Question 1

1. Algorithm

A. Data Structures

a) Adjacency Matrix (Adj)

- Adj(x) ← List of all the adjacent nodes with “x”

b) Partition Sets (P)

- P (0) stores all the nodes of Partition A
- P (1) stores all the nodes of Partition B

c) Storing visited set of the nodes for DFS (V)

- If node is in V, that means it is visited

B. IS_BIPARTITE

- Check for every connected component in graph, if any violates the rule then return false

```
V ← {}
For each node in Adj
    If node is not in V
        If DFS_BP (node, 0) is TRUE
            Return FALSE
Return TRUE
```

C. DFS_BP

- DFS_BP will return TRUE if the given connected component violates the bipartition, it will store the Label of that component into “Bad” Set
- DFS will return TRUE immediately if any of the following conditions apply

a) Self-Loop is found
b) If any subgraph of adjacent node is violating the rule
c) If adjacent node is in the same set

```
add x to V
add x to P(clr)

For each node in Adj(x)
    If node is equal to x
        Return TRUE
```

```
        If node is not in V
           If DFS_BP (child, reverse(clr)) is TRUE
               Return TRUE
        Else if node is in P(clr)
            Add Label(x) to Bad set
            Return TRUE

    Return FALSE
```

2. Proof of Correctness

- We are checking for every component's bipartiteness using DFS_BP
- Les Prove that DFS_BP always return a bipartite set otherwise return TRUE
- 1. Proof of Coverage

  * By using set V, we make sure that every node is visited only once

  * A node will not get visited only when graph is not bipartite

- 2. Proof by Contradiction

  * Let's say that there are two nodes "u" and "v" in G, s.t. u and v both belong to same set

  * Now during DFS let's say "u" is visited first and added to set(0)

  * If at any point of time if "v" is also added to set(0), then during the traversal of adjacency list of "v" we will encounter that

  * "u" is already visited and is in the same set, then we immediately return TRUE, meaning that graph is not bipartite

  * Hence there will be never be a case when two adjacent nodes are in the same set

3. Time Complexity (worst-case)

N : Number of nodes currently present in the graph

M : Number of edges currently present in the graph

DFS_BP:

- Every node only visited once = O(n)

- Every Edge only visited twice = O(2*m) = O(m)
- T : O (n + m)

Over All: T = O (N*log(N) + M)

Question 2

1. Dynamic Algorithm

D. Data Structures

d) Dynamic Adjacency Matrix (Adj)

- Adj(x) ← List of all the adjacent nodes with "x"
- Adj(x)(y) ← Number of edges between x and y

e) Label Map (Label)

- We will be keeping track of connected components to reduce the time complexity
- We use label to represent all nodes of the same connected component
- Label(x) ← Label of Component in which "x" node is

f) Storing Size of Components with given Label (Size)

- Size(label) ← Size of component with given "label"

g) Partition Sets (P)

- P (0) stores all the nodes of Partition A
- P (1) stores all the nodes of Partition B

h) Colour Map (Colour)

- Colour(x) ← Colour representing the partition set of node "x"

i) Bad Component Labels Set (Bad)

- Bad stores the Label of Connected Component if that component is not following some rule of bipartiteness

j) Storing visited set of the nodes for DFS (V)

- If node is in V, that means it is visited

E. Initialization

```
Adj ← {}        // Initially all maps are empty
Label ← {}
Size ← {}
Colour ← {}
P ← {0: set (), 1: set ()} // 0 for partition-A, 1 for partition-B
Bad ← set ()
V ← set ()
```

F. Modifying DFS_BP used in Previous algorithm

- DFS_BP will return TRUE if the given connected component violates the bipartition, it will store the Label of that component into "Bad" Set
- If swapping is applied on component, take the current node from one set to another
- DFS will return TRUE immediately if any of the following conditions apply

d) Self-Loop is found
e) If any subgraph of adjacent node is violating the rule
f) If adjacent node is in the same set

```
DFS_BP (x, clr)

    add x to V
    Colour(x) ← clr

    If x is not in P(clr)
      add x to P(clr)
      remove x from P(clr ^ 1)

    For each node in Adj(x)
       If node is equal to x
          Add Label(x) to Bad set
          Return TRUE
       If node is not in V
          If DFS_BP (child, reverse(clr)) is TRUE
              Add Label(x) to Bad set
              Return TRUE
       Else if node is in P(clr)
           Add Label(x) to Bad set
           Return TRUE

    Return FALSE
```

G. DFS_LBL

- We use this DFS to Relabel the nodes when connected components are merged or separated

```
DFS_LBL (x, lbl)
    Label(x) ← lbl
    Size(p) ← Size(p) + 1
    Add x to V

    For each node in Adj(x)
      If node is not in V
        DFS_LBL(node, lbl)
```

H. Add an edge

- If both nodes are in same component and in the same set then bipartition is violated, we add the label of this component to the "Bad" set and return
- Now the nodes were from different components,
- Let's swap the u and v such that Size(u) is less then Size(v)
- Now relabel the component u using Label(v) with DFS_LBL
- There are two cases possible regarding the bipartiteness
  1. Both the components that are getting merged are bipartite
     a) If u and v belonged to same set, we can swap the colours of component "u" and then add the edge.
     b) Otherwise we can simply add the edge, component will still be bipartite
  2. At least one of the components which is getting merged is not bipartite
     c) In this case the merged component will also be non-bipartite

```
Add EDGE (u, v)

  Raise error if u or v are not in the graph

  If Label(u) is equal to Label(v)
    If Colour(u) is equal to Colour(v)
      Add Label(u) to Bad
      Add the edge
    Return

  If Size(u) is greater than Size(v)
    Swap (u, v)

  If Any of the Component is not in Bad
```

```
    If u and v are in same set
        DFS_BP (u, Colour(u) ^ 1)

Remove Label(u) from Size
Remove u from Label

DFS_LBL (u, Label(v))
```

I. Add a Node

- We will add the new node to partition A always.
- Assign empty set to the Adj(x)
- Label the node with itself initially

```
Add NODE (x)

    Raise error if u is already in the graph

    Adj(x) ← {}
    Label(x) ← x
    Size(x) ← 1
    Colour(x) ← 0
    Add x to P(0)
```

J. Remove an edge

- Remove the edge from Adj(u)(v)
- If there are multiple edges then removing the edge will not change the bipartiteness of the component, return
- If component is bipartite then removing the edge will not affect the bipartiteness, we simply relabel the components in case of separation
- If component was non-bipartite then
    - a) Remove Label of component from Bad set
    - b) Run DFS_BP to update the bipartiteness and Bad set accordingly
    - c) Relabel with DFS_LBL if two components are separated

```
Remove EDGE (u, v)

    Raise error if u or v is not in the graph
    Raise error if there is no edge between u and v
```

```
Adj(u)(v) ← Adj(u)(v) – 1

If Adj(u)(v) is positive
    Return

Check ← If Label(u) is in Bad

If Check is TRUE then
    Remove Label(u) from Bad

Remove Label(u) from Size

V ← {}
DFS_LBL (u, u)


If Check is TRUE then
    DFS_BP (u, Colour(u))

If v is not in V
    DFS_LBL (v, v)
    If Check is TRUE then
        DFS_BP (v, Colour(v))
```

K. Remove a Node

- Remove node from Graph, Labelling and Bipartiteness Data Structures
- Remove the node from all the adjacent node lists
- If the component was already bipartite then removing the node will not affect the state, we simply relabel the components which got separated due to removal of node
- If component was non-bipartite then
    a) Remove Label of component from Bad set
    b) Run DFS_BP to update the bipartiteness and Bad set accordingly
    c) Relabel with DFS_LBL
    d) Do this for all the adjacent nodes

```
Remove NODE (x)

    Raise error if u or v is not in the graph
    Raise error if there is no edge between u and v
```

```
adj ← Adj(x)
Check ← If Label(u) is in Bad

If Check is TRUE then
    Remove Label(x) from Bad

Remove x from Graph
Remove Label(x) from Size
Remove x from Label
Remove x from P(Colour(x))
Remove x from Colour

For each node in adj
    Remove x from Adj(node)



V ← set()
For each node in adj
  If node not in V
    DFS_LBL(node, node)

If Check is TRUE
  V ← set()
  For each node in adj
      If node not in V
          DFS_BP (node, Colour(node))
```

L. Is_Bipartite

- If at any point of time Bad Components set is empty then and only then the Graph is bipartite.

M. Print Partitions

- Check using **is_bipartite()** , if returns False then print "Not a Bipartite Graph!!!"
- Partitions are stores in sets P(0) and P(1)
- We iterate over each element and print the sets

4. Proof of Correctness

- We already proved in the previous Algorithm that DFS_BP will always maintain the set P0 and P1, such that bipartiteness is followed
- If the "Bad" set has at least one element then the graph will not be bipartite

- Relabelling is pretty straight forward
    a) If two components are getting merged, we relabel the smaller component, with label of bigger component
    b) If components are getting separated, we relabel each component using one of nodes in those components
- Now lets prove that after every update this state remain as it is

A. Add Node

- Adding the node does not affect the bipartiteness, hence we can add it to any of the sets, we are adding it to the P0
- Also, by doing this we are making sure that any point of time, node will always be in one of the sets (P0 or P1): Proof of Coverage

B. Add Edge

- We can have two cases here
    a) Adding the edge inside the connected component
        - If the component is non-bipartite adding the edge will not change the state
        - Otherwise, there are two possibilities
            1) Adding the edge between different sets, state remains the same
            2) Otherwise, we add Label of component to the Bad set
            3) Size of Bad set becomes positive
            4) Hence violation of bipartiteness is stored in the structures
    b) Add edge between two connected components
        - If any of the component is non-bipartite, the resulting component will be non-bipartite, no change in the state
        - Otherwise, there are two possibilities
            1) Adding the edge between different sets, state remains the same

2) Otherwise, we swap the Partitions for one of the components and merge both components, here again bipartition is not violated, since edge will be now added between different sets, due to swapping

3) Hence bipartition state is still intact

C. Remove Node

- If graph is bipartite removal of node will not affect the state
- Else, we will first remove label of removed node from Bad set
- Now, we apply DFS_BP on every component that got separated due to removal, if any of the component is not bipartite, DFS_BP will take care of it and add to the Bad set
- Hence size of Bad set will be positive if any of the separated component is not bipartite, If its empty is_bipartite() will return true

D. Remove Edge

- If graph is bipartite removal of edge will not affect the state
- Else, we will first remove label of component of the edge from Bad set
- Now, we apply DFS_BP on both nodes connecting the edge, if any of the component is not bipartite, DFS_BP will take care of it and add to the Bad set
- We will not revisit the second node if it was already visited in DFS_BP
- Hence again if size of Bad set will be positive if any of the separated component is not bipartite
- Otherwise the Bad will be empty and so we can say that the graph is now bipartite

5. Time Complexity (worst-case)

N : Number of nodes currently present in the graph

M : Number of edges currently present in the graph

n : Number of nodes in given connected component

m : Number of edges in given connected component

1. Add Node: O (1)
2. Add Edge, Remove Edge, Remove Node:
   - DFS_BP: O (n + m)
   - DFS_LBL: O (n + m)
   - Search and Deletion is in log(n) since we used maps

- T = O (2*(n*log(n) + m)) = O (n*log(n) + m)
- Worst-case: Graph is fully connected (n=N, m=M)
- T = O (N*log(N) + M)

3. Over All: T = O (N*log(N) + M)
4. Note. In general, what happens is n << N and m << M, in those cases out algorithm does not need to traverse the whole graph, and hence becomes very efficient

Question 3

1. Distributed Algorithm

A. Machine Model

- Each node represents a processor
- Each edge represents a communication channel between processors

B. Assumptions

- We will follow the similar assumptions taken in the session
- Graph is a connected graph, i.e. all pair of nodes have at least one communication channel path between them
- Each node has some id, also an adjacency list (adj) which contains ids of its neighbour nodes
- Each node has id of the root node, root node(v0) knows that it is such node

C. Algorithm

- Initially root node will mark itself as one colour (here "0") and pass the opposite values (here "1") as message to its adjacent nodes
- Other nodes will wait until the first message is arrived
- For the sake of simplicity, we will use "flag" attribute with message, to know the type of the message.
    1. Flag = 1 => Marking / Initialization Message
    2. Flag = 2 => Violation / Termination Message
    3. Flag = 3 => Is Bipartite / Root query Message
    4. Flag = 4 => Bipartiteness / Root reply Message
- Each node will have to attributes My_mark and My_visit
- Each node will be only visited maximum twice, since after second visit that node will not circulate the message again

- If at any point of time we encounter the violation of bipartiteness we return this information to root and all the neighbours also.
- After the Round 1 every node will be visited at least once, and either marked or unmarked based on bipartiteness
- If the root has got any message of flag = 2 then it will get detected in Round 2
- Now at any point of time query can be done from any node about bipartiteness of the graph to the root_id

```
Round 1:

If Vi is equal to Vo
   My_mark ← 0
   My_Visit ← 1
   For each receiver_id in adj
      Send (flag ← 1, mark ← 1, id ← receiver_id)
Else
   (flag, marker, sender_id) ← Wait (Receive ())

   If flag == 1
      If My_visit == 0
         My_visit ← 1
         My_mark ← marker
         For each adj_id in adj
            Send (flag ← 1, mark ← not(marker), id = adj_id)
      Else If marker is not equal to My_mark
         Send (flag ← 2, id ← root_id)
         For each adj_id in adj
            Send (flag ← 2, id ← adj_id)
   Else If flag == 2
      If My_visit == 0
         My_visit ← 1
         For each adj_id in adj
            Send (flag ← 2, id ← adj_id)

Round 2:

   If Vi == Vo
      Bipartite ← True
```

```
    While ((flag, sender_id) ← Receive())
        If flag == 2
            Bipartite ← False
  Else
    Send (flag ← 3, id ← root_id)

/// User query about bipartiteness from any of the nodes
Round 3:
  If Vi == 0
    While ((flag, sender_id) ← Receive ())
        If flag == 3
         Send (flag ← 4, bipartiteness ← Bipartite, id ← sender_id)
  Else
     (flag, bipartiteness) ← Wait(Receive ())
     If flag == 4
          Bipartite ← bipartiteness
```

2. Proof of Correctness

E. Proof of Coverage

- We keep track of visiting nodes by using My_Visit object
- We only propagate the received message further if we were already not visited, hence each node will always be visited **at most twice**
- If we encounter the violation, we share this information with root node as well as other components hence storing the information about bipartiteness

B. Proof of Correctness of Bipartiteness

- If there is a case that two adjacent nodes u, v are marked same then,
    - The first node "u" will send the information about opposite of its colour (here "0") to all the adjacent nodes
    - The second "v" will also get information to change its colour (here let's say "0") from all its adjacent nodes
    - Now after second Node is marked, it send the information of opposite colour(here "1") to "u"

- "u" will receive the message, since it is already marked it will check that the colour it received from the "v" is same as its own my_mark or not
- If this condition is not met than "u" will send the TERMINATION message to all of the nodes including ROOT node.
- Root node will be waiting for such message and when received it will change the bipartiteness to FALSE
- Hence ROOT will have stored that graph is non-bipartite
- Hence No two adjacent nodes could be marked without root getting the information about it and storing the correct state
- Hence Bipartiteness is always correct in the root after all the nodes are visited

3. Time Complexity (worst-case)

V : Number of nodes currently present in the graph

E : Number of edges currently present in the graph

* Every Node only Visited = 2* Degree of that node

- $Tt = \sum 2*d(i) = 4*E$

4. Message Complexity (worst-case)

* Every Edge is visited at most twice since after that the node will be marked visited

- Tm = 2*E

5. Overall Complexity (worst-case)

- T = 2*E* (2*Cost of Visiting One Node + Cost of Message passing in one edge)