Lec 37

[illegible]

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 connected).
- 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**.

```
class Solution {
    public boolean isBipartite(int[][] graph) {
        Queue<IsBipartite> q = new LinkedList<>();
        HashMap<Integer, Integer> visited = new HashMap<>;
        for (int vtx = 0; vtx < graph.length; vtx++) {
            if (visited.containsKey(vtx)) {
                continue;
            }
            q.add(new IsBipartite(vtx, 0));
            // TODO
            while (q.isEmpty()) {
                // remove
                IsBipartite bp = q.poll();
                int vtx = bp.vtx, dis = bp.dis;
                if(visited.containsKey(vtx,vtx)) {
                    if(visited.get(bp.vtx)!=bp.dis) {
                        return false;
                    }
                    continue;
                }
                // 3. marked visited
                visited.put(bp.vtx, bp.dis);
                // 4. self work

                // 5. add unvisited nbrs
                for(int nbrs:graph[vtx]) {
                    if(!visited.containsKey(nbrs)) {
                        q.add(new IsBipartite(nbrs, bp.dis+1));
                    }
                }
            }
        }
        return true;
    }
}
class IsBipartite {
    int vtx;
    int dis;

    public IsBipartite(int vtx, int dis) {
        // TODO Auto-generated constructor stub
        this.vtx = vtx;
        this.dis = dis;
    }
}
```

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)

```
public void Prims_Algo() {
    PriorityQueue<PrimsPair> pq = new PriorityQueue<>(new Comparator<>() {

        @Override
        public int compare(PrimsPair o1, PrimsPair o2) {
            // TODO Auto-generated method stub
            return o1.cost-o2.cost;
        }

    });
    HashSet<Integer> visited = new HashSet<>();
    pq.add(new PrimsPair(1, 1, 0));
    while (!pq.isEmpty()) {
        // 1. remove
        PrimsPair pr = pq.poll();
        // 2. Ignore if already visited
        if (visited.contains(pr.vtx)) {
            continue;
        }
        // 3. marked visited
        visited.add(pr.vtx);
        System.out.println(pr);// sum = sum + pr.cost;
        // 5. Add nbrs
        for(int nbrs:map.get(pr.vtx).keySet()) {
            if(!visited.contains(nbrs)) {
                int cost=map.get(pr.vtx).get(nbrs);
                pq.add(new PrimsPair(nbrs, pr.vtx, cost));
            }
        }
    }
}
```