Lec-54

16/10/25 10:18 AM

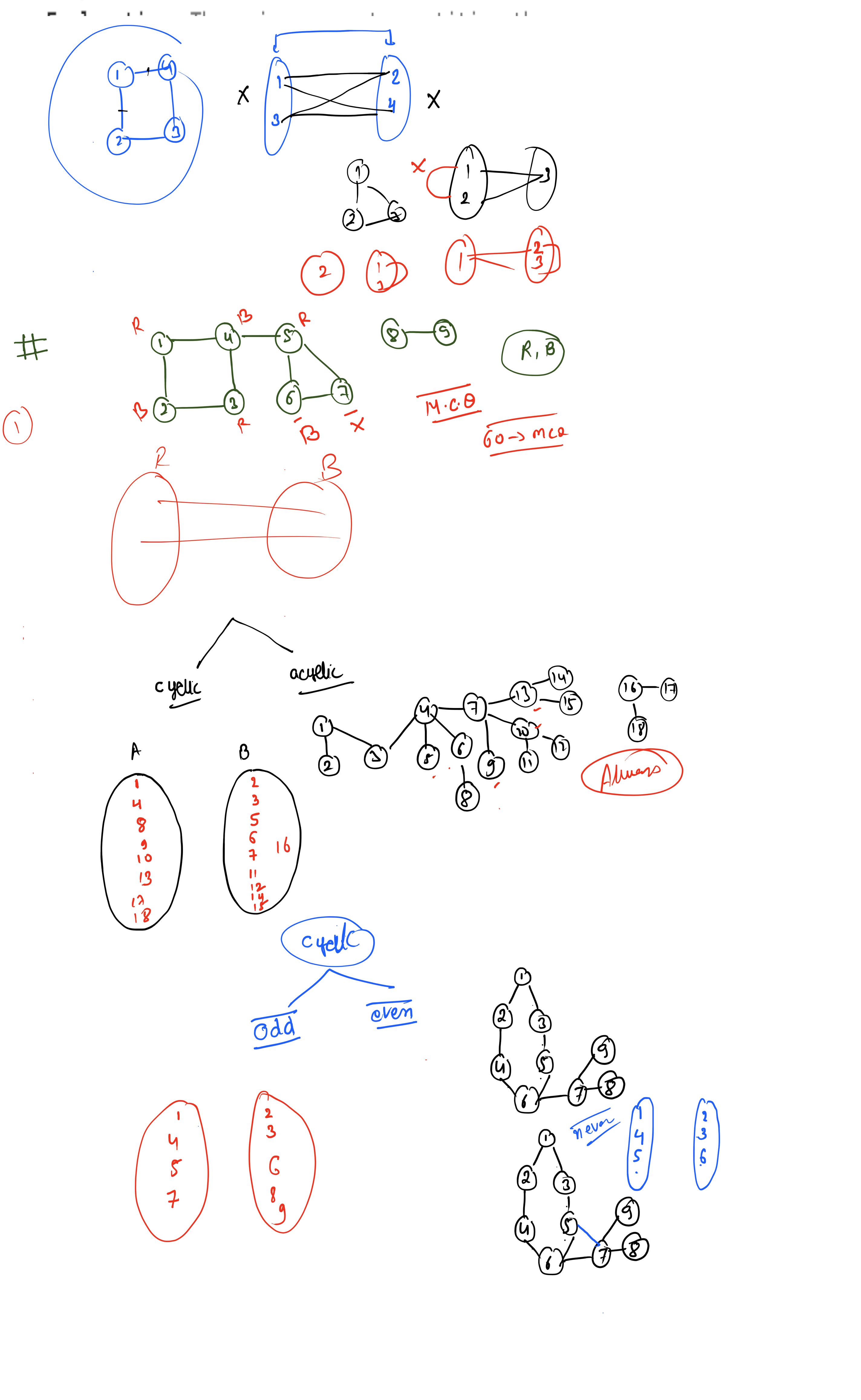
There is an **undirected** graph with `n` nodes, where each node is numbered between `0` and `n - 1`. You are given a 2D array `graph`, where `graph[u]` is an array of nodes that node `u` is adjacent to. More formally, for each `v` in `graph[u]`, there is an undirected edge between node `u` and node `v`. The graph has the following properties:

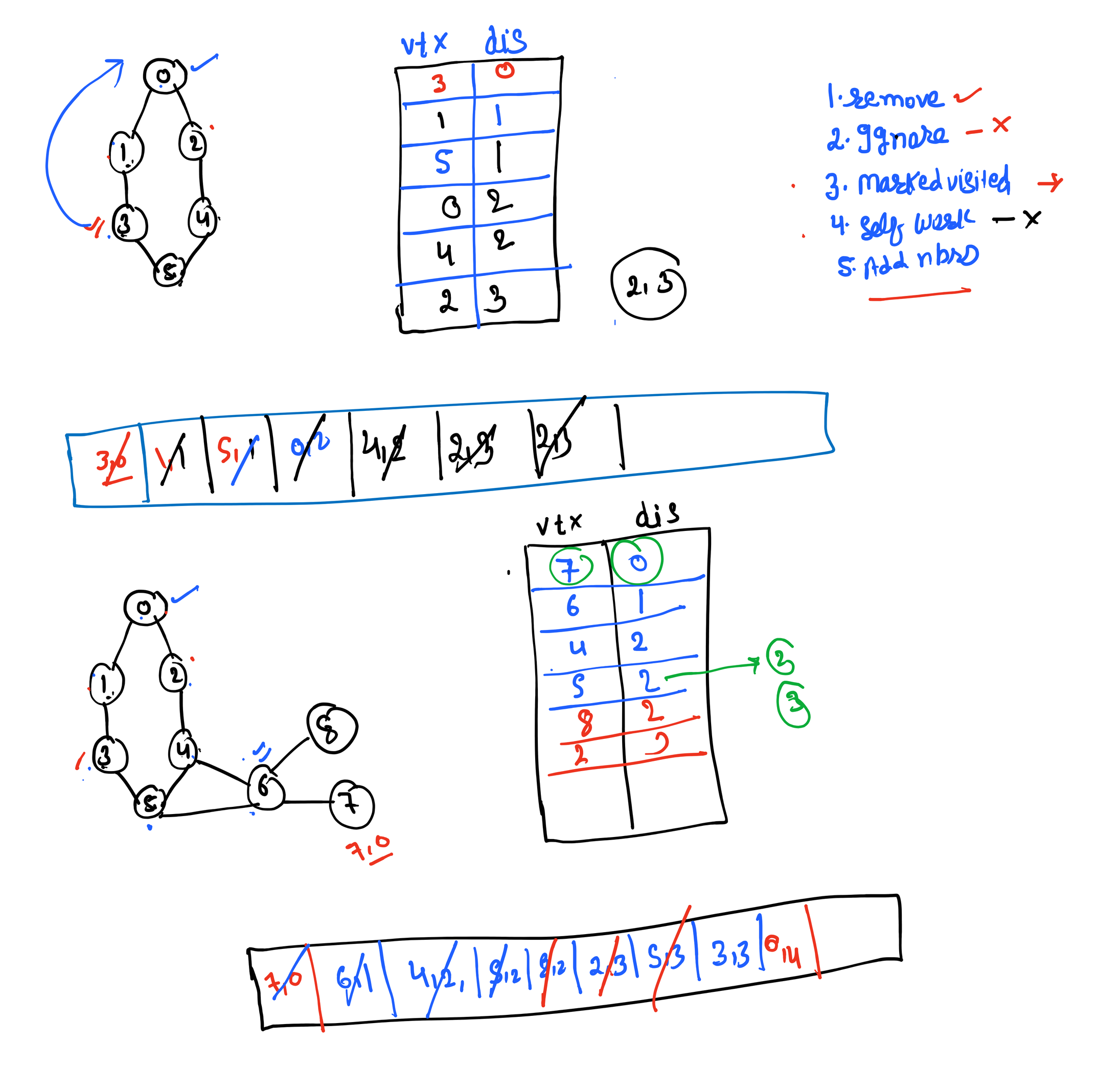
- There are no self-edges (`graph[u]` does not contain `u`).
- There are no parallel edges (`graph[u]` does not contain duplicate values).
- If `v` is in `graph[u]`, then `u` is in `graph[v]` (the graph is undirected).
- The graph may not be connected, meaning there may be two nodes `u` and `v` such that there is no path between them.

A graph is **bipartite** if the nodes can be partitioned into two independent sets `A` and `B` such that **every** edge in the graph connects a node in set `A` and a node in set `B`.

Return `true` *if and only if it is* ***bipartite***.

Input: graph = [[1,2,3],[0,2],[0,1,3],[0,2]]
Output: false



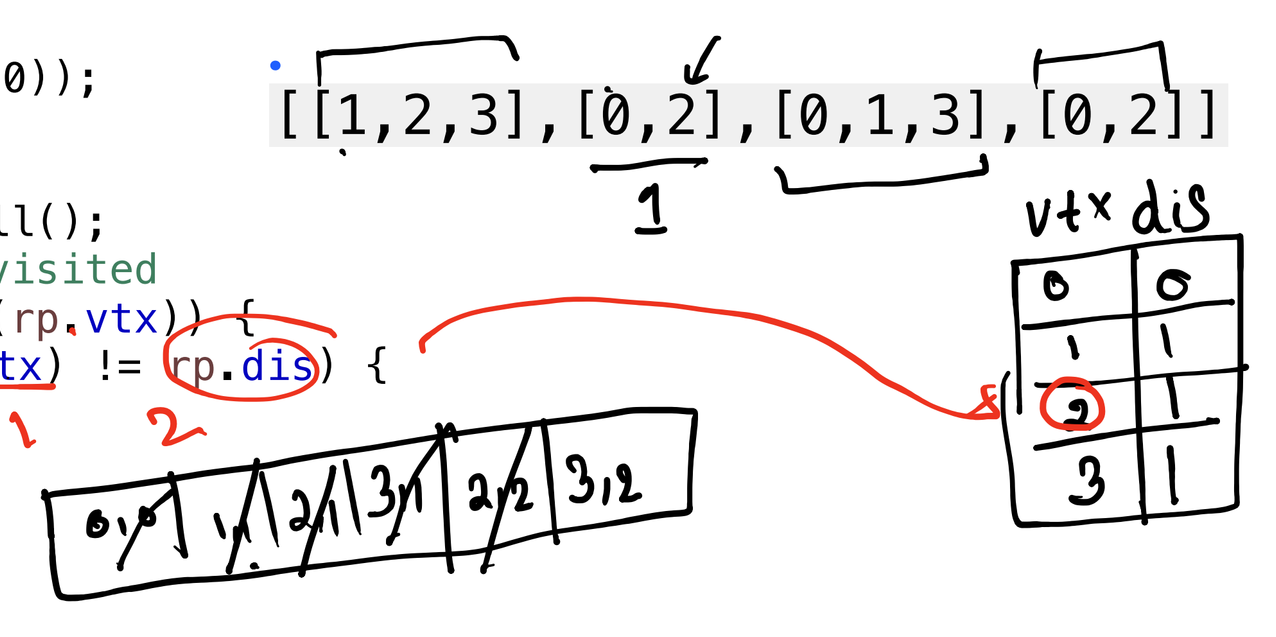


```
public boolean isBipartite(int[][] graph) {
        Queue<BipartitePair> q = new LinkedList<>();
        HashMap<Integer, Integer> visited = new HashMap<>();
        for (int i = 0; i < graph.length; i++) {// vtx
            if(visited.containsKey(i)) {
                continue;
            }
                q.add(new BipartitePair(i,0));
        }
    }
```

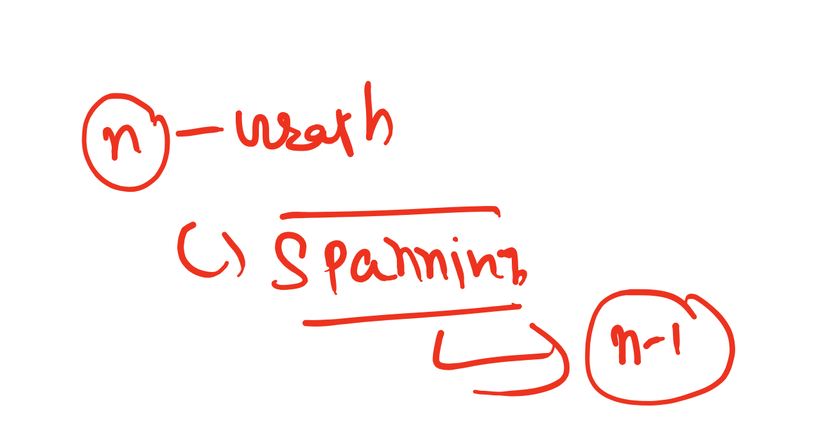
Input: graph = [[1,2,3],[0,2],[0,1,3],[0,2]]
Output: false

```
public boolean isBipartite(int[][] graph) {
        Queue<BipartitePair> q = new LinkedList<>();
        HashMap<Integer, Integer> visited = new HashMap<>();
        for (int i = 0; i < graph.length; i++) {// vtx
            if (visited.containsKey(i)) {
                continue;
            }
            q.add(new BipartitePair(i, 0));
            while (!q.isEmpty()) {
                // 1. remove
                BipartitePair rp = q.poll();
                // 2. Ignore if Alredv visited
                if (visited.containsKey(rp.vtx)) {
                    if (visited.get(rp.vtx) != rp.dis) {
                        return false;
                    }
                    continue;
                }
                // 3. makred visited
                visited.put(rp.vtx, rp.dis);
                // 4. self work
                // 5. Add unvisited nbrs
                for (int nbrs : graph[rp.vtx]) {
                    if (!visited.containsKey(nbrs)) {
                        q.add(new BipartitePair(nbrs, rp.dis + 1));
                    }
                }
            }
        }
        return true;

    }
```

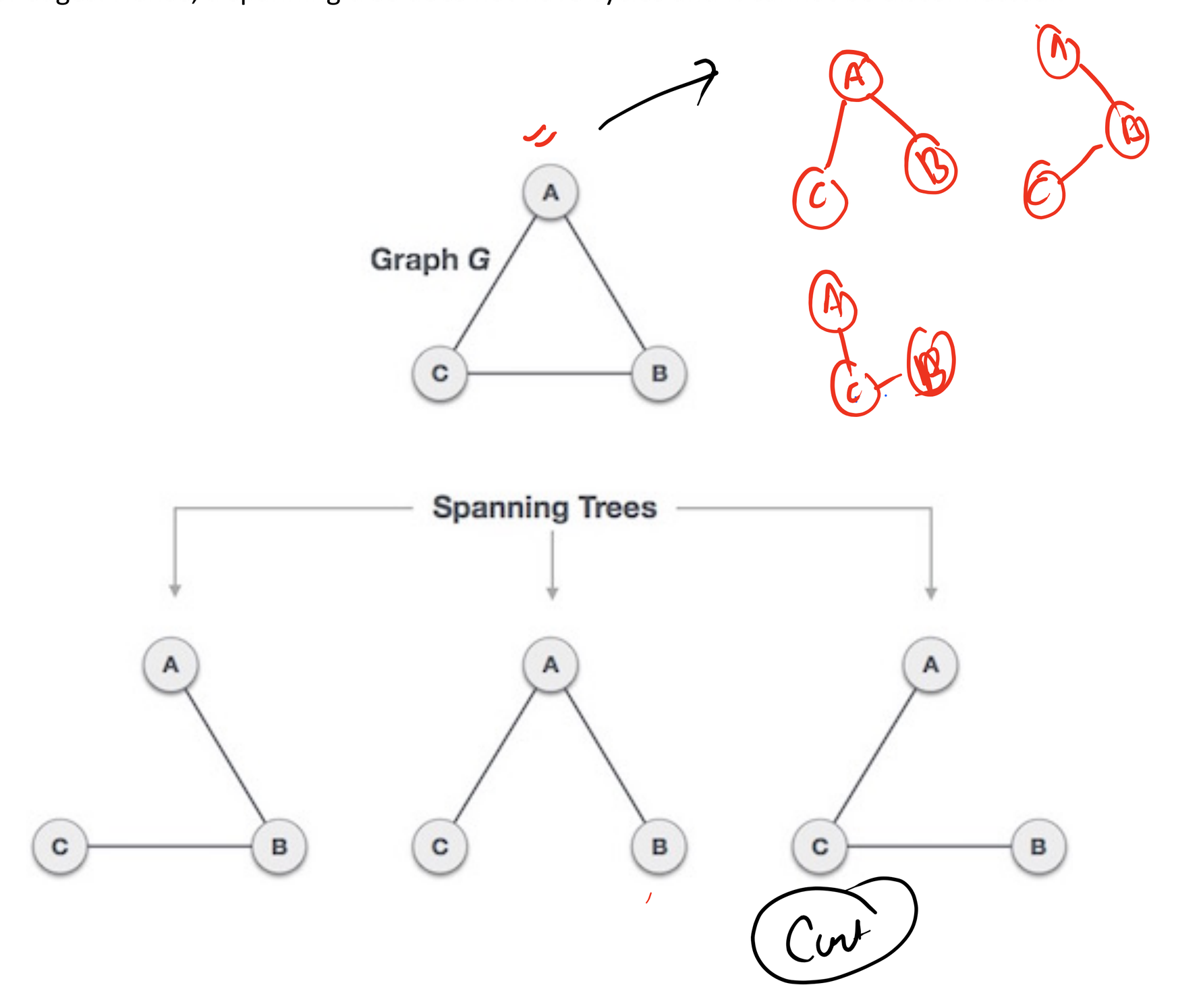


Spanning Tree



Spanning Tree

A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected..



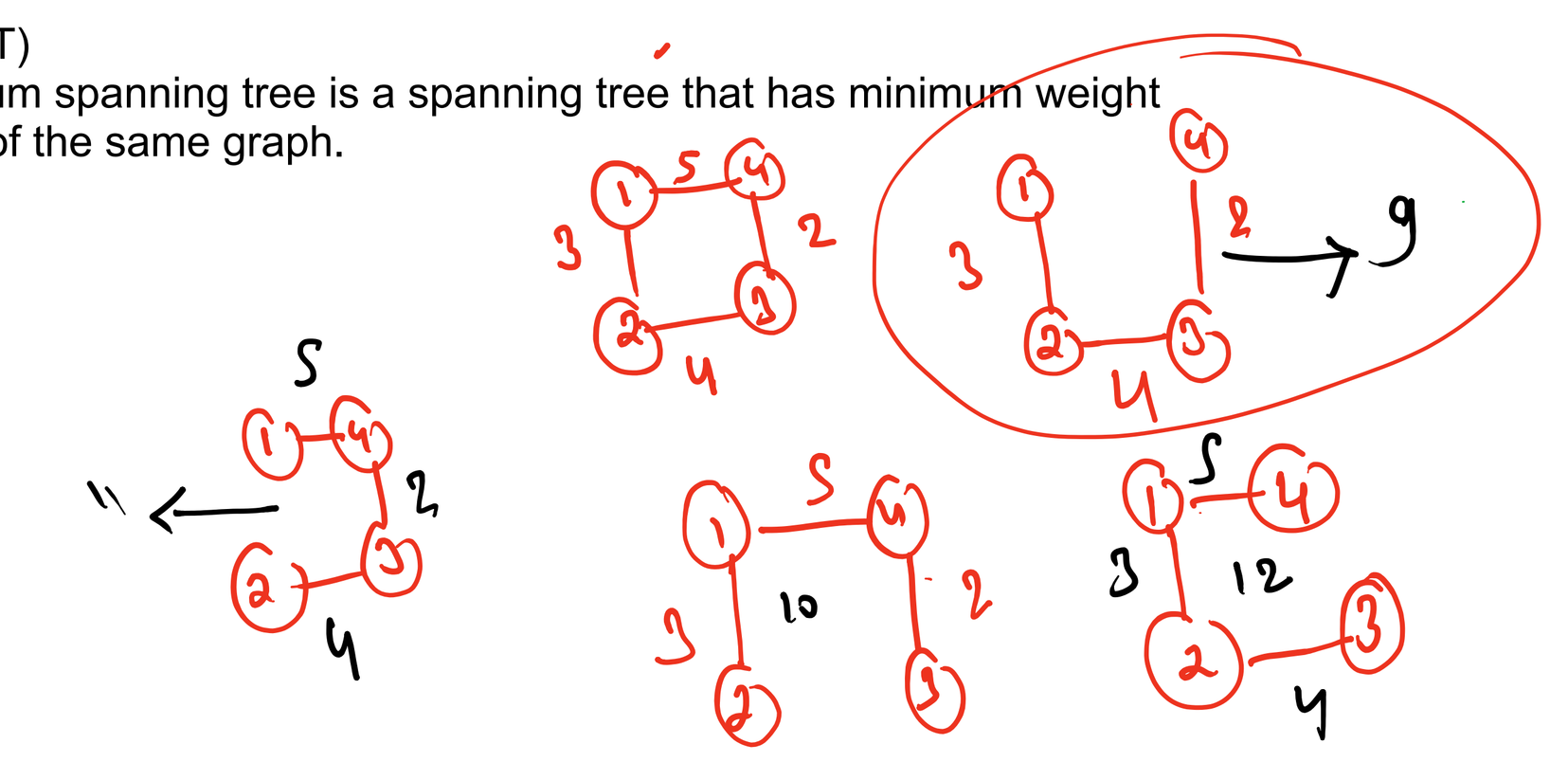
Properties of Spanning Tree

- A connected graph G can have more than one spanning tree.
- Spanning Tree has n-1 edges, where n is the number of Node (Vertices)
- All possible spanning trees of graph G, have the same number of edges and vertices.
- The spanning tree does not have any cycle (loops).
- Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is minimally connected.
- Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is maximally acyclic.

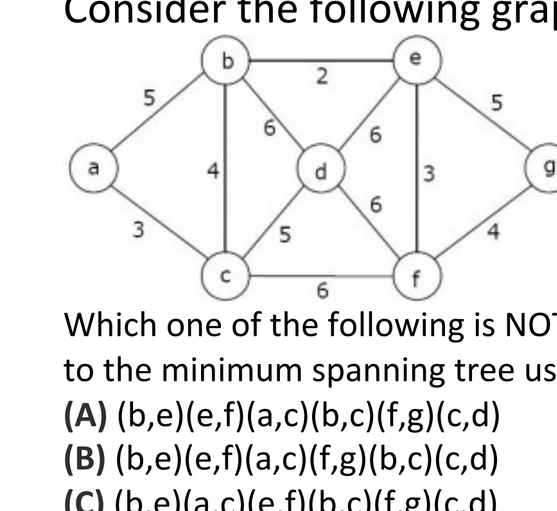
Minimum Spanning Tree (MST)
In a weighted graph, a minimum spanning tree is a spanning tree that has minimum weight than all other spanning trees of the same graph.

- Kruskal's Algorithm

Prim's Algorithm
Both are greedy algorithms.



Consider the following graph:



Which one of the following is NOT the sequence of edges added to the minimum spanning tree using Kruskal's algorithm?
(A) (b,e)(e,f)(a,c)(b,c)(f,g)(c,d)
(B) (b,e)(e,f)(a,c)(f,g)(b,c)(c,d)
(C) (b,e)(a,c)(e,f)(b,c)(f,g)(c,d)
(D) (b,e)(e,f)(b,c)(a,c)(f,g)(c,d)