Q2> Algorithm assignment 6.2
Brute force approach would be to remove one island at a time and check if the islands still stays connected by running DFS algorithm. If it disconnects then add this island to the result set. Do this for all the islands. This approach takes I * (I + B) time, where I is the number of islands and B is the number of bridges.

Efficient approach: : Let i1, i2, i3,…in be the islands in Micronesia. Let b1, b2, b3, b4…...bm be the bridges connecting the islands. This forms a graph representation with islands as nodes and bridges as edges of the Graph.

The idea involved here is to use DFS algorithm to traverse the graph which is formed by islands and bridges. An island removal disconnect the islands of Micronesia if the following case holds true:

- **i>** If Island is the starting point of the DFS traversal i.e. if it's root node and it has more than one adjacent island connected to it by bridge
- **ii>** If Island not the starting point the it is said to be a disconnecting island iff none of the islands in the DFS subtree rooted at the island has a bridge connecting to it's ancestor.

Even If one of the above condition holds true then we add the island to the resStack which holds all the islands whose removal will disconnect the islands of Micronesia.

Steps involved:

1. Each island has an index which is lowest possible number that can be assigned. It also has a field which keeps track of lowest index island that is reachable from any island using one or more brides or an back edge which connects the island to an already visited node. Each island has an Boolean visited field. Previous to keep track of the previous island visited.
2. Let resStack be a stack which is used to hold all islands whose removal will disconnect Micronesia.
3. Mark all islands as unvisited initially by setting it to false
4. Loop through all the unvisited islands to perform depth first
   4.1 Let neighborIsland keep track of count of all neighboring islands
   4.1 We assign the island the smallest index possible
   4.2 Mark the current island as visited and set the lowest index island Reachable island as itself initially.
   4.3 Add this island on to the stack
   4.4 Loop through all the neighboring islands of the currently considered Island
      4.4.1 If island is not visited then Increase the neighbour count, Set the previous for the neighbour as current island and recur on this island and update the lowest index island reachable from the current node to the minimum value i.e. minimum(isl1.low, isl2.low) where isl1 is current island and isl2 is neighbouring island Check if it's an disconnecting island based on whether it is first node Or not. If it is then add it to the resStack
      4.4.2 If island is already visited and if it's previous is not the current island

then update the lowest possible index island that is reachable to the minimum value.i.e. minimum(isl1.low, isl2.index) where isl1 is current island and isl2 is neighbouring island

5. return resStack

Pseudocode :

We assume the island provided is of type struct Island as described below

```
Struct Island
{
        Integer Indx
        Integer Previous
        Integer low
        Boolean visited
}

set index = 0;
FetchIslandsWhichDisconnects( islands, bridges )
{
        resStack stack used to store all the islands whose removal disconnects
                    islands

       // Setting initial values for fields of island
        for( int i=0;  i < bridges; i++)
        {
                Islands.Indx = -1
                Islands.low = -1
                Islands.Previous = -1
                Islands.visited = false
        }

        // Looping through all islands
        for (Island  isl in islands)
        {
                If( isl.visited == false)
                {
                        FetchDisconnectingIslands ( isl, bridges, resStack);
                }
        }
}


FetchDisconnectingIslands( isl1, bridges, resStack)
```

```
{
        // To handle the root node case we keep track of the count of the islands
        // connected  to the current island
        Int  neighborIsland =0;

        //  mark the island as visited
        Isl1.visited = true;

        // Number the island and initialize the lowest index island that is
        // reachable from the current node as itself.
        Isl1.indx = index;
        Isl1.low = index;

        for (isl1, isl2) in bridges
        {
                // if neighboring island is not visited yet then recurse on it
                If (isl2.visited == false)
                {
                        // increment the number of neighbors count
                        neighborIsland++

                        // set previous island for the neighbor as current island
                        Isl2.previous = isl1

                        // recurse on neighboring island
                        FetchDisconnectingIslands (isl2, bridges, resStack)

                        // If any of the islands visited during the recursive call
                        //  has an alternate route to the island which is visited before
                        // isl1 then update low
                        Isl1.low =  minimum( isl1.low, isl2.low)

                        // handling the first island visited case
                        If( isl1.previous == -1 and neighbourIsland > 1 )
                                resStack.push(isl1)

                        // handling the interior node case.
                        // It's removal will disconnect the islands iff there is no
                        // alternate route to the isl1 ancestors in the traversal path
                        else if( isl1.previous != -1 and isl2.low  >=  isl1.indx)
                                resStack.push(isl1)
                }
                // If island is already visited update the lowest index island that is
                // reachable from isl1
                else if ( isl1.previous != isl2)
                {
                        Isl1.low = minimum ( isl1.low, isl2.indx)
                }
        }
}
```

Time complexity :

This involves visiting all islands and bridges at-most two times . Therefore the time complexity **T(n) = (I + B)** where I is the number of islands and B is the number of bridges.

Space complexity : Since this algorithm involves a stack to keep track of the result which has all the disconnecting islands, **Space Complexity = I** where I is the number of islands. All the other variables take constant space hence ignored.