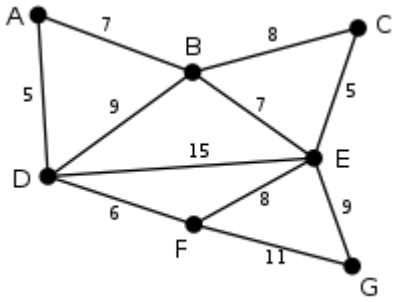
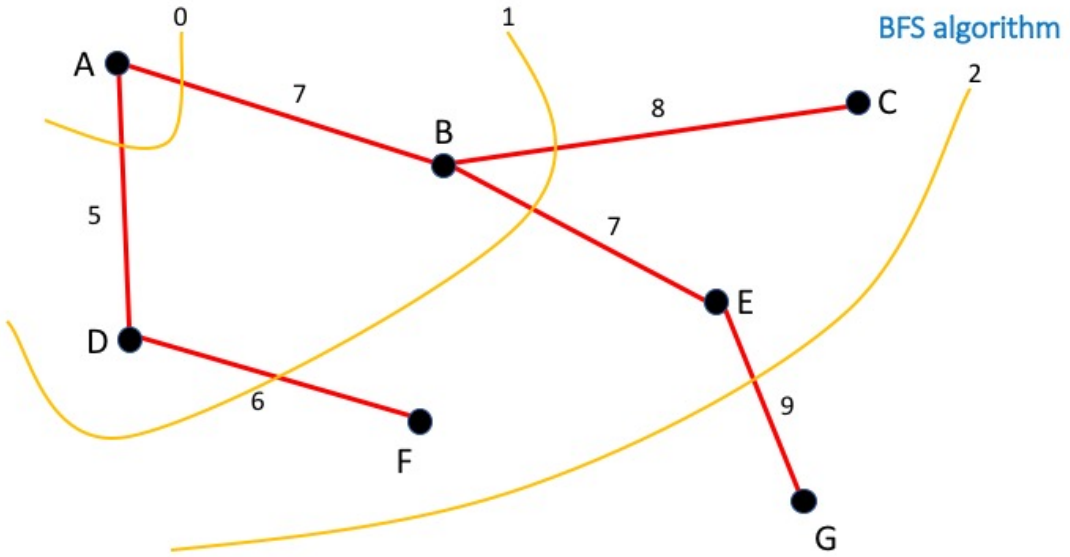
Exercise 4. Answer Sheet

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

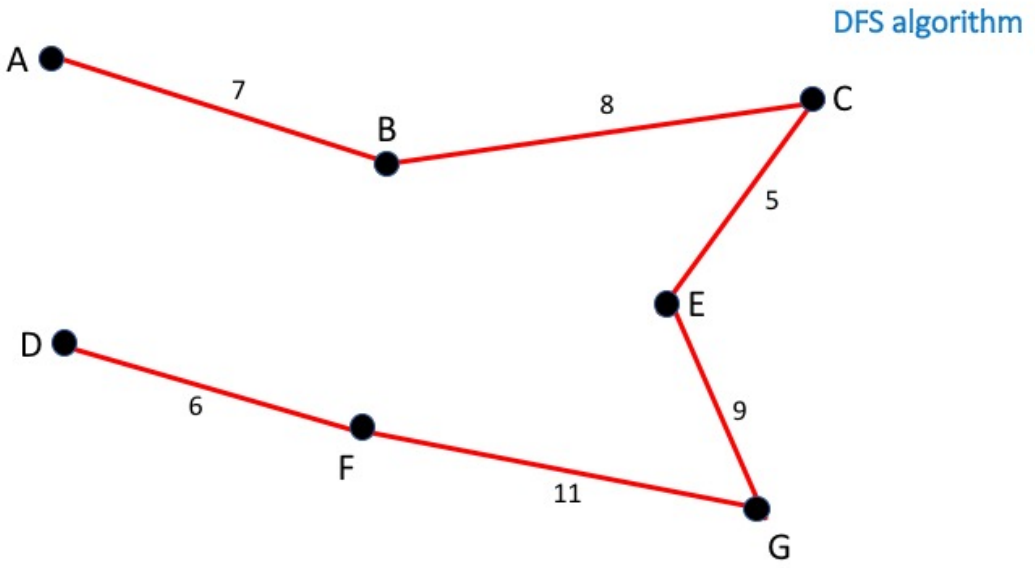
Problem 1. (*50 points*) Consider the following graph and assume node A as a root.



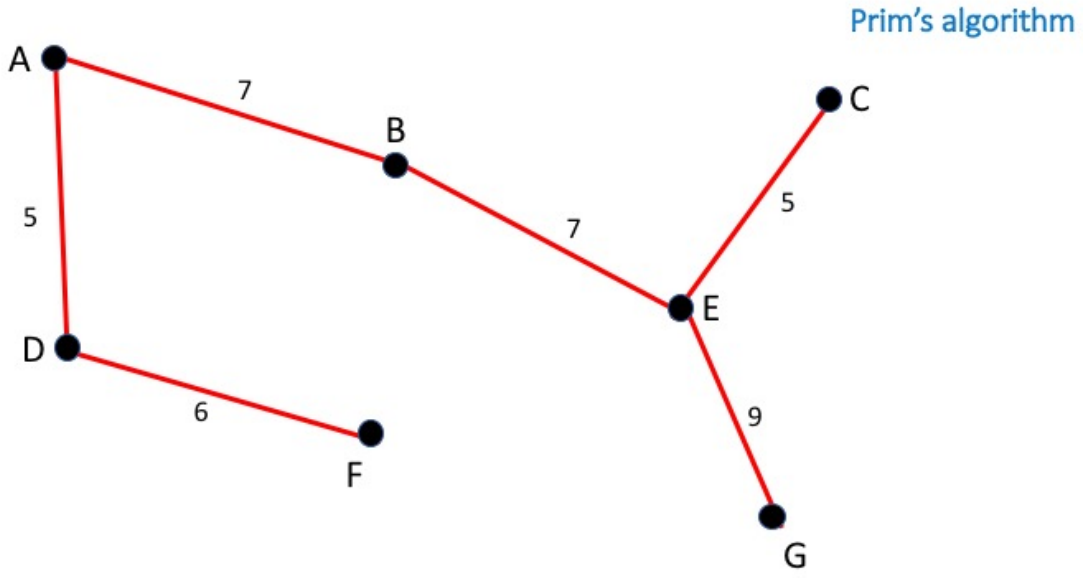
a) Draw a spanning tree obtained by using the Breadth Fist Search (BFS) algorithm.



b) Draw a spanning tree obtained by using the Depth First Search (DFS) algorithm.



c) Draw the minimum spanning tree obtained by the Prim's algorithm.



Problem 2. *(50 points)* Write a program implementing Kruskal's algorithm. Upload your source code. Show your input graph and the obtained MST in the space below.

<How to compile/run>

Command: **javac Graph.java**
java Graph

Input(Example): A B 10
B C 10

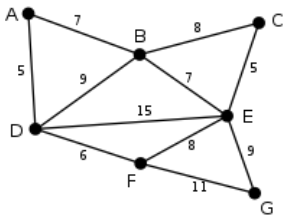
End

When you input the data of the edge of graph, please enter the elements separated by space in order of "**Node name of a side of edge**, **Node name of the other side of edge**, **The weight of edge**". And after you finished this operation, please input the string "End" to end the input operation.

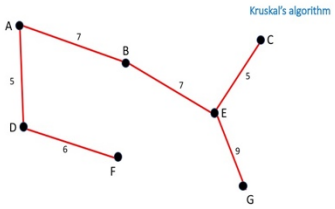
<Output result>

!!ATTENTION!! The input data in this part is just a EXAMPLE to show you how to run this program I get from the problem above and the lecture slide. Not a unique answer of this problem.

Case 1(From the problem above):



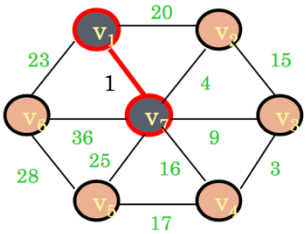

```
Input: A B 7
       B C 8
       C E 5
       B E 7
       E G 9
       F G 11
       F E 8
       A D 5
       D B 9
       D E 15
       D F 6
```



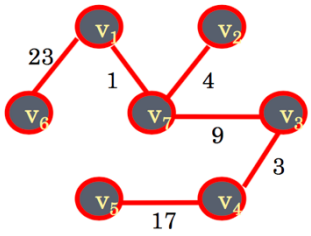
Output:

```
       B to E
       C to E
       A to D
       E to G
       A to B
       D to F
```

Case 2(From the lecture slide):




```
Input: v1 v2 20
       v2 v3 15
       v3 v4 3
       v4 v5 17
       v5 v6 28
       v6 v1 23
       v1 v7 1
       v2 v7 4
       v3 v7 9
       v4 v7 16
       v5 v7 25
       v6 v7 36
Output:
       v4 to v5
       v1 to v7
       v3 to v4
       v6 to v1
       v3 to v7
       v2 to v7
```



<Actual Interface>

!!ATTENTION!! The input data in this part is just a EXAMPLE to show you how to run this program I get from the problem above and the lecture slide. Not a unique answer of this problem.

```
std6dc28{s1240234}64: javac Graph.java
std6dc28{s1240234}65: java Graph
Please enter the information of directed edges in this graph in order of "Origin-Node Destination-Node
weight"
Example: a b 1 (It means the directed edge from node a to node b has the weight 1.)
(Enter the string "End" to finish the input operation):
A B 7
B C 8
C E 5
B E 7
E G 9
F G 11
```

```
F E 8
A D 5
D B 9
D E 15
D F 6
End
<Edges in the Minimum Spanning Tree of this graph>
Edge from the Node B to the Node E with weight 7.0
Edge from the Node C to the Node E with weight 5.0
Edge from the Node A to the Node D with weight 5.0
Edge from the Node E to the Node G with weight 9.0
Edge from the Node A to the Node B with weight 7.0
Edge from the Node D to the Node F with weight 6.0
std6dc28{s1240234}66: java Graph
Please enter the information of directed edges in this graph in order of "Origin-Node Destination-Node
weight"
Example: a b 1 (It means the directed edge from node a to node b has the weight 1.)
(Enter the string "End" to finish the input operation):
v1 v2 20
v2 v3 15
v3 v4 3
v4 v5 17
v5 v6 28
v6 v1 23
v1 v7 1
v2 v7 4
v3 v7 9
v4 v7 16
v5 v7 25
v6 v7 36
End
<Edges in the Minimum Spanning Tree of this graph>
Edge from the Node v4 to the Node v5 with weight 17.0
Edge from the Node v1 to the Node v7 with weight 1.0
Edge from the Node v3 to the Node v4 with weight 3.0
Edge from the Node v6 to the Node v1 with weight 23.0
Edge from the Node v3 to the Node v7 with weight 9.0
Edge from the Node v2 to the Node v7 with weight 4.0
```

<Source Code>

Graph.java

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.Comparator;
import java.util.HashSet;
import java.util.Iterator;
import java.util.List;
import java.util.stream.Collectors;

public class Graph {

        // List of Nodes
        private NodeList nodes = new NodeList();
        // ArrayList of WeightedDirectedEdge
        private ArrayList<WeightedDirectedEdge> edges = new ArrayList<WeightedDirectedEdge>();

        /* Constructor */
        public Graph() {
                InputModule inMod = new InputModule(this.nodes, this.edges);
        }

        /* Returns the MinimumSpanningTree */
        public ArrayList<WeightedDirectedEdge> getMST(){
                KruskalsAlgorithm ka = new KruskalsAlgorithm();
                ArrayList<WeightedDirectedEdge> result = new ArrayList<WeightedDirectedEdge>();
                result.addAll(ka.MST_KRUSKAL(this.edges, this.nodes));
                return result;
        }

        /* Output the given ArrayList */
        public void outList(ArrayList<WeightedDirectedEdge> argEdge) {
                OutputModule outMod = new OutputModule(argEdge);
                outMod.output();
        }

        /* Main Method */
```

```
	public static void main(String[] args) {
		Graph graph = new Graph();
		System.out.println("<Edges in the Minimum Spanning Tree of this graph>");
		graph.outList(graph.getMST());
	}

	/* Sub class KruskalsAlgorithm */
	class KruskalsAlgorithm{
		// HashSet A for making MST
		private HashSet<WeightedDirectedEdge> A;
		// HashSet vertices for storing the set of set of nodes
		private HashSet<HashSet<Node>> setOfSetOfNode;
		/* Constructor */
		public KruskalsAlgorithm() {

		}
		/* MST-KRUSKAL Method */
		public HashSet<WeightedDirectedEdge> MST_KRUSKAL(ArrayList<WeightedDirectedEdge> edges, NodeList
nodes) {
			// Initialize the Set A
			this.A = new HashSet<WeightedDirectedEdge>();
			// Initialize the Set setOfSetOfNode
			this.setOfSetOfNode = new HashSet<HashSet<Node>>();
			// Make set of vertices and store it to setOfSetOfNode
			for(int i = 0; i < nodes.size(); i++)
this.setOfSetOfNode.add(this.MAKE_SET(nodes.get(i)));
			// Sort edges into non-decreasing order by weight
			List<WeightedDirectedEdge> sortedEdges =
edges.stream().sorted(Comparator.naturalOrder()).collect(Collectors.toList());
			Iterator<WeightedDirectedEdge> eit = sortedEdges.iterator();
			while(eit.hasNext()) {
				WeightedDirectedEdge nextEdge = eit.next();
				if(!(this.FIND_SET(nextEdge.getOrigin()) ==
this.FIND_SET(nextEdge.getDestination()))) {
					A.add(nextEdge);
					this.UNION(nextEdge.getOrigin(), nextEdge.getDestination());
				}
			}
			return A;
		}
		/* UNION Method */
		public void UNION(Node u, Node v) {
			HashSet<Node> uSet = this.FIND_SET(u);
			HashSet<Node> vSet = this.FIND_SET(v);
			uSet.addAll(vSet);
			vSet.clear();
			this.setOfSetOfNode.remove(vSet);
		}
		/* FIND-SET Method */
		public HashSet<Node> FIND_SET(Node vertice){
			Iterator<HashSet<Node>> nit = this.setOfSetOfNode.iterator();
			while(nit.hasNext()) {
				HashSet<Node> nextSet = nit.next();
				if(nextSet.contains(vertice)) return nextSet;
			}
			System.err.println("Set of Node " + vertice.getName() + "isn't generated!!");
			return null;
		}
		/* MAKE-SET Method */
		public HashSet<Node> MAKE_SET(Node v){
			HashSet<Node> result = new HashSet<Node>();
			result.add(v);
			return result;
		}
	}

	/* 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;
		}
		@Override
		public int compareTo(WeightedDirectedEdge o) {
			return (int) (this.weight - o.weight);
		}
	}

	/* Sub class Node */
	class Node{
		// Keep the name of this object
		private String name;
		/* Constructor */
		public Node(String name) {
			this.name = name;
		}
		/* Return the name of this Node */
		public String getName() {
			return this.name;
		}
	}

	/* Sub class NodeList */
	class NodeList extends ArrayList<Node>{
		/* 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 InputModule */
	class InputModule{
		// Keep the object of given ArrayList
		private NodeList nodes;
		private ArrayList<WeightedDirectedEdge> edges;
		private final String endWord = "End";

		/* Constructor */
		public InputModule(NodeList nodes, ArrayList<WeightedDirectedEdge> edges) {
			this.nodes = nodes;
			this.edges = edges;
			this.getInputOfEdge();
		}

		/* Operate the ArrayList of Node and Edge given in the constructor */
		public void getInputOfEdge(){
			// Prepare BufferedReader
			BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
			// Output the message
			System.out.println("Please enter the information of directed edges in this graph in order
of \"Origin-Node Destination-Node weight\"");
			System.out.println("Example: a b 1 (It means the directed edge from node a to node b has
the weight 1.)");
			System.out.println("(Enter the string \"" + this.endWord + "\" to finish the input
operation):");
			while(true) {
				// Declare the variable for storing the inputed line temporary.
				String loadLine;
				// Read a line.
				try { loadLine = br.readLine();} catch (IOException e)
{e.printStackTrace();break;}
				// Store it in the array of String.
```

```
                                String[] lineElement = loadLine.split(" ", 0);
                                // If some element is match to the word "End", finish the input operation.
                                if(lineElement[0].equals(this.endWord)) break;
                                if(lineElement[1].equals(this.endWord)) break;
                                if(lineElement[2].equals(this.endWord)) break;
                                // If the nodeList doesn't contain the particular node, add it.
                                if(!this.nodes.contains(lineElement[0])) this.nodes.add(new
Node(lineElement[0]));
                                if(!this.nodes.contains(lineElement[1])) this.nodes.add(new
Node(lineElement[1]));
                                // Generate new Edge and store it in the ArrayList
                                this.edges.add(new WeightedDirectedEdge(this.nodes.get(lineElement[0]),
this.nodes.get(lineElement[1]), Double.valueOf(lineElement[2])));
                        }
                }
        }

        /* Sub class OutputModule */
        class OutputModule{
                // Keep the given ArrayList of Edge
                private ArrayList<WeightedDirectedEdge> outEdges;
                /* Constructor */
                public OutputModule(ArrayList<WeightedDirectedEdge> edges) {
                        this.outEdges = edges;
                }
                /* Output the content of ArrayList of edges */
                public void output() {
                        Iterator<WeightedDirectedEdge> eit = this.outEdges.iterator();
                        while(eit.hasNext()) {
                                WeightedDirectedEdge wde = eit.next();
                                System.out.println("Directed edge from the Node " + wde.getOrigin().getName() + "
to the Node " + wde.getDestination().getName() + " with weight " + wde.getWeight());
                        }
                }
        }

}
```