**Problem Statement**

Given an undirected graph represented by an adjacency matrix, implement a breadth-first search algorithm to traverse the graph starting from a specified vertex. Print the order of visited vertices during the traversal.

**Input format :**

The first line of input consists of the number of vertices **n.**

The second line consists of the number of edges**m.**

The following lines consist of **m** pairs of integers representing the edges of the graph.

The last line of input consists of the starting vertex.

**Output format :**

The output prints the order in which the nodes are visited starting from the specified vertex, separated by space.

**Refer to the sample output for formatting specifications.**

**Code constraints :**

1 ≤ n, m ≤ 10

**Sample test cases :**

**Input 1 :**

5

5

0 1

0 4

1 2

1 3

3 4

0

**Output 1 :**

0 1 4 2 3

**Input 2 :**

6

7

0 1

0 2

1 3

1 4

2 4

3 5

4 5

2

**Output 2 :**

2 0 4 1 5 3

**Problem Statement**

**Traverse a Graph using Depth-First Search**

A software company is developing a network analysis tool that visualizes connections between various servers in a data centre. Given an undirected graph represented by an adjacency matrix, the tool needs to implement a depth-first search algorithm to explore the network starting from a specified server. Then display the order in which the servers are visited during the traversal.

As a developer in the company, your task is to write a program for the same.

**Input format :**

The first line of input consists of the number of vertices.

The following lines consist of the adjacency matrix.

The last line consists of the starting vertex.

**Output format :**

The output prints the vertices visited during the depth-first search traversal in the order they were visited.

**Refer to the sample output for formatting specifications.**

**Code constraints :**

2 ≤ number of vertices ≤ 10

The matrix is binary.

**Sample test cases :**

**Input 1 :**

8

0 1 1 1 1 0 0 0

1 0 0 0 0 1 0 0

1 0 0 0 0 1 0 0

1 0 0 0 0 0 1 0

1 0 0 0 0 0 1 0

0 1 1 0 0 0 0 1

0 0 0 1 1 0 0 1

0 0 0 0 0 1 1 0

0

**Output 1 :**

0 1 5 2 7 6 3 4

**Input 2 :**

8

0 1 1 1 1 0 0 0

1 0 0 0 0 1 0 0

1 0 0 0 0 1 0 0

1 0 0 0 0 0 1 0

1 0 0 0 0 0 1 0

0 1 1 0 0 0 0 1

0 0 0 1 1 0 0 1

0 0 0 0 0 1 1 0

2

**Output 2 :**

2 0 1 5 7 6 3 4

**Problem Statement**

Given a connected undirected graph. Perform a depth-first traversal of the graph.

**Note:** Use a recursive approach.

**Example**

**Input:**

2

5 4

0 1 0 2 0 3 2 4

4 3

0 1 1 2 0 3

**Output:**

0 1 2 4 3

0 1 2 3

**Explanation:**

Visit all the vertices using a depth-first search algorithm using 0 as a source node.

**Input format :**

The first line of input contains an integer T denoting the number of test cases.

Then T test cases follow. Each test case consists of two lines.

The first line of each test case contains two integers N and E which denote the no. of vertices and no. of edges respectively.

The second line of each test case contains E space-separated pairs u and v denoting that there is an edge from u to v.

**Output format :**

For each test case, the output displays the graph vertices in the depth-first traversal order.

**Refer to the sample output for formatting specifications.**

**Code constraints :**

1 ≤ T ≤ 100

2 ≤ N ≤ 104

1 ≤ E ≤ (N\*(N-1))/2

The graph doesn't contain multiple edges and self-loops.

**Sample test cases :**

**Input 1 :**

2

5 4

0 1 0 2 0 3 2 4

4 3

0 1 1 2 0 3

**Output 1 :**

0 1 2 4 3

0 1 2 3

**Input 2 :**

1

2 1

0 1

**Output 2 :**

0 1

**Detailed Explanation:**

1. **Imports**:
   * import java.util.\*;: Import all classes from the java.util package, including Scanner, List, ArrayList, Queue, LinkedList.
2. **Main Class and Method**:
   * class GraphBFS: Define the main class.
   * public static void main(String[] args): Main method to run the program.
3. **Reading Input**:
   * Scanner scanner = new Scanner(System.in);: Initialize Scanner to read input from the standard input.
   * int n = scanner.nextInt();: Read the number of vertices.
   * int m = scanner.nextInt();: Read the number of edges.
4. **Initializing Adjacency Matrix**:
   * int[][] adjMatrix = new int[n][n];: Create a 2D array for the adjacency matrix to represent the graph.
5. **Populating Adjacency Matrix**:
   * Loop through the number of edges (m) to read each edge:
     + int u = scanner.nextInt();: Read the first vertex of the edge.
     + int v = scanner.nextInt();: Read the second vertex of the edge.
     + adjMatrix[u][v] = 1;: Mark the edge from u to v.
     + adjMatrix[v][u] = 1;: Mark the edge from v to u (since the graph is undirected).
6. **Reading Starting Vertex**:
   * int startVertex = scanner.nextInt();: Read the starting vertex for BFS.
7. **Performing BFS**:
   * List<Integer> result = bfs(adjMatrix, startVertex, n);: Call the bfs method to perform BFS and store the result.
8. **Printing Result**:
   * Loop through the result list and print each vertex.
9. **BFS Method**:
   * boolean[] visited = new boolean[n];: Create a boolean array to track visited vertices.
   * List<Integer> orderOfVisit = new ArrayList<>();: Create a list to store the order of visited vertices.
   * Queue<Integer> queue = new LinkedList<>();: Create a queue for BFS traversal.
   * visited[startVertex] = true;: Mark the starting vertex as visited.
   * queue.add(startVertex);: Add the starting vertex to the queue.
10. **Processing Queue**:
    * while (!queue.isEmpty()): Loop until the queue is empty.
    * int vertex = queue.poll();: Remove the front vertex from the queue.
    * orderOfVisit.add(vertex);: Add the current vertex to the result list.
    * Loop through all possible neighbors of the current vertex:
      + Check if there is an edge and if the neighbor is not visited.
      + If true, add the neighbor to the queue and mark it as visited.
11. **Returning Result**:
    * return orderOfVisit;: Return the list of vertices in the order they were visited.

This code efficiently performs a breadth-first search (BFS) on an undirected graph represented by an adjacency matrix and prints the order of visited vertices starting from a specified vertex.

import java.util.\*;

class GraphBFS {

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in); // Initialize scanner to read input from the console

        // Step 1: Read number of vertices

        int n = scanner.nextInt(); // Read the number of vertices in the graph

        // Step 2: Read number of edges

        int m = scanner.nextInt(); // Read the number of edges in the graph

        // Step 3: Initialize adjacency matrix

        int[][] adjMatrix = new int[n][n]; // Create a 2D array (adjacency matrix) to store graph edges

        // Step 4: Read edges and populate the adjacency matrix

        for (int i = 0; i < m; i++) { // Loop through each edge

            int u = scanner.nextInt(); // Read the first vertex of the edge

            int v = scanner.nextInt(); // Read the second vertex of the edge

            adjMatrix[u][v] = 1; // Mark the edge from u to v in the adjacency matrix

            adjMatrix[v][u] = 1; // Mark the edge from v to u since the graph is undirected

        }

        // Step 5: Read starting vertex

        int startVertex = scanner.nextInt(); // Read the starting vertex for the BFS traversal

        // Step 6: Perform BFS

        List<Integer> result = bfs(adjMatrix, startVertex, n); // Call the bfs function and store the result

        // Step 7: Print result

        for (int vertex : result) { // Loop through the BFS result

            System.out.print(vertex + " "); // Print each visited vertex

        }

    }

    public static List<Integer> bfs(int[][] adjMatrix, int startVertex, int n) {

        boolean[] visited = new boolean[n]; // Create an array to keep track of visited vertices

        List<Integer> orderOfVisit = new ArrayList<>(); // List to store the order of visited vertices

        Queue<Integer> queue = new LinkedList<>(); // Queue to manage the BFS process

        visited[startVertex] = true; // Mark the start vertex as visited

        queue.add(startVertex); // Add the start vertex to the queue

        while (!queue.isEmpty()) { // Loop until there are no more vertices to process

            int vertex = queue.poll(); // Remove and get the front vertex of the queue

            orderOfVisit.add(vertex); // Add this vertex to the result list

            // Iterate through all possible neighbors of the current vertex

            for (int neighbor = 0; neighbor < n; neighbor++) {

                // Check if there is an edge to the neighbor and if the neighbor is not visited

                if (adjMatrix[vertex][neighbor] == 1 && !visited[neighbor]) {

                    queue.add(neighbor); // Add the neighbor to the queue for further processing

                    visited[neighbor] = true; // Mark the neighbor as visited

                }

            }

        }

        return orderOfVisit; // Return the list of vertices in the order they were visited

    }

}

For java visualizer

import java.util.\*;

public class GraphBFS {

    // Static graph data

    private static final int n = 6; // Number of vertices

    private static final int m = 7; // Number of edges

    private static final int[][] adjMatrix = {

        {0, 1, 1, 0, 0, 0},

        {1, 0, 1, 1, 0, 0},

        {1, 1, 0, 1, 1, 0},

        {0, 1, 1, 0, 1, 1},

        {0, 0, 1, 1, 0, 1},

        {0, 0, 0, 1, 1, 0}

    };

    private static final int startVertex = 0; // Starting vertex for BFS

    public static void main(String[] args) {

        // Perform BFS using the static data

        List<Integer> result = bfs(adjMatrix, startVertex, n);

        // Print result

        for (int vertex : result) {

            System.out.print(vertex + " ");

        }

    }

    public static List<Integer> bfs(int[][] adjMatrix, int startVertex, int n) {

        boolean[] visited = new boolean[n]; // Create an array to keep track of visited vertices

        List<Integer> orderOfVisit = new ArrayList<>(); // List to store the order of visited vertices

        Queue<Integer> queue = new LinkedList<>(); // Queue to manage the BFS process

        visited[startVertex] = true; // Mark the start vertex as visited

        queue.add(startVertex); // Add the start vertex to the queue

        while (!queue.isEmpty()) { // Loop until there are no more vertices to process

            int vertex = queue.poll(); // Remove and get the front vertex of the queue

            orderOfVisit.add(vertex); // Add this vertex to the result list

            // Iterate through all possible neighbors of the current vertex

            for (int neighbor = 0; neighbor < n; neighbor++) {

                // Check if there is an edge to the neighbor and if the neighbor is not visited

                if (adjMatrix[vertex][neighbor] == 1 && !visited[neighbor]) {

                    queue.add(neighbor); // Add the neighbor to the queue for further processing

                    visited[neighbor] = true; // Mark the neighbor as visited

                }

            }

        }

        return orderOfVisit; // Return the list of vertices in the order they were visited

    }

}

Q2.

import java.util.\*;

class GraphDFS {

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in); // Initialize scanner to read input from the console

        // Read number of vertices

        int n = scanner.nextInt(); // Read the number of vertices in the graph

        // Initialize adjacency matrix

        int[][] adjMatrix = new int[n][n]; // Create a 2D array (adjacency matrix) to store graph edges

        // Read adjacency matrix

        for (int i = 0; i < n; i++) { // Loop through each row

            for (int j = 0; j < n; j++) { // Loop through each column

                adjMatrix[i][j] = scanner.nextInt(); // Read the value for each entry in the adjacency matrix

            }

        }

        // Read starting vertex

        int startVertex = scanner.nextInt(); // Read the starting vertex for the DFS traversal

        // Perform DFS

        List<Integer> result = dfs(adjMatrix, startVertex, n); // Call the dfs function and store the result

        // Print result

        for (int vertex : result) { // Loop through the DFS result

            System.out.print(vertex + " "); // Print each visited vertex

        }

    }

    // Function to perform DFS traversal

    public static List<Integer> dfs(int[][] adjMatrix, int startVertex, int n) {

        boolean[] visited = new boolean[n]; // Create an array to keep track of visited vertices

        List<Integer> orderOfVisit = new ArrayList<>(); // List to store the order of visited vertices

        // Helper function for DFS traversal

        dfsUtil(startVertex, adjMatrix, visited, orderOfVisit);

        return orderOfVisit; // Return the list of vertices in the order they were visited

    }

    // Recursive helper function for DFS traversal

    public static void dfsUtil(int vertex, int[][] adjMatrix, boolean[] visited, List<Integer> orderOfVisit) {

        visited[vertex] = true; // Mark the current vertex as visited

        orderOfVisit.add(vertex); // Add this vertex to the result list

        // Iterate through all possible neighbors of the current vertex

        for (int neighbor = 0; neighbor < adjMatrix.length; neighbor++) {

            // Check if there is an edge to the neighbor and if the neighbor is not visited

            if (adjMatrix[vertex][neighbor] == 1 && !visited[neighbor]) {

                dfsUtil(neighbor, adjMatrix, visited, orderOfVisit); // Recursively visit the neighbor

            }

        }

    }

}

**Explanation:**

1. **Imports and Class Definition:**

java

Copy code

import java.util.\*;

class GraphDFS {

* + import java.util.\*; imports the required classes from the java.util package such as Scanner, List, ArrayList.
  + class GraphDFS defines the class that contains the main method and DFS methods.

1. **Main Method:**

java

Copy code

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

int n = scanner.nextInt();

int[][] adjMatrix = new int[n][n];

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

for (int j = 0; j < n; j++) {

adjMatrix[i][j] = scanner.nextInt();

}

}

int startVertex = scanner.nextInt();

List<Integer> result = dfs(adjMatrix, startVertex, n);

for (int vertex : result) {

System.out.print(vertex + " ");

}

}

* + Scanner scanner = new Scanner(System.in); initializes a scanner to read from the console.
  + int n = scanner.nextInt(); reads the number of vertices.
  + int[][] adjMatrix = new int[n][n]; creates a 2D array to represent the adjacency matrix of the graph.
  + The nested for loops read the adjacency matrix values from the console.
  + int startVertex = scanner.nextInt(); reads the starting vertex for the DFS.
  + List<Integer> result = dfs(adjMatrix, startVertex, n); calls the dfs method to perform DFS and returns the result.
  + for (int vertex : result) { System.out.print(vertex + " "); } prints the result of the DFS traversal.

1. **DFS Method:**

java

Copy code

public static List<Integer> dfs(int[][] adjMatrix, int startVertex, int n) {

boolean[] visited = new boolean[n];

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

dfsUtil(startVertex, adjMatrix, visited, orderOfVisit);

return orderOfVisit;

}

* + boolean[] visited = new boolean[n]; creates an array to keep track of which vertices have been visited.
  + List<Integer> orderOfVisit = new ArrayList<>(); creates a list to store the order in which vertices are visited.
  + dfsUtil(startVertex, adjMatrix, visited, orderOfVisit); calls the dfsUtil method to start the DFS traversal from the starting vertex.
  + return orderOfVisit; returns the list of vertices in the order they were visited.

1. **DFS Utility Method:**

java

Copy code

public static void dfsUtil(int vertex, int[][] adjMatrix, boolean[] visited, List<Integer> orderOfVisit) {

visited[vertex] = true;

orderOfVisit.add(vertex);

for (int neighbor = 0; neighbor < adjMatrix.length; neighbor++) {

if (adjMatrix[vertex][neighbor] == 1 && !visited[neighbor]) {

dfsUtil(neighbor, adjMatrix, visited, orderOfVisit);

}

}

}

* + visited[vertex] = true; marks the current vertex as visited.
  + orderOfVisit.add(vertex); adds the current vertex to the list of visited vertices.
  + The for loop iterates through all possible neighbors of the current vertex.
  + if (adjMatrix[vertex][neighbor] == 1 && !visited[neighbor]) checks if there is an edge to the neighbor and if the neighbor has not been visited.
  + dfsUtil(neighbor, adjMatrix, visited, orderOfVisit); recursively calls dfsUtil to visit the neighbor.

This code performs a depth-first traversal of a graph starting from a specified vertex and prints the vertices in the order they are visited.

**Understanding DFS Order**

1. **Recursive DFS:**
   * In a recursive DFS implementation, the order of visiting nodes is determined by the order of recursive calls.
   * The function explores each neighbor fully before backtracking. Hence, if a vertex v has neighbors w and x, it will explore all of w's neighbors before moving to x.
2. **Iterative DFS Using Stack:**
   * In an iterative DFS implementation using a stack, vertices are pushed onto the stack in the order they are discovered.
   * When you pop from the stack, you process the most recently added vertex first. This means that the order of traversal can depend on the order neighbors are pushed onto the stack.

**Using stack**

import java.util.\*;

class GraphDFS {

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in); // Initialize scanner to read input from the console

        // Read number of vertices

        int n = scanner.nextInt(); // Read the number of vertices in the graph

        // Initialize adjacency matrix

        int[][] adjMatrix = new int[n][n]; // Create a 2D array (adjacency matrix) to store graph edges

        // Read adjacency matrix

        for (int i = 0; i < n; i++) { // Loop through each row

            for (int j = 0; j < n; j++) { // Loop through each column

                adjMatrix[i][j] = scanner.nextInt(); // Read the value for each entry in the adjacency matrix

            }

        }

        // Read starting vertex

        int startVertex = scanner.nextInt(); // Read the starting vertex for the DFS traversal

        // Perform DFS

        List<Integer> result = dfs(adjMatrix, startVertex, n); // Call the dfs function and store the result

        // Print result

        for (int vertex : result) { // Loop through the DFS result

            System.out.print(vertex + " "); // Print each visited vertex

        }

    }

    // Function to perform DFS traversal using an explicit stack

    public static List<Integer> dfs(int[][] adjMatrix, int startVertex, int n) {

        boolean[] visited = new boolean[n]; // Create an array to keep track of visited vertices

        List<Integer> orderOfVisit = new ArrayList<>(); // List to store the order of visited vertices

        Stack<Integer> stack = new Stack<>(); // Create a stack to manage DFS traversal

        stack.push(startVertex); // Push the starting vertex onto the stack

        while (!stack.isEmpty()) { // Loop until the stack is empty

            int vertex = stack.pop(); // Pop a vertex from the stack

            // If the vertex has not been visited

            if (!visited[vertex]) {

                visited[vertex] = true; // Mark the vertex as visited

                orderOfVisit.add(vertex); // Add the vertex to the result list

                // Iterate through all possible neighbors of the current vertex

                for (int neighbor = 0; neighbor < adjMatrix.length; neighbor++) {

                    // Check if there is an edge to the neighbor and if the neighbor is not visited

                    if (adjMatrix[vertex][neighbor] == 1 && !visited[neighbor]) {

                        stack.push(neighbor); // Push the neighbor onto the stack for future exploration

                    }

                }

            }

        }

        return orderOfVisit; // Return the list of vertices in the order they were visited

    }

}

**Explanation of Non-Recursive DFS Using Stack**

1. **Stack Initialization:**

java

Copy code

Stack<Integer> stack = new Stack<>();

stack.push(startVertex);

* + Create a Stack to manage the vertices for DFS traversal.
  + Push the starting vertex onto the stack.

1. **Traversal Loop:**

java

Copy code

while (!stack.isEmpty()) {

int vertex = stack.pop();

if (!visited[vertex]) {

visited[vertex] = true;

orderOfVisit.add(vertex);

for (int neighbor = 0; neighbor < adjMatrix.length; neighbor++) {

if (adjMatrix[vertex][neighbor] == 1 && !visited[neighbor]) {

stack.push(neighbor);

}

}

}

}

* + While the stack is not empty, pop a vertex from the stack.
  + If the vertex is not visited, mark it as visited and add it to the result list.
  + Iterate through all possible neighbors of the current vertex.
  + If there is an edge to a neighbor and it hasn’t been visited, push the neighbor onto the stack.

1. **Return the Result:**

java

Copy code

return orderOfVisit;

* + After all reachable vertices are processed, return the list of vertices in the order they were visited.

This iterative approach using a stack mimics the behavior of recursion by explicitly managing the vertices to be explored and ensures that the DFS traversal is performed correctly without relying on the call stack of recursive method calls.

Static data

import java.util.\*;

import java.util.ArrayList;

public class GraphDFS {

    // Static graph data

    private static final int n = 6; // Number of vertices

    private static final int[][] adjMatrix = {

        {0, 1, 1, 0, 0, 0},

        {1, 0, 1, 1, 0, 0},

        {1, 1, 0, 1, 1, 0},

        {0, 1, 1, 0, 1, 1},

        {0, 0, 1, 1, 0, 1},

        {0, 0, 0, 1, 1, 0}

    };

    private static final int startVertex = 0; // Starting vertex for DFS

    public static void main(String[] args) {

        // Perform DFS using the static data

        List<Integer> result = dfs(adjMatrix, startVertex, n);

        // Print result

        for (int vertex : result) {

            System.out.print(vertex + " ");

        }

    }

    // Function to perform DFS traversal

    public static List<Integer> dfs(int[][] adjMatrix, int startVertex, int n) {

        boolean[] visited = new boolean[n]; // Create an array to keep track of visited vertices

        List<Integer> orderOfVisit = new ArrayList<>(); // List to store the order of visited vertices

        // Helper function for DFS traversal

        dfsUtil(startVertex, adjMatrix, visited, orderOfVisit);

        return orderOfVisit; // Return the list of vertices in the order they were visited

    }

    // Recursive helper function for DFS traversal

    public static void dfsUtil(int vertex, int[][] adjMatrix, boolean[] visited, List<Integer> orderOfVisit) {

        visited[vertex] = true; // Mark the current vertex as visited

        orderOfVisit.add(vertex); // Add this vertex to the result list

        // Iterate through all possible neighbors of the current vertex

        for (int neighbor = 0; neighbor < adjMatrix.length; neighbor++) {

            // Check if there is an edge to the neighbor and if the neighbor is not visited

            if (adjMatrix[vertex][neighbor] == 1 && !visited[neighbor]) {

                dfsUtil(neighbor, adjMatrix, visited, orderOfVisit); // Recursively visit the neighbor

            }

        }

    }

}

Q3.

import java.util.\*;

class GraphDFSTraversal {

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in); // Initialize scanner to read input from the console

        // Read number of test cases

        int T = scanner.nextInt();

        for (int t = 0; t < T; t++) {

            // Read number of vertices and edges

            int N = scanner.nextInt(); // Number of vertices

            int E = scanner.nextInt(); // Number of edges

            // Initialize adjacency list

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

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

                adjList.add(new ArrayList<>()); // Create an empty list for each vertex

            }

            // Read edges and populate adjacency list

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

                int u = scanner.nextInt(); // Read vertex u

                int v = scanner.nextInt(); // Read vertex v

                adjList.get(u).add(v); // Add v to the adjacency list of u

                adjList.get(v).add(u); // Add u to the adjacency list of v (undirected graph)

            }

            // Perform DFS starting from vertex 0

            boolean[] visited = new boolean[N]; // Array to keep track of visited vertices

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

            dfs(0, adjList, visited, result); // Call DFS from vertex 0

            // Print result for current test case

            for (int i = 0; i < result.size(); i++) {

                if (i > 0) System.out.print(" "); // Print a space between vertices

                System.out.print(result.get(i)); // Print each vertex in the result list

            }

            System.out.println(); // Move to the next line after printing all vertices for a test case

        }

    }

    // Recursive DFS function

    public static void dfs(int vertex, List<List<Integer>> adjList, boolean[] visited, List<Integer> result) {

        visited[vertex] = true; // Mark the current vertex as visited

        result.add(vertex); // Add the current vertex to the result list

        // Iterate through all neighbors of the current vertex

        for (int neighbor : adjList.get(vertex)) {

            if (!visited[neighbor]) { // If neighbor hasn't been visited

                dfs(neighbor, adjList, visited, result); // Recursively call DFS on the neighbor

            }

        }

    }

}

**Detailed Explanation**

1. **Reading Input:**
   * int T = scanner.nextInt(); reads the number of test cases.
   * For each test case, it reads the number of vertices N and edges E.
2. **Graph Representation:**
   * An adjacency list is used to represent the graph. Each vertex has a list of its neighbors.
   * List<List<Integer>> adjList = new ArrayList<>(); creates a list of lists where each inner list holds the neighbors of a vertex.
   * Edges are read and added to the adjacency list. Since the graph is undirected, each edge is added to both vertices' lists.
3. **DFS Traversal:**
   * The dfs method is a recursive function that performs Depth-First Search starting from the given vertex.
   * It marks the current vertex as visited, adds it to the result list, and recursively explores all unvisited neighbors.
4. **Output:**
   * After DFS completes for a test case, the result list is printed. Each vertex is separated by a space.

**Notes:**

* **Starting Vertex:** In this implementation, DFS starts from vertex 0. If a different starting vertex is required, you can adjust the starting point in the dfs call.
* **Handling Multiple Test Cases:** The code handles multiple test cases by iterating over them and processing each graph separately.
* **Edge Cases:** Ensure that the input format matches the expected structure (e.g., valid vertex indices). If the graph is disconnected, only the component containing the starting vertex will be fully traversed.

This implementation provides a clear and efficient way to handle graph traversal using DFS with an adjacency list and can be adapted for various use cases and input formats.

**Static data**

import java.util.\*;

class GraphDFSTraversal {

    public static void main(String[] args) {

        // Static data for test cases

        List<int[][]> graphs = Arrays.asList(

            new int[][] {

                {0, 1, 0, 0, 1},

                {1, 0, 1, 1, 0},

                {0, 1, 0, 1, 1},

                {0, 1, 1, 0, 1},

                {1, 0, 1, 1, 0}

            },

            new int[][] {

                {0, 1, 0, 0},

                {1, 0, 1, 1},

                {0, 1, 0, 1},

                {0, 1, 1, 0}

            }

        );

        // Static starting vertices for each graph

        int[] startingVertices = {0, 0};

        // Iterate over test cases

        for (int t = 0; t < graphs.size(); t++) {

            int[][] adjMatrix = graphs.get(t);

            int startVertex = startingVertices[t];

            // Convert adjacency matrix to adjacency list

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

            int N = adjMatrix.length;

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

                adjList.add(new ArrayList<>());

            }

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

                for (int j = 0; j < N; j++) {

                    if (adjMatrix[i][j] == 1) {

                        adjList.get(i).add(j);

                    }

                }

            }

            // Perform DFS

            boolean[] visited = new boolean[N];

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

            dfs(startVertex, adjList, visited, result);

            // Print result for current test case

            System.out.print("DFS result for graph " + (t + 1) + ": ");

            for (int i = 0; i < result.size(); i++) {

                if (i > 0) System.out.print(" ");

                System.out.print(result.get(i));

            }

            System.out.println();

        }

    }

    // Recursive DFS function

    public static void dfs(int vertex, List<List<Integer>> adjList, boolean[] visited, List<Integer> result) {

        visited[vertex] = true; // Mark the current vertex as visited

        result.add(vertex); // Add the current vertex to the result list

        // Iterate through all neighbors of the current vertex

        for (int neighbor : adjList.get(vertex)) {

            if (!visited[neighbor]) { // If neighbor hasn't been visited

                dfs(neighbor, adjList, visited, result); // Recursively call DFS on the neighbor

            }

        }

    }

}

**Explanation**

1. **Static Data Setup:**
   * We define a list of adjacency matrices for different test cases. Each matrix represents a graph.
   * We also define an array of starting vertices for each graph.
2. **Convert Adjacency Matrix to Adjacency List:**
   * The adjacency matrix is converted to an adjacency list format because DFS is performed more naturally on adjacency lists.
3. **Perform DFS:**
   * For each graph, perform DFS starting from the specified vertex and store the result.
4. **Print Results:**
   * Output the DFS traversal results for each graph.