# Answer of 1

# A piece of paper with writing on it Description automatically generated

A notebook with writing on it

Description automatically generatedA piece of paper with writing on it

Description automatically generatedA notebook with writing on it

Description automatically generatedA piece of paper with writing on it

Description automatically generated

# Answer of 2

Algorithm: IsReachableFrom(G, u, v)

**Input:** A directed graph G, vertices u, v in G

**Output:** TRUE if there is a directed path from u to v in G, false otherwise.

Initialize a stack S //supports backtracking

Mark s as visited

S.push(s)

while S ≠ ∅ do

v’ ← S.peek()

if some out vertex of v’ not yet visited then

w ← next out vertex of v (i.e. (v’,w) in E)

if w = v then

return TRUE

mark w

push w onto S

else //if can’t find such a w,

backtrack S.pop()

return FALSE

# Answer of 3

import java.util.\*;

class Solution {

public int[] findOrder(int numCourses, int[][] prerequisites) {

// Edge case: no prerequisites, return a range in order

if (prerequisites.length == 0) {

int[] order = new int[numCourses];

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

order[i] = i;

}

return order;

}

// Create adjacency lists for in-degrees and out-degrees

Map<Integer, List<Integer>> inList = new HashMap<>();

Map<Integer, List<Integer>> outList = new HashMap<>();

Set<Integer> vertices = new HashSet<>();

// Initialize the maps

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

inList.put(i, new ArrayList<>());

outList.put(i, new ArrayList<>());

vertices.add(i);

}

// Build the graph

for (int[] prerequisite : prerequisites) {

int inVertex = prerequisite[1];

int outVertex = prerequisite[0];

outList.get(inVertex).add(outVertex);

inList.get(outVertex).add(inVertex);

}

// Find vertices with zero in-degrees to start the process

List<Integer> zeroInDegree = new ArrayList<>();

for (int v : vertices) {

if (inList.get(v).isEmpty()) {

zeroInDegree.add(v);

}

}

List<Integer> result = new ArrayList<>(); // List to store the topological order

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

Set<Integer> path = new HashSet<>(); // Set to detect cycles during DFS

// Perform DFS from all vertices with zero in-degrees

for (int vertex : zeroInDegree) {

if (!dfs(vertex, outList, visited, path, result)) {

return new int[0]; // If a cycle is detected, return an empty array

}

}

// If not all courses are visited, return an empty list (unreachable nodes)

if (visited.size() != numCourses) {

return new int[0];

}

// Reverse the result to get the correct topological order

Collections.reverse(result);

// Convert the result list to an array

return result.stream().mapToInt(i -> i).toArray();

}

// Helper function for DFS

public static boolean dfs(int vertex, Map<Integer, List<Integer>> outList, Set<Integer> visited, Set<Integer> path, List<Integer> result) {

if (path.contains(vertex)) {

return false; // Cycle detected

}

if (visited.contains(vertex)) {

return true; // Vertex already processed

}

path.add(vertex);

visited.add(vertex);

for (int v : outList.get(vertex)) {

if (!dfs(v, outList, visited, path, result)) {

return false; // Cycle detected in path starting at v

}

}

path.remove(vertex);

result.add(vertex);

return true;

}

}

# Answer of 4

import java.util.\*;

class Solution {

public boolean canFinish(int numCourses, int[][] prerequisites) {

// Edge case: no prerequisites, return true (all courses can be finished)

if (prerequisites.length == 0) {

return true;

}

// Create adjacency lists for in-degrees and out-degrees

Map<Integer, List<Integer>> inList = new HashMap<>();

Map<Integer, List<Integer>> outList = new HashMap<>();

Set<Integer> vertices = new HashSet<>();

// Initialize the maps

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

inList.put(i, new ArrayList<>());

outList.put(i, new ArrayList<>());

vertices.add(i);

}

// Build the graph

for (int[] prerequisite : prerequisites) {

int inVertex = prerequisite[1];

int outVertex = prerequisite[0];

outList.get(inVertex).add(outVertex);

inList.get(outVertex).add(inVertex);

}

// Find vertices with zero in-degrees to start the process

List<Integer> zeroInDegree = new ArrayList<>();

for (int v : vertices) {

if (inList.get(v).isEmpty()) {

zeroInDegree.add(v);

}

}

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

Set<Integer> path = new HashSet<>(); // Set to detect cycles during DFS

// Perform DFS from all vertices with zero in-degrees

for (int vertex : zeroInDegree) {

if (!dfs(vertex, outList, visited, path)) {

return false; // If a cycle is detected, return false

}

}

// If not all courses are visited, return false (some nodes are unreachable)

return visited.size() == numCourses;

}

// Helper method for DFS

private boolean dfs(int vertex, Map<Integer, List<Integer>> outList, Set<Integer> visited, Set<Integer> path) {

if (path.contains(vertex)) {

return false; // Cycle detected

}

if (visited.contains(vertex)) {

return true; // Vertex already processed

}

path.add(vertex);

visited.add(vertex);

for (int v : outList.get(vertex)) {

if (!dfs(v, outList, visited, path)) {

return false; // Cycle detected in path starting at v

}

}

path.remove(vertex);

return true;

}

}