# KONGU ENGINEERING COLLEGE

(Autonomous)

Perundurai,Erode – 638060

## DEPARTMENT OF INFORMATION TECHNOLOGY

# RECURSIVE HEIGHT CALCULATION OF A BINARY TREE USING DIVIDE AND CONQUER ALGORITHM

## A MICRO PROJECT REPORT

## FOR

## DESIGN AND ANALYSIS OF ALGORITHMS(22ITT31)


**SUBMITTED BY**

**SAMBAVI S (23ITR141)**

# KONGU ENGINEERING COLLEGE

(Autonomous)

Perundurai,Erode – 638060

## DEPARTMENT OF INFORMATION TECHNOLOGY

# RECURSIVE HEIGHT CALCULATION OF A BINARY TREE USING DIVIDE AND CONQUER ALGORITHM

## A MICRO PROJECT REPORT

## FOR

## DESIGN AND ANALYSIS OF ALGORITHMS(22ITT31)


**SUBMITTED BY**

**SAMBAVI S (23ITR141)**

# KONGU ENGINEERING COLLEGE

**(Autonomous)**

Perundurai,Erode – 638060

## DEPARTMENT OF INFORMATION TECHNOLOGY

## BONAFIDE CERTIFICATE

| | |
|---|---|
| Name | : SAMBAVI S |
| Course Code | : 22ITT31 |
| Course Name | : DESIGN AND ANALYSIS OF ALGORITHMS |
| Semester | : IV |

Certified that this is a bonafide record of work for application project done by the above student for 22ITT31-DESIGN AND ANALYSIS OF ALGORITHMS during the academic year 2024-2025.

Submitted for the Viva Voice Examination held on _____

Faculty Incharge Head of the Department

## ABSTRACT

This project presents the design and implementation of a divide-and-conquer algorithm to compute the number of levels in a binary tree. The algorithm operates recursively by dividing the tree into its left and right subtrees, computing the height (or depth) of each, and combining the results by taking the maximum of the two and adding one to account for the current root level. Special attention is given to the base cases: the algorithm returns 0 for an empty tree and 1 for a single-node tree. This divide-and-conquer strategy ensures a clear and efficient approach to navigating and processing tree structures. The time efficiency class of the algorithm is **O(n)**, where *n* is the number of nodes in the tree, as each node is visited exactly once during the traversal. This method not only simplifies the problem-solving process but also demonstrates the power and clarity of recursive thinking in structural computations within computer science.

# TABLE OF CONTENTS

## 1.0 INTRODUCTION

Determining the structure and depth of data is a fundamental task in computer science, especially in tree-based data structures. One such task is computing the number of levels (or height) in a binary tree, which represents the maximum depth from the root to any leaf node. This problem can be efficiently solved using a **Divide and Conquer** approach.

In this project, we design and implement a divide-and-conquer algorithm to compute the number of levels in a binary tree. The core idea is to recursively divide the tree into its left and right subtrees, calculate their individual depths, and then combine the results by taking the maximum of the two and adding one (for the current level). This strategy efficiently reduces the problem size at each step and ensures that every node is visited only once, making the algorithm optimal.

Compared to brute-force or iterative methods, the divide-and-conquer technique offers clearer logic and better performance for large and complex binary trees. The project also includes an interactive visualization to help users understand how the algorithm traverses the tree and computes the level count step by step. This educational tool aims to enhance comprehension of recursive problem-solving and demonstrate the practical utility of divide-and-conquer algorithms.

## 1.1PURPOSE

The main purpose of this project is:

- Make complex algorithmic concepts more accessible and understandable, especially for students and learners.
- Highlight the efficiency and elegance of divide and conquer techniques in reducing problem-solving time and complexity.
- Encourage logical thinking, problem decomposition, and strategic decision-making through an interactive platform.
- Serve as a learning aid for computer science education, particularly in the areas of algorithms and problem-solving strategies.

## 1.2OBJECTIVE

The main objective of this project is to design and implement an efficient algorithm to compute the number of levels (height) in a binary tree using the Divide and Conquer strategy. The specific goals include:

- To analyze binary tree structures and determine their depth in an optimal, recursive manner.
- To apply the divide and conquer approach to break the problem into smaller subtrees and combine their results efficiently.
- To reduce computational overhead by avoiding redundant tree traversal and ensuring logarithmic depth exploration.
- To enhance understanding of recursive algorithm design through a clear and structured implementation.
- To demonstrate the effectiveness of divide and conquer techniques in solving hierarchical data problems in computer science.

## 1.3 METHODOLOGY OVERVIEW

The project follows a structured approach to determine the number of levels in a binary tree using the Divide and Conquer methodology. The steps involved in the implementation are:

1. **Input&Initialization**:
   The algorithm receives the root node of a binary tree as input. If the tree is empty (i.e., the root is null), it immediately returns 0, indicating no levels.
2. **RecursiveDivision**:
   The binary tree is recursively divided into left and right subtrees. The algorithm computes the number of levels in each subtree independently using recursive calls.
3. **Comparison&Combination**:
   Once the left and right subtree heights are calculated, the algorithm takes the maximum of the two and adds 1 (to include the current level), effectively merging the sub-solutions into the overall height.
4. **BaseCase**:
   The recursion terminates when a null node is encountered, which represents an empty subtree and contributes 0 to the height.
5. **ResultGeneration**:
   The final output is the height of the binary tree, which equals the number of levels from the root to the deepest leaf node.
6. **Visualization(Optional)**:
   For educational purposes, the process can be visualized step-by-step, showing how the tree is split, levels are calculated, and merged back, illustrating the efficiency of the divide and conquer strategy.

## 2. PROBLEM STATEMENT

Determining the number of levels in a binary tree is a fundamental problem in computer science, often encountered in tree traversal and structural analysis. The challenge lies in accurately computing the height of the tree—defined as the number of levels from the root node down to the deepest leaf—using an efficient and recursive approach. For an empty tree, the level count should return 0, and for a single-node tree, it should return 1.

This project focuses on implementing a Divide and Conquer algorithm that recursively breaks the tree into its left and right subtrees, computes their respective heights, and combines the results by selecting the maximum and adding one. This recursive strategy ensures that each subtree is processed only once, leading to a time-efficient and elegant solution. The approach demonstrates how divide and conquer can simplify complex hierarchical data problems like tree depth calculation.

## 3.0 Methodology for Computing the Number of Levels in a Binary Tree

### 3.1 Input & Initialization

- Accept or generate a binary tree structure, which could be empty, have a single node, or be more complex.
- Initialize the root node and define left and right child nodes recursively.
- Prepare the data structure (e.g., a class or object representation) to store nodes and their relationships.
- Ensure input validation for malformed or incomplete trees.

### 3.2 Divide & Compare

- Use a recursive function to divide the binary tree into left and right subtrees at each node.
- For each node, recursively calculate the height of the left subtree and the right subtree.
- Use a comparison: take the maximum height of the two subtrees and add 1 to account for the current level.

### 3.3 Recursive Detection

- The recursive logic continues until a base case is reached (i.e., the node is null), which returns a height of 0.
- If the current node is a leaf (no children), return 1 as its level.
- Propagate the calculated height upward through the recursive calls to determine the overall tree height (number of levels).

### 3.4 Visualization & Output

- Optionally visualize the binary tree structure and highlight how the recursive calls are made at each node.
- After computation, display:
    - The total number of levels in the binary tree.
    - If the tree is empty, return 0; if it contains only a root node, return 1.
    - Optionally show intermediate steps, subtree heights, and recursion depth for better understanding.

## IMPLEMENTATION :

### 4.1 Input & Initialization

```
class TreeNode {
constructor(
public value: number,
public left: TreeNode | null = null,
public right: TreeNode | null = null
) {}
const root = new TreeNode(1);
root.left = new TreeNode(2);
root.right = new TreeNode(3);
root.left.left = new TreeNode(4);
root.left.right = new TreeNode(5);
```

**4.2 Divide & Compare**

```
class TreeNode {
value: number;
left: TreeNode | null;
right: TreeNode | null;
constructor(value: number) {
this.value = value;
this.left = null;
this.right = null;
}
}
function computeLevels(node: TreeNode | null): number {
if (node === null) {
return 0;
}
const leftLevels = computeLevels(node.left);
const rightLevels = computeLevels(node.right);
return 1 + Math.max(leftLevels, rightLevels);
}
```

**4.3 Breadth-First Search**

```
function computeLevelsBFS(root) {
if (root === null) {
return 0;
}
```

```
let queue = [root];
let levels = 0;
while (queue.length > 0) {
let levelSize = queue.length;  // Number of nodes at current level
for (let i = 0; i < levelSize; i++) {
let currentNode = queue.shift();
if (currentNode.left !== null) {
queue.push(currentNode.left);
}
if (currentNode.right !== null) {
queue.push(currentNode.right);
}
}
levels++;
}
return levels;
}
```

**4.4 Visualization & Output**

```
const levels = computeLevels(root);
console.log("Level Count Computation Complete!");
console.log(`The number of levels in the binary tree is: ${levels}`);
```

## DIFFERENCE BETWEEN DIVIDE AND CONQUER AND BREADTH FIRST SEARCH (BFS):

**Divide and Conquer:**

**Concept:**

- Breaks the problem into smaller subproblems, solves each subproblem independently, and combines their results.
- Works recursively by dividing the input data into parts, conquering each part, and merging the results.

**How it works:**

1. Divide the binary tree into left and right subtrees.
2. Recursively compute the number of levels in each subtree.
3. Combine the results by taking the maximum level from both sides and adding one for the root node.

**Time Complexity:**

- O(n), where n is the number of nodes, because every node is visited once during recursion.

**Breadth First Search (BFS):**

**Concept:**

- Traverses the binary tree level by level, visiting all nodes at the current depth before moving to the next level.
- Uses a queue data structure to keep track of nodes at each level.

**How it works:**

1. Start from the root node and enqueue it.
2. While the queue is not empty, dequeue a node, and enqueue its children.
3. Keep track of the number of levels traversed until all nodes are visited.

**Time Complexity:**

- O(n), where n is the number of nodes, because each node is visited once.

**Pros:**

- Efficient and fast due to recursive problem breakdown.
- Well-suited for hierarchical or tree-based structures.
- Reduces redundant computations when subproblems overlap.

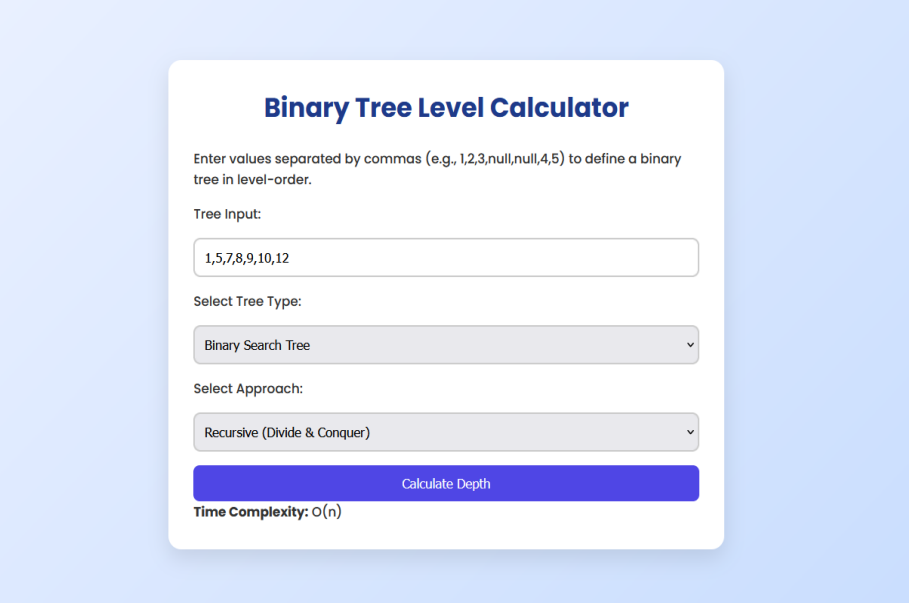
**Cons:**

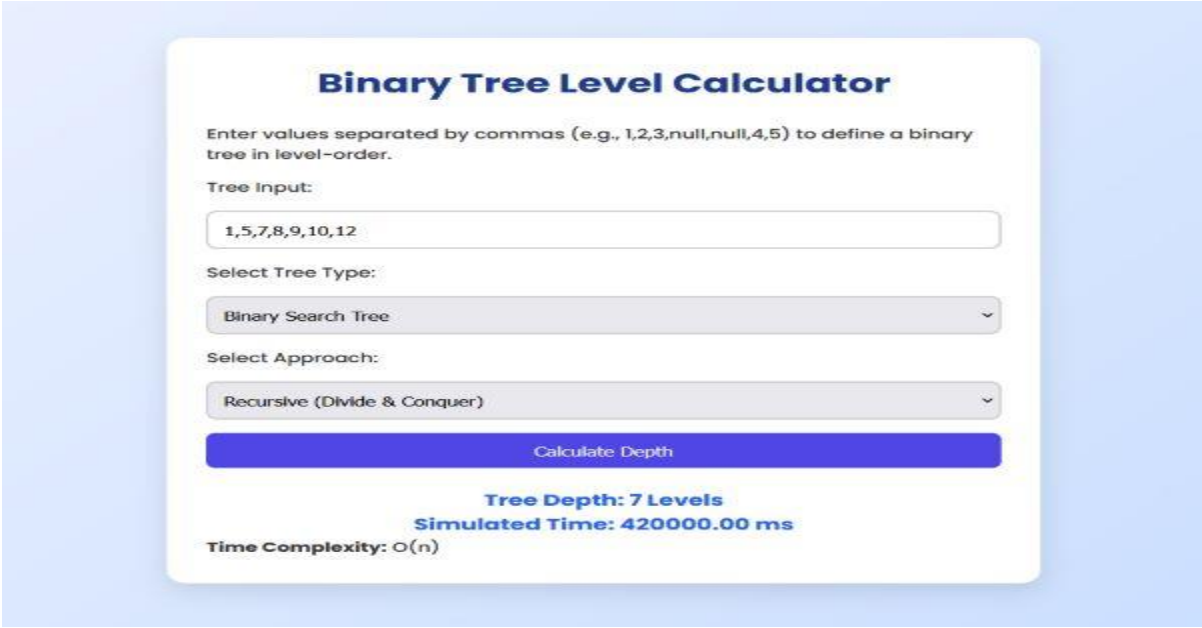
- Logic can become complex and harder to debug.
- Requires careful base case handling and correct merging of results.

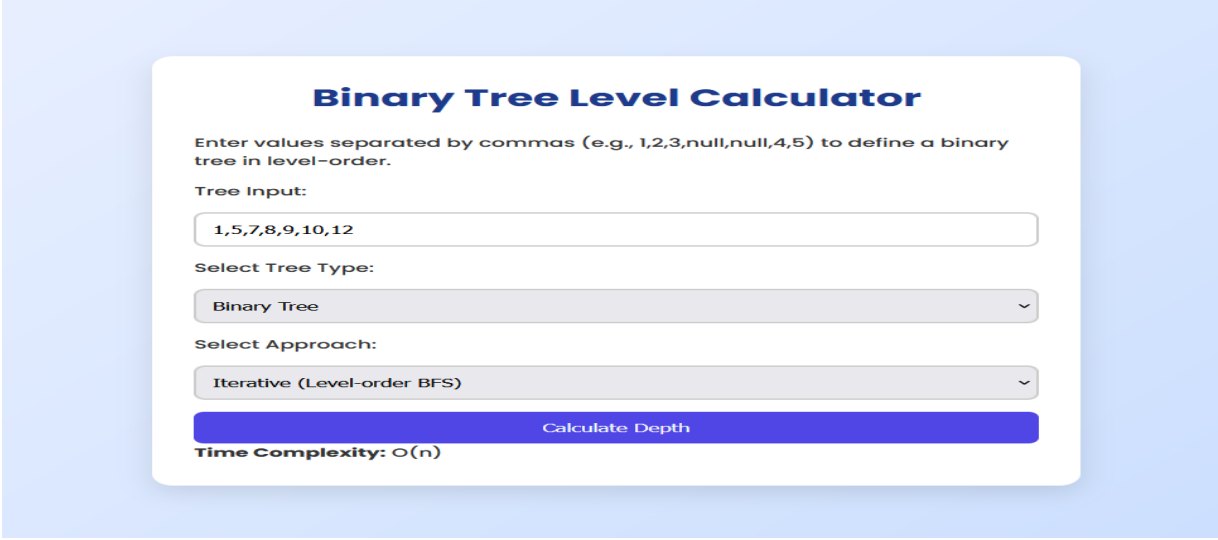
| **Feature** | **Divide and Conquer** | **Breadth First Search (BFS)** |
|---|---|---|
| Strategy | Recursive subdivision | Level-order traversal |
| Time Complexity | O(log n) (for balanced problems) | O(n) (visits every node once) |
| Efficiency | High for divide-friendly problems | Moderate, can be memory-intensive |

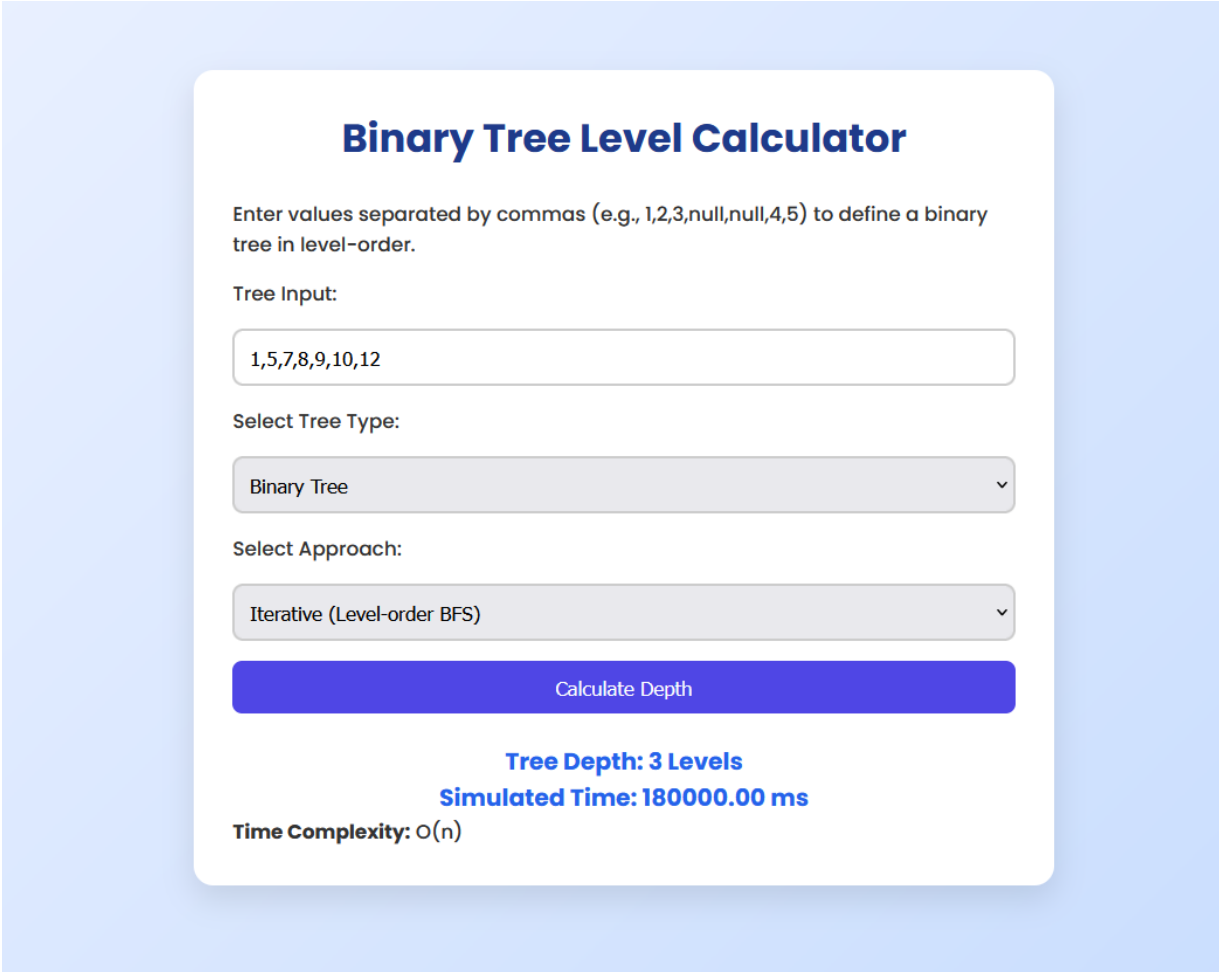
| Feature | Divide and Conquer | Breadth First Search (BFS) |
|---|---|---|
| Ideal for | Tree depth, search, optimization | Shortest path, level discovery |
| Use of Resources | Uses call stack efficiently | Requires queue for level tracking |
| Logic Complexity | Moderate to High | Simple and easy to implement |

## 5.0. RESULTS:




Binary Tree Level Calculator
Enter values separated by commas (e.g., 1,2,3,null,null,4,5) to define a binary tree in level-order.
Tree Input:
1,5,7,8,9,10,12
Select Tree Type:
Binary Search Tree
Select Approach:
Recursive (Divide & Conquer)
Calculate Depth
Tree Depth: 7 Levels
Simulated Time: 420000.00 ms
Time Complexity: O(n)


Binary Tree Level Calculator
Enter values separated by commas (e.g., 1,2,3,null,null,4,5) to define a binary tree in level-order.
Tree Input:
1,5,7,8,9,10,12
Select Tree Type:
Binary Tree
Select Approach:
Iterative (Level-order BFS)
Calculate Depth
Time Complexity: O(n)




**GITHUB LINK: https://github.com/sambavi29/daa**