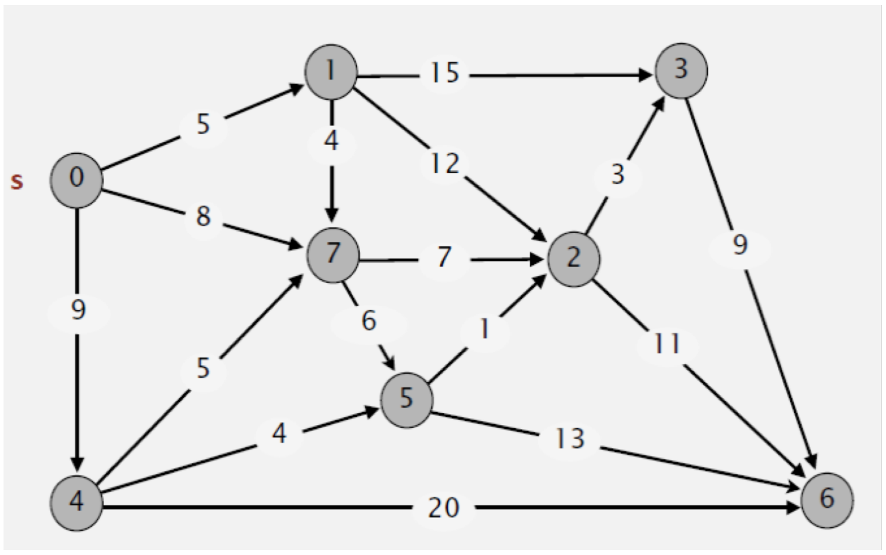
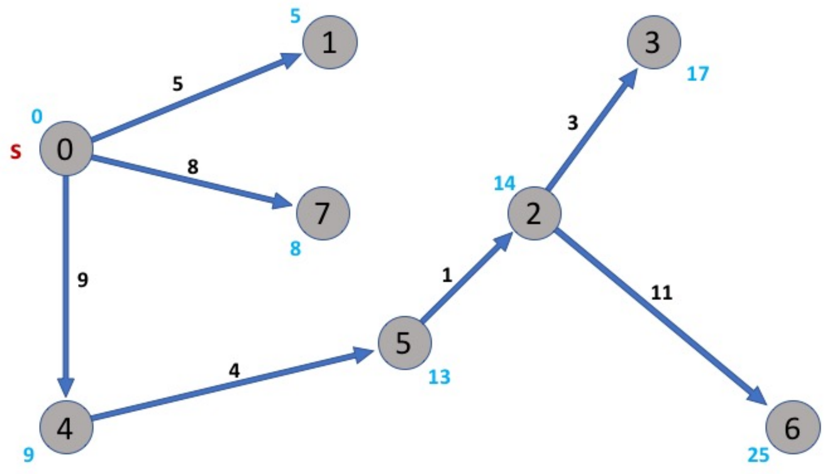
Exercise 5. Answer Sheet

Student's Name: Yuta Nemoto Student's ID: s1240234

Problem 1. *(15 points)* Consider the graph below.

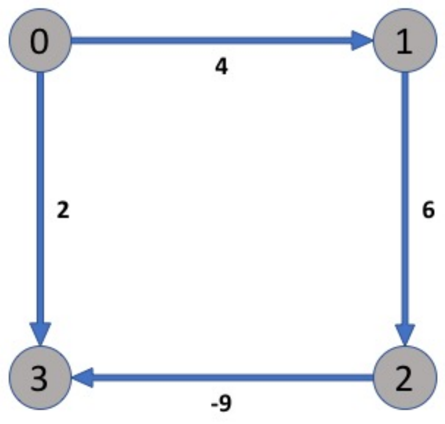


Draw a shortest path spanning tree with root at vertex ***s***. Show the cost (weight) of paths to each vertex.



Problem 2. *(15 points)* Dijksta's algorithm cannot handle negative weights. Show an example and explain what happens.

For example, for the graph below,



Dijksta’s algorithm selects vertex 3 immediately after beginning the discovering from vertex 0, because the edge from 0 to 3 is the shortest path in the directed edges from vertex 0. However, actually the shortest path from vertex 0 to 3 is $0 \rightarrow 1 \rightarrow 2 \rightarrow 3$, because

the weight of path $(0 \rightarrow 3) = \mathbf{2 > 1} = 4 + 6 + (-9) =$ weight of path $(0 \rightarrow 1 \rightarrow 2 \rightarrow 3)$

So, in such a case, Dijksta’s algorithm doesn’t work correctly. In other words, it doesn’t found the shortest path rightly. That’s why Dijksta’s algorithm cannot handle negative weights.

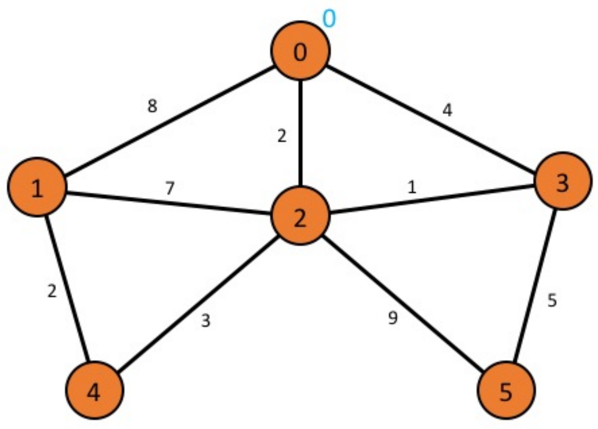
To avoid this problem, we can add the value 9 for all weight of edges to make them positive.

Problem 3. *(20 points)* Extend the pseudocode of the Bellman-Ford algorithm given at the lecture so it can detect negative cycles.

```
def Bellman-Ford-modified (G,s,w):
Init-SS (G,s)
for i=1 to |G.V|-1
    for each edge (u,v) in G.E
            RELAX (u,v,w)
            if v.d > u.d + w(u, v)
                Detect Negative Cycle!
```

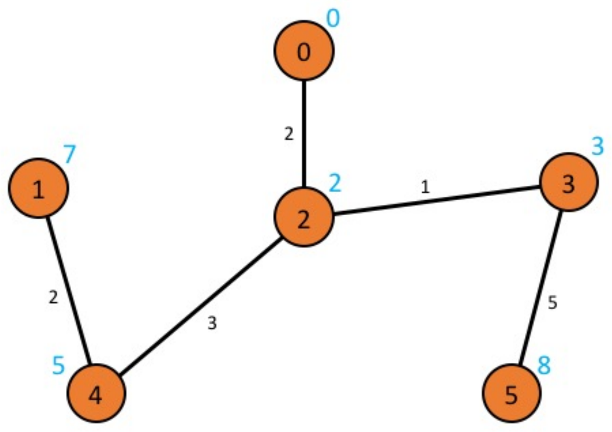
Problem 4. *(50 points)* Write a program implementing Dijksta's algorithm. Upload your source code. Show your input graph and the obtained shortest path spanning tree in the space below.

Input Graph:
(The same dataset with the lecture slide p. 12 – 13)



Obtained Shortest Path Spanning Tree (Output Graph):

(The Same dataset with the lecture slide p.12 – 13)



Program's Result Output (Input is did in the main method by following the example above):

```
<Result of DIJKSTRA>
Edges:
Edge from 1 to 4, weight: 2.0
Edge from 2 to 0, weight: 2.0
Edge from 3 to 2, weight: 1.0
Edge from 4 to 2, weight: 3.0
Edge from 5 to 3, weight: 5.0
Distance of each edges from the root node:
Node 0: 0.0 from the root node
Node 1: 7.0 from the root node
Node 2: 2.0 from the root node
Node 3: 3.0 from the root node
Node 4: 5.0 from the root node
Node 5: 8.0 from the root node
```

How to compile/run:

Please execute the command:

```
javac Graph.java
java Graph
```

Source Code (The original one is uploaded separately):

<Graph.java>

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;
import java.util.PriorityQueue;

public class Graph {

        public static int numberOfNodes = 6;
        public static Node start = new Node("0");
        public NodeList nodes = new NodeList();
        public NodeList[] Adj = new NodeList[Graph.numberOfNodes];

        /* Constructor */
        public Graph(double[][] weightMatrix) {
                this.nodes.add(start);
                for(int i = 1; i < Graph.numberOfNodes; i++) this.nodes.add(new Node(String.valueOf(i)));
                for(int i = 0; i < Graph.numberOfNodes; i++) Adj[i] = new NodeList();
                for(int i = 0; i < Graph.numberOfNodes; i++)
                        for(int j = 0; j < Graph.numberOfNodes; j++)
                                if(weightMatrix[i][j] != 0.0 && weightMatrix[i][j] != Node.DISTANCE_INFINITY)
                                        this.Adj[i].add(this.nodes.get(j));
        }

        /* Main Method */
        public static void main(String[] args) {
                // Prepare the weight matrix
                double[][] weightMatrix = new double[Graph.numberOfNodes][Graph.numberOfNodes];
                for(int i = 0; i < Graph.numberOfNodes; i++)
                        for(int j = 0; j < Graph.numberOfNodes; j++)
                                weightMatrix[i][j] = Node.DISTANCE_INFINITY;
                weightMatrix[0][1] = 8.0; // A-B: 8
                weightMatrix[0][2] = 2.0; // A-C: 2
                weightMatrix[0][3] = 4.0; // A-D: 4
                weightMatrix[1][0] = 8.0; // B-A: 8
                weightMatrix[1][2] = 7.0; // B-C: 7
                weightMatrix[1][4] = 2.0; // B-E: 2
                weightMatrix[2][0] = 2.0; // C-A: 2
                weightMatrix[2][1] = 7.0; // C-B: 7
                weightMatrix[2][3] = 1.0; // C-D: 1
                weightMatrix[2][4] = 3.0; // C-E: 3
                weightMatrix[2][5] = 9.0; // C-F: 9
                weightMatrix[3][0] = 4.0; // D-A: 4
                weightMatrix[3][2] = 1.0; // D-C: 1
                weightMatrix[3][5] = 5.0; // D-F: 5
                weightMatrix[4][1] = 2.0; // E-B: 2
                weightMatrix[4][2] = 3.0; // E-C: 3
                weightMatrix[5][2] = 9.0; // F-C: 9
                weightMatrix[5][3] = 5.0; // F-D: 5

                Graph graph = new Graph(weightMatrix);
                graph.DIJKSTRA(graph, Graph.start, weightMatrix);
                graph.output(graph);
        }

        /* DIJKSTRA Method */
        public void DIJKSTRA(Graph G, Node s, double[][] w) {
                // Dijkstra's algorithm based on min-priority queue
                // Input: Graph G, start node s, weight matrix w.
                INIT_SS(G, s);
                PriorityQueue<Node> Q = new PriorityQueue<Node>(G.nodes);

                while(Q.size() != 0) {
                        Node u = Q.poll();
//                      System.out.println(u + " Taken from Q!");
                        for(Node v: G.Adj[Integer.valueOf(u.getName())])
                                RELAX(u, v, w);
```

```
			}
		}

		/* INIT-SS Method */
		public void INIT_SS(Graph G, Node s) {
			// Initializes shortest
			// single source tree
			// Input: Graph G
			// source vertex s.
			for(Node u: G.nodes) {
				u.setDistance(Node.DISTANCE_INFINITY);
				u.setParent(null);
			}
			s.setDistance(0.0);
		}

		/* RELAX Method */
		public void RELAX(Node u, Node v, double[][] weightMatrix) {
			// Performs relaxation // step on edge (u, v)
			// Input: nodes u, v and // weight matrix w.
			double weightBetweenUandV = weightMatrix[Integer.valueOf(u.getName())][Integer.valueOf(v.getName())];
			if(v.getDistance() > u.getDistance() + weightBetweenUandV) {
				v.setDistance(u.getDistance() + weightBetweenUandV);
				v.setParent(u);
			}
		}

		public void output(Graph graph) {
			System.out.println("<Result of DIJKSTRA>");
			System.out.println("Edges:");
			for(Node n: graph.nodes)
				if(n != Graph.start)
					System.out.println("Edge from " + n + " to " + n.getParent() + ", weight: " + (n.getDistance()
	- n.getParent().getDistance()));
			System.out.println("Distance of each edges from the root node:");
			for(Node n: graph.nodes)
				System.out.println("Node " + n + ": " + n.getDistance() + " from the root node");
		}

		/* Sub class Node */
		static class Node implements Comparable<Node>{
			// Constant value means infinity
			public static final double DISTANCE_INFINITY = Double.MAX_VALUE;
			// Keep the name of this object
			private String name;
			// Keep the distance from the source Node
			private double distance;
			// Keep the parent Node of this object
			private Node parent;
			/* Constructor */
			public Node(String name) {
				this.name = name;
			}
			/* toString Method */
			public String toString() {
				return this.name;
			}
			/* Return the name of this Node */
			public String getName() {
				return this.name;
			}
			/* Edit the distance from the source Node */
			public void setDistance(double distance) {
				this.distance = distance;
			}
			/* Return the distance from the source Node */
			public double getDistance() {
				return this.distance;
			}
			/* Edit the parent of this Node */
			public void setParent(Node parent) {
				this.parent = parent;
			}
```

```
        /* Return the parent of this Node */
        public Node getParent() {
                return this.parent;
        }
        @Override
        public int compareTo(Node n) {
                return (int) (this.getDistance() - n.getDistance());
        }
}

/* Sub class DirectedEdge */
class DirectedEdge{
        // Keep the Origin Node
        private Node origin;
        // Keep the Destination Node
        private Node destination;
        /* Constructor */
        public DirectedEdge(Node from, Node to) {
                this.origin = from;
                this.destination = to;
        }
        /* Return the Origin Node of this Edge */
        public Node getOrigin() {
                return this.origin;
        }
        /* Return the Destination Node of this Edge */
        public Node getDestination() {
                return this.destination;
        }
}

/* Sub class WeightedDirectedEdge */
class WeightedDirectedEdge extends DirectedEdge implements Comparable<WeightedDirectedEdge>{
        // Keep the weight of this Edge
        private double weight;
        /* Constructor */
        public WeightedDirectedEdge(Node from, Node to, double weight) {
                super(from, to);
                this.weight = weight;
        }
        /* Return the value of weight */
        public double getWeight() {
                return this.weight;
        }
        /* Return the edge of this Edge is same with the Node given in argument or not */
        public boolean isSame(Node u, Node v) {
                if(u.equals(v)) return false;
                if(u.equals(this.getOrigin())) {
                        if(v.equals(this.getDestination())) return true;
                        else return false;
                }
                else if(u.equals(this.getDestination())) {
                        if(v.equals(this.getOrigin())) return true;
                        else return false;
                }
                else return false;
        }
        @Override
        public int compareTo(WeightedDirectedEdge o) {
                return (int) (this.weight - o.weight);
        }
}

/* Sub class NodeList */
class NodeList extends ArrayList<Node>{
        /* Constructor (No  arguments) */
        public NodeList() {
                super();
        }
        /* Constructor (Argument is any collection) */
        public NodeList(Collection c) {
                super(c);
        }
```

```
		/* Override the contains method to return result by searching the name */
		public boolean contains(String name) {
			Iterator<Node> it = this.iterator();
			while(it.hasNext())
				if(it.next().getName().equals(name)) return true;
			return false;
		}
		/* Return the element designated by the name in argument */
		public Node get(String name) {
			Iterator<Node> it = this.iterator();
			while(it.hasNext()) {
				Node next = it.next();
				if(next.getName().equals(name)) return next;
			}
			return null;
		}
	}

	/* Sub class EdgeList */
	class EdgeList extends ArrayList<WeightedDirectedEdge>{
		/* Override the contains method to return result by searching the name of Node */
		public boolean contains(Node u, Node v) {
			Iterator<WeightedDirectedEdge> it = this.iterator();
			while(it.hasNext())
				if(it.next().isSame(u, v)) return true;
			return false;
		}
		/* Return the element has the Node u and v as the edges of this Edge */
		public WeightedDirectedEdge get(Node u, Node v) {
			Iterator<WeightedDirectedEdge> it = this.iterator();
			while(it.hasNext()) {
				WeightedDirectedEdge next = it.next();
				if(next.isSame(u, v)) return next;
			}
			return null;
		}
		/* Return the element has the Node u as the origin of the edge */
		public NodeList getDestinationFrom(Node u) {
			NodeList result = new NodeList();
			Iterator<WeightedDirectedEdge> nit = this.iterator();
			while(nit.hasNext()) {
				WeightedDirectedEdge next = nit.next();
				if(next.getOrigin() == u) result.add(next.getDestination());
			}
			return result;
		}
	}
}
```