# Answer of 1

A piece of paper with writing on it

Description automatically generatedA notebook with writing on it

Description automatically generatedA piece of paper with writing on it

Description automatically generated

# Answer of 2

# A notebook with writing on it Description automatically generated

# A piece of paper with writing on it Description automatically generatedA notebook with writing on it Description automatically generatedA notebook with writing on it Description automatically generated

# Answer of 3

import java.util.\*;

class Solution {

public int networkDelayTime(int[][] times, int n, int k) {

Map<Integer, Map<Integer, Integer>> outList = getGraphDetails(times);

PriorityQueue<int[]> pq = new PriorityQueue<>(Comparator.comparingInt(a -> a[0]));

Set<Integer> visited = new HashSet<>();

Map<Integer, Integer> distance = new HashMap<>();

pq.offer(new int[]{0, k});

distance.put(k, 0);

while (!pq.isEmpty()) {

int[] current = pq.poll();

int uWeight = current[0];

int u = current[1];

distance.put(u, Math.min(distance.getOrDefault(u, Integer.MAX\_VALUE), uWeight));

if (!visited.contains(u)) {

visited.add(u);

if (outList.containsKey(u)) {

for (Map.Entry<Integer, Integer> entry : outList.get(u).entrySet()) {

int v = entry.getKey();

int w = entry.getValue();

if (!visited.contains(v)) {

pq.offer(new int[]{distance.get(u) + w, v});

}

}

}

}

}

int result = distance.values().stream().max(Integer::compare).orElse(-1);

return result == -1 || visited.size() < n ? -1 : result;

}

private Map<Integer, Map<Integer, Integer>> getGraphDetails(int[][] times) {

Map<Integer, Map<Integer, Integer>> outList = new HashMap<>();

for (int[] time : times) {

int inVertex = time[0];

int outVertex = time[1];

int weight = time[2];

outList.putIfAbsent(inVertex, new HashMap<>());

outList.get(inVertex).put(outVertex, weight);

}

return outList;

}

}

# Answer of 4

class Solution {

public int[] findRedundantConnection(int[][] edges) {

int n = edges.length;

int[] parent = new int[n + 1];

int[] rank = new int[n + 1];

for (int i = 1; i <= n; i++) {

parent[i] = i;

rank[i] = 0;

}

for (int[] edge : edges) {

int u = edge[0];

int v = edge[1];

if (!union(u, v, parent, rank)) {

return edge;

}

}

return new int[0];

}

private int find(int x, int[] parent) {

if (parent[x] != x) {

parent[x] = find(parent[x], parent);

}

return parent[x];

}

private boolean union(int x, int y, int[] parent, int[] rank) {

int rootX = find(x, parent);

int rootY = find(y, parent);

if (rootX != rootY) {

if (rank[rootX] > rank[rootY]) {

parent[rootY] = rootX;

} else if (rank[rootX] < rank[rootY]) {

parent[rootX] = rootY;

} else {

parent[rootY] = rootX;

rank[rootX]++;

}

return true;

}

return false;

}

}

# Answer of 5

import java.util.\*;

class Solution {

public int minCostConnectPoints(int[][] points) {

PriorityQueue<int[]> edgePQ = new PriorityQueue<>(Comparator.comparingInt(a -> a[0]));

for (int i = 0; i < points.length; i++) {

for (int j = i + 1; j < points.length; j++) {

int distance = Math.abs(points[i][0] - points[j][0]) + Math.abs(points[i][1] - points[j][1]);

edgePQ.offer(new int[]{distance, i, j});

}

}

int[] parent = new int[points.length];

int[] rank = new int[points.length];

for (int i = 0; i < points.length; i++) {

parent[i] = i;

rank[i] = 0;

}

int find(int x) {

if (parent[x] != x) {

parent[x] = find(parent[x]);

}

return parent[x];

}

boolean union(int x, int y) {

int rootX = find(x);

int rootY = find(y);

if (rootX != rootY) {

if (rank[rootX] > rank[rootY]) {

parent[rootY] = rootX;

} else if (rank[rootX] < rank[rootY]) {

parent[rootX] = rootY;

} else {

parent[rootY] = rootX;

rank[rootX]++;

}

return true;

}

return false;

}

int totalCost = 0;

int edgesUsed = 0;

while (!edgePQ.isEmpty() && edgesUsed < points.length - 1) {

int[] edge = edgePQ.poll();

int cost = edge[0], u = edge[1], v = edge[2];

if (union(u, v)) {

totalCost += cost;

edgesUsed++;

}

}

return totalCost;

}

}