Q1.A directed graph G is semi-connected if for every pair of nodes u, v ∈ V (G), there is a path from u to v or a path from v to u. Give an efficient algorithm in pseudo code which, given G, determines whether G is semi-connected or not. Prove the correctness and analyze the running time of your algorithm.

- Code:

```
IsSemiconnected(G)    //G (v1, v2, ......, vn) is a directed graph
    Using Depth-first-search find all strongly connected components G1, G2, ..., Gk;
    V[V1, V2, ......, Vk];    //Vi is the vertex set of Gi
    Topological-sorting( V [V1,V2, ......, Vk] );
    flag = true;    //flag is a bool type variable
    for i = 1 up to k do
        If  edge_exist(V[i], V[i+1]) = false then
            flag = false;
    return flag;
```

- Prove:

If a directed graph G is semi-connected, then there exists a path that passes all vertex in G. If we have G1, ..., Gk as strongly connected components in G, we know that the graph is semi-connected for G1, ..., Gk. If there exists a path in G1, ..., Gk, such that contains at least one vertex of every graph in G1, ..., Gk, we ca conclude that the directed G is semi-connected since all sub graph are semi-connected, and are connected with each other.

- Time analysis:

Depth-first-search find all strongly connected components G1, G2, ..., Gk costs O(V+E), Topological-sorting costs O(V+E), loop costs O(V). Thus the total time cost is T(n) = O(V+E).

Q2. Prove that if all edge weights of a graph are positive, then any subset of edges that connects all vertices and has minimum total weight must be a tree. Give an example to show that the same conclusion does not follow if we allow some weights to be non-positive.

- Prove:

Suppose we have a graph, vertexes still connected after removing the edge form the cycle. We call the new graph G'. The weight of G' is equal to weight of G minus weight of the removed edge. If G' is a tree which means there is no cycle. Adding any edge between any two vertex will increases the total weight and forming cycle, which contradicts the definition of a tree. Thus then any subset of edges that connects all vertices and has minimum total weight must be a tree.

If we allow negative weight edges in graph G. Suppose G is a tree containing only one cycle which consist of edges e1 and e2 between two vertex v1 and v2. Assume weight of e1 and e2 are negative, removing any edges between e1 and e2 would increase the total weight, which contradicts the minimum weight constrains.

Q3.

◆ Running time report of **Prim's algorithm**:

[1]. Prim's minimum spanning tree algorithm using **adjacent list**

E / V	3*n	N^1.5	(n-1)/2
100	1744800 ns	2243300 ns	1342700 ns
200	2547100 ns	2614400 ns	1831400 ns
400	3171600 ns	3527000 ns	2074600 ns
800	3977300 ns	5267200 ns	3015900 ns

[2]. Prim's minimum spanning tree algorithm using **adjacent matrix**

E / V	3*n	N^1.5	(n-1)/2
100	17607100 ns	2243300 ns	16723500 ns
200	30210400 ns	29648400 ns	36947700 ns
400	145349200 ns	147127400 ns	120842600 ns
800	881566000 ns	882888100 ns	888191100 ns

◆ Running time report of **creating input graphs**:

[1].Procedure of creating graph by **adjacent list**

```
public static void fillTheGraph(Graph G, int e)
{
   for (int i = 0; i<G.V-1; i++){
      // 1 to 50 random weight
      addEdge(G, i, i+1, (int)(1+Math.random()*(50-1+1)));
   }
   int remain = e - G.V + 1;
   for (int j = 0; j<remain; j++){
      // 1 to 50 random weight
      addEdge(G, (int)(0+Math.random()*(G.V-1-0+0)), (int)(0+Math.random()*(G.V-1-0+0)), (int)(1+Math.random()*(50-1+1)));
   }
}
```

E / V	3*n	N^1.5	(n-1)/2
100	927400 ns	1210400 ns	609400 ns
200	1040100 ns	1601500 ns	743800 ns
400	1432600 ns	2377800 ns	1071900 ns
800	1548400 ns	4898000 ns	1215100 ns

[2].Procedure of creating graph by **adjacent matrix**

```
public static int [][] fillMatrix(int V, int E){
   int[][] matrix = new int [V][V];
   for(int i=0; i<V; i++){
      for(int j=0; j<V; j++){
         if(j == i+1){
            int weight = (int)(1+Math.random()*(50-1+1));
            matrix[i][j] = weight;
            //System.out.println(matrix[i][j]);
         }
         else{
            matrix[i][j] = INT_MAX;
         }

      }
   }

   int remain = E-V+1;
   for(int k = 0; k<remain; k++){
      int src = (int)(0+Math.random()*(V-1-0+0));
      int des = (int)(0+Math.random()*(V-1-0+0));
      int weight = (int)(1+Math.random()*(50-1+1));

      while (matrix[src][des] != INT_MAX){
         src = (int)(0+Math.random()*(V-1-0+0));
         des = (int)(0+Math.random()*(V-1-0+0));
      }

      matrix[src][des] = weight;

   }

   return matrix;
}
```

E V	3*n	N^1.5	(n-1)/2
100	315200 ns	402600 ns	332000 ns
200	1076500 ns	1299800 ns	1028500 ns
400	3108400 ns	4046600 ns	2954600 ns
800	6171500 ns	7713700 ns	6373800 ns

◆ Code:

```
public class prims {
   static class Mynode {
      int dest;
      int weight;
      Mynode(int a, int b)
      {
         dest = a;
         weight = b;
      }
   }
   static class Graph {
      int V;

      // List of adjacent nodes of a given vertex
      LinkedList<Mynode>[] adj;


      // Constructor
      Graph(int e)
      {
         V = e;
         adj = new LinkedList[V];
         for (int o = 0; o < V; o++)
            adj[o] = new LinkedList<>();
      }

   }
   class node {
      int vertex;
      int key;
   }
   class comparator implements Comparator<node> {

      @Override
      public int compare(node node0, node node1)
      {
         return node0.key - node1.key;
      }
   }
   static void addEdge(Graph graph, int src, int dest, int weight)
   {

      Mynode node0 = new Mynode(dest, weight);
```

```
        Mynode node = new Mynode(src, weight);
        graph.adj[src].addLast(node0);
        graph.adj[dest].addLast(node);
    }

    void prims_mst(Graph graph)
    {
        Boolean[] mstset = new Boolean[graph.V];
        node[] e = new node[graph.V];

        int[] parent = new int[graph.V];

        for (int o = 0; o < graph.V; o++)
            e[o] = new node();

        for (int o = 0; o < graph.V; o++) {
            mstset[o] = false;

            e[o].key = Integer.MAX_VALUE;
            e[o].vertex = o;
            parent[o] = -1;
        }

        mstset[0] = true;

        e[0].key = 0;

        TreeSet<node> queue = new TreeSet<node>(new comparator());

        for (int o = 0; o < graph.V; o++)
            queue.add(e[o]);


        while (!queue.isEmpty()) {
            node node0 = queue.pollFirst();
            mstset[node0.vertex] = true;

            for (Mynode iterator : graph.adj[node0.vertex]) {

                if (mstset[iterator.dest] == false) {
                    if (e[iterator.dest].key > iterator.weight) {
                        queue.remove(e[iterator.dest]);
                        e[iterator.dest].key = iterator.weight;
                        queue.add(e[iterator.dest]);
                        parent[iterator.dest] = node0.vertex;
                    }
                }
            }
        }


        // Prints the vertex pair of mst
//        for (int o = 1; o < graph.V; o++) {
//            System.out.println(parent[o] + " "
//                    + "-"
//                    + " " + o);
//        }

    }

    public static void fillTheGraph(Graph G, int e)
    {

        for (int i = 0; i<G.V-1; i++){
            // 1 to 50 random weight
            addEdge(G, i, i+1, (int)(1+Math.random()*(50-1+1)));
        }
        int remain = e - G.V + 1;
        for (int j = 0; j<remain; j++){
            // 1 to 50 random weight
            addEdge(G, (int)(0+Math.random()*(G.V-1-0+0)), (int)(0+Math.random()*(G.V-1-0+0)), (int)(1+Math.random()*(50-1+1)));
        }
    }


//Adjacent Matrix

    static int INT_MAX = Integer.MAX_VALUE;

    static boolean isValidEdge(int u, int v, boolean[] inMST)
    {
        if (u == v)
            return false;
        if (inMST[u] == false && inMST[v] == false)
            return false;
        else if (inMST[u] == true && inMST[v] == true)
            return false;
        return true;
    }

    public static int [][] fillMatrix(int V, int E){
        int[][] matrix = new int [V][V];
        for(int i=0; i<V; i++){
            for(int j=0; j<V; j++){
                if(j == i+1){
                    int weight = (int)(1+Math.random()*(50-1+1));
                    matrix[i][j] = weight;
                    //System.out.println(matrix[i][j]);
                }
                else{
                    matrix[i][j] = INT_MAX;
                }
```

```
            }
        }

        int remain = E-V+1;
        for(int k = 0; k<remain; k++){
            int src = (int)(0+Math.random()*(V-1-0+0));
            int des = (int)(0+Math.random()*(V-1-0+0));
            int weight = (int)(1+Math.random()*(50-1+1));

            while (matrix[src][des] != INT_MAX){
                src = (int)(0+Math.random()*(V-1-0+0));
                des = (int)(0+Math.random()*(V-1-0+0));
            }

            matrix[src][des] = weight;

        }

        return matrix;
    }


    static void primMST(int cost[][], int V)
    {
        boolean []inMST = new boolean[V];

        inMST[0] = true;

        int edge_count = 0, mincost = 0;
        while (edge_count < V - 1)
        {

            int min = INT_MAX, a = -1, b = -1;
            for (int i = 0; i < V; i++)
            {
                for (int j = 0; j < V; j++)
                {
                    if (cost[i][j] < min)
                    {
                        if (isValidEdge(i, j, inMST))
                        {
                            min = cost[i][j];
                            //System.out.println("Mini = "+min);
                            a = i;
                            b = j;
                        }
                    }
                }
            }

            if (a != -1 && b != -1)
            {
                edge_count++;
                //System.out.printf("Edge %d:(%d, %d) cost: %d \n",edge_count, a, b, min);
                mincost = mincost + min;
                inMST[b] = inMST[a] = true;
            }
        }
        System.out.printf("\n Minimum cost = %d \n", mincost);
    }

    public static void main(String[] args)
    {
        int V = 100;
        int E = 3*V;
        Graph graph = new Graph(V);
        fillTheGraph(graph, E);

        long begin1, end1;
        long time1;

        prims e = new prims();

        begin1 = System.nanoTime();
        e.prims_mst(graph);
        end1 = System.nanoTime();
        time1 = end1 -  begin1;

        System.out.println(time1+" ns"+"\n");


        int[][] matrix = fillMatrix(V, E);

        long begin2, end2;
        long time2;
        begin2 = System.nanoTime();
        primMST(matrix, V);
        end2 = System.nanoTime();
        time2 = end2 -  begin2;

        System.out.println(time2+" ns"+"\n");



    }
}
```

Q4.

◆ Structure of optimal solution

Dx(y): the least cost from x to y

Goal: each node notifies neighbors only when its Dv changes, then neighbors notify their neighbors

- Case 1: source node is the destination, x = y, Dx(y) = 0.
- Case 2: source node differs from destination, x != y, Dx(y) = min(v){c(x, v) + Dv(y)}.

◆ Bellman equation:

```
Dx(y) = { 0                                    ,x = y
           min(v){ c(x, v) + Dv(y) }     ,x != y     }
```

◆ Code

```
Distance_vector_algorithm(G, i, s, v)   //G is a graph represented by adjacent matrix
    If s is equal to v then
        return 0;
    w;            //w is neighbor nodes of s
    while there are neighbor node w not visited do
            Path = mini{ opt(i-1, w1, v), opt(i-2, w2, v), ....., opt(i-k, wk, v) };
    return path + cost(s, wk);
```

◆ Time analysis

From time-to-time, each node sends its own distance vector estimate to neighbors, which takes O(V). When a node x receives new DV estimate from any neighbor v, it saves v's distance vector and it updates its own DV, which takes O(V^3). Thus the time complexity is T(n) = O(V^3).

Q5.

◆ Running time report of **Dijkstra's algorithm**:

[1]. Dijkstra's algorithm using **adjacent list**

E V	3*n	N^1.5	(n-1)/2
100	1368200 ns	19771500 ns	293600 ns
200	2460600 ns	77584400 ns	365700 ns
400	2533700 ns	355401400 ns	499000 ns
800	3502100 ns	2929802100 ns	832000 ns

[2]. Dijkstra's algorithm using **adjacent matrix**

E V	3*n	N^1.5	(n-1)/2
100	17607100 ns	2243300 ns	16723500 ns
200	30210400 ns	29648400 ns	36947700 ns
400	145349200 ns	147127400 ns	120842600 ns
800	881566000 ns	882888100 ns	888191100 ns

◆ Running time report of **creating input graphs**:

[1].Procedure of creating graph by **adjacent list**

```
public static void fillTheGraph(List<List<Node>> adj, int V, int E) {

    for (int i = 0; i < V - 1; i++) {
        int weight = (int) (1 + Math.random() * (20 - 1 + 1));
        adj.get(i).add(new Node(i + 1, weight));
    }
    int remain = E - V + 1;
    for (int j = 0; j < remain; j++) {
        int weight = (int) (1 + Math.random() * (20 - 1 + 1));
        adj.get((int) (0 + Math.random() * (V - 1 - 0 + 0))).add(new Node((int) (0 + Math.random() * (V - 1 - 0 + 0)), weight));
    }
}
```

E / V	3*n	N^1.5	(n-1)/2
100	1000500 ns	1267700 ns	728400 ns
200	1082200 ns	1873300 ns	882500 ns
400	1398800 ns	3064900 ns	1054500 ns
800	1555400 ns	5734600 ns	1180800 ns

[2].Procedure of creating graph by **adjacent matrix**

```
public static int [][] fillMatrix(int V, int E){
    int[][] matrix = new int [V][V];
    for(int i=0; i<V; i++){
        for(int j=0; j<V; j++){
            if(j == i+1){
                int weight = (int)(1+Math.random()*(50-1+1));
                matrix[i][j] = weight;
                //System.out.println(matrix[i][j]);
            }
            else{
                matrix[i][j] = INT_MAX;
            }

        }
    }

    int remain = E-V+1;
    for(int k = 0; k<remain; k++){
        int src = (int)(0+Math.random()*(V-1-0+0));
        int des = (int)(0+Math.random()*(V-1-0+0));
        int weight = (int)(1+Math.random()*(50-1+1));

        while (matrix[src][des] != INT_MAX){
            src = (int)(0+Math.random()*(V-1-0+0));
            des = (int)(0+Math.random()*(V-1-0+0));
        }

        matrix[src][des] = weight;

    }

    return matrix;
}
```

E / V	3*n	N^1.5	(n-1)/2
100	352100 ns	448700 ns	367600 ns
200	1350600 ns	1429100 ns	1173600 ns
400	3111700 ns	3618700 ns	499000 ns
800	7334400 ns	8793300 ns	6459800 ns

◆ Code

```
class Graph_pq {
    int dist[];
    Set<Integer> visited;
    PriorityQueue<Node> pqueue;
    int V;
    List<List<Node>> adj_list;

    public Graph_pq(int V) {
        this.V = V;
        dist = new int[V];
        visited = new HashSet<Integer>();
        pqueue = new PriorityQueue<Node>(V, new Node());
    }

    public void algo_dijkstra(List<List<Node>> adj_list, int src_vertex) {
        this.adj_list = adj_list;

        for (int i = 0; i < V; i++)
            dist[i] = Integer.MAX_VALUE;


        pqueue.add(new Node(src_vertex, 0));

        dist[src_vertex] = 0;
        while (visited.size() != V) {

            int u = pqueue.remove().node;

            visited.add(u);
            graph_adjacentNodes(u);
        }
    }

    private void graph_adjacentNodes(int u) {
        int edgeDistance = -1;
        int newDistance = -1;

        for (int i = 0; i < adj_list.get(u).size(); i++) {
            Node v = adj_list.get(u).get(i);

            if (!visited.contains(v.node)) {
                edgeDistance = v.cost;
                newDistance = dist[u] + edgeDistance;

                if (newDistance < dist[v.node])
                    dist[v.node] = newDistance;

                pqueue.add(new Node(v.node, dist[v.node]));
            }
        }
    }


    public static void fillTheGraph(List<List<Node>> adj, int V, int E) {

        for (int i = 0; i < V - 1; i++) {
            int weight = (int) (1 + Math.random() * (20 - 1 + 1));
            adj.get(i).add(new Node(i + 1, weight));
        }
        int remain = E - V + 1;
        for (int j = 0; j < remain; j++) {
            int weight = (int) (1 + Math.random() * (20 - 1 + 1));
            adj.get((int) (0 + Math.random() * (V - 1 - 0 + 0))).add(new Node((int) (0 + Math.random() * (V - 1 - 0 + 0)), weight));
        }
    }

    // ------------------------Adjacent Matrix----------------------------


    private static char[] points;
    private static int[][] arc;
    private static final int INF = Integer.MAX_VALUE;


    private static void dijkstra(int start) {
        boolean[] flag = new boolean[points.length];
        int[] distence = arc[start];
        int[] prev = new int[points.length];

        for (int i = 0; i < points.length; i++) {
            prev[i] = start;
        }

        flag[start] = true;
        prev[start] = start;
        Stack path = new Stack();
        int currentSmall = 0;


        for (int j = 1; j < points.length; j++) {
            int minDistence = INF;
            for (int i = 0; i < points.length; i++) {

                if (!flag[i] && distence[i] < minDistence) {
                    minDistence = distence[i];
                    currentSmall = i;
                }
            }
            flag[currentSmall] = true;


            for (int i = 0; i < points.length; i++) {
                if (!flag[i])
                {
                    int temp = (arc[currentSmall][i] == INF ? INF : (minDistence + arc[currentSmall][i]));
                    if (temp < distence[i])
                    {
                        distence[i] = temp;
                        prev[i] = currentSmall;
                    }
                }
```

```
            }
        }

        System.out.printf("dijkstra(%c): \n", points[start]);
        for (int i = 0; i < points.length; i++) {

            int j = prev[i];
            while (j != start) {
                path.push(points[j]);
                j = prev[j];
            }
            System.out.print("shortestDistenct(" + points[start] + "," + points[i] + ") = " + distence[i] + "    ");
            System.out.print("path = " + points[start] + " ");
            while (!path.isEmpty()) {
                System.out.print(path.pop() + " ");
            }
            System.out.println(points[i]);
        }
    }

    public static int [][] fillMatrix(int V, int E){
        int[][] matrix = new int [V][V];
        for(int i=0; i<V; i++){
            for(int j=0; j<V; j++){
                if(j == i+1 || j== i-1){
                    int weight = (int)(1+Math.random()*(50-1+1));
                    matrix[i][j] = weight;
                    matrix[j][i] = weight;
                    //System.out.println(matrix[i][j]);
                }
                else{
                    matrix[i][j] = INF;
                }

            }
        }

        int remain = E-V+1;
        for(int k = 0; k<remain; k++){
            int src = (int)(0+Math.random()*(V-1-0+0));
            int des = (int)(0+Math.random()*(V-1-0+0));
            int weight = (int)(1+Math.random()*(50-1+1));

            while (matrix[src][des] != INF){
                src = (int)(0+Math.random()*(V-1-0+0));
                des = (int)(0+Math.random()*(V-1-0+0));
            }

            matrix[src][des] = weight;

        }


        return matrix;
    }




    public static void main(String arg[]) {
        long begin1, end1;
        long time1;
        long begin2, end2;
        long time2;
        long begin3, end3;
        long time3;

        int V = 800;
        int E = (V-1)/2;
        int source = 0;

        List<List<Node>> adj_list = new ArrayList<List<Node>>();

        for (int i = 0; i < V; i++) {
            List<Node> item = new ArrayList<Node>();
            adj_list.add(item);
        }


        begin1 = System.nanoTime();
        fillTheGraph(adj_list, V, E);
        end1 = System.nanoTime();
        time1 = end1 - begin1;

        System.out.println("Fill the list: "+time1);


        begin2 = System.nanoTime();
        arc = fillMatrix(V, E);
        end2 = System.nanoTime();
        time2 = end2 - begin2;

        System.out.println("Fill the matrix: "+time2);


        // call Dijkstra's algo method
        Graph_pq dpq = new Graph_pq(V);


        begin3 = System.nanoTime();
        dpq.algo_dijkstra(adj_list, source);
        end3 = System.nanoTime();
        time3 = end3 - begin3;

        System.out.println("List cost time: "+time3);

        // Print the shortest path from source node to all the nodes
//        System.out.println("The shorted path from source node to other nodes:");
//        System.out.println("Source\t\t" + "Node#\t\t" + "Distance");
//        for (int i = 0; i < dpq.dist.length; i++)
//            System.out.println(source + " \t\t " + i + " \t\t " + dpq.dist[i]);


        arc = fillMatrix(V, E);
```

```
        dijkstra(1);
    }

    // Node class
    static class Node implements Comparator<Node> {
        public int node;
        public int cost;

        public Node() {
        }

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

        @Override
        public int compare(Node node1, Node node2) {
            if (node1.cost < node2.cost)
                return -1;
            if (node1.cost > node2.cost)
                return 1;
            return 0;
        }
    }
}
```

Q6.

```
EXTEND-SHORTEST-PATHS(∏, L, W)
        n = L.row;
        L' = l'[i, j];          // L' is a n × n matrix
        ∏' = π'[i, j];       // ∏' is a n × n matrix
        for i = 1 to n do
            for j = 1 to n do
                    l'[i, j] = INFINITE; π'[i, j] = NIL;
                    for k = 1 to n do
                        if l[i, k] + l[k, j] < l[i, j] then
                            l[i, j] = l[i, k] + l[k, j];
                                if k != j then π'[i, j] = k;
                                else π'[i, j] = π[i, j];
        return (∏', L');


SLOW-ALL-PAIRS-SHORTEST-PATHS(W)
        n = W.rows;
        L(1) = W;
        ∏(1) = π[i, j](1);
        if there is an edge from i to j then
            π[i, j](1) = i;
        else π[i, j](1) = NIL;

        for m = 2 to n - 1 do
            ∏(m), L(m) = EXTEND-SHORTEST-PATHS(∏(m-1), L(m-1), W);
        return (∏(n-1), L(n-1));
```

Q7.

```
Faster-All-Pairs-Shortest-Paths(W)
    matrix = W; r = 1;      // matrix is 2 x 2 matrix
    while r < n - 1 do
        matrix =Extend-Shortest-Paths(matrix, matrix);
        r = 2r;
    return matrix.
```