**Count the Number of Complete Components (Leetcode 2685)**

**Problem Statement**

You are given an integer n, representing the number of vertices in an **undirected graph** numbered from 0 to n - 1. You are also given a 2D integer array edges, where edges[i] = [a\_i, b\_i] denotes an edge between vertices a\_i and b\_i.

A **connected component** is a subgraph in which all vertices are connected, and no vertex in the subgraph shares an edge with a vertex outside the subgraph.

A **connected component is complete** if there exists an edge between every pair of its vertices.

**Example 1**

**Input:**

n = 6, edges = [[0,1],[0,2],[1,2],[3,4]]

**Output:**

3

**Explanation:**

* Component {0,1,2} is complete since every pair of nodes has an edge.
* Component {3,4} is complete.
* Isolated node {5} is also complete.

**Example 2**

**Input:**

n = 6, edges = [[0,1],[0,2],[1,2],[3,4],[3,5]]

**Output:**

1

**Explanation:**

* {0,1,2} is complete.
* {3,4,5} is not complete because there is no edge between 4 and 5.
* Only 1 complete component.

**Constraints**

* 1 <= n <= 50
* 0 <= edges.length <= n \* (n - 1) / 2
* edges[i].length == 2
* 0 <= a\_i, b\_i <= n - 1
* a\_i != b\_i
* No repeated edges.

**Approach**

**1. Graph Representation**

* Use an **adjacency list** to store the graph as List<List<Integer>>.
* Iterate through edges and populate the adjacency list.

**2. Finding Connected Components**

* Use **DFS (Depth-First Search)** or **BFS (Breadth-First Search)** to explore each component.
* Maintain a visited array to track explored nodes.
* Store all nodes in a component inside a list or set.

**3. Checking Completeness**

* A component with k nodes must have exactly **(k \* (k - 1)) / 2** edges.
* Count the edges within the component and compare it with the expected count.

**Algorithm (Steps)**

1. **Build the adjacency list** from edges.
2. **Find all connected components** using DFS or BFS.
3. **Check completeness**:
   * If the number of edges in the component equals (k \* (k - 1)) / 2, it's complete.
4. **Return the count of complete components**.

**Java Implementation**

import java.util.\*;

class Solution {

public int countCompleteComponents(int n, int[][] edges) {

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

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

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

}

for (int[] edge : edges) {

graph.get(edge[0]).add(edge[1]);

graph.get(edge[1]).add(edge[0]);

}

boolean[] visited = new boolean[n];

int completeComponents = 0;

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

if (!visited[i]) {

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

dfs(i, graph, visited, component);

if (isComplete(component, graph)) {

completeComponents++;

}

}

}

return completeComponents;

}

private void dfs(int node, List<List<Integer>> graph, boolean[] visited, List<Integer> component) {

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

stack.push(node);

visited[node] = true;

while (!stack.isEmpty()) {

int curr = stack.pop();

component.add(curr);

for (int neighbor : graph.get(curr)) {

if (!visited[neighbor]) {

visited[neighbor] = true;

stack.push(neighbor);

}

}

}

}

private boolean isComplete(List<Integer> component, List<List<Integer>> graph) {

int size = component.size();

int expectedEdges = (size \* (size - 1)) / 2;

int actualEdges = 0;

for (int node : component) {

actualEdges += graph.get(node).size();

}

actualEdges /= 2; // Since it's an undirected graph, we counted each edge twice

return actualEdges == expectedEdges;

}

}

**Time Complexity Analysis**

* **Building the graph:** O(E)
* **DFS traversal:** O(V + E)
* **Checking completeness:** O(V)
* **Overall complexity:** O(V + E)

**Edge Cases Considered**

✅ Graph with **no edges** (all isolated nodes).  
✅ Graph where **all nodes are connected** (single complete component).  
✅ Graph where **some components are complete** and others are not.  
✅ Graph with **only one node** (n = 1).

**Summary**

* **Use DFS/BFS** to find connected components.
* **Check completeness** by comparing actual vs. expected edges.
* **Return the count of complete components.**

This approach ensures an efficient solution for n ≤ 50 and works well for all edge cases. 🚀