i++) {

                TreeNode node = queue.poll();

                if(node.left != null && !visited.contains(node.left)) {

                    queue.offer(node.left);

                    visited.add(node.left);

                }

                if(node.right != null && !visited.contains(node.right)) {

                    queue.offer(node.right);

                    visited.add(node.right);

                }

                TreeNode parent = parentMap.get(node);

                if(parent != null && !visited.contains(parent)) {

                    queue.offer(parent);

                    visited.add(parent);

                }

            }

            depth++;

        }

        return depth;

    }

    private TreeNode findStartNode(TreeNode node, int start) {

        if(node == null) {

            return null;

        }

        if(node.val == start) {

            return node;

        }

        TreeNode left = findStartNode(node.left, start);

        if(left != null) {

            return left;

        }

        return findStartNode(node.right, start);

    }

    private void buildParentMap(TreeNode node, TreeNode parent,

        Map<TreeNode, TreeNode> parentMap) {

        if(node == null) {

            return;

        }

        parentMap.put(node, parent);

        buildParentMap(node.left, node, parentMap);

        buildParentMap(node.right, node, parentMap);

    }

}

Time Complexity: O(n)

Space Complexity: O(n)

**Refer to Deepseek**

**Approach**

To calculate the time it takes for the entire binary tree to get infected starting from an initial infected node:

**Build Parent Map**: Create a mapping from each node to its parent to enable upward traversal.

**BFS Traversal**: Perform a breadth-first search starting from the infected node to track the infection spread.

**Track Time**: Count the levels in BFS to determine the maximum time taken for complete infection.

**Solution Code**

class Solution {

public int amountOfTime(TreeNode root, int start) {

// Build parent map

Map<TreeNode, TreeNode> parentMap = new HashMap<>();

buildParentMap(root, null, parentMap);

// Find the start node

TreeNode startNode = findNode(root, start);

if (startNode == null) return 0;

// BFS setup

Queue<TreeNode> queue = new LinkedList<>();

Set<TreeNode> visited = new HashSet<>();

queue.offer(startNode);

visited.add(startNode);

int time = -1; // Initialize to -1 to account for level counting

while (!queue.isEmpty()) {

int size = queue.size();

time++;

for (int i = 0; i < size; i++) {

TreeNode current = queue.poll();

// Add left child

if (current.left != null && !visited.contains(current.left)) {

queue.offer(current.left);

visited.add(current.left);

}

// Add right child

if (current.right != null && !visited.contains(current.right)) {

queue.offer(current.right);

visited.add(current.right);

}

// Add parent

TreeNode parent = parentMap.get(current);

if (parent != null && !visited.contains(parent)) {

queue.offer(parent);

visited.add(parent);

}

}

}

return time;

}

private void buildParentMap(TreeNode node, TreeNode parent, Map<TreeNode, TreeNode> parentMap) {

if (node == null) return;

parentMap.put(node, parent);

buildParentMap(node.left, node, parentMap);

buildParentMap(node.right, node, parentMap);

}

private TreeNode findNode(TreeNode root, int val) {

if (root == null) return null;

if (root.val == val) return root;

TreeNode left = findNode(root.left, val);

if (left != null) return left;

return findNode(root.right, val);

}

}

**Explanation**

**Parent Mapping**: The buildParentMap function recursively creates a mapping of each node to its parent, enabling upward traversal from any node.

**Finding Start Node**: The findNode function locates the initially infected node in the tree.

**BFS Initialization**: Start BFS from the infected node, marking it as visited.

**Time Calculation**: Each level in BFS represents one minute of infection spread. The maximum level reached gives the total time for complete infection.

**Three-Way Expansion**: At each node, the infection spreads to:

Left child (if exists and uninfected)

Right child (if exists and uninfected)

Parent node (if exists and uninfected)

This approach efficiently tracks the infection spread in O(n) time (visiting each node once) and O(n) space (for parent map and queue), where n is the number of nodes in the tree.

**How about DFS solution ?**

Here's a Depth-First Search (DFS) approach to solve this problem, which provides an alternative to the BFS solution.

**DFS Approach**

**Build Parent Map**: Create a mapping from each node to its parent (same as BFS approach)

**DFS Traversal**: Perform a DFS starting from the infected node, tracking the current infection time

**Track Maximum Time**: Maintain a global maximum time variable that gets updated during the DFS

**Solution Code**

class Solution {

private int maxTime = 0;

public int amountOfTime(TreeNode root, int start) {

// Build parent map

Map<TreeNode, TreeNode> parentMap = new HashMap<>();

buildParentMap(root, null, parentMap);

// Find the start node

TreeNode startNode = findNode(root, start);

if (startNode == null) return 0;

// DFS traversal

Set<TreeNode> visited = new HashSet<>();

dfs(startNode, 0, parentMap, visited);

return maxTime;

}

private void dfs(TreeNode node, int currentTime,

Map<TreeNode, TreeNode> parentMap, Set<TreeNode> visited) {

if (node == null || visited.contains(node)) {

return;

}

visited.add(node);

maxTime = Math.max(maxTime, currentTime);

// Recursively infect adjacent nodes with increased time

dfs(node.left, currentTime + 1, parentMap, visited);

dfs(node.right, currentTime + 1, parentMap, visited);

TreeNode parent = parentMap.get(node);

dfs(parent, currentTime + 1, parentMap, visited);

}

private void buildParentMap(TreeNode node, TreeNode parent, Map<TreeNode, TreeNode> parentMap) {

if (node == null) return;

parentMap.put(node, parent);

buildParentMap(node.left, node, parentMap);

buildParentMap(node.right, node, parentMap);

}

private TreeNode findNode(TreeNode root, int val) {

if (root == null) return null;

if (root.val == val) return root;

TreeNode left = findNode(root.left, val);

if (left != null) return left;

return findNode(root.right, val);

}

}

**Explanation**

**Parent Mapping**: The buildParentMap function recursively creates a parent-child relationship map, same as in the BFS solution.

**Finding Start Node**: The findNode function locates the initially infected node in the tree.

**DFS Traversal**:

Start DFS from the infected node

Track visited nodes to avoid cycles

Increment time at each level of recursion

Update global maximum time when finding longer infection paths

**Three-Way Recursion**:

Infect left child with time+1

Infect right child with time+1

Infect parent (using the parent map) with time+1

Key differences from BFS:

Uses recursion stack instead of a queue

Tracks maximum depth (time) during traversal

Still maintains O(n) time complexity

Call stack depth could be O(n) in worst case (unbalanced tree)

This DFS solution provides the same result as BFS but with a different traversal order. The DFS approach might be more intuitive for some as it naturally tracks the depth/time through recursion.

**For findNode method below, what's the basic idea behind ? its a pre-order traversal or something ?**

The findNode method in the solution is indeed using a **pre-order traversal** approach to search for the target node in the binary tree. Here's a breakdown of how it works:

**Basic Idea:**

**Pre-order Traversal**:

Visits nodes in the order: **Root → Left subtree → Right subtree**

This is why we check root.val first, then recursively search the left subtree, and finally the right subtree.

**Termination Condition**:

If root is null, return null (base case for recursion).

If root.val matches the target value val, return root immediately (target found).

**Recursive Search**:

**Priority to Left Subtree**: First search the left subtree (findNode(root.left, val)).

If found (left != null), return the result immediately without searching the right subtree.

**Fallback to Right Subtree**: If not found in the left, search the right subtree (findNode(root.right, val)).

**Key Characteristics:**

**Efficiency**:

In the worst case (target is in the rightmost leaf), it visits all N nodes → **O(N) time**.

Space complexity is **O(H)** (height of tree) due to recursion stack.

**Short-Circuit Evaluation**:

The moment the target is found in the left subtree, the right subtree is **not searched**, saving unnecessary work.

**Example:**

For the tree below and val = 5:

1

/ \

2 3

/ \

4 5

Checks 1 (not 5) → recurse left to 2.

Checks 2 (not 5) → recurse left to 4.

Checks 4 (not 5) → left/right are null → backtrack to 2.

Now checks right subtree of 2 → finds 5 → returns immediately.

**Why Pre-order?**

**Natural Fit for Search**: Pre-order is intuitive for "find" operations because it checks the root first, which can terminate early if the root itself is the target.

**Explicit Priority**: By checking the left subtree before the right, it mirrors how one might visually scan a tree from left to right.

**Alternative Traversals:**

**In-order (Left → Root → Right)**: Would still work but might take longer if the target is in the right subtree.

**Post-order (Left → Right → Root)**: Least efficient for search as it checks leaves first.

**Summary:**

The findNode method is a **pre-order traversal** optimized with early termination. It efficiently searches for the target by prioritizing the root and left subtree before the right subtree, making it both simple and effective for this problem.

private TreeNode findNode(TreeNode root, int val) {

if (root == null) return null; // Base case: not found

if (root.val == val) return root; // Found at current node

TreeNode left = findNode(root.left, val); // Recurse left first

if (left != null) return left; // Short-circuit if found

return findNode(root.right, val); // Recurse right if not found left

}

**I have seen another solution called "one pass DFS" like below, can you try with similar way ?**

This approach is an elegant single-pass DFS solution that calculates the infection time while simultaneously determining whether the current subtree contains the infected node. Here's how it works:

**Key Concepts**

**Dual-Purpose Return Values**:

Positive return value: Maximum depth of a non-infected subtree

Negative return value: Depth of the infected node (converted to negative to distinguish)

**Three Scenarios**:

Current node is the infected node

Current subtree doesn't contain the infected node

Current subtree contains the infected node in one of its children

**Solution Code**

class Solution {

private int maxTime = 0;

public int amountOfTime(TreeNode root, int start) {

traverse(root, start);

return maxTime;

}

private int traverse(TreeNode node, int start) {

if (node == null) {

return 0;

}

int leftDepth = traverse(node.left, start);

int rightDepth = traverse(node.right, start);

if (node.val == start) {

// Current node is the infection source

maxTime = Math.max(leftDepth, rightDepth);

return -1; // Mark as infected

}

if (leftDepth >= 0 3333&& rightDepth >= 0) {

// Neither subtree contains the infected node

return Math.max(leftDepth, rightDepth) + 1;

}

// One subtree contains the infected node

int infectedDepth = Math.min(leftDepth, rightDepth); // The negative value

int healthyDepth = Math.max(leftDepth, rightDepth); // The positive value

maxTime = Math.max(maxTime, healthyDepth - infectedDepth);

return infectedDepth - 1; // Propagate the infection depth upwards

}

}

**Explanation**

**Base Case**: If node is null, return 0 (no depth).

**Recursive Calls**: Get depths from left and right subtrees.

**Infected Node Found**:

Set maxTime to the maximum depth of its subtrees (time to infect all nodes below)

Return -1 to mark this path as infected

**No Infection in Subtrees**:

Return normal maximum depth + 1 (for current node)

**Infection in One Subtree**:

Calculate the maximum infection time as the sum of:

Distance from infected node to current node (abs(infectedDepth))

Maximum depth of the healthy subtree

Propagate the infection depth upwards (decreasing by 1 each level)

**Why This Works**

The solution cleverly uses negative values to track the infection path while simultaneously calculating maximum depths in healthy subtrees

By comparing these values at each node, it can determine the longest infection path

The single traversal makes it very efficient with O(n) time complexity and O(h) space complexity (recursion stack)

**How to convert our earlier DFS solution to this one pass DFS solution ? Or how the converting logic behind ?**

**Converting Standard DFS to One-Pass DFS**

The key difference between the standard DFS and the one-pass DFS solution lies in how we track and propagate information about the infected node. Here's how to transform the standard approach into the more efficient one-pass version:

**Key Conversion Steps**

**Eliminate Parent Map**:

The one-pass solution doesn't need explicit parent tracking because it propagates infection information through return values

**Dual-Purpose Return Values**:

Instead of just tracking depth, return values now indicate both depth and infection status

Positive values: Maximum depth in healthy subtrees

Negative values: Infection path depth

**Single Traversal Logic**:

Combine the parent/child infection tracking into a single recursive pass

Calculate maximum infection time during traversal rather than after

**Conversion Process**

**Original DFS Components:**

Parent map construction

Separate infected node search

DFS with visited set tracking

Explicit three-way traversal (left, right, parent)

**One-Pass DFS Components:**

**Unified traversal** that handles both depth calculation and infection tracking

**Return value encoding** to communicate infection status

**Implicit parent tracking** through recursive returns

**Why This Conversion Works**

**Information Compression**:

Combines parent/child relationships into recursive returns

Uses sign of return value to indicate infection presence

**Time Calculation**:

When infection is found, calculates time to infect descendants

When propagating infection, calculates time to infect ancestors and other branches

**Efficiency Gain**:

Eliminates separate parent mapping step

Removes need for visited set

Performs all calculations in a single traversal

This conversion demonstrates how we can optimize tree traversal problems by creatively using return values to encode multiple pieces of information, reducing the need for auxiliary data structures.

**Can you show an example for how One-Pass DFS works ?**

The one-pass DFS solution efficiently calculates the infection time by propagating infection depth information through recursive returns. Here's the solution with an example walkthrough:

**Solution Code**

**Example Walkthrough**

**Tree Structure:**

3

/ \

2 1

\

4

**Infection Start:** Node 4

**Expected Output:** 3 (minutes)

**Step-by-Step Execution:**

**Traverse Node 3 (Root):**

Recursively process left child (Node 2)

Recursively process right child (Node 1)

**Traverse Node 2:**

Left child: null → returns 0

Right child: null → returns 0

Node 2 ≠ 4 → return max(0, 0) + 1 = 1

**Traverse Node 1:**

Left child: null → returns 0

Recursively process right child (Node 4)

**Traverse Node 4:**

Node 4 is infection source:

maxTime = max(0, 0) = 0

Return -1

**Back to Node 1:**

leftDepth = 0 (non-negative)

rightDepth = -1 (negative)

One subtree contains infection:

infectedDepth = min(0, -1) = -1

healthyDepth = max(0, -1) = 0

maxTime = max(0, 0 - (-1)) = max(0, 1) = 1

Return -1 - 1 = -2

**Back to Node 3 (Root):**

leftDepth = 1 (from Node 2)

rightDepth = -2 (from Node 1)

One subtree contains infection:

infectedDepth = min(1, -2) = -2

healthyDepth = max(1, -2) = 1

maxTime = max(1, 1 - (-2)) = max(1, 3) = 3

Return -2 - 1 = -3

**Result:** maxTime = 3

**Key Concepts**

**Return Values:**

**Positive:** Maximum depth of a non-infected subtree

**Negative:** Depth of infection path (converted to negative to distinguish)

**Three Cases:**

Infected Node Found:

Update maxTime with maximum subtree depth

Return -1 to mark infection path

No Infection in Subtrees:

Return maximum subtree depth + 1

Infection in One Subtree:

Calculate potential infection time: healthyDepth - infectedDepth

Propagate infection depth: infectedDepth - 1

**Efficiency:**

Time Complexity: O(n) - Single tree traversal

Space Complexity: O(h) - Recursion stack height (h = tree height)

This approach efficiently calculates infection time by propagating infection depth information while simultaneously tracking the maximum infection time encountered during traversal.

**To fully understand the One Pass process, its better to go through the debugging workflow:**

import java.util.\*;

public class TreeSolution {

class TreeNode {

int val;

TreeNode left;

TreeNode right;

TreeNode() {}

TreeNode(int val) { this.val = val; }

TreeNode(int val, TreeNode left, TreeNode right) {

this.val = val;

this.left = left;

this.right = right;

}

}

private int maxTime = 0;

public int amountOfTime(TreeNode root, int start) {

traverse(root, start);

return maxTime;

}

private int traverse(TreeNode node, int start) {

if (node == null) {

return 0;

}

int leftDepth = traverse(node.left, start);

int rightDepth = traverse(node.right, start);

if (node.val == start) {

// Current node is the infection source

maxTime = Math.max(leftDepth, rightDepth);

return -1; // Mark as infected

}

if (leftDepth >= 0 && rightDepth >= 0) {

// Neither subtree contains the infected node

return Math.max(leftDepth, rightDepth) + 1;

}

// One subtree contains the infected node

int infectedDepth = Math.min(leftDepth, rightDepth); // The negative value

int healthyDepth = Math.max(leftDepth, rightDepth); // The positive value

maxTime = Math.max(maxTime, healthyDepth - infectedDepth);

return infectedDepth - 1; // Propagate the infection depth upwards

}

public static void main(String[] args) {

/\*\*

\* 3

\* / \

\* 2 1

\* \

\* 4

\*/

TreeSolution so = new TreeSolution();

TreeNode one = so.new TreeNode(1);

TreeNode two = so.new TreeNode(2);

TreeNode three = so.new TreeNode(3);

TreeNode four = so.new TreeNode(4);

three.left = two;

three.right = one;

one.right = four;

int result = so.amountOfTime(three, 4);

System.out.println(result);

}

}

**Refer to**

[L863.All Nodes Distance K in Binary Tree (Ref.L2385)](note://WEB5cbc9ab83d04acc976ba0ed5cbb8ff07)