You have a graph of n nodes labeled from 0 to $n-1$. You are given an integer n and a list of edges where $edges[i] = [a_i, b_i]$ indicates that there is an undirected edge between nodes a_i and b_i in the graph.

Return `true` *if the edges of the given graph make up a valid tree, and* `false` *otherwise.*

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**.

```
Public boolean isBipartite(int[][] graph){
    queue=BipartitePair pp new LinkedList<>();
    HashMap<Integer, Integer> visited= new HashMap<>();
    for(int i=0;i<graph.length; i++){
        if(visited.get(i)==null){
            BipartitePair rp=pp.poll();
            //add even cycle ignore steps
            if(visited.containsKey(rp.vtx)){
                if(visited.get(rp.vtx)!=rp.dis){
                    return false;
                } else{
                    continue;
                }
            }
            visited.put(rp.vtx, r.dis);
            for(int[]nbrs: graph[r.vtx]) {
                if(!visited.containsKey(nbrs[0]) {
                    q.add(new BipartitePair(nbrs, r.dis+ 1));
                }
            }
        }
    }
```

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),(f,g),(c,d)
(B) (b,e),(e,f),(a,c),(f,g),(b,c),(c,d)
(C) (b,e),(e,f),(a,c),(f,g),(c,d)
(D) (b,e),(e,f),(a,c),(f,g),(c,d)


```
Public int Prim(HashMap<Integer, HashMap<Integer, Integer>> map) {
    PriorityQueue<PrimInfo> pq= new PriorityQueue<>();
    HashSet<Integer> visited= new HashSet<>();
    int sum= 0;
    pq.add(new PrimInfo(1, 1, 0));
    while(pq.isEmpty()) {
        PrimInfo rp= pq.poll();
        if(visited.contains(rp.vtx)) {
            continue;
        }
        visited.add(rp.vtx);
        System.out.println(rp);
        sum= rp.cost;
        for(int[]nbr: map.get(rp.vtx).keySet()) {
            if(!visited.contains(nbr[0])) {
                pq.add(new PrimInfo(nbr[0], rp.vtx, nbr[1]));
                pq.add(new PrimInfo(nbr[0], rp.vtx, cost));
            }
        }
    }
    return sum;
}
```



```
Public void Dijkstra() {
    PriorityQueue<DijkstraPair> pq= new PriorityQueue<>(new Comparator<DijkstraPair>() {
        @Override
        public int compare(DijkstraPair o1, DijkstraPair o2) {
            throw new UnsupportedOperationException();
            return o1.cost- o2.cost;
        }
    });
    HashSet<Integer> visited= new HashSet<>();
    pq.add(new DijkstraPair(1, "1", 0));
    while(pq.isEmpty()) {
        DijkstraPair rp= pq.poll();
        if(visited.contains(rp.vtx)) {
            continue;
        }
        visited.add(rp.vtx);
        System.out.println(rp);
        for(int[]nbr: map.get(rp.vtx).keySet()) {
            if(!visited.contains(nbr[0])) {
                int cost= map.get(rp.vtx).get(nbr[0]);
                pq.add(new DijkstraPair(nbr[0], rp.vtx+ "->"+ nbr[0], rp.cost+ cost));
            }
        }
    }
}
```