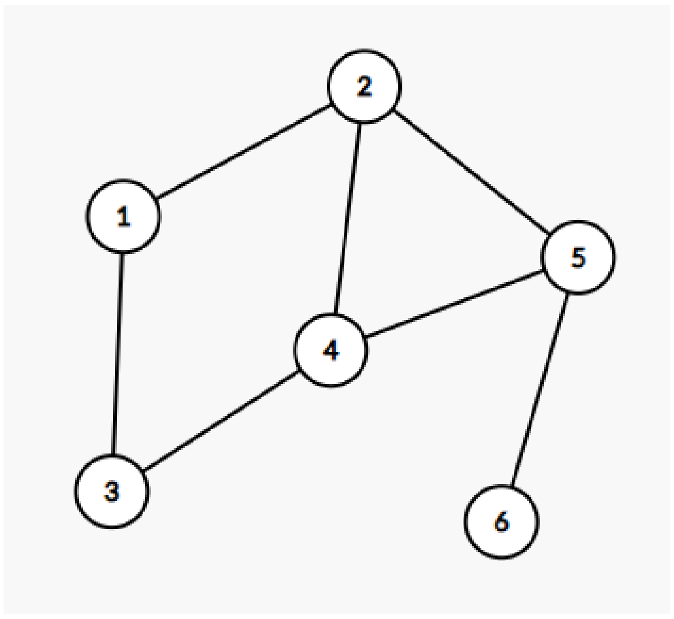
Biconnected Components

Biconnected Graph

Given an undirected graph G, G is said to be biconnected iff:

1. G is connected i.e there exists a simple path between every pair of vertices
2. G has no articulation points i.e the removal of any vertex does not increase the number of connected components of G.

For Example:

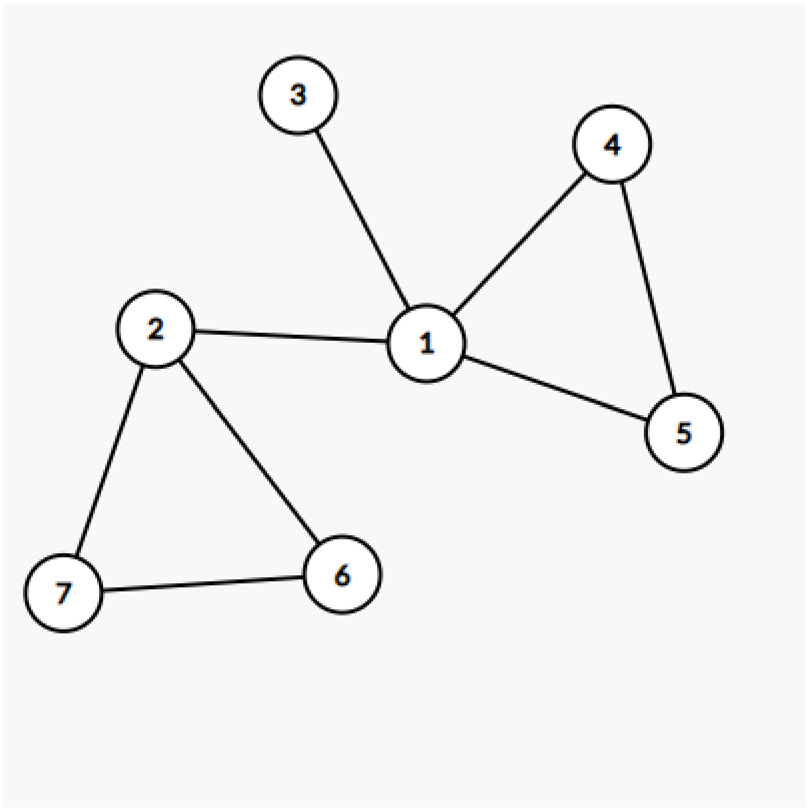


The graph in the above diagram is **connected** and has **no articulation points**.

Biconnected Component

A biconnected component is a maximal biconnected subgraph of a graph. According to the definition a disconnected graph too has biconnected components.

For Example:



In the above graph, the different biconnected components are:

1. 2-7, 6-7, 2-6
2. 1-3
3. 1-2
4. 1-4, 4-5, 4-5

The following algorithm helps us to efficiently find all the biconnected components of a graph: Since we need to make sure that our biconnected component should be connected and does not contain any articulation point, we maintain a stack of edges while performing DFS traversal of the graph and push the edges we traverse during the DFS into the stack.
As we backtrack from the DFS traversal, say currently we are at edge **v-to**, we check if v is an articulation point i.e **low[to] >= tin[v]**, if v is an articulation point then we pop the edges from the stack until the top of the stack is 'v-to' and include 'v-to' too. The popped set of edges will form one biconnected component of the given graph.

Pseudocode:

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

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


child = 0
for to in G[v]
    if to == par
        continue
    //   If to is already visited and node to is still in the current stack
    if vis[to] == true and tin[to] < low[v]
        low[v] = tin[to]
        st.push(v, to)
    else
        st.push(v, to)
        child++
        articulationPoints(G, to, v, tin, low, vis)
        //   Checking for minimum value of low[v]
        low[v] = min(low[v], low[to])
        //   Checking if v is an articulation point
        if low[to] >= tin[v] and par != -1
            art_point[v] = true
            //  Popping off the elements of stack until we get edge(v,to)
            while st.top() != (v, to)
                print st.top()
                st.pop()
            //   Popping off the edge (v, to)
            print st.top()
            st.pop()

//   If the current node is the root node and has greater than 1 child
if par == -1 and child > 1
    art_point[v] = true
    //   Popping off the elements of the stack until we get edge (v, to)
    while st.top() != (v, to)
        print st.top()
        st.pop()
    //   Popping off the edge (v, to)
    print st.top()
    st.pop()
```

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.