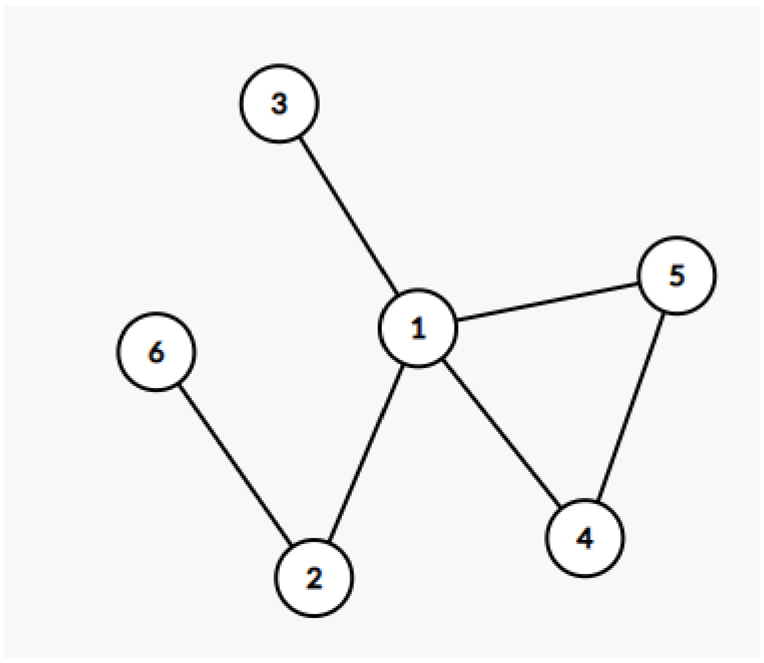
Bridges

Given an undirected graph G, we say that an edge u-v is a **bridge(cut edge)** if the removal of edge u-v increases the number of connected components in G.

For Example:



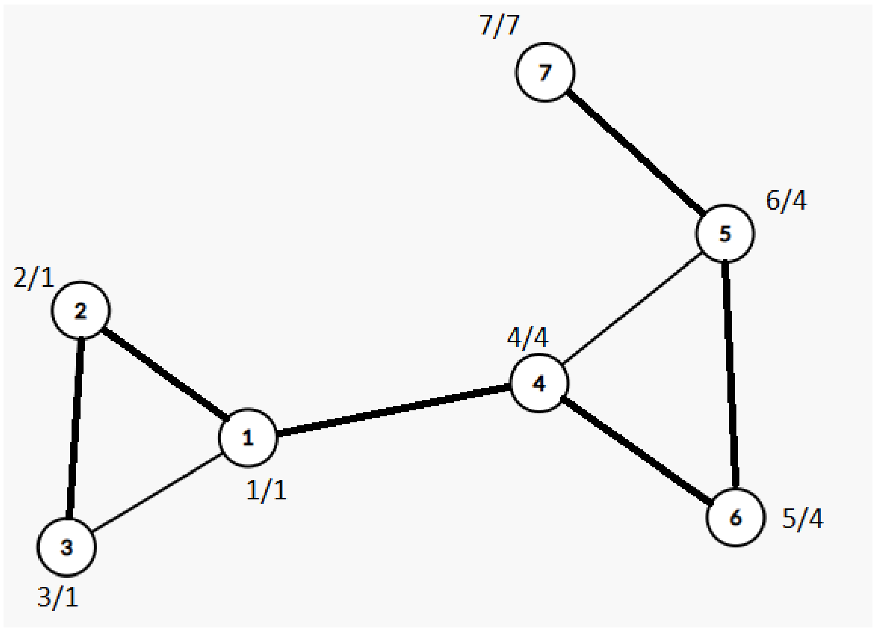
In the above diagram, edges **2-6**,**1-2**,**1-3** are **bridges**, while the remaining edges are not bridges as they are a part of a **cycle** i.e the graph is supported by a back edge even if the edge is removed.

The approach to finding all the bridges in a graph is again based upon DFS similar to that of finding articulation points in the graph.

Perform a DFS by taking any arbitrary vertex as the root resulting in the formation of a DFS tree having tree edges and back edges. Let's say we are in the DFS looking at the edges starting from the vertex **v**, we say that the edges **v-to** is a bridge where to is the direct descendant of v or the child of v in the DFS tree, when **there is no back edge from 'to' or any of its descendants to the vertex 'v' or any of its ascendants**.

Hence we can check this fact efficiently by using the same approach as discussed in finding the articulation point with a slight modification that **there exists a back edge from 'to' or any of its descendants to the vertex 'v' or any of its ascendants iff low[to] <= tin[v]**. So, the **edge v-to in the DFS tree is a bridge iff low[to] > tin[v]**.

For Example:



Let us consider the above graph where we start the DFS from node **1**, which results in the formation of a DFS tree with the edges belonging to the DFS tree being marked as bold, we mark the discovery and update the low times of the nodes as we traverse the graph. Each node is labelled with **a/b**, where **a** represents the **discovery time or tin[v]**, whereas **b** represents the **low time or low[v]**, for a node v.
The DFS traversal is:
1->2->3->4->6->5->7

According to the algorithm, it can be clearly observed that due to the condition low[to] > tin[v], edges
1-4 and 5-7 are **bridges**.

Pseudocode:

```
/*
        The function takes an input of graph G(adjacencyList), tin, low and vis arrays,
        and the current node 'v' and parent 'par'.
*/
function findBridges(G, v, par, tin, low, vis)

        vis[v] = true
        tin[v] = curtime
        low[v] = curtime
        curtime = curtime + 1


        for to in G[v]
```

```
//   If to == par, then the edge is back to the parent, we skip the iteration
if to == par
        continue
//   If to is already visited, then we may have a back edge from 'v' to 'to'
if vis[to] == true
        low[v] = min(low[v], tin[to])
else
        findBridges(G, to, v, tin, low, vis)
        //   Checking for minimum value of low[v]
        low[v] = min(low[v], low[to])
        //   Checking if the edge (v, to) is a bridge
        if low[to] > tin[v]
                Bridge(v,to) = true
```

Time Complexity: O(V + E), as we perform a DFS traversal of the given graph, where V is the number of vertices and E is the number of edges in the graph.

NOTE: The above function fails for graphs having multiple edges, in that case we need to pass the id of the edge(edge number) instead of par or we can check if the edge(u,v) we are considering as a bridge is not a multiple edge.