AP LAB-II

22BCS10608

SAHIL GUPTA

IOT-614/B

200. Number of Islands

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*.

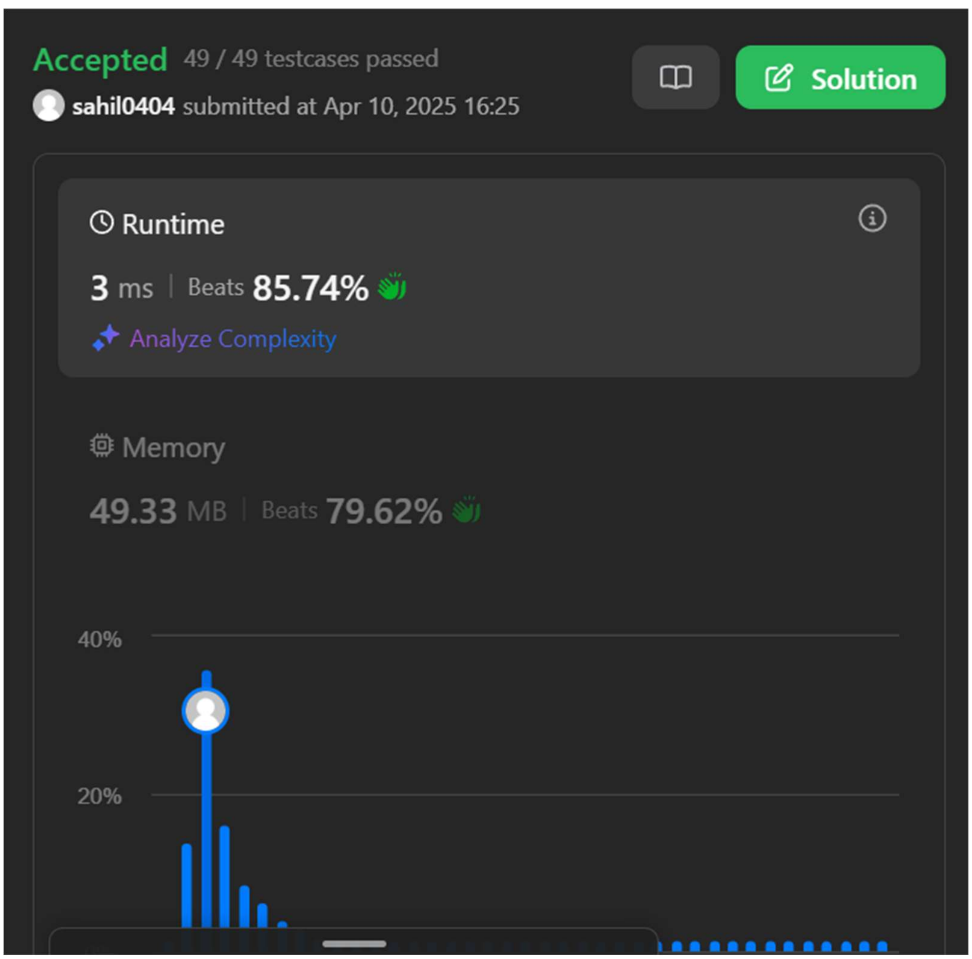
An **island** is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

Code:

```
public class Solution {
    public int numIslands(char[][] grid) {
        int count = 0;

        for (int i = 0; i < grid.length; i++) {
            for (int j = 0; j < grid[i].length; j++) {
                if (grid[i][j] == '1') {
                    count++;
                    clearRestOfLand(grid, i, j);
                }
            }
        }
        return count;
    }

    private void clearRestOfLand(char[][] grid, int i, int j) {
        if (i < 0 || j < 0 || i >= grid.length || j >= grid[i].length || grid[i][j] == '0') return;
```

```
        grid[i][j] = '0';
        clearRestOfLand(grid, i+1, j);
        clearRestOfLand(grid, i-1, j);
        clearRestOfLand(grid, i, j+1);
        clearRestOfLand(grid, i, j-1);
        return;
    }
}
```

Output:



127. Word Ladder

A **transformation sequence** from word beginWord to word endWord using a dictionary wordList is a sequence of words beginWord -> s_1 -> s_2 -> ... -> s_k such that:

- Every adjacent pair of words differs by a single letter.
- Every s_i for 1 <= i <= k is in wordList. Note that beginWord does not need to be in wordList.
- s_k == endWord

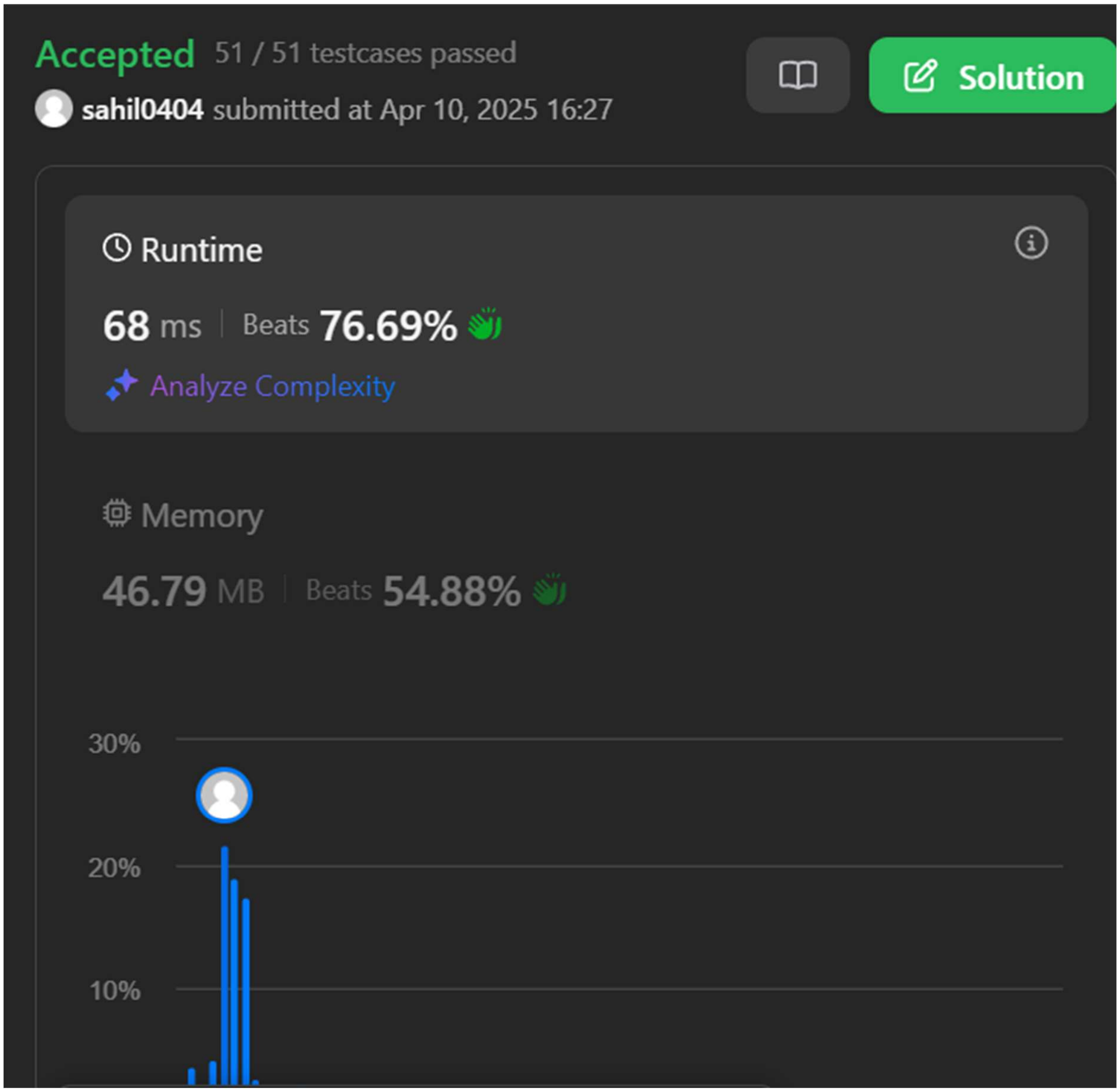
Given two words, beginWord and endWord, and a dictionary wordList, return *the **number of words** in the **shortest transformation sequence** from* beginWord *to* endWord*, or* 0 *if no such sequence exists.*

Code:

```
class Solution {
    public int ladderLength(String beginWord, String endWord, List<String> wordList) {

        Set<String> wordSet = new HashSet<>(wordList);
        if (!wordSet.contains(endWord)) return 0;

        Queue<Pair<String, Integer>> queue = new LinkedList<>();
        queue.offer(new Pair<>(beginWord, 1));
        wordSet.remove(beginWord);

        while (!queue.isEmpty()) {
            String word = queue.peek().getKey();
            int steps = queue.poll().getValue();

            if (word.equals(endWord)) return steps;

            for (int i = 0; i < word.length(); i++) {
```

```
            char[] wordArr = word.toCharArray();
            char original = wordArr[i];
            for (char ch = 'a'; ch <= 'z'; ch++) {
                wordArr[i] = ch;
                String newWord = new String(wordArr);
                if (wordSet.contains(newWord)) {
                    wordSet.remove(newWord);
                    queue.offer(new Pair<>(newWord, steps + 1));
                }
            }
        }
    }

    return 0;
}
static class Pair<K, V> {
    private K key;
    private V value;

    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public K getKey() { return key; }
    public V getValue() { return value; }
}
}
```

Output:



[130. Surrounded Regions]

You are given an m x n matrix board containing **letters** 'X' and 'O', **capture regions** that are **surrounded**:

- **Connect**: A cell is connected to adjacent cells horizontally or vertically.
- **Region**: To form a region **connect every** 'O' cell.
- **Surround**: The region is surrounded with 'X' cells if you can **connect the region** with 'X' cells and none of the region cells are on the edge of the board.

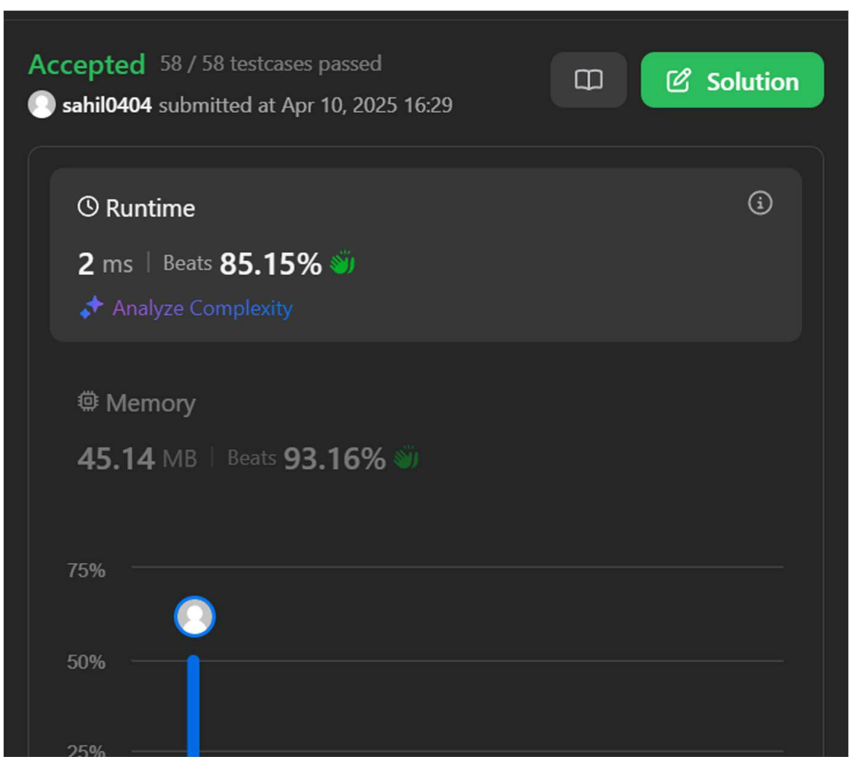
To capture a **surrounded region**, replace all 'O's with 'X's **in-place** within the original board. You do not need to return anything.

Code:

```
class Solution {
    public void solve(char[][] board) {
        int rows = board.length;
        int cols = board[0].length;
        for(int i=0; i<rows; i++){
            for(int j=0; j<cols; j++){
                if(i*j==0 || i==rows-1 || j==cols-1){

                    if(board[i][j] == 'O'){
                        dfs(board,i,j);
                    }
                }
            }
        }

        for(int i=0; i<rows; i++){
            for(int j=0; j<cols; j++){
                if(board[i][j] == 'O'){
                    board[i][j] = 'X';
                }
                else if(board[i][j] == 'V'){
                    board[i][j] = 'O';
                }
            }
```

```
        }
    }

    private void dfs(char[][] board, int i, int j){
        if(i<0 || j<0 || i>=board.length || j>=board[0].length || board[i][j] != 'O'){
            return;
        }
        board[i][j] = 'V';
        dfs(board,i+1,j);
        dfs(board,i-1,j);
        dfs(board,i,j+1);
        dfs(board,i,j-1);
    }

}
```

Output:



[124. Binary Tree Maximum Path Sum]

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.

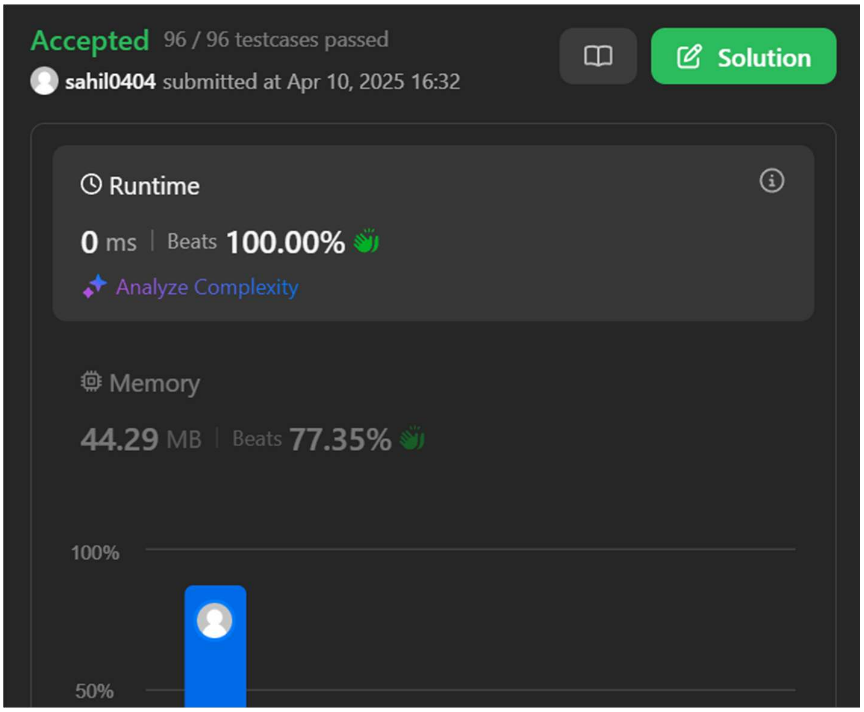
The **path sum** of a path is the sum of the node's values in the path.

Given the root of a binary tree, return *the maximum* ***path sum*** *of any* ***non-empty*** *path*.

Code:

```
public class Solution {
  int max = Integer.MIN_VALUE;

  public int maxPathSum(TreeNode root) {
    helper(root);
    return max;
  }
  int helper(TreeNode root) {
    if (root == null) return 0;

    int left = Math.max(helper(root.left), 0);
    int right = Math.max(helper(root.right), 0);

    max = Math.max(max, root.val + left + right);

    return root.val + Math.max(left, right);
  }
}
```

Output:



[547. Number of Provinces]

There are n cities. Some of them are connected, while some are not. If city a is connected directly with city b, and city b is connected directly with city c, then city a is connected indirectly with city c.

A **province** is a group of directly or indirectly connected cities and no other cities outside of the group.

You are given an n x n matrix isConnected where isConnected[i][j] = 1 if the i^{th} city and the j^{th} city are directly connected, and isConnected[i][j] = 0 otherwise.

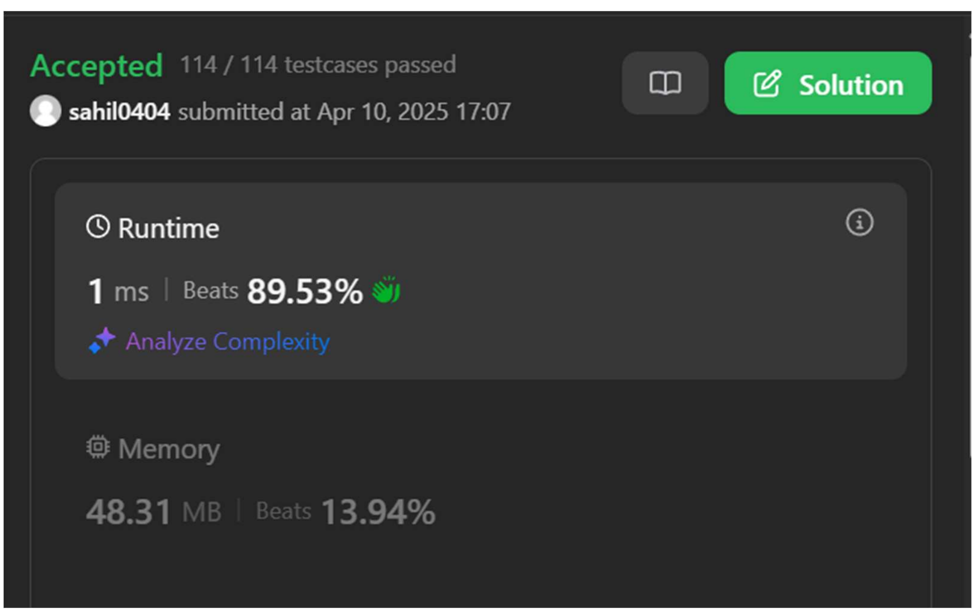
Return *the total number of **provinces***.

Code:

```
public class Solution {
    public void dfs(int[][] M, int[] visited, int i) {
        for (int j = 0; j < M.length; j++) {
            if (M[i][j] == 1 && visited[j] == 0) {
```

```
                visited[j] = 1;
                dfs(M, visited, j);
            }
        }
    }
    public int findCircleNum(int[][] M) {
        int[] visited = new int[M.length];
        int count = 0;
        for (int i = 0; i < M.length; i++) {
            if (visited[i] == 0) {
                dfs(M, visited, i);
                count++;
            }
        }
        return count;
    }
}
```

Output:



236. Lowest Common Ancestor of a Binary Tree

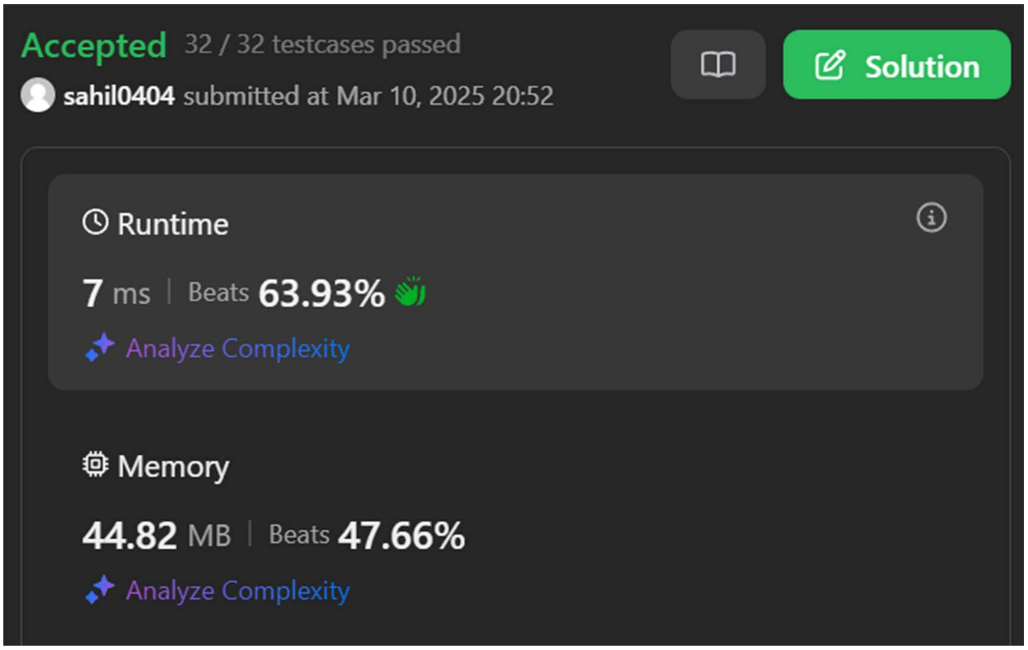
Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in the tree.

According to the definition of LCA on Wikipedia: “The lowest common ancestor is defined between two nodes p and q as the lowest node in T that has both p and q as descendants (where we allow **a node to be a descendant of itself**).”

Code:

```
class Solution {
  public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {

    if (root == null || root == p || root == q) {
      return root;
    }
    TreeNode left = lowestCommonAncestor(root.left, p, q);
    TreeNode right = lowestCommonAncestor(root.right, p, q);


    if(left == null) {
      return right;
    }
    else if(right == null) {
      return left;
    }
    else {
      return root;
    }
  }
}
```

Output:



[207. Course Schedule]

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.

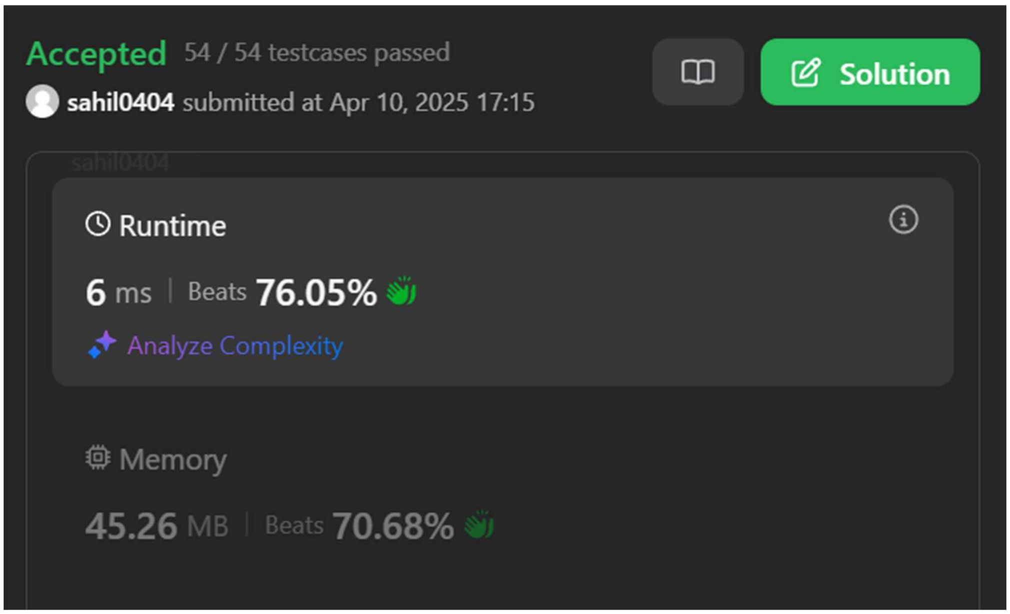
Code:

```
class Solution {
    public boolean canFinish(int n, int[][] prerequisites) {
        List<Integer>[] adj = new List[n];
        int[] indegree = new int[n];
        List<Integer> ans = new ArrayList<>();

        for (int[] pair : prerequisites) {
```

```
    int course = pair[0];
    int prerequisite = pair[1];
    if (adj[prerequisite] == null) {
        adj[prerequisite] = new ArrayList<>();
    }
    adj[prerequisite].add(course);
    indegree[course]++;
}

Queue<Integer> queue = new LinkedList<>();
for (int i = 0; i < n; i++) {
    if (indegree[i] == 0) {
        queue.offer(i);
    }
}

while (!queue.isEmpty()) {
    int current = queue.poll();
    ans.add(current);

    if (adj[current] != null) {
        for (int next : adj[current]) {
            indegree[next]--;
            if (indegree[next] == 0) {
                queue.offer(next);
            }
        }
    }
```

```
        }

        return ans.size() == n;
    }
}
```

Output:



[329. Longest Increasing Path in a Matrix]

Given an m x n integers matrix, return *the length of the longest increasing path in* matrix.

From each cell, you can either move in four directions: left, right, up, or down. You **may not** move **diagonally** or move **outside the boundary** (i.e., wrap-around is not allowed).

Code:

```
class Solution {
    public int longestIncreasingPath(int[][] matrix) {
        int ROW = matrix.length;
        int COL = matrix[0].length;
```

```
    int[][] memo = new int[ROW][COL];

    int max = Integer.MIN_VALUE;

    for(int i = 0; i < ROW; i++) {
      for(int j = 0; j < COL; j++) {
        int currMax = dfs(matrix, i, j, -1, memo);
        max = Math.max(max, currMax);
      }
    }
    return max;
  }

  private int dfs(int[][] matrix, int i, int j, int previous, int[][] memo) {
    if(!isInBound(matrix, i, j)) return 0;
    if(previous >= matrix[i][j]) return 0;
    if(memo[i][j] != 0) return memo[i][j];



    int curr = matrix[i][j];

    int top = dfs(matrix, i-1, j, curr, memo);
    int bottom = dfs(matrix, i+1, j, curr, memo);
    int left = dfs(matrix, i, j-1, curr, memo);
    int right = dfs(matrix, i, j+1, curr, memo);

    int max = top;
    max = Math.max(max, bottom);
```

```
        max = Math.max(max, left);
        max = Math.max(max, right);



        memo[i][j] = max + 1;
        return memo[i][j];
    }

    private boolean isInBound(int[][] matrix, int i, int j) {
        return i >=0 && j >= 0 && i < matrix.length && j < matrix[i].length;
    }
}
```

Output: