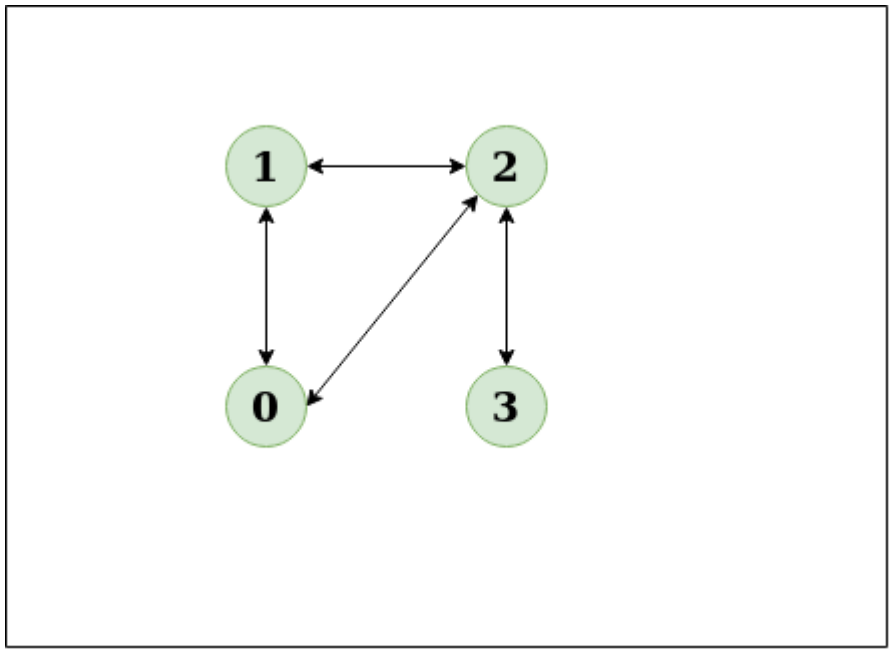
Detect cycle in Undirected graph

Given an undirected graph, The task is to check if there is a cycle in the given graph.

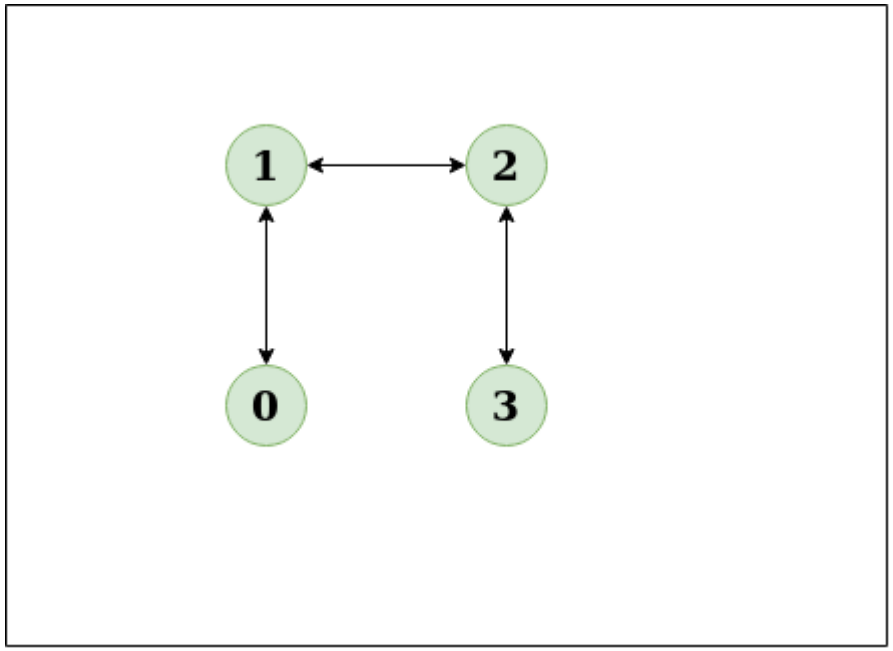
Example:

Input: N = 4, E = 4



Output: Yes
Explanation: The diagram clearly shows a cycle 0 to 2 to 1 to 0

Input: N = 4, E = 3 , 0 1, 1 2, 2 3



Output: No
Explanation: There is no cycle in the given graph

Articles about cycle detection:

- cycle detection for directed graph.
- union-find algorithm for cycle detection in undirected graphs.

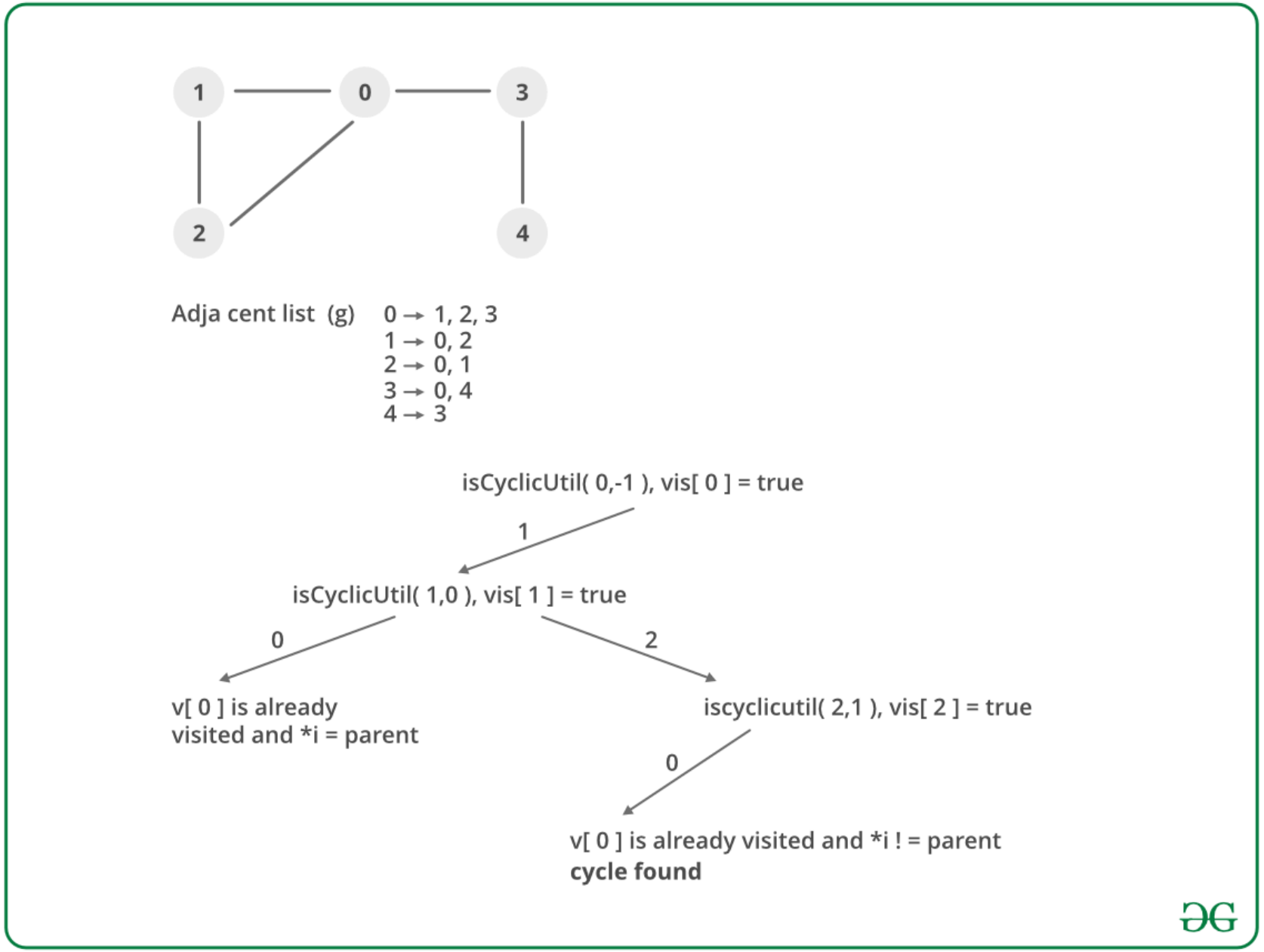
Find cycle in undirected Graph using DFS:

> Use DFS from every unvisited node. Depth First Traversal can be used to detect a cycle in a Graph. There is a cycle in a graph only if there is a back edge present in the graph. A back edge is an edge that is indirectly joining a node to itself (self-loop) or one of its ancestors in the tree produced by DFS.
>
> To find the back edge to any of its ancestors keep a visited array and if there is a back edge to any visited node then there is a loop and return **true**.

Follow the below steps to implement the above approach:

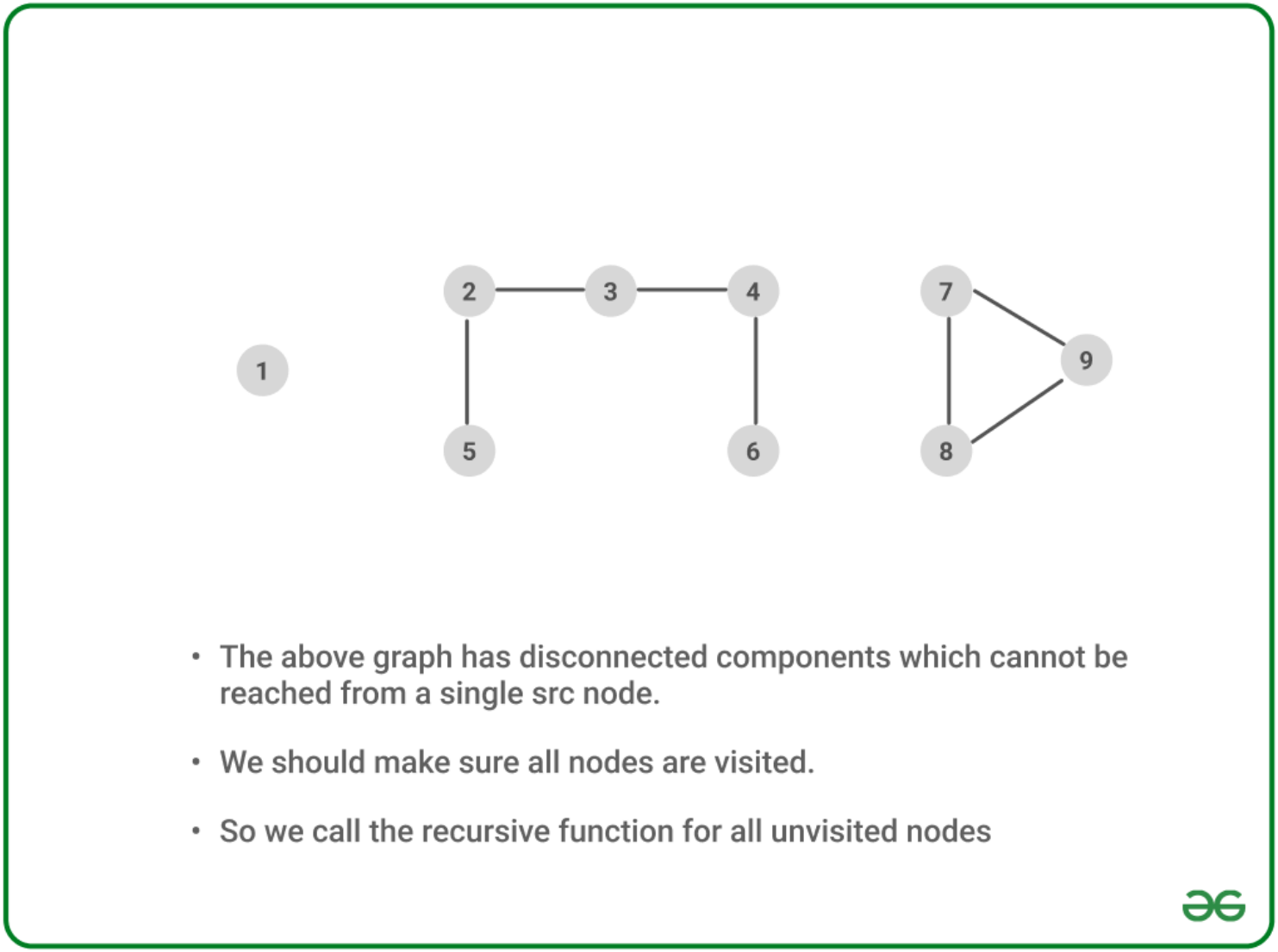
- Iterate over all the nodes of the graph and Keep a visited array **visited[]** to track the visited nodes.
- Run a Depth First Traversal on the given subgraph connected to the current node and pass the parent of the current node. In each recursive
  - Set visited[root] as **1**.
  - Iterate over all adjacent nodes of the current node in the adjacency list
    - If it is not visited then run DFS on that node and return **true** if it returns **true**.
    - Else if the adjacent node is visited and not the parent of the current node then return **true**.
  - Return **false**.

Dry Run:



Another possible scenario:

If No cycle is detected after running Depth First Traversal for every subgraph the there exists no cycle as shown below



Graph with disconnected components

Below is the implementation of the above approach:

C++ Java Python C#

```java
// A Java Program to detect cycle in an undirected graph
import java.io.*;
import java.util.*;
@SuppressWarnings("unchecked")
// This class represents a
// directed graph using adjacency list
// representation
class Graph {

    // No. of vertices
    private int V;

    // Adjacency List Representation
    private LinkedList<Integer> adj[];

    // Constructor
    Graph(int v)
    {
        V = v;
        adj = new LinkedList[v];
        for (int i = 0; i < v; ++i)
            adj[i] = new LinkedList();
    }

    // Function to add an edge
    // into the graph
    void addEdge(int v, int w)
    {
        adj[v].add(w);
        adj[w].add(v);
    }

    // A recursive function that
    // uses visited[] and parent to detect
```

```
// cycle in subgraph reachable
// from vertex v.
Boolean isCyclicUtil(int v, Boolean visited[],
                     int parent)
{
    // Mark the current node as visited
    visited[v] = true;
    Integer i;

    // Recur for all the vertices
    // adjacent to this vertex
    Iterator<Integer> it = adj[v].iterator();
    while (it.hasNext()) {
        i = it.next();

        // If an adjacent is not
        // visited, then recur for that
        // adjacent
        if (!visited[i]) {
            if (isCyclicUtil(i, visited, v))
                return true;
        }

        // If an adjacent is visited
        // and not parent of current
        // vertex, then there is a cycle.
        else if (i != parent)
            return true;
    }
    return false;
}

// Returns true if the graph
// contains a cycle, else false.
Boolean isCyclic()
{

    // Mark all the vertices as
    // not visited and not part of
    // recursion stack
    Boolean visited[] = new Boolean[V];
```

```
        for (int i = 0; i < V; i++)
            visited[i] = false;

        // Call the recursive helper
        // function to detect cycle in
        // different DFS trees
        for (int u = 0; u < V; u++) {

            // Don't recur for u if already visited
            if (!visited[u])
                if (isCyclicUtil(u, visited, -1))
                    return true;
        }

        return false;
    }

    // Driver method to test above methods
    public static void main(String args[])
    {

        // Create a graph given
        // in the above diagram
        Graph g1 = new Graph(5);
        g1.addEdge(1, 0);
        g1.addEdge(0, 2);
        g1.addEdge(2, 1);
        g1.addEdge(0, 3);
        g1.addEdge(3, 4);
        if (g1.isCyclic())
            System.out.println("Graph contains cycle");
        else
            System.out.println("Graph doesn't contain cycle");

        Graph g2 = new Graph(3);
        g2.addEdge(0, 1);
        g2.addEdge(1, 2);
        if (g2.isCyclic())
            System.out.println("Graph contains cycle");
        else
            System.out.println("Graph doesn't contain cycle");
```

```
        }
    }
    // This code is contributed by Aakash Hasija
```


Output

```
Graph contains cycle
Graph doesn't contain cycle
```

Time Complexity: O(V+E), The program does a simple DFS Traversal of the graph which is represented using an adjacency list. So the time complexity is O(V+E).
Auxiliary Space: O(V), To store the visited array O(V) space is required.