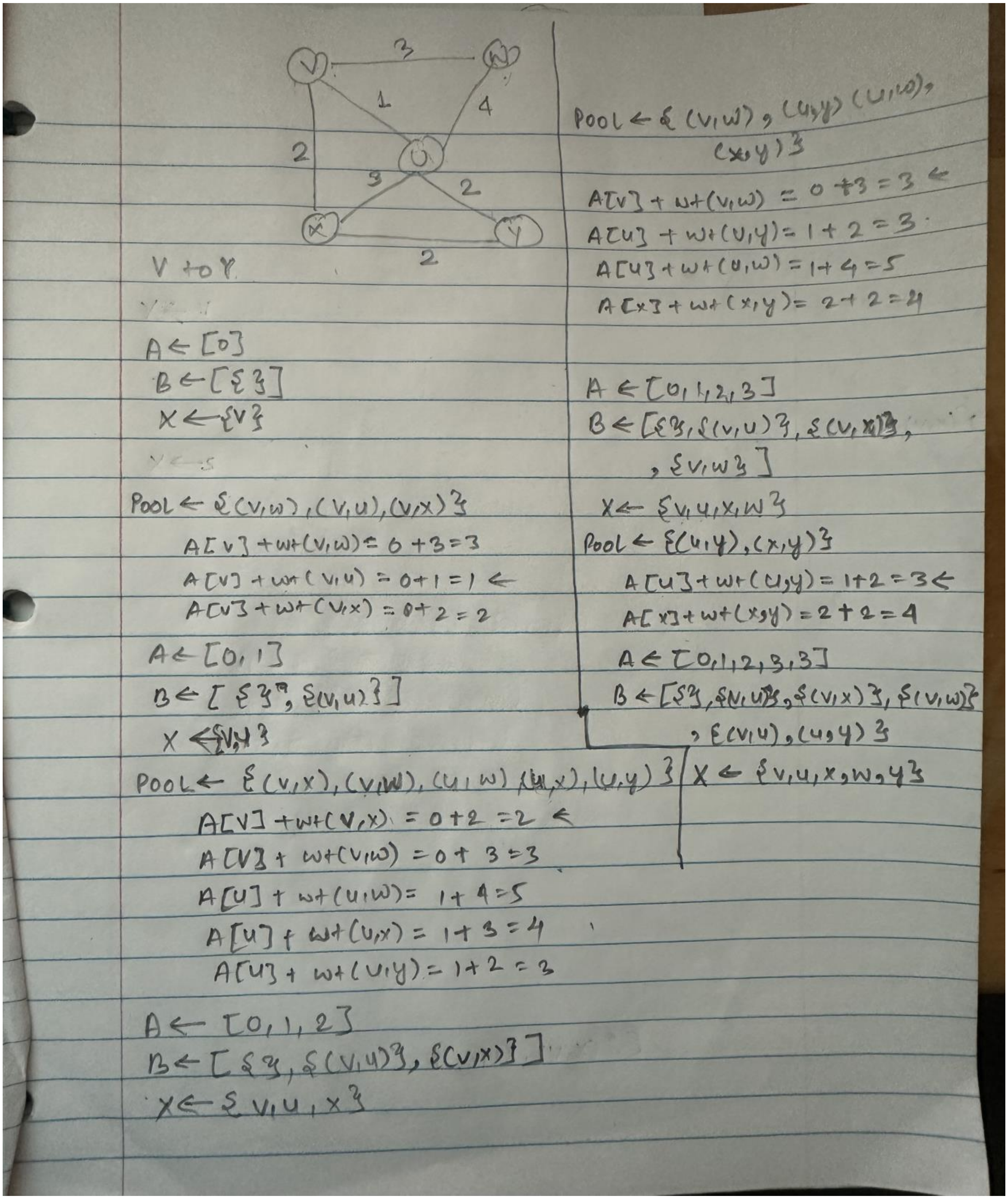
Q1.


V
W
U
X
Y
3
1
4
2
3
2
2

V to Y

A ← [0]
B ← [{ }]
X ← {V}

Pool ← {(V,W),(V,U),(V,X)}
A[V] + wt(V,W) = 0 + 3 = 3
A[V] + wt(V,U) = 0 + 1 = 1 ←
A[V] + wt(V,X) = 0 + 2 = 2

A ← [0, 1]
B ← [{ }, {(V,U)}]
X ← {V,U}

Pool ← {(V,X), (V,W), (U,W), (U,X), (U,Y)}
A[V] + wt(V,X) = 0 + 2 = 2 ←
A[V] + wt(V,W) = 0 + 3 = 3
A[U] + wt(U,W) = 1 + 4 = 5
A[U] + wt(U,X) = 1 + 3 = 4
A[U] + wt(U,Y) = 1 + 2 = 3

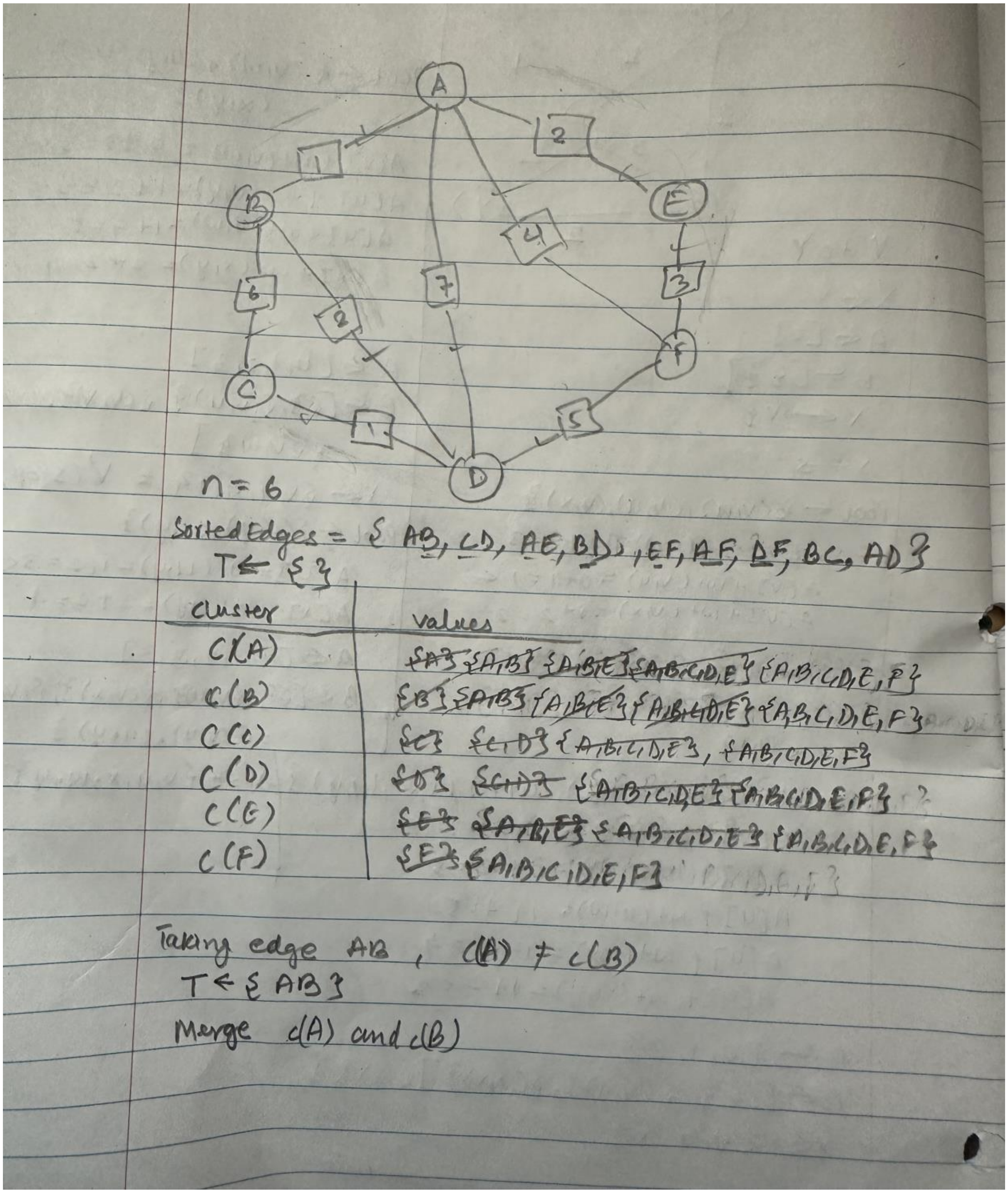
A ← [0, 1, 2]
B ← [{ }, {(V,U)}, {(V,X)}]
X ← {V, U, X}

Pool ← {(V,W), (U,Y) (U,W),
(X,Y)}
A[V] + wt(V,W) = 0 + 3 = 3 ←
A[U] + wt(U,Y) = 1 + 2 = 3
A[U] + wt(U,W) = 1 + 4 = 5
A[X] + wt(X,Y) = 2 + 2 = 4

A ← [0, 1, 2, 3]
B ← [{ }, {(V,U)}, {(V,X)},
, {V,W}]
X ← {V, U, X, W}
Pool ← {(U,Y), (X,Y)}
A[U] + wt(U,Y) = 1 + 2 = 3 ←
A[X] + wt(X,Y) = 2 + 2 = 4
A ← [0, 1, 2, 3, 3]
B ← [{ }, {(V,U)}, {(V,X)}, {(V,W)}
, {(V,U), (U,Y)}
X ← {V, U, X, W, Y}

a. 3
b. A = [0,1,2,3,3]
c. B = [{ } , { (v,u) }, { (v,x) }, { (v,w) }, { (v,u) , (u,y) }]

Q2.



n = 6

Sorted Edges = { AB, CD, AE, BD, , EF, AF, DF, BC, AD }

T ⇐ { }

cluster	values
C(A)	~~{A}~~ ~~{A,B}~~ ~~{A,B,E}~~ ~~{A,B,C,D,E}~~ {A,B,C,D,E,F}
C(B)	~~{B}~~ ~~{A,B}~~ ~~{A,B,E}~~ ~~{A,B,C,D,E}~~ {A,B,C,D,E,F}
C(C)	~~{C}~~ ~~{C,D}~~ ~~{A,B,C,D,E}~~, ~~{A,B,C,D,E,F}~~
C(D)	~~{D}~~ ~~{C,D}~~ ~~{A,B,C,D,E}~~ ~~{A,B,C,D,E,F}~~
C(E)	~~{E}~~ ~~{A,B,E}~~ ~~{A,B,C,D,E}~~ {A,B,C,D,E,F}
C(F)	~~{F}~~ {A,B,C,D,E,F}

Taking edge AB , c(A) ≠ c(B)

T ← { AB }

Merge c(A) and c(B)

Taking edge CD, $c(C) \neq c(D)$, $T \leftarrow \{AB, CD\}$
Merge $c(C)$, $c(D)$.

Taking edge AE, $c(A) \neq c(E)$, $T \leftarrow \{AB, CD, AE\}$.
Merge $c(A)$ and $c(E)$

Taking edge BD, $c(B) \neq c(D)$, $T \leftarrow \{AB, CD, AE, BD\}$.
Merge $c(B)$ and $c(D)$

Taking edge EF, $c(E) \neq c(F)$, $T \leftarrow \{AB, CD, AE, BD, EF\}$
Merge $c(E)$ and $c(F)$

T has $(n-1) = 6-1 = 5$ edges ~~{AB, CD, AE, BD, EF, AF}~~
~~Merge c(A) and c(F)~~

~~Taking edge DF, c(D) ≠ c(F), T ← {AB, CD, AE, BD, EF, AF, DF}~~
~~Merge c(D) and c(F)~~

Q3.

```
class Solution {
  static class Node implements Comparable<Node>{
    int weight;
    int node;

    Node(int node,int weight){
      this.weight = weight;
      this.node = node;
    }

    @Override
    public int compareTo(Node o) {
      return Integer.compare(this.weight,o.weight);
    }
  }

  public int networkDelayTime(int[][] times, int n, int k) {
   int[] result = new int[n+1];

    ArrayList<ArrayList<Node>> adjList = new ArrayList<>(n+1);
    // since there is no node 0. reduce indices by 1
    for(int i=0;i<=n;i++){
      adjList.add(i,new ArrayList<>());
    }

    for(int[] arr: times){
      adjList.get(arr[0]).add(new Node(arr[1],arr[2]));
    }

    PriorityQueue<Node> pq = new PriorityQueue<>();
    pq.add(new Node(k,0)); // from k to k it costs 0
    Set<Integer> visited = new HashSet<>();

    while (!pq.isEmpty()){
      Node p = pq.remove();
      if(!visited.contains(p.node)){
        visited.add(p.node);
        result[p.node] = p.weight;

        for(Node node: adjList.get(p.node)){
```

```
                    int d =  p.weight + node.weight;
                    pq.add(new Node(node.node,d));
                }

            }
        }

        if(visited.size() != n) return -1;

        int max = Integer.MIN_VALUE;
        for(int ar: result){
            if(ar > max){
                max = ar;
            }
        }

        return max;
    }
}
```

Q4.

```
class Solution {
    static class UnionFind {
        private final int[] parent;
        private final int[] rank;

        public UnionFind(int n) {
            this.parent = new int[n];
            this.rank = new int[n];

            for (int i = 0; i < n; i++) {
                parent[i] = i; // initially each node is its own parent
            }
        }

        public int find(int x) {
            if (parent[x] != x) {
                // Path compression step: make all nodes point directly to the root
                parent[x] = find(parent[x]);
```

```
      }
      return parent[x];
    }

    // Union two sets containing 'x' and 'y' (by rank)
    public boolean union(int x, int y) {
      int rootX = find(x);
      int rootY = find(y);

      if (rootX != rootY) {
        int comp = Integer.compare(this.rank[rootX], this.rank[rootY]);
        switch (comp) {
          case -1 -> this.parent[rootX] = rootY;
          case 0 -> {
            this.parent[rootX] = rootY;
            this.rank[rootY]++;
          }
          case 1 -> this.parent[rootY] = rootX;
        }
        return true;
      }
      // already same parent
      return false;
    }
  }

  public static int[] findRedundantConnection(int[][] edges) {
   UnionFind un = new UnionFind(edges.length+1);
   for(int[] edge: edges){
     if(!un.union(edge[0],edge[1])){
       return edge;
     }
   }
   // guaranteed to have a redundant edge so it below
   // will not execute
   return new int[]{};
  }
}
```

Q5.

```
class Solution {

  static class UnionFind {
    private final int[] parent;
    private final int[] rank;

    public UnionFind(int n) {
      this.parent = new int[n];
      this.rank = new int[n];

      for (int i = 0; i < n; i++) {
        parent[i] = i; // initially each node is its own parent
      }
    }

    public int find(int x) {
      if (parent[x] != x) {
        // Path compression step: make all nodes point directly to the root
        parent[x] = find(parent[x]);
      }
      return parent[x];
    }

    // Union two sets containing 'x' and 'y' (by rank)
    public boolean union(int x, int y) {
      int rootX = find(x);
      int rootY = find(y);

      if (rootX != rootY) {
        int comp = Integer.compare(this.rank[rootX], this.rank[rootY]);
        switch (comp) {
          case -1 -> this.parent[rootX] = rootY;
          case 0 -> {
            this.parent[rootX] = rootY;
            this.rank[rootY]++;
          }
          case 1 -> this.parent[rootY] = rootX;
        }
        return true;
      }
      // already same parent
      return false;
    }
  }

  public int minCostConnectPoints(int[][] points) {
    int[][] paths = new int[points.length*points.length][3];
    int cursor = 0;
    for(int i=0;i<points.length;i++){
      for (int j=i+1;j<points.length;j++){
        int dist = Math.abs(Math.abs(points[i][0]-points[j][0]) + Math.abs(points[i][1]-
points[j][1]));
        paths[cursor++] = new int[]{dist,i,j};
      }
    }
    Arrays.sort(paths,(a,b)->a[0]-b[0]);
    int result = 0;
    UnionFind uf = new UnionFind(paths.length);
    for(int[] arr: paths){
      if(arr[0]==0&&arr[1]==0&&arr[2]==0) continue;
      if(uf.union(arr[1],arr[2])){
        result += arr[0];
      }
    }

    return result;

 }
}
```