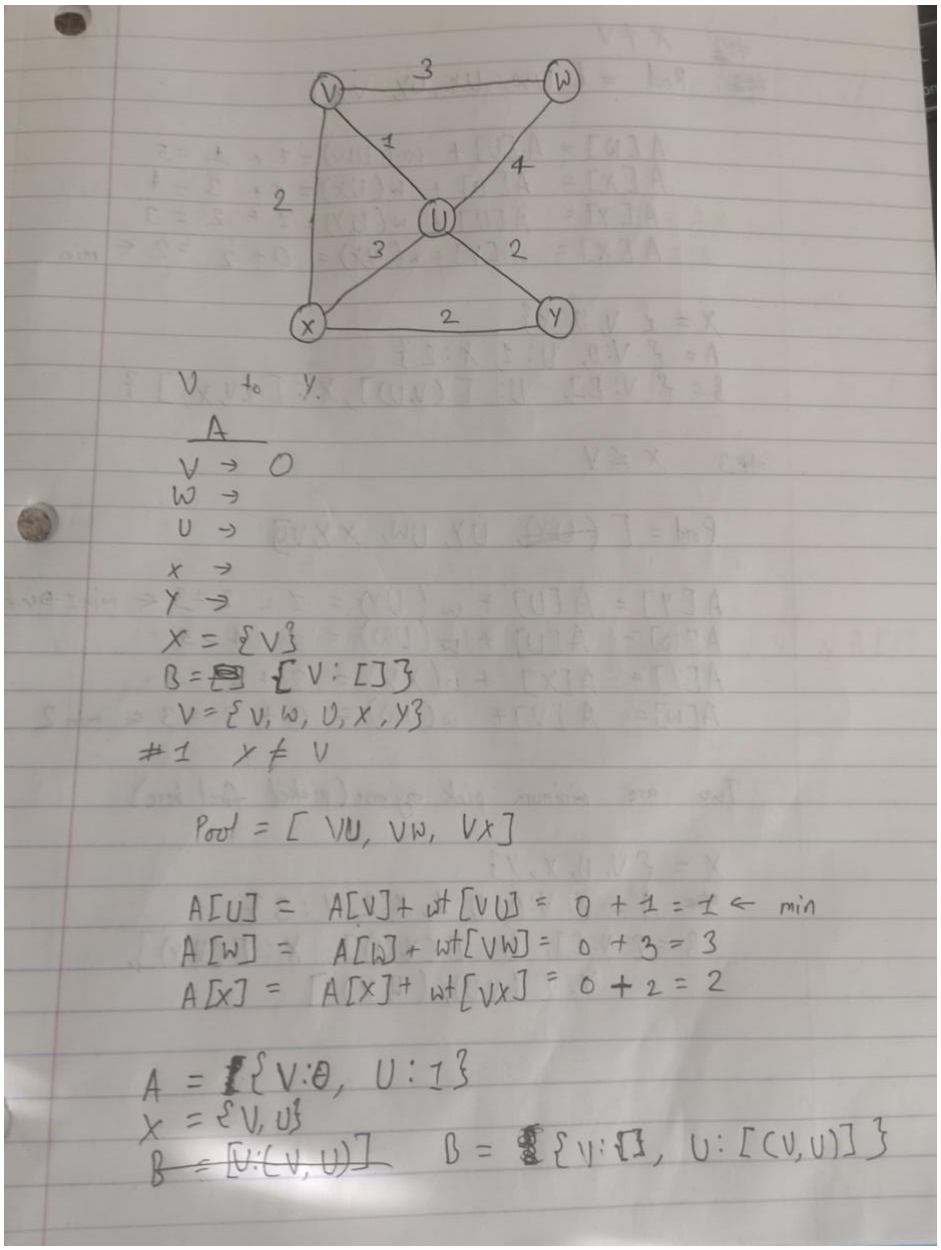
Answer of 1



V to Y.

A

V → 0
W →
U →
x →
Y →

X = {V}

B = { V : [] }

V = {V, W, U, x, Y}

#1 Y ≠ V

Pool = [VU, VW, Vx]

A[U] = A[V] + wt[VU] = 0 + 1 = 1 ← min
A[W] = A[V] + wt[VW] = 0 + 3 = 3
A[x] = A[x] + wt[Vx] = 0 + 2 = 2

A = { V:0, U:1 }
X = {V, U}
~~B = [V:(V, U)]~~ B = { V: [], U: [(V, U)] }

#2 $X \neq V$

~~#3~~: Pool = [UW, UX, UY, VX]

$A[W] = A[U] + w(UW) = 1 + 4 = 5$
$A[X] = A[U] + w(UX) = 1 + 3 = 4$
$A[Y] = A[U] + w(UY) = 1 + 2 = 3$
$A[X] = A[V] + w(VX) = 0 + 2 = 2 \leftarrow$ min

$X = \{V, U, X\}$
$A = \{V:0,\ U:1,\ X:2\}$
$B = \{V:[],\ U:[(V,U)],\ X:[(V,X)]\}$

#3 $X \neq V$

Pool = [~~(UV)~~ UY, UW, XY, VW]

$A[Y] = A[U] + w(UY) = 1 + 2 = 3 \leftarrow$ min1 picked
$A[W] = A[U] + w(UW) = 1 + 4 = 5$
$A[Y] = A[X] + w(XY) = 2 + 2 = 4$
$A[W] = A[V] + w(VW) = 0 + 3 = 3 \leftarrow$ min 2

Two are minimum pick any one (picked first here)

$X = \{V, U, X, Y\}$
~~A~~ $A = \{V:0,\ U:1,\ X:2,\ Y:3\}$
$B = \{V:[],\ U:[(V,U)],\ X:[(V,X)],$
$Y:[(V,U),(U,Y)]\}$

#4 $X \neq V$

Pool = [VW, UW]

$A[W] = A[V] + w(VW) = 0 + 3 = 3 \leftarrow$ min
$A[W] = A[U] + w(UW) = 1 + 4 = 5$

$X = \{ V, U, W, X, Y \}$
$A = \{ V:0,\ U:1,\ X:2,\ Y:3,\ W:3 \}$
$B = \{ V:[],\ U:[(V,U)],\ X:[(V,X)],$
$Y:[(V,U),(U,Y)],\ W:[(V,W)]$
$\}$

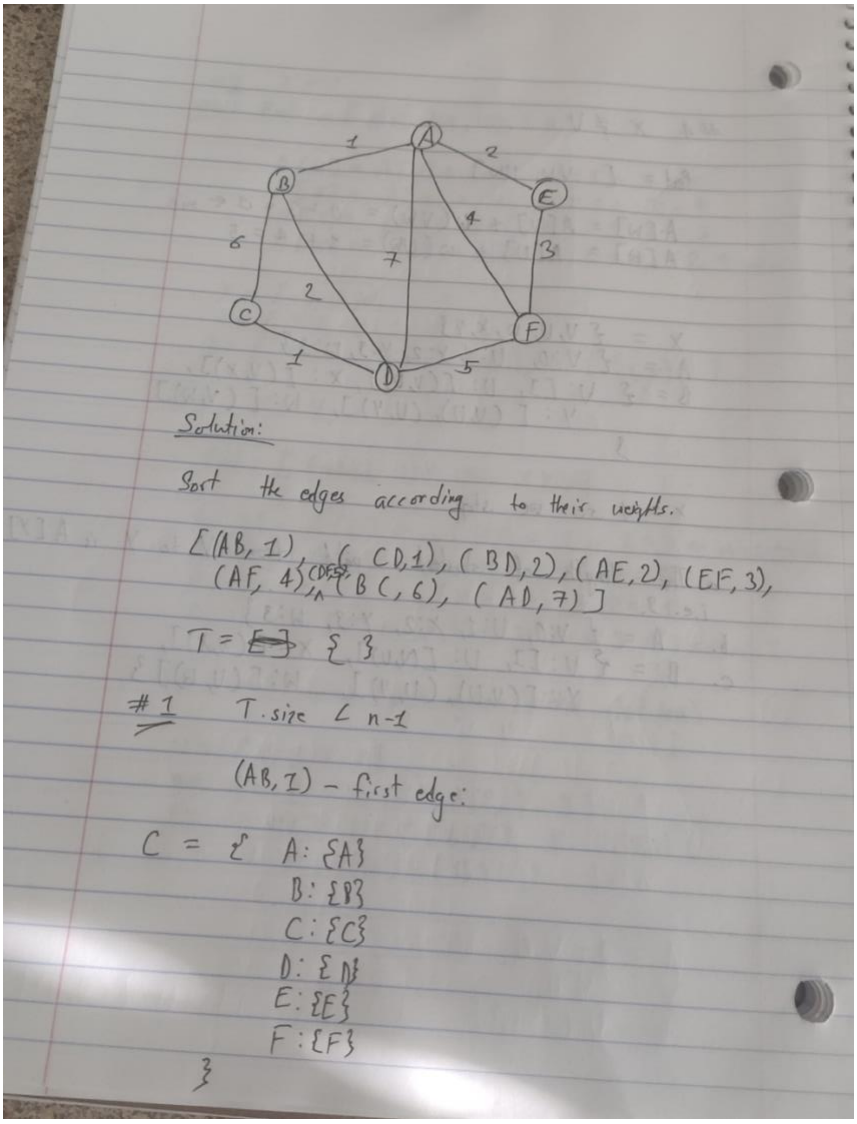
$X = V$ so we stop.

a. The length of shortest path from V to Y is A
i.e. 3.

b. $A = \{ V:0,\ U:1,\ X:2,\ Y:3,\ W:3 \}$

c. $B = \{ V:[],\ U:[(V,U)],\ X:[(V,X)],$
$Y:[(V,U),(U,Y)],\ W:[(V,W)] \}$

Answer of 2



Solution:

Sort the edges according to their weights.

[(AB, 1), (CD, 1), (BD, 2), (AE, 2), (EF, 3), (AF, 4), (BC, 6), (AD, 7)]

T = { }

#1 T.size < n-1

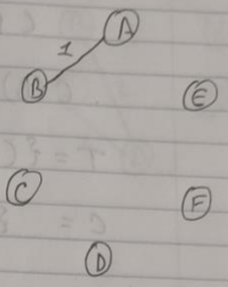
(AB, 1) - first edge:

C = { A: {A}
B: {B}
C: {C}
D: {D}
E: {E}
F: {F}
}

As, $C(A) \neq C(B)$

$T = \{ (AB, 1) \}$

$C = \{ A: \{A, B\}$
$D: \{A, B\}$
$C: \{C\}$
$D: \{D\}$
$E: \{E\}$
$F: \{F\}$
$\}$



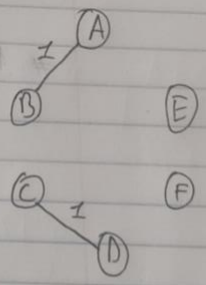
#2 T. size $< n-1$

$(CD, 1)$ is next.

$C(C) \neq C(D)$

$T = \{ (AB, 1), (CD, 1) \}$

$C = \{ A: \{A, B, \}$,
$B: \{A, B, \}$
$C: \{A, B, C\}$ $\{C, D\}$
$D: \{C, D\}$
$E: \{E\}$
$F: \{F\}$
$\}$



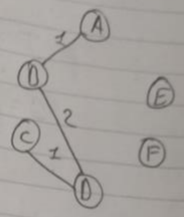
#3 T.size $< n-1$

$(BD, 2) \in next$

$C(B) \neq C(D)$

$T = \{(AB, 1), (CD, 1), (BD, 2)\}$

$C = \{ A: \{A, B, C, D\},$
$B: \{A, B, C, D\},$
$C: \{A, B, C, D\}$
$D: \{A, B, C, D\},$
$E: \{E\}$
$F: \{F\}$
$\}$



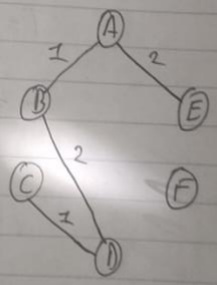
#4 T. size $< n-1$

$(AE, 2) \in next$

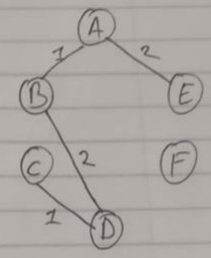
$C(A) \neq C(E)$

$T = [(AB,1), (CD,2), (BD,2), (AE,2)]$

$C = \{ A: \{A, B, C, D, E\}$
$B: \{A, B, C, D, E\},$
$C: \{A, B, C, D, E\},$
$D: \{A, B, C, D, E\}$
$E: \{A, B, C, D, E\}$
$F: \{F\}$
$\}$



#5 T.size $< n-1$



$(EF, 3) \Leftarrow$ next

$C(E) \neq C(F)$

$T = [(AB, 1), (CD, 2), (BD, 2), (AE, 2), (EF, 3)]$

$C = \{ A: \{A, B, C, D, E, F\},$
$B: \{A, B, C, D, E, F\}$
$C: \{A, B, C, D, E, F\}$
$D: \{A, B, C, D, E, F\}$
$E: \{A, B, C, D, E, F\}$
$F: \{A, B, C, D, E, F\}$
$\}$

T.size $== n-1$ so we stop.

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;
    }
}
```