Let G be an undirected graph. We use DFS in G to solve a couple of problems associated with G.

Part I

Read the number n of vertices and the number e of edges in G. The vertices of G will be numbered $0, 1, \ldots, n - 1$. The e edges (with each edge being specified by two different vertex numbers in the range mentioned above) are then read from the user. Store the graph in the *adjacency-list* format. Each adjacency list should be a linked list storing the numbers of the neighboring vertices. For an undirected edge (u, v), store both u in the adjacency list of v, and v in the adjacency list of u. Print the adjacency lists for each vertex.

Part II

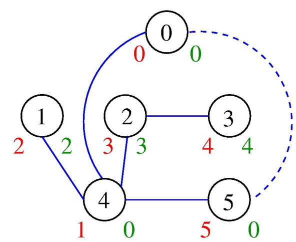
In this part, you check whether G is bipartite. Recall that G is bipartite if and only if it does not contain any cycle of odd length. Modify the DFS function taught in the class to detect if G has any cycle of odd length. Note that the input graph G need not be connected.

Part III

Let u, v, w be three different vertices in G. By the removal of v from G, we mean a graph H which is the same as G except that H does not contain (1) the vertex v, and (2) all the edges in G in which v is an endpoint. Suppose that in G, the vertices u and w are connected. That means that there are one or more paths from u to w. If all these paths go via v, then the removal of v from G gives a graph H in which u and w are disconnected from one another. We call v a *critical vertex* if its removal disconnects at least one pair of different vertices. In this part, your task is to locate all critical vertices in G.

The obvious algorithm is to consider each vertex v, delete it from G, and check whether the new graph has more connected components than G. This takes a total running time of $O(n\,(n + e))$.

Design an $O(n + e)$-time algorithm to locate all critical vertices in G. You need to modify the standard DFS procedure. Give serial numbers to the vertices in increasing sequence as they are visited. Also maintain a criticalness value for each vertex v. This is initialized to the serial number of v, and is meant to store the minimum of this initial value, the criticalness values of all vertices (excluding v) in the DFS subtree rooted at v, and the serial numbers of all vertices that have back edges from v. Handling back edges is necessary, because these edges provide escape routes from a vertex to an ancestor without following the tree edges. Update the criticalness values appropriately, and use them to detect critical vertices. Also note that the root vertex requires a separate treatment (because it has no proper ancestor to escape to).



Consider the graph shown in the adjacent figure. This graph is not bipartite, since it contains the 3-cycle (0, 4, 5). The solid edges are DFS tree edges, and the dotted edge is a back edge. The serial numbers of the vertices are shown in red, and the criticalness values in green. In this graph, Vertices 2 and 4 are critical. The removal of Vertex 4 disconnects Vertices 2 and 3 from Vertex 1, for example. The back edge (5, 0) changes the criticalness value of Vertex 5 to 0. Because of this, Vertex 4 too receives a criticalness value of 0.

Sample Run

For the graph shown in the adjacent figure, the program runs as follows.

```
+++ n = 6
+++ Neighbor list:
    0 :  4  5
    1 :  4
    2 :  3  4
    3 :  2
    4 :  0  1  2  5
    5 :  0  4
+++ Running DFS
    0  4  1  2  3  5
The graph is not bipartite
+++ The critical vertices of G are:
    4 is critical for 1
    2 is critical for 3
    4 is critical for 2
```