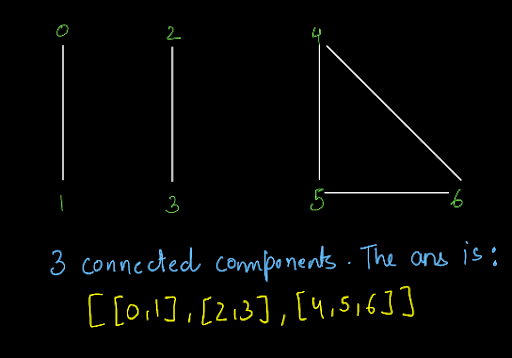
**1. Problem Discussion:**

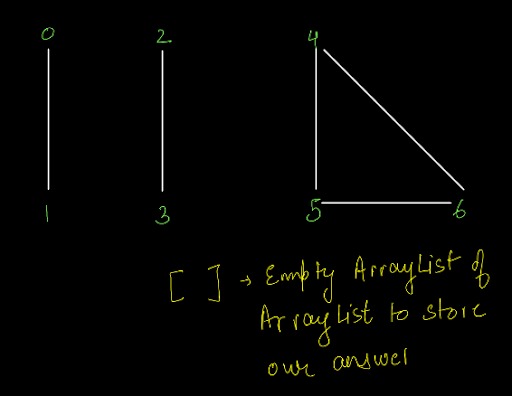
We are given two integers: the first represents the number of vertices of the graph and the second represents the number of edges of the graph. Then, we are given the Edges of the graph in the form of source-neighbor-weight triplets. We have to find the connected components of the graph and return them in the form of an ArrayList of ArrayList. For instance, have a look at the image shown below:

****

So, in this graph, we had three connected components and we filled the vertices of each connected component in a separate ArrayList and put all the ArrayLists in one ArrayList.

**2. Approach:**

So, let us see the procedure that we have to follow. Consider the graph shown below:

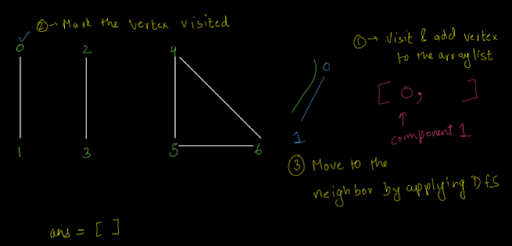
****

Let's first take a look at hasPath function code: We have taken an example graph for which we will fill the connected components. The prerequisite for this problem is the Has Path Problem where we learnt how to apply DFS on a graph.

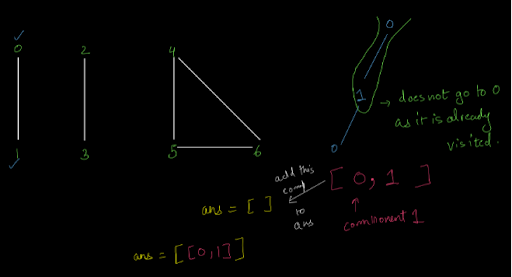
Let us understand the procedure in a nutshell:

1• We will apply DFS for every unvisited vertex of the graph. 2• While we encounter any vertex that has not already been visited, we will mark it visited and add it to the ArrayList. 3• Once we complete the traversal of one component, we will add that component's arraylist to our answer ArrayList of ArrayList.

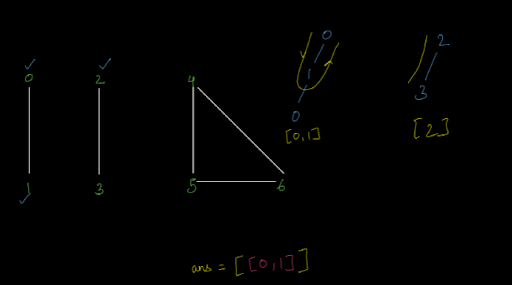
So, let's start implementing this procedure. Let us start from the first vertex i.e. vertex 0. We will create a new arraylist as we are going to traverse a new component. Now, we are at vertex 0. We will mark it visited and then add it to the new ArrayList that we have created. Then, by applying DFS, we will reach its neighboring vertex. Since the only neighbor of vertex 0 is 1, we will reach vertex 1.

****

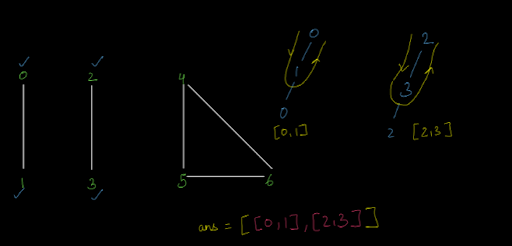
Remember that this is the DFS of a graph. We will visit only those neighbors that are unvisited. Here, the only neighbor is 1 and it is unvisited, so we move to it. Repeat the same steps as above i.e. add the vertex to the component ArrayList and mark it visited. Now, 1 has only one neighbor i.e. vertex 0 and it is already visited. So, we can not go there. Since there is no more neighboring vertex to visit, this component is complete and we will add this component's ArrayList to our answer ArrayList of ArrayList as shown below:

****

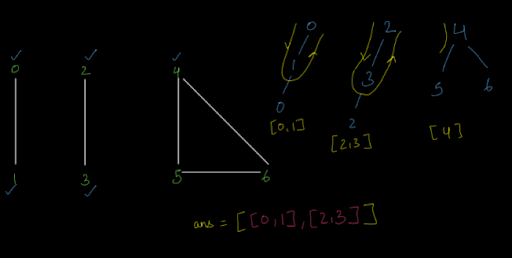
Now, as discussed above, we have to apply DFS for every vertex. So, we will move to the next vertex that is vertex 1. Now, this vertex is already marked visited. So, we will not apply DFS on it. We move to the next vertex i.e. vertex 2. Since this vertex is not visited, we apply DFS on it and create a new ArrayList as we are traversing into a new component. Note: We get to know that we are traversing a new component as after traversing 0 and 1, we did not have any neighbor to traverse to but there are a total of 7 vertices in the graph numbered 0-6. This means that 0 and 1 were part of one connected component that is not connected to the rest of the graph.

****

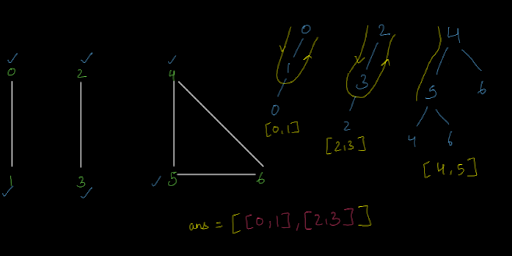
So, as we can see from the image above, we have traversed the vertex 2 and marked it visited. Now, by applying DFS, we move to the next and the only unvisited neighbor is vertex 3.

****

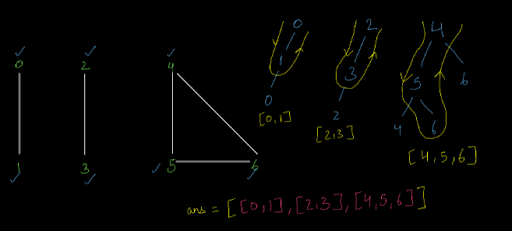
So, we have filled one more component. We applied the DFS for vertex 3. Now, let us apply the DFS for vertex 4. So, we see that vertex 4 is also already visited. So, we won't apply DFS at vertex 4 also. Now, let us move at vertex 5. Here again, we have created a new ArrayList and in that ArrayList, we have added 5 and marked it as visited.

****

Here, we have 2 neighbors that are unvisited. So, we can go to any neighbor that comes in the Euler path. So, we will now visit vertex 5.

****

Now, vertex 5 has only one unvisited neighbor i.e. vertex 6. So, we will visit this vertex too. Also, after visiting vertex 6, we have no other vertex to go to. So, this component will also be completed and added to our answer ArrayList.

****

So, we have completed the DFS for vertex 4 as well. Now, let us try the DFS for vertex 5. Since it is already visited, we will not traverse it and the same will be the case for vertex 6. Now that we have applied DFS on all the vertices, we have got our answer. This is how we will find the connected components of a graph. Now that we have understood the complete procedure, let us write the code for the same.

ConsoleJava

import java.io.\*;

import java.util.\*;

public class Main {

static class Edge {

int src;

int nbr;

int wt;

Edge(int src, int nbr, int wt) {

this.src = src;

this.nbr = nbr;

this.wt = wt;

}

}

public static void main(String[] args) throws Exception {

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

int vtces = Integer.parseInt(br.readLine());

ArrayList<Edge>[] graph = new ArrayList[vtces];

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

graph[i] = new ArrayList<>();

}

int edges = Integer.parseInt(br.readLine());

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

String[] parts = br.readLine().split(" ");

int v1 = Integer.parseInt(parts[0]);

int v2 = Integer.parseInt(parts[1]);

int wt = Integer.parseInt(parts[2]);

graph[v1].add(new Edge(v1, v2, wt));

graph[v2].add(new Edge(v2, v1, wt));

}

boolean[] visited = new boolean[vtces];

ArrayList<ArrayList<Integer>> comps = new ArrayList<>();

for(int v = 0; v < vtces; v++){

if(visited[v] == false){

ArrayList<Integer> comp = new ArrayList<>();

gcc(graph, v, visited, comp);

comps.add(comp);

}

}

System.out.println(comps);

}

public static void gcc(ArrayList<Edge>[] graph, int src, boolean[] visited, ArrayList<Integer> comp) {

comp.add(src);

visited[src] = true;

for (Edge e : graph[src]) {

if (!visited[e.nbr]) {

gcc(graph, e.nbr, visited, comp);

}

}

}

}

**3. Analysis:**

Time Complexity:

The time complexity of the above code is O(V) as we are going to visit every vertex exactly once.

Space Complexity:

The space complexity of the above code is O(h) where h is the height of the recursion stack.