## Construction

**How to construct a Bridge Tree from a given undirected connected graph G?**
The algorithm to construct a Bridge Tree is based upon DFS and BFS traversals of the graph:

1. Start a DFS from any arbitrary node.
2. In every step of DFS, start a BFS from the current node and visit all the nodes until you find a bridge edge. Basically, this BFS will visit all the nodes in a single bridge component.
3. When a bridge edge(v - to) is encountered during BFS, start the DFS from 'to', this is meant to jump to the node belonging to another bridge component, and hence again start a BFS from this node(to) to visit all the nodes present in the bridge component of node 'to'.
4. Once the exploration from node 'to' is complete because of the recursive nature of DFS, we return to the previous BFS and again start visiting the remaining nodes belonging to the bridge component node 'v'.

Hence the recursion helps us in finding all the bridge components of the graph where DFS is used to make jumps through the bridge edges and BFS is used to visit all the nodes belonging to the bridge component of the current node in the DFS.

**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



/*
    The function takes an input of graph G(adjacencyList), vis and Bridge arrays, and
    the current node ‘v’, and the number of bridge components represented by
    nocomp
*/
function bridgeComp(G, v, nocomp, Bridge, vis)

    //  Initalizing curr_comp to the Bridge component number ‘nocomp’
    curr_comp = nocomp
    //  Initializing a separate queue for the current bridge component
    Q[curr_comp].push(v)
    vis[v] = true

    while Q[curr_comp] is not empty
        node = Q[curr_comp].front()
        Q[curr_comp].pop()
        for to in G[node]
            If the node is already visited skip the current iteration
            if vis[to] == true
                continue
            e = (node, to)
            /*
                If the edge (node, to) is a bridge, then we make a DFS call
                from the node ‘to’ to make a jump to explore the next
                bridge component
            */
            if Bridge[e] == true
                nocomp++
                /*
                    Building the bridge tree in form of adjacency list
                    where the edge is (curr_comp, nocomp)
                */
                Bridge_Tree[curr_comp].push(nocomp)
                Bridge_Tree[nocomp].push(curr_comp)
                bridgeComp(G, to, nocomp, Bridge, vis)
            /*
```

```
                Else continue the exploration of nodes with BFS of the
                current bridge component
        */
        else
                Q[curr_comp].push(to)
                vis[to] = true
```

**Time Complexity: O(V + E)**, as we perform a DFS and BFS 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.