1. The complexity of **BFS** implemented using an Adjacency Matrix will be $O(|V|^2)$. This is because every time we want to find what are the edges adjacent to a given vertex u, we would have to traverse the whole array $AdjacencyMatrix[u]$, which is of length $|V|$.

2. We will use **DFS** to solve this problem in $O(V + E)$ time.

   (a) Perform a DFS of G starting at an arbitrary vertex. The path required by the problem can be obtained from the order in which DFS explores the edges in the graph.

   (b) When exploring an edge (u, v) that goes to an un-visited node the edge (u, v) is included for the first time in the path.

   (c) When DFS backtracks to u again after v is marked visited, the edge (u, v) is included for the second time in the path, this time in the opposite direction (from v to u).

   (d) When DFS explores an edge (u, v) that goes to a visited node we add (u, v) and (v, u) to the path. In this way each edge is added to the path exactly twice.

3. Finding connected components for an un-directed graph can be done using either **BFS** or **DFS** starting from each un-visited vertex. Below are steps based on DFS.

   (a) This algorithm utilizes the property that nodes of strongly connected component form a sub-tree in the DFS spanning tree of the graph.

   (b) A DFS is run over the nodes and the sub-trees of SCCs are removed and recorded as they are encountered.

   (c) Two values $dfs\_num(u)$ and $dfs\_low(u)$ are maintained for each of the users. $dfs\_num(u)$ is the value of the counter when the node u is explored for the first time. $dfs\_low(u)$ stores the lowest dfs_num reachable from u which is not the part of another SCC.

   (d) As the nodes are explored, they are pushed onto a stack.

   (e) The unexplored children of a node are explored and $dfs\_low(u)$ is accordingly updated.

   (f) A node is encountered with $dfs\_low(u) == dfs\_num(u)$ is the first explored node in its strongly connected component and all the nodes above it in the stack are popped out and assigned the appropriate SCC number. These can then be printed.

   Time complexity of is $O(V + E)$ - where V is the number of vertices, and E the number of edges, in a graph. Space is $O(V)$.

4. Algorithm: Create a set MST that keeps track of vertices already included. Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign key value as 0 for the first vertex so that it is picked

first. While MST doesn't include all vertices. Pick a vertex u which is not there in MST and has minimum key value. Include u to MST. Update key value of all adjacent vertices of u. To update the key values, iterate through all adjacent vertices. For every adjacent vertex v, if weight of edge $u - v$ is less than the previous key value of v, update the key value as weight of $u - v$ The idea of using key values is to pick the minimum weight edge from cut.

```
bool isValidEdge(int u, int v, vector <bool> MST)
{
    if (u == v)
    {
        return false;
    }

    if (MST[u] == false && MST[v] == false)
    {
        return false;
    }
    else if (MST[u] == true && MST[v] == true)
    {
        return false;
    }

    return true;
}

void primAlgo(int adjacencyMatrix[][V])
{
    vector <bool> MST(V, false);

    // Include first vertex in MST
    MST[0] = true;
    while (edge_count < V - 1)
    {
        for (int i = 0; i < V; i++)
        {
            for (int j = 0; j < V; j++)
            {
                if (adjacencyMatrix[i][j] < min)
                {
                    if (isValidEdge(i, j, MST))
                    {
                        min = adjacencyMatrix[i][j];
                        a = i;
                        b = j;
```

```
                    }
                }
            }
        }

        if (a != -1 && b != -1)
        {
            // Print the respective edges
            // Mark index a and b true in MST
        }
    }
    // Print minimum cost here
}
```

Time Complexity is $O(V^2)$ for matrix representation. If the input graph is represented using adjacency list, then the time complexity of Prim's algorithm can be reduced to $O(ElogV)$

5. One method for computing the maximum weight spanning tree of a graph G – using Kruskal – can be summarized as follows (Negate all edge weights, then run the minimum spanning tree algorithm):

   (a) Sort the edges of G into decreasing order by weight. Let T be the set of edges comprising the maximum weight spanning tree. Set $T = \phi$.

   (b) Add the first edge to T.

   (c) Add the next edge to T if and only if it does not form a cycle in T. If there are no remaining edges exit and report G to be disconnected.

   (d) If T has $n - 1$ edges (where n is the number of vertices in G) stop and output T. Otherwise go to step 3.