Question1

a)

We need N-1 bridges to visit any village starting from the 1st village where N is count of cities in graph. Since each city separated each other, we need to make sure connection between cities. In this first example we have 4 cities according to reach each village in 1st village we need to construct bridge to other cities so that we need 3 bridges to reach each village.

b)

We need to find city count in given graph. If we find count of connected components in graph we directly found number of cities. Since we have different components and they are not connected each connected component will be a city. To find count of cities we can use a non-weighted search algorithm to find connected components. We can select random node until the visit each node in given graph. So we can search in given node then check visited nodes in unique node list. We can use a while loop until the reach each node visited in given graph. In loop we can select random unvisited node and apply search algorithm to find neighbour nodes and check visited, after each search algorithm applied we can increase bridge count since each search algorithm we select a randomly connected component but we need to decrease 1 at the before of while loop since we start a specific city therefore we don't need a bridge for starting city.

c)

Algorithm 1 Breadth-first search algorithm for given graph

```
BFS(graph: Input, s: Input, nodeNames: Input)
  $visited \leftarrow array[False * (max(graph) + 1]$
  $queue \leftarrow array[empty]$
  $queue.append(s)$
  $visited[s] = true$
  **while** $len(queue) \neq 0$ **do**
    $output \leftarrow queue.pop()$
    $nodeNames.remove(output)$
    **for** node in graph[output] **do**
      **if** $visited[node] \neq True$ **then**
        $queue.append(node)$
        $visited[node] \leftarrow True$
      **end if**
    **end for**
  **end while**
  return nodeNames
```

In Algorithm 1, we implement breadth-first search algorithm. It takes an graph, start node and unique node names. Firstly create an array which contains only False values in beginning to check visited state. After that we use a queue and enqueue our start node and until the queue will be empty we simply search neighbour nodes and check visited with that function our aim is to find a specific city and remove these city's village names from nodeNames variable and return these variable to MINBRIDGE function.

In Algorithm 2, we implement MINBRIDGE function, firstly we find unique node names from our adjacency list like hashtable representation then we use a while loop until the reach each village in given graph. In while loop we select first index node then apply BFS to these node therefore each BFS function call we simply find a city and villages after

Algorithm 2 An algorithm to find count of minimum bridges in given graph

```
MIN BRIDGE(graph: Input, n: Input)
  $bridgeCount \leftarrow 0$
  $nodeNames \leftarrow graph.keys()$
  while $len(nodeNames) \neq 0$ do
      $startNode \leftarrow nodeNames[0]$
      $nodeNames \leftarrow BFS(graph, startNode, nodeNames)$
      $bridgeCount \leftarrow bridgeCount + 1$
  end while
  return bridgeCount -1
```

that we increase bridgeCount variable with 1. At the end of while loop we decrease value of bridgeCount with 1 because we don't want to count start city to construct a bridge since we already can reach each village current city.

d) In c section of Question 1, i show that pseudocode to find minimum number of bridges. In that code we use a while loop to find different connected components in graph and we call BFS function in each step of loop. In BFS function we use a inner loop; first loop is a while loop to check if queue is empty or not and second loop check if current node is visited or not. Time complexity of BFS function is O($V + E$) where V is a number of vertices and E is a number of edges in the graph. Since we have a loop out of BFS function and these loop counts until the reach number of connected components count, time complexity of MINBRIDGE function will be O($N(V + E)$) where N is count of connected components in given graph.

QUESTION2

a)

Euler tour can define as a single cycle that traverse each edge of given graph only once but it can be complex cycle too. So given graph is a strongly connected we can separate G into multiple edge-disjoint cycles so that their combination will give us an Euler tour. If we want to if G has an Euler tour we need to look simple cycles. In each cycle each vertex in the cycle has one entering edge and one leaving edge therefore each vertex as v has in-degree(v) = out-degree(v) where the degrees are either 1(simple cycle) or 0 (if v is not on the simple cycle). If we want to add new in and out degrees over all edges it proves that G has an Euler tour. So we simply divide our graph problem more specific cycles using edge-disjoint cycles and check vertexes with given condition and prove if we have a situation like in-degree(v) != out-degree(v) it can't be an Euler tour because we can't go back other edges and walk with them since graph structure will be more strict. For example if we have mostly out-degree edges we cannot go back and visit other edges. In H1 A Node have 2 out-degree but 0 in-degree, if we start from A and tour like D and C and B, we cannot back A using (A-B) edge since it's directed graph but we have to instead if we had (B->A) relation instead of (A>B) we can say that given H1 have an Euler tour. In H2 every node have equal in-degree and out-degree.

b)

In given graph there can be an Euler tour if we provide in-degree(v) = out-degree(v) condition. To find an example Euler tour we can select a node and pick any outgoing edge for example v, since in-degree and out-degree must be equal we can pick some outgoing edge of u (v, u) and continue to visiting edges. Since we started the cycle by visiting one of v node's outgoing edges and we remain a leaving edge so we can visit for any vertex other than v, at the end of process cycle must return to v to prove claim in above. In EULER TOUR algorithm we select all edges as unvisited and we use while loop to traverse arbitrarily edges so time complexity of given algorithm will be O(E). At the start of algorithm we create an empty list and then select an initial node to start arbitrarily walk process and also we put firstNode variable in visitedNodes list We search unvisited out-edge with outSample function if these edge not in visitedEdges we use these edge and reach secondNode and put these variable on visitedNodes list then we continue to search another unvisited edge but now with new node and process keep continue until the reach end of euler tour then return euler tour path in visitedNodes path.

Algorithm 3 Check given graph has an Euler tour or not

```
CHECK EULER TOUR(graph: Input)
  $visitedEdges \leftarrow array[empty]$
  $firstNode \leftarrow graph[0]$
  $visitedNodes \leftarrow array[firstNode]$
  while $outSample(firstNode)$ not in $visitedEdges$ do
      $secondNode \leftarrow outSample(firstNode)$
      $firstNode \leftarrow secondNode$
      $visitedNodes.append(firstNode)$
  end while
  return visitedNodes
```