Construction Cost

Problem Description

Given a graph with **A** nodes and **C** weighted edges. Cost of constructing the graph is the sum of weights of all the edges in the graph.

Find the **minimum cost** of constructing the graph by selecting some given edges such that we can reach every other node from the **1st** node.

NOTE: Return the answer modulo 10^9+7 as the answer can be large.

```java
public class Solution {
    private static final int MOD = 1000000007;
    public int solve(int A, ArrayList<ArrayList<Integer>> B) {
        // Convert the edge list B into an adjacency list
        List<List<int[]>> graph = new ArrayList<>();
        for (int i = 0; i < A; i++) {
            graph.add(new ArrayList<>());
        }
        for (ArrayList<Integer> edge : B) {
            int u = edge.get(0) - 1;
            int v = edge.get(1) - 1;
            int weight = edge.get(2);
            graph.get(u).add(new int[]{v, weight});
            graph.get(v).add(new int[]{u, weight});
        }

        // Prim's algorithm
        PriorityQueue<int[]> pq = new PriorityQueue<>(Comparator.comparingInt(o -> o[1]));
        boolean[] inMST = new boolean[A];
        int[] minEdge = new int[A];
        Arrays.fill(minEdge, Integer.MAX_VALUE);
        minEdge[0] = 0;
        pq.add(new int[]{0, 0});  // {node, weight}

        long minCost = 0;

        while (!pq.isEmpty()) {
            int[] current = pq.poll();
            int u = current[0];
            int weight = current[1];

            if (inMST[u]) continue;

            inMST[u] = true;
            minCost = (minCost + weight) % MOD;

            for (int[] neighbor : graph.get(u)) {
                int v = neighbor[0];
                int w = neighbor[1];

                if (!inMST[v] && w < minEdge[v]) {
                    minEdge[v] = w;
                    pq.add(new int[]{v, w});
                }
            }
        }
        return (int) minCost;
    }
}
```

Commutable Islands

Problem Description

There are **A** islands and there are **M** bridges connecting them. Each bridge has some **cost** attached to it.

We need to find bridges with **minimal cost** such that all islands are connected.

It is guaranteed that input data will contain **at least one** possible scenario in which all islands are connected with each other.

```
public class Solution {
    public int solve(int A, int[][] B) {
        // Step 1: Build the adjacency list
        List<List<int[]>> adj = new ArrayList<>();
        for (int i = 0; i < A; i++) {
            adj.add(new ArrayList<>());
        }
        for (int[] bridge : B) {
            int from = bridge[0] - 1; // Convert 1-based to 0-based index
            int to = bridge[1] - 1;   // Convert 1-based to 0-based index
            int cost = bridge[2];
            adj.get(from).add(new int[]{to, cost});
            adj.get(to).add(new int[]{from, cost});
        }
        // Step 2: Use Prim's algorithm to find MST
        int minCost = primMST(adj, A);
        return minCost;
    }

    private int primMST(List<List<int[]>> adj, int A) {
        // Priority queue to store the minimum cost edges
        PriorityQueue<int[]> pq = new PriorityQueue<>(Comparator.comparingInt(edge -> edge[1]));
        // Visited array to mark visited islands
        boolean[] visited = new boolean[A];
        // Start from island 0 (arbitrary choice)
        pq.offer(new int[]{0, 0});
        int minCost = 0;
        int edgesUsed = 0;
        while (!pq.isEmpty()) {
            int[] current = pq.poll();
            int island = current[0];
            int cost = current[1];
            if (visited[island]) {
                continue;
            }
            // Add cost of the edge to the MST
            minCost += cost;
            visited[island] = true;
            edgesUsed++;
            // If we have added A-1 edges, we are done
            if (edgesUsed == A) {
                break;
            }
            // Add all adjacent edges of the current island to the priority queue
            for (int[] neighbor : adj.get(island)) {
                if (!visited[neighbor[0]]) {
                    pq.offer(neighbor);
                }
            }
        }
        return minCost;
    }
}
```

Dijkstra

Problem Description

Given a **weighted undirected graph** having **A nodes** and **M weighted edges**, and a **source node C**.

You have to **find an integer array D of size A** such that:

D[i]: Shortest distance from the **C node** to **node i**.
If **node i** is not reachable from C then **-1**.

Note:

There are no self-loops in the graph.
There are no multiple edges between two pairs of vertices.
The graph may or may not be connected.
Nodes are numbered from 0 to A-1.
Your solution will run on multiple test cases. If you are using global variables, make sure to clear them.

```java
public class Solution {
    public int[] solve(int A, int[][] B, int C) {
        List<List<Node>> adjList = new ArrayList<>();
        for (int i = 0; i < A; i++)
            adjList.add(new ArrayList<>());
        // Build the adjacency list
        for (int[] edge : B) {
            int u = edge[0];
            int v = edge[1];
            int weight = edge[2];
            adjList.get(u).add(new Node(v, weight));
            adjList.get(v).add(new Node(u, weight)); // Since it's an undirected graph
        }
        // Distance array to store shortest distances from source C
        int[] distances = new int[A];
        Arrays.fill(distances, Integer.MAX_VALUE);
        distances[C] = 0;
        // Min-heap priority queue for Dijkstra's algorithm
        PriorityQueue<Node> pq = new PriorityQueue<>(Comparator.comparingInt(node -> node.distance));
        pq.offer(new Node(C, 0));
        while (!pq.isEmpty()) {
            Node currNode = pq.poll();
            int u = currNode.vertex;
            int distU = currNode.distance;
            // If distU is already greater than the known shortest path, skip it
            if (distU > distances[u])
                continue;
            // Traverse all neighbors of u
            for (Node neighbor : adjList.get(u)) {
                int v = neighbor.vertex;
                int weightUV = neighbor.distance;

                // Calculate the new distance
                int newDist = distU + weightUV;

                // If a shorter path to v is found, update distances and push to pq
                if (newDist < distances[v]) {
                    distances[v] = newDist;
                    pq.offer(new Node(v, newDist));
                }
            }
        }
        // Replace unreachable nodes (still with Integer.MAX_VALUE) with -1
        for (int i = 0; i < A; i++) {
            if (distances[i] == Integer.MAX_VALUE && i != C) {
                distances[i] = -1;
            }
        }
        return distances;
    }
    // Node class to represent vertices and distances in the graph
    static class Node {
        int vertex;
        int distance;

        public Node(int vertex, int distance) {
            this.vertex = vertex;
            this.distance = distance;
        }
    }
}
```