# WORKSHEET 8

**Student Name:** Mohit
**UID:** 22BCS50051
**Branch:** BE-CSE
**Section/Group:** 22BCS_NTPP-602-A
**Semester:** 6th
**Date of Performance:** 17/03/2025
**Subject Name:** AP LAB - II
**Subject Code:** 22CSP-351

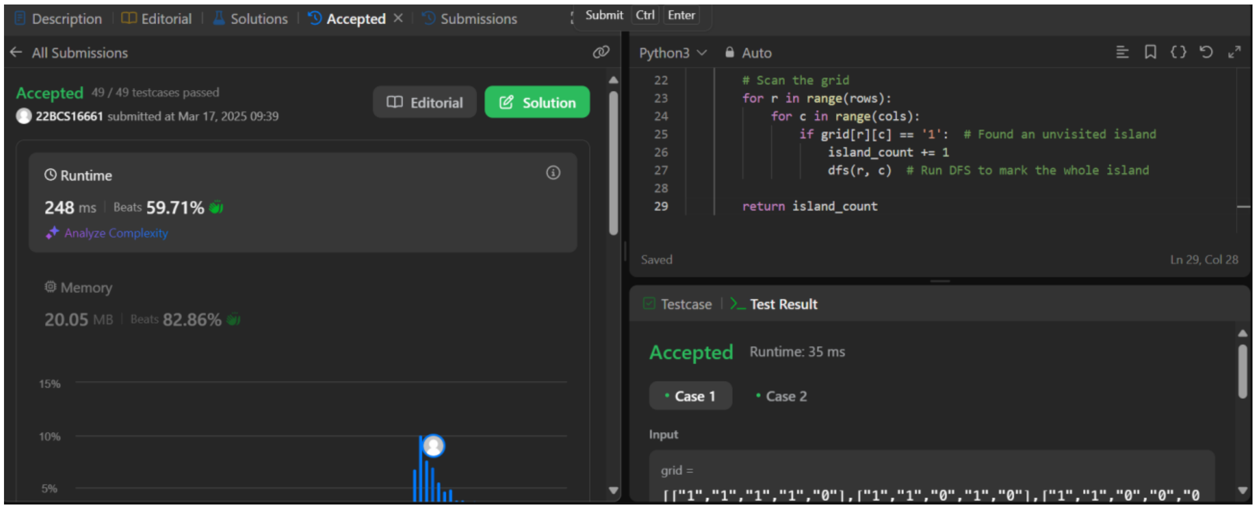
1. **Aim:** Given an m x n 2D binary grid grid which represents a map of '1's (land) and '0's (water), return *the number of islands*.
   An **island** is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

2. **Source Code:**

```
from typing import List

class Solution:
    def numIslands(self, grid: List[List[str]]) -> int:
        if not grid:
            return 0

        rows, cols = len(grid), len(grid[0])
        island_count = 0

        def dfs(r, c):
            """Mark all connected land ('1') as visited ('0') using DFS"""
            if 0 <= r < rows and 0 <= c < cols and grid[r][c] == '1':
                grid[r][c] = '0'  # Mark as visited

                # Explore all four directions
                dfs(r + 1, c)  # Down
                dfs(r - 1, c)  # Up
                dfs(r, c + 1)  # Right
                dfs(r, c - 1)  # Left

        # Scan the grid
        for r in range(rows):
```

```
        for c in range(cols):
            if grid[r][c] == '1':  # Found an unvisited island
                island_count += 1
                dfs(r, c)  # Run DFS to mark the whole island

    return island_count
```

## 3. Screenshots of outputs:



Accepted 49 / 49 testcases passed
22BCS16661 submitted at Mar 17, 2025 09:39

Runtime
248 ms | Beats 59.71%

Memory
20.05 MB | Beats 82.86%

```
        # Scan the grid
        for r in range(rows):
            for c in range(cols):
                if grid[r][c] == '1':  # Found an unvisited island
                    island_count += 1
                    dfs(r, c)  # Run DFS to mark the whole island

        return island_count
```

Saved    Ln 29, Col 28

Accepted Runtime: 35 ms

Case 1    Case 2

Input

```
grid =
[["1","1","1","1","0"],["1","1","0","1","0"],["1","1","0","0","0
```

## 2.

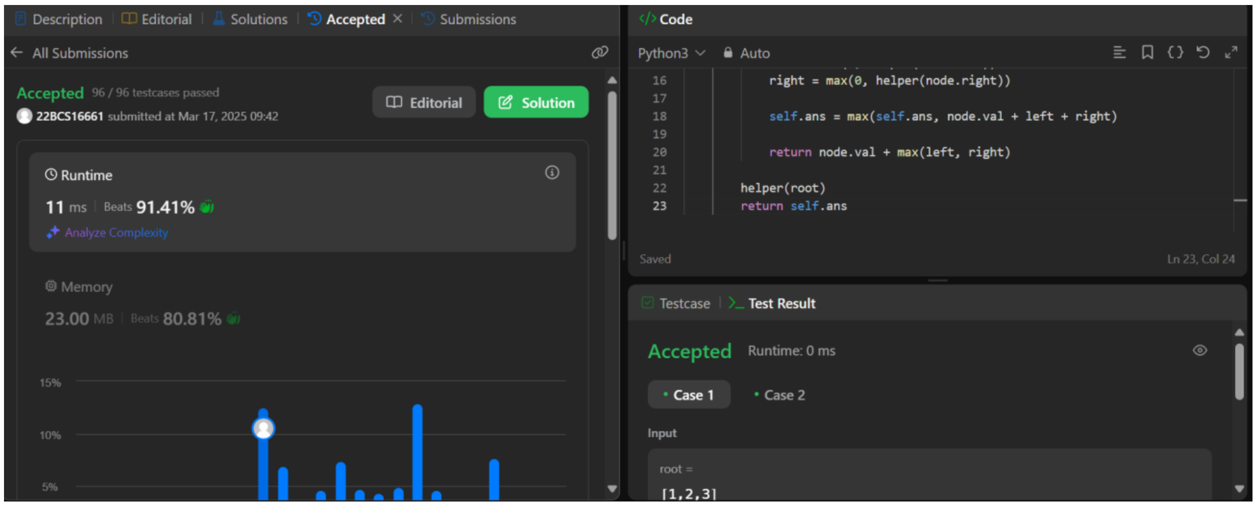
**Aim:** A **path** in a binary tree is a sequence of nodes where each pair of adjacent nodes in the sequence has an edge connecting them. A node can only appear in the sequence **at most once**. Note that the path does not need to pass through the root.
The **path sum** of a path is the sum of the node's values in the path.
Given the root of a binary tree, return *the maximum* ***path sum*** *of any* ***non-empty*** *path*.

## Source Code:

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

```
class Solution:
    def maxPathSum(self, root: TreeNode) -> int:
        self.ans = float('-inf')

        def helper(node):
            if not node:
                return 0

            left = max(0, helper(node.left))
            right = max(0, helper(node.right))

            self.ans = max(self.ans, node.val + left + right)

            return node.val + max(left, right)

        helper(root)
        return self.ans
```

**Screenshots of outputs:**



Accepted 96 / 96 testcases passed
22BCS16661 submitted at Mar 17, 2025 09:42

Runtime
11 ms | Beats 91.41%

Memory
23.00 MB | Beats 80.81%

15%
10%
5%

```
            right = max(0, helper(node.right))

            self.ans = max(self.ans, node.val + left + right)

            return node.val + max(left, right)

        helper(root)
        return self.ans
```

Saved
Ln 23, Col 24

Accepted Runtime: 0 ms

Input

root =
[1,2,3]

**3.**

**Aim:** There are a total of numCourses courses you have to take, labeled from 0 to numCourses - 1. You are given an array prerequisites where prerequisites[i] = [$a_i$, $b_i$] indicates that you **must** take course $b_i$ first if you want to take course $a_i$.

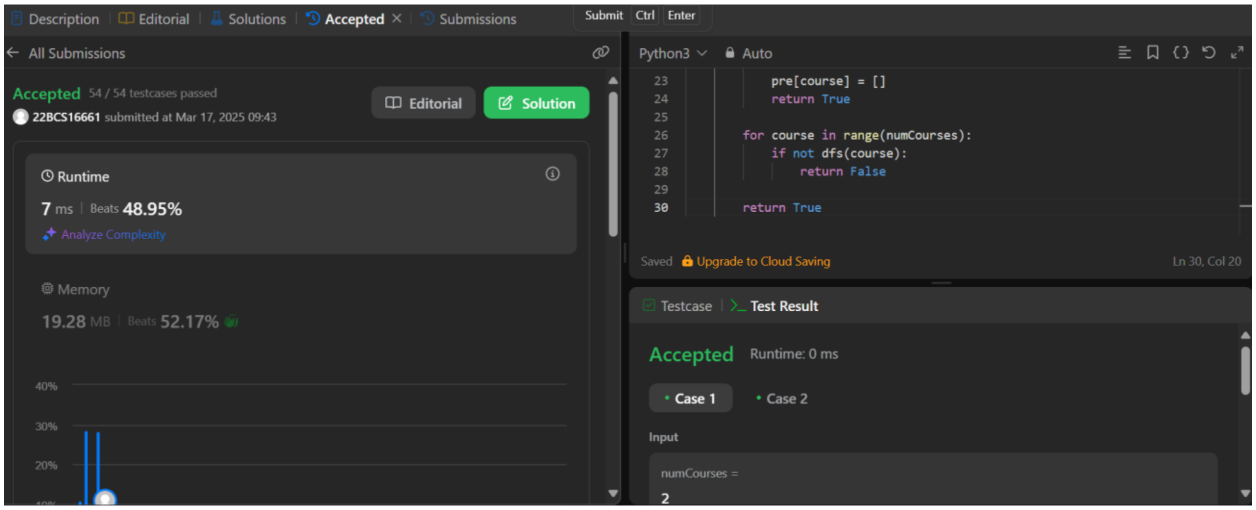
- For example, the pair [0, 1], indicates that to take course 0 you have to first take course 1.

Return true if you can finish all courses. Otherwise, return false.

## Source Code:

```
class Solution:
    def canFinish(self, numCourses: int, prerequisites: List[List[int]]) -> bool:
        pre = defaultdict(list)
        for course, p in prerequisites:
            pre[course].append(p)

        taken = set()
        def dfs(course):
            if not pre[course]:
                return True
            if course in taken:
                return False
            taken.add(course)
            for p in pre[course]:
                if not dfs(p): return False
            pre[course] = []
            return True

        for course in range(numCourses):
            if not dfs(course):
                return False
        return True
```

## 4. Screenshots of outputs:



Accepted 54 / 54 testcases passed
22BCS16661 submitted at Mar 17, 2025 09:43

Runtime
7 ms | Beats 48.95%

Memory
19.28 MB | Beats 52.17%

```
                pre[course] = []
                return True

        for course in range(numCourses):
            if not dfs(course):
                return False

        return True
```

Saved   Ln 30, Col 20

Accepted Runtime: 0 ms
Case 1   Case 2
Input
numCourses =
2