Experiment 3 A

Student Name: Sachin **UID: 22BCS15411**

Branch: CSE **Section/Group: Ntpp 602-A**

Semester: 6TH **Date of Performance:03/02/25**

Subject Name: AP Lab-2 **Subject Code: 22CSH-352**

1. TITLE:

Maximum Depth of Binary Tree

2. AIM:

Given the root of a binary tree, return its maximum depth.

A binary tree's maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

3. Algorithm

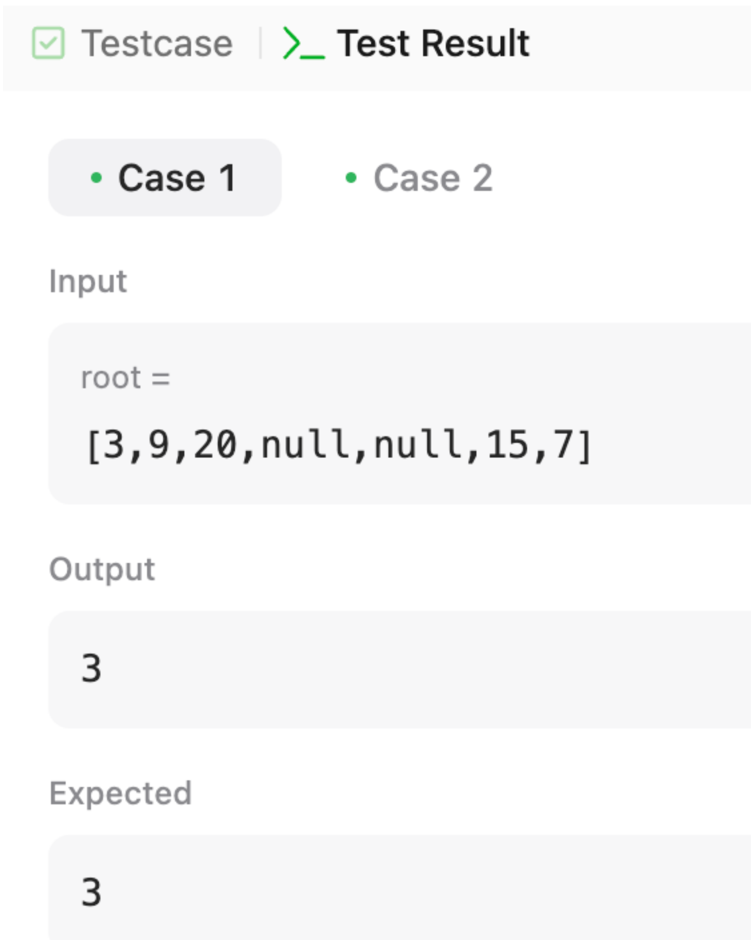
- Start DFS with the root node at depth 0.
- If the node is null, return the current depth.
- Recursively explore left and right children, increasing depth by 1.
- Return the maximum depth from left or right subtree.

Implemetation/Code

```
      class Solution {
public:
    int maxDepth(TreeNode* root) {
        if (root == nullptr) {
            return 0;
        } else {
            int leftDepth = maxDepth(root->left);
            int rightDepth = maxDepth(root->right);
            return 1 + std::max(leftDepth, rightDepth);
```

```
        }
    }
};
```

Output



Time Complexity : O(n)

Space Complexity : O(h)

Learning Outcomes:-

- Understand how to use depth-first search for tree traversal.
- Gain skills in calculating the depth or height of binary trees

Experiment 3 B

Student Name: Sachin **UID: 22BCS15411**

Branch: CSE **Section/Group: Ntpp 602-A**

Semester: 6TH **Date of Performance:03/02/25**

Subject Name: AP Lab-2 **Subject Code: 22CSH-352**

1. TITLE:

KTH Smallest Element in a BST

2. AIM:

Given the root of a binary search tree, and an integer k, return *the* kth *smallest value* ***(1-indexed)*** *of all the values of the nodes in the tree.*

3. Algorithm

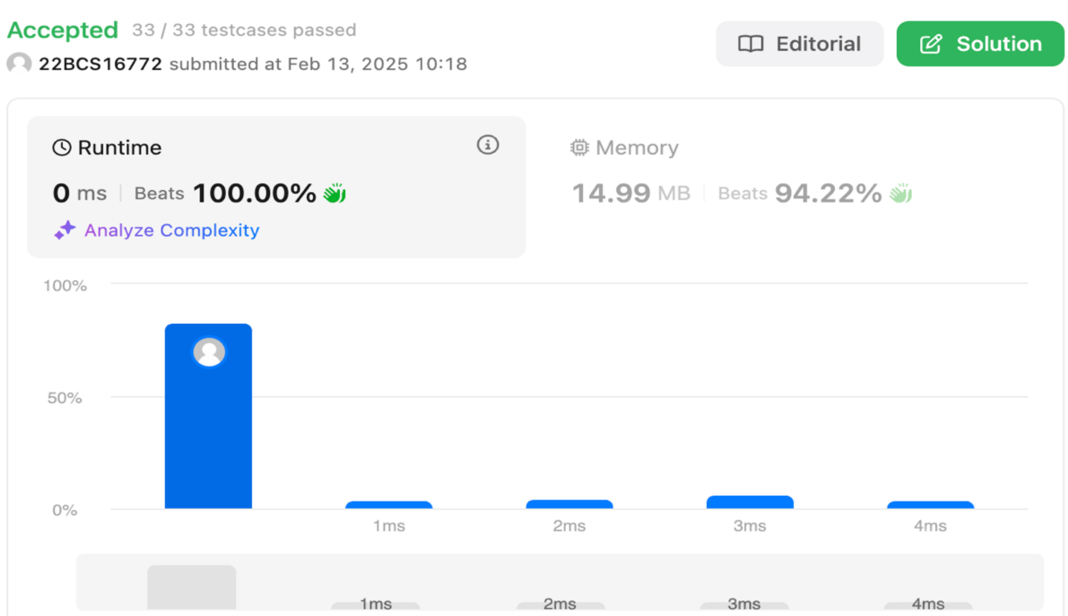
- Perform an in-order traversal of the binary tree starting from the root.
- Use a generator to yield nodes' values one by one in their in-order sequence.
- Iterate up to the kth element of the generator.
- Return the kth smallest element from the traversal.

Implemetation/Code:

```
class Solution {
public:
    std::vector<std::vector<int>> zigzagLevelOrder(TreeNode* root) {
        std::vector<std::vector<int>> result;
        if (root == nullptr) {
            return result;
        }

        std::queue<TreeNode*> q;
        q.push(root);
        bool leftToRight = true;

        while (!q.empty()) {
            int levelSize = q.size();
            std::vector<int> currentLevel(levelSize);

            for (int i = 0; i < levelSize; ++i) {
```

```
                TreeNode* node = q.front();
                q.pop();

                // Store the value in the correct order
                if (leftToRight) {
                    currentLevel[i] = node->val;
                } else {
                    currentLevel[levelSize - 1 - i] = node->val; // Reverse order
                }

                if (node->left) {
                    q.push(node->left);
                }
                if (node->right) {
                    q.push(node->right);
                }
            }
            result.push_back(currentLevel);
            leftToRight = !leftToRight; // Toggle direction for the next level
        }
        return result;
    }
};
```

Output



Time Complexity : O(k)

Space Complexity : O(h)

Learning Outcomes:-

- Learn how to perform and apply in-order traversal in binary trees to solve problems.
- Python generators to manage state and produce results on demand during tree traversal.

Experiment 3 C

Student Name: Sachin **UID: 22BCS15411**

Branch: CSE **Section/Group:NTPP-602-A**

Semester: 5 **Date of Performance:11/10/24**

Subject Name: AP Lab **Subject Code: 22CSH-311**

1. **TITLE: Fibonacci Numbers**

2. **AIM:** The Fibonacci sequence appears in nature all around us, in the arrangement of seeds in a sunflower and the spiral of a nautilus for example. The Fibonacci sequence begins with Fibonacci (0)=0 and Fibonacci(1)=1 and as its first and second terms. After these first two elements, each subsequent element is equal to the sum of the previous two elements

3. **Algorithm**
   - Base Case Check: If n is 1, return 1 as the first Fibonacci number.
   - Initialize Variables: Set a and b to 1, representing the first two Fibonacci numbers.
   - Iterate Through Positions: Loop from position 2 to n-1.
   - Update Fibonacci Values: In each iteration, update a to b and b to a + b.
   - Return Result: After completing the loop, return a as the n-th Fibonacci number.

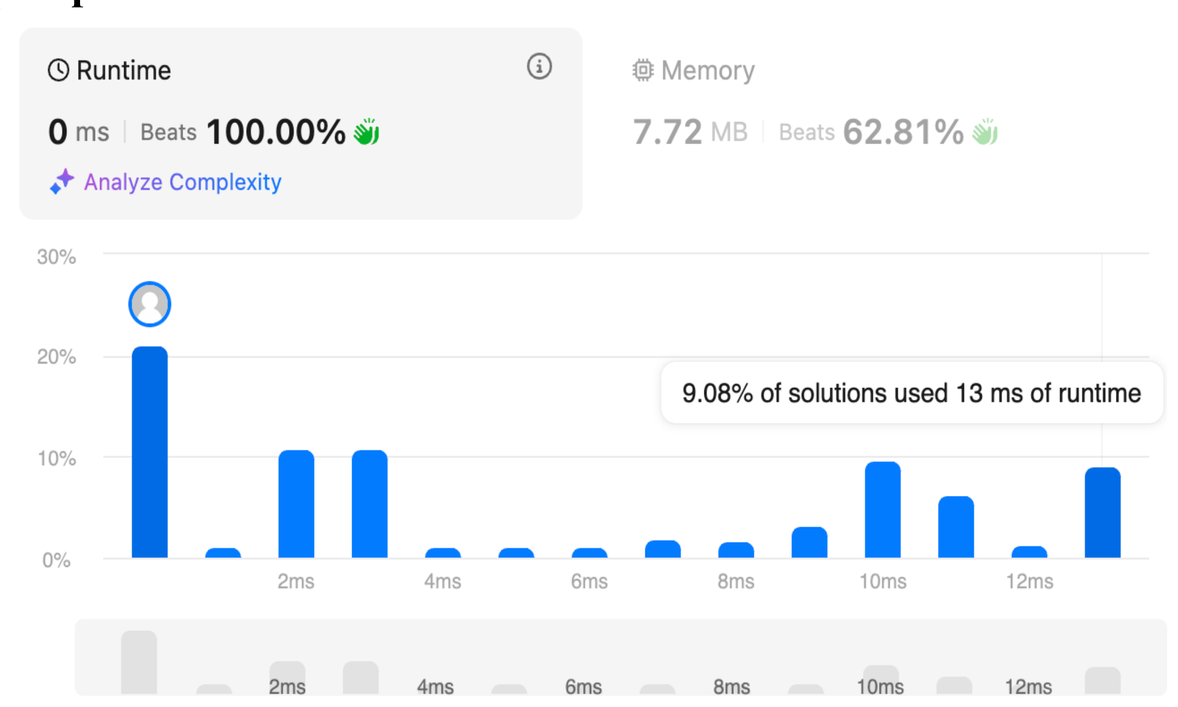
4. **Implemetation/Code**

```
class Solution {
public:
    int fib(int n) {
        if (n <= 1) {
            return n;
        }

        // Efficient iterative approach (using only two variables)
        int a = 0; // F(0)
        int b = 1; // F(1)
        for (int i = 2; i <= n; ++i) {
            int temp = a + b; // Calculate F(i)
            a = b;           // Update F(i-2)
            b = temp;          // Update F(i-1)
```

```
        }
        return b; // F(n) is now stored in b
    }
};
```

Time Complexity : O(N)
Space Complexity : O(1)

5. Output



6. Learning Outcomes:-

1. Understand how to compute Fibonacci numbers using an iterative approach.
2. Learn to optimize space by maintaining only two variables instead of using an entire array.
3. Gain proficiency in implementing efficient loop constructs for sequential computations.