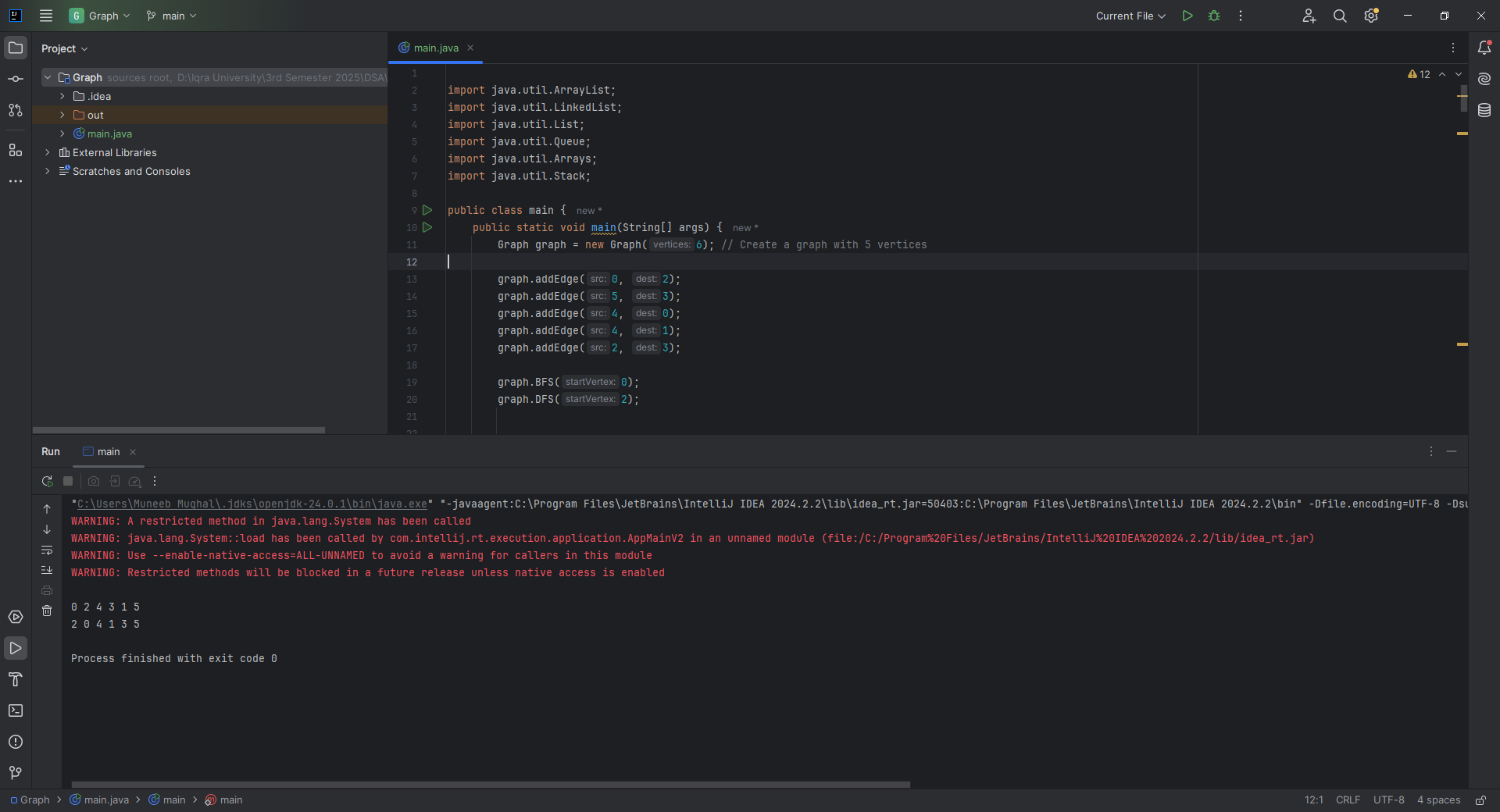
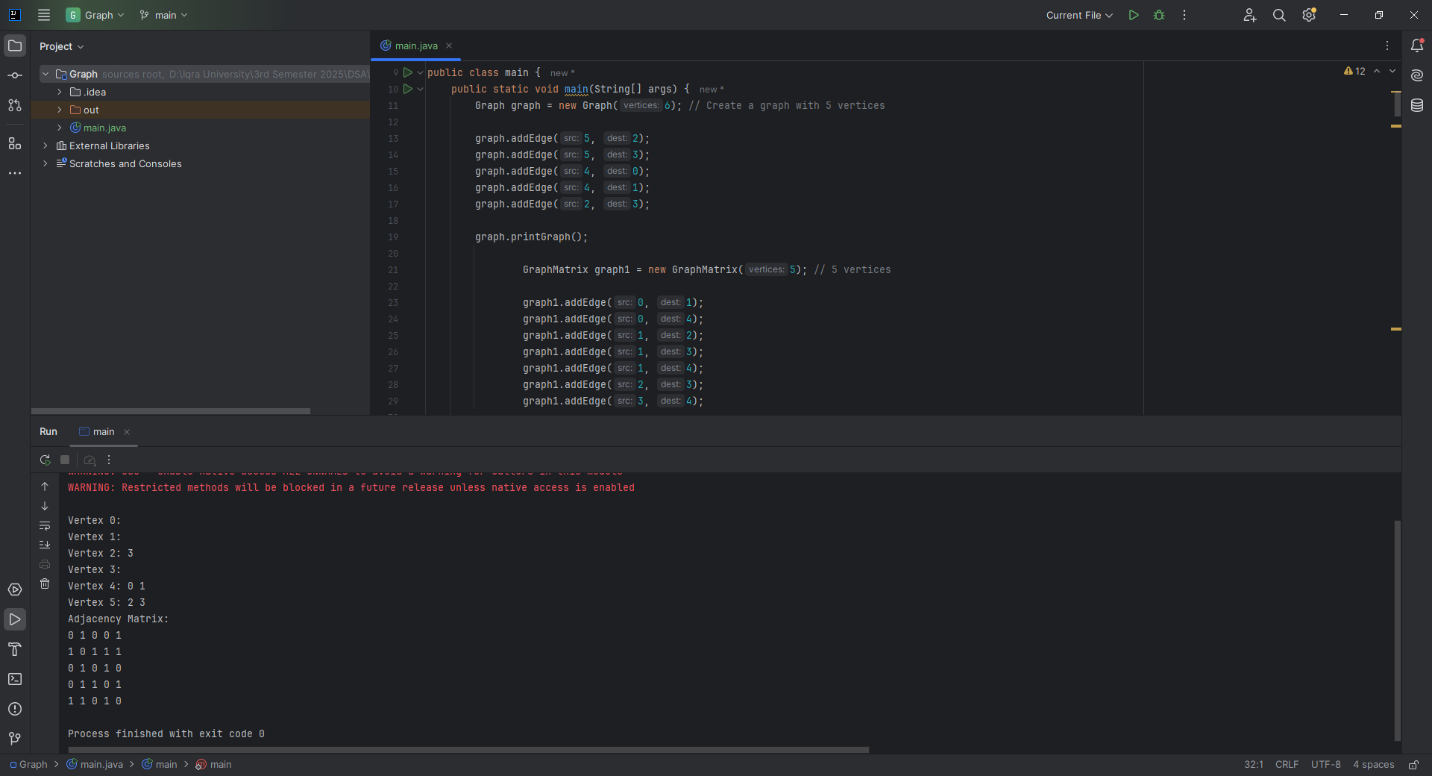
**CLASS TASK All:**

**CODE:**

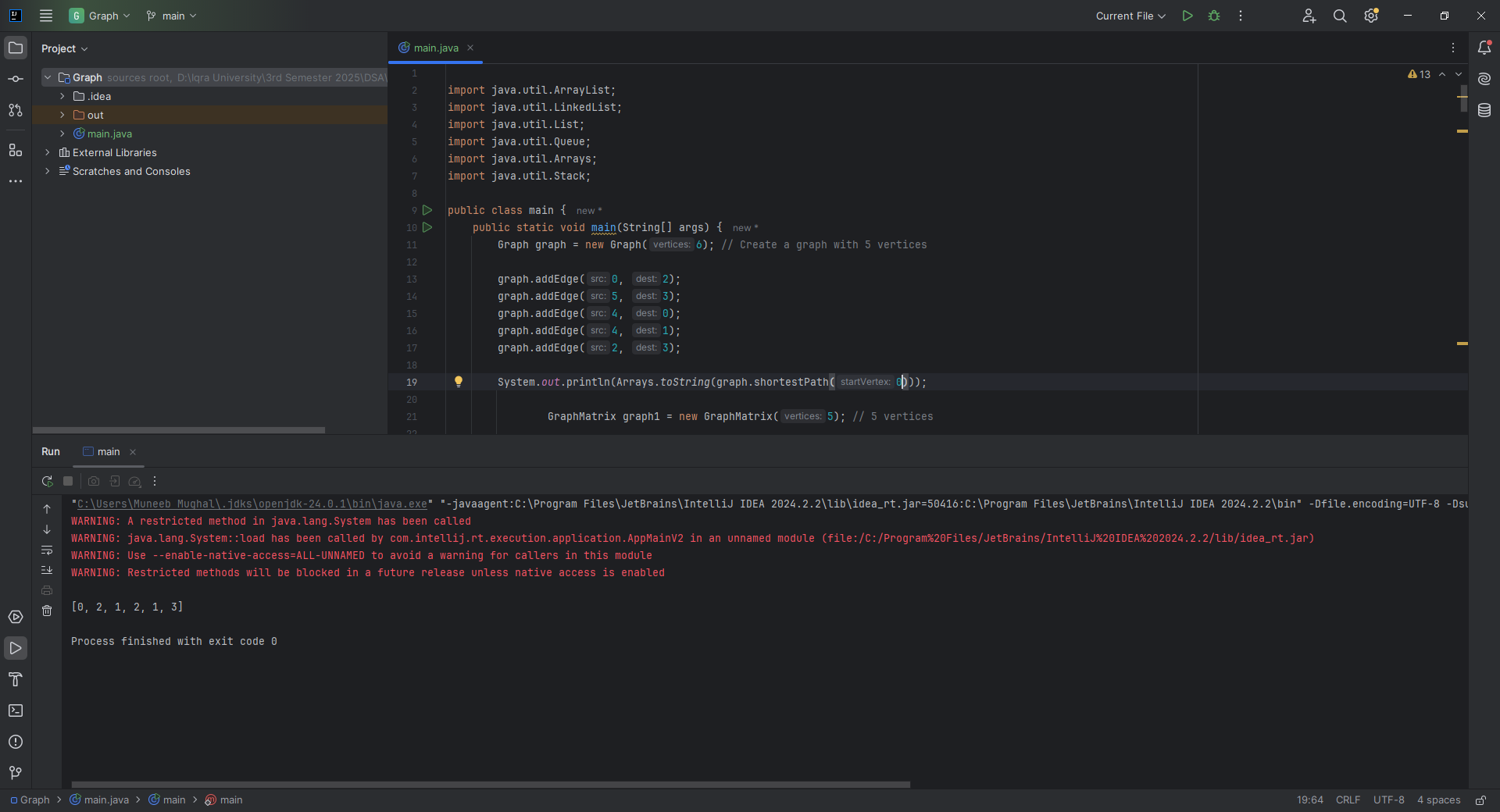
import java.util.\*;  
  
public class main {  
 public static void main(String[] args) {  
 Graph graph = new Graph(6); // Create a graph with 5 vertices  
  
 graph.addEdge(5, 2);  
 graph.addEdge(5, 3);  
 graph.addEdge(4, 0);  
 graph.addEdge(4, 1);  
 graph.addEdge(2, 3);  
  
  
  
 int[] path = graph.shortestPath(1);  
 System.*out*.println(Arrays.*toString*(path));  
 List<Integer> topologi = graph.topologicalSort();  
 System.*out*.println(topologi);  
  
 GraphMatrix graph1 = new GraphMatrix(5); // 5 vertices  
  
 graph1.addEdge(0, 1);  
 graph1.addEdge(0, 4);  
 graph1.addEdge(1, 2);  
 graph1.addEdge(1, 3);  
 graph1.addEdge(1, 4);  
 graph1.addEdge(2, 3);  
 graph1.addEdge(3, 4);  
  
 graph1.DFS(3);  
 graph1.BFS(3);  
  
 }  
}  
  
class Graph {  
 private int vertices; // Number of vertices  
 private List<List<Integer>> adjacencyList; // Adjacency list representation  
  
 // Constructor to initialize the graph  
 public Graph(int vertices) {  
 this.vertices = vertices;  
 adjacencyList = new ArrayList<>(vertices);  
  
 // Initialize each vertex with an empty LinkedList  
 for (int i = 0; i < vertices; i++) {  
 adjacencyList.add(new LinkedList<>());  
 }  
 }  
  
 // Method to add an edge between two vertices  
 public void addEdge(int src, int dest) {  
 adjacencyList.get(src).add(dest); // Add destination to source's list  
// adjacencyList.get(dest).add(src); // Add source to destination's list (undirected)  
 }  
  
 // Optional: Display the graph  
 public void printGraph() {  
 for (int i = 0; i < vertices; i++) {  
 System.*out*.print("Vertex " + i + ": ");  
 for (Integer neighbor : adjacencyList.get(i)) {  
 System.*out*.print(neighbor + " ");  
 }  
 System.*out*.println();  
 }  
 }  
 // Public method to start DFS from a given vertex  
 public void DFS(int startVertex) {  
 boolean[] visited = new boolean[vertices]; // Track visited vertices  
 DFSRecursive(startVertex, visited); // Call the recursive helper  
 System.*out*.println(); // New line after traversal  
 }  
  
 // Private recursive helper for DFS  
 private void DFSRecursive(int vertex, boolean[] visited) {  
 visited[vertex] = true; // Mark the current vertex as visited  
 System.*out*.print(vertex + " "); // Print the vertex  
  
 // Visit all unvisited adjacent vertices  
 for (int adjVertex : adjacencyList.get(vertex)) {  
 if (!visited[adjVertex]) {  
 DFSRecursive(adjVertex, visited); // Recursively visit the neighbor  
 }  
 }  
 }  
  
 public void BFS(int startVertex) {  
 boolean[] visited = new boolean[vertices];  
 Queue<Integer> queue = new LinkedList<>();  
  
 visited[startVertex] = true;  
 queue.offer(startVertex);  
  
 while (!queue.isEmpty()) {  
 int current = queue.poll();  
 System.*out*.print(current + " ");  
  
 for (int neighbor : adjacencyList.get(current)) {  
 if (!visited[neighbor]) {  
 visited[neighbor] = true;  
 queue.offer(neighbor);  
 }  
 }  
 }  
  
 System.*out*.println();  
 }  
}  
  
class GraphMatrix {  
 private int vertices; // Number of vertices  
 private int[][] adjacencyMatrix; // Adjacency matrix representation  
  
 // Constructor to initialize the graph  
 public GraphMatrix(int vertices) {  
 this.vertices = vertices;  
 adjacencyMatrix = new int[vertices][vertices];  
 }  
  
 // Method to add an undirected edge  
 public void addEdge(int src, int dest) {  
 adjacencyMatrix[src][dest] = 1;  
 adjacencyMatrix[dest][src] = 1; // Since the graph is undirected  
 }  
  
 // Method to print the adjacency matrix  
 public void printMatrix() {  
 System.*out*.println("Adjacency Matrix:");  
 for (int i = 0; i < vertices; i++) {  
 for (int j = 0; j < vertices; j++) {  
 System.*out*.print(adjacencyMatrix[i][j] + " ");  
 }  
 System.*out*.println();  
 }  
 }  
 public void DFS(int startVertex) {  
 boolean[] visited = new boolean[vertices];  
 DFSRecursive(startVertex, visited);  
 System.*out*.println();  
 }  
  
 private void DFSRecursive(int vertex, boolean[] visited) {  
 visited[vertex] = true;  
 System.*out*.print(vertex + " ");  
  
 for (int i = 0; i < vertices; i++) {  
 if (adjacencyMatrix[vertex][i] == 1 && !visited[i]) {  
 DFSRecursive(i, visited);  
 }  
 }  
 }  
 public void BFS(int startVertex) {  
 boolean[] visited = new boolean[vertices];  
 Queue<Integer> queue = new LinkedList<>();  
  
 visited[startVertex] = true;  
 queue.offer(startVertex);  
  
 while (!queue.isEmpty()) {  
 int current = queue.poll();  
 System.*out*.print(current + " ");  
  
 for (int i = 0; i < vertices; i++) {  
 if (adjacencyMatrix[current][i] == 1 && !visited[i]) {  
 visited[i] = true;  
 queue.offer(i);  
 }  
 }  
 }  
  
 System.*out*.println();  
 }  
}

****

**Home Task 1:** : Detecting a Cycle in an Undirected Graph

**CODE:**

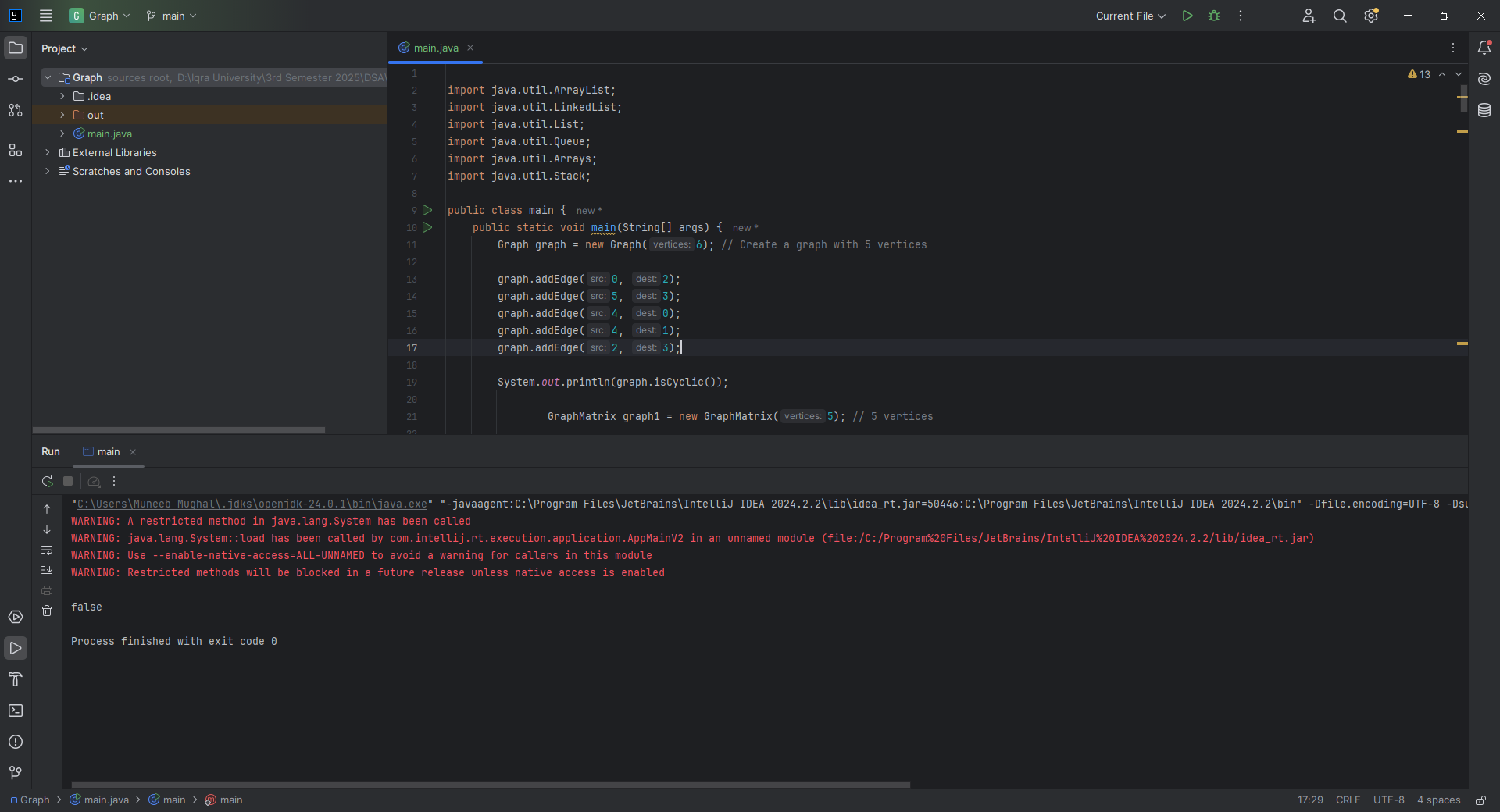
public boolean isCyclic() {  
 boolean[] visited = new boolean[vertices];  
  
 for (int i = 0; i < vertices; i++) {  
 if (!visited[i]) {  
 if (isCyclicDFS(i, visited, -1)) {  
 return true; // Cycle found  
 }  
 }  
 }  
  
 return false; // No cycle found  
}  
  
// DFS helper for cycle detection  
private boolean isCyclicDFS(int current, boolean[] visited, int parent) {  
 visited[current] = true;  
  
 for (int neighbor : adjacencyList.get(current)) {  
 if (!visited[neighbor]) {  
 if (isCyclicDFS(neighbor, visited, current)) {  
 return true;  
 }  
 } else if (neighbor != parent) {  
 // Visited and not coming from the parent → Cycle detected  
 return true;  
 }  
 }  
  
 return false;  
}



**Home Task 2:** Finding the Shortest Path Using BFS

**CODE:**

public int[] shortestPath(int startVertex) {  
 int[] distance = new int[vertices];  
 boolean[] visited = new boolean[vertices];  
 Queue<Integer> queue = new LinkedList<>();  
  
 // Initialize all distances to -1 (meaning unreachable initially)  
 Arrays.*fill*(distance, -1);  
  
 queue.offer(startVertex);  
 visited[startVertex] = true;  
 distance[startVertex] = 0;  
  
 while (!queue.isEmpty()) {  
 int current = queue.poll();  
  
 for (int neighbor : adjacencyList.get(current)) {  
 if (!visited[neighbor]) {  
 visited[neighbor] = true;  
 distance[neighbor] = distance[current] + 1;  
 queue.offer(neighbor);  
 }  
 }  
 }  
  
 return distance;  
}



**Home Task 3:** Topological Sorting of a Directed Acyclic Graph (DAG)

**CODE:**

public List<Integer> topologicalSort() {  
 boolean[] visited = new boolean[vertices];  
 Stack<Integer> stack = new Stack<>();  
  
 for (int i = 0; i < vertices; i++) {  
 if (!visited[i]) {  
 topologicalSortDFS(i, visited, stack);  
 }  
 }  
  
 List<Integer> sortedList = new ArrayList<>();  
 while (!stack.isEmpty()) {  
 sortedList.add(stack.pop());  
 }  
  
 return sortedList;  
}  
  
private void topologicalSortDFS(int vertex, boolean[] visited, Stack<Integer> stack) {  
 visited[vertex] = true;  
  
 for (int neighbor : adjacencyList.get(vertex)) {  
 if (!visited[neighbor]) {  
 topologicalSortDFS(neighbor, visited, stack);  
 }  
 }  
  
 stack.push(vertex); // Add to stack after visiting all neighbors  
}

