Data structures assignment - 5.

1) Question #2 can be solved using Prim's algorithm i.e computing an MST of the input graph.

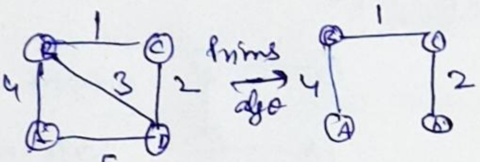
Algorithm :

a) Compute the MST using Prim's algorithm.

b) Traverse the MST from the source node to the destination node.

c) Calculate the weight using the max function in C++. Max function gives the maximum weight encountered during the traversal.

eg -



Undirected graph MST.

$A \to D$

$\max(\max(4,1), 2)$

$= 4.$

2) For the first part of the question, I'll give a proof and for the second part, an counter example.

~~Proof that every minimum spanning tree is a minimum bottleneck spanning tree: Let T be MST, T' be MBST. Consider the edge of T and T', that is having a maximum weight. Following 3 cases are possible -~~

~~a) If both the edges are same - Then T is a MBST i.e. every MST is an MBST, due to the G selected arbitrarily. G(V, E) represents a graph and T and T' are its MST and MBST respectively.~~

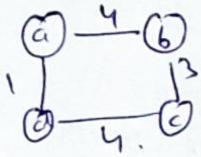
~~b) If both the edges are different -~~

Proof that every minimum spanning tree is a minimum bottleneck spanning tree - Let T is a MST. suppose there is some edge in it (a, b) that has a weight which is greater than the weight of MBST. Then, let $V_1 \subset V$ that are reachable from a in T, without going through b. Let us define V_2 symmetrically. Then we will have a cut that separates V_1 and V_2. The edge that we could add across this cut is the one of minimum weight, so we know that there are no edge across this cut of weight less than $w(u,v)$.

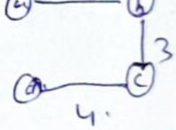
But we ~~have~~ knew that there is a bottleneck spanning tree with less than that weight. This ~~a~~ is a contradiction, as a MBST must have an edge across this cut.

<u>Counter-example</u> -

Graph G



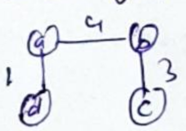
. For this graph we have this as a MBST i.e.



weight = 11.

bottleneck edge weight = 4.

However, MST for this is



weight = 8.

Thus, above MBST is not a MST. Hence, proved that every MBST is not a MST.

4) Given an adjacency-list representation, Adj of a directed graph, the out-degree of a vertex u is equal to the length of Adj[u], and if we sum it over all lengths, we will get |E|. Hence, time to compute the out-degree of one vertex is $\Theta(|Adj(u)|)$ and for all, it's <u>$\Theta(V+E)$</u>.

The in-degree of a vertex u is equal to the number of times it appears in all the lists in Adj. Therefore the time to compute the in-degrees of all vertices is <u>$\Theta(VE)$</u>, if we search all the lists for each vertex.

5) Algorithm for computing G^2 from G. -

a) Convert the adjacency matrix to adjacency list. T.C. = $O(V^2)$

b) Apply BFS to compute G^2.

step (a) - To convert an adjacency matrix into the adjacency list, create an array of lists and traverse the adjacency matrix. If for any position (i, j) in the ~~array~~ matrix i.e. mat[i][j] = 1, it means there is an edge from i to j, so insert j in the list at i^{th} position in the array of lists. Time complexity is $O(V^2)$ as we are traversing the whole matrix.

step (b) - BFS basically traverses the graph level by level. i.e. it traverses all the vertices at a distance of 1 from source vertex x, then at a distance of 2 and so on. Below is the pseudocode -

```
For each vertex v in V
{
    do a BFS with v as source vertex
    {
        for all vertices u at a distance of 2 from v, add u to adjacency
        list of v and terminate BFS.
    }.
}.
```

~~So, total~~ Time complexity is $O(V(V+E))$ as BFS takes $O(V+E)$ as we are traversing each vertex.

Final time complexity is $O(V(V+E)) + O(V^2) = \underline{O(VE)}$.

3) A cliq is basically a strongly connected components. A directed graph is said to be strongly connected if $\exists$ a path between all pairs of vertices. Strongly connected component of a directed graph is a maximal strongly connected subgraph, Kosaraju's algorithm can be used to find a cliq, which is as follows -

a) Create a empty stack S and traverse the graph using DFS.

b) In the traversal, push the vertex to the stack, once you have called recursive DFS for adjacent vertices of a vertex.

c). Now, reverse the directions of all arcs. Arc is a basically a directed edge. This will give you a transpose graph.

d) Pop a vertex from S, one by one, until it's empty. If the popped vertex is 'v', say. Then take it as a source and do DFS starting from v. It will print the strongly connected components of v.

Kosaraju's algorithm only deals with DFS. Actually, it does DFS two times.

Time complexity – In short, we can say that the above algorithm is calling DFS, finding reverse of graph and again calls DFS. DFS takes O(V+E) and reversing a graph takes O(V+E). Note that these are for adjacency lists representation of graphs.