Name: _____ Andrew Id: _____

**15-121 Fall 2024 Assessment 6**

Up to 50 minutes. No calculators, no notes, no books, no computers. Show your work!

1. **Searching and Sorting**

   Consider the following list of integers:

   27, 17, 3, 40, 18, 29, 32, 50, 16

   (a) (3 points) Assuming that the array provided at the beginning of this problem receives *two passes* from the loop in the Selection Sort algorithm (assume it is the first and second pass, with `i = 0` and then `i = 1`), what will be the new state of the array? Write your answer in the provided boxes below. Show your work. Answers without appropriate work will receive no credit.

   **Final Answer for Part a:**

   | | | | | | | | | |
   |---|---|---|---|---|---|---|---|---|
   | | | | | | | | | |

   (b) (4 points) Assuming that the array provided at the beginning of this problem receives a single pass from the loop in the Bubble Sort algorithm, what will be the new state of the array? Write your answer in the provided boxes below. Show your work. Answers without appropriate work will receive no credit.

   **Final Answer for Part b:**

   | | | | | | | | | |
   |---|---|---|---|---|---|---|---|---|
   | | | | | | | | | |

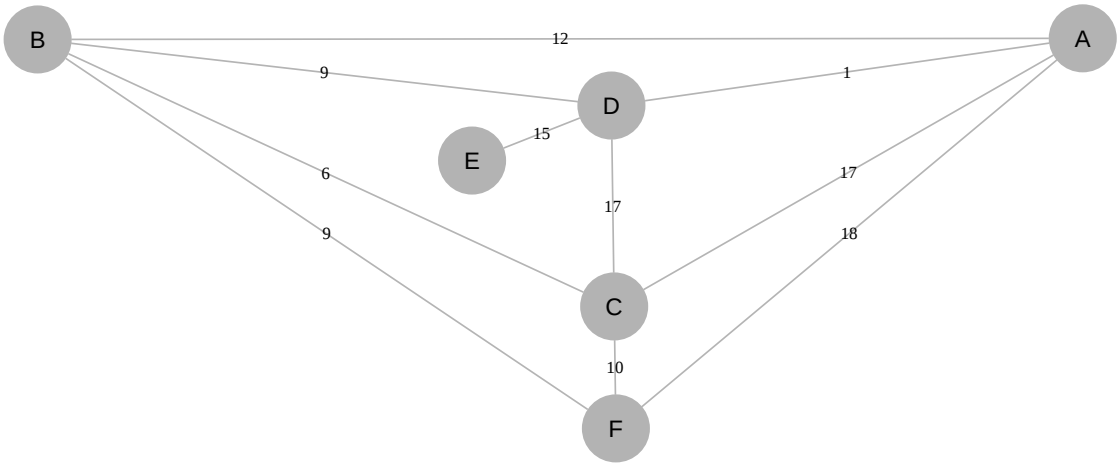
   (c) (4 points) Assuming that the array provided at the beginning of this problem is partitioned using the Quick Sort partition algorithm (use 27 as your pivot) as presented in class, what will be the new state of the array? Write your answer in the boxes provided below. Show your work. Answers without appropriate work will not receive credit.

   **Final Answer for Part c:**

   | | | | | | | | | |
   |---|---|---|---|---|---|---|---|---|
   | | | | | | | | | |

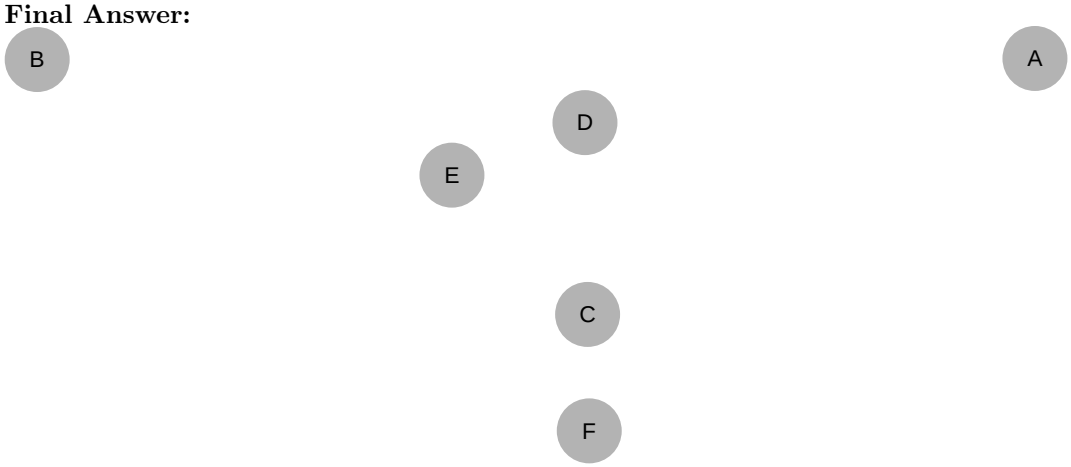
2. **Minimum Spanning Trees**

Consider the following weighted graph:



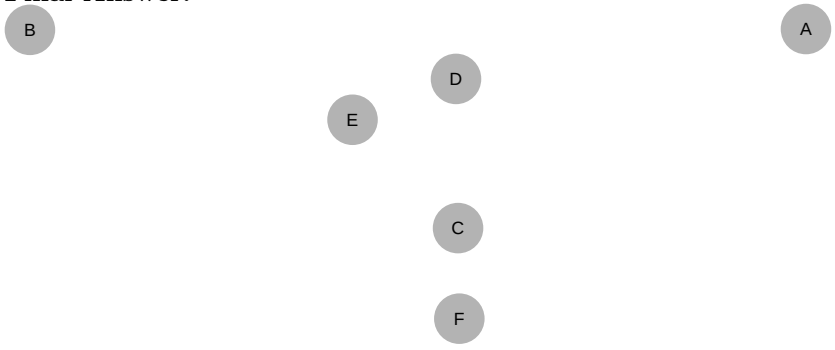

(a) (5 points) Find a minimum spanning tree for this graph using Kruskal's algorithm. Be sure to show your work in such a way that it is clear you understand how the algorithm works (for example, by explictly stating the order in which edges are added to the MST). Answers without appropriate work will receive no credit. Draw your final MST onto the provided graph below.

**Final Answer:**



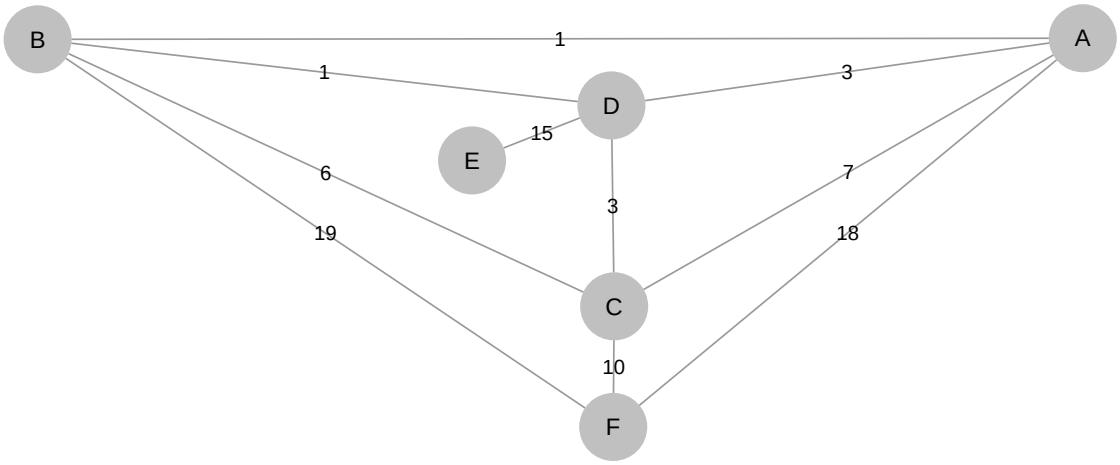

(b) (5 points) Find a minimum spanning tree for this graph using Prims's algorithm starting from E. Be sure to show your work in such a way that it is clear you understand how the algorithm works (for example, by explictly stating the order in which edges are added to the MST). Answers without appropriate work will receive no credit. Draw your final MST onto the provided graph below.

**Final Answer:**



3. (10 points) **Dijkstra**

Using Dijkstra's algorithm, find the shortest paths from F to every node in the following graph. Be sure to show your work as you run the algorithm. Answers without appropriate work will not receive credit.




Write your final answer in the table below. Do *not* write your unvisited / working table in the table below.

**Final Answer:**

| Node | Cost | Previous |
|---|---|---|
| F | 0 | |
| | | |
| | | |
| | | |
| | | |
| | | |

4. (9 points) Graph Code

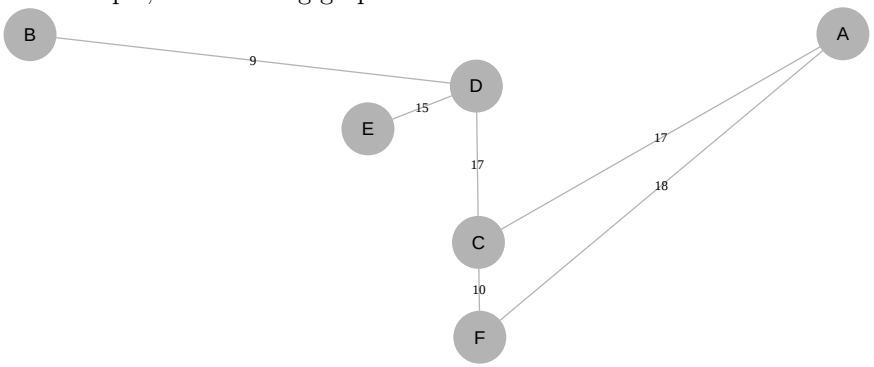
Recall the following code, slightly simplified from what we wrote in class, for implementing a graph:

```
public class Edge {
    private Vertex src;
    private Vertex dst;
    private int weight;

    public Edge(Vertex src, Vertex dst, int weight) {
        this.src = src;
        this.dst = dst;
        this.weight = weight;
    }

    public Vertex getSrc() {
        return this.src;
    }

    public Vertex getDst() {
        return this.dst;
    }

    public int getWeight() {
        return this.weight;
    }
}

public class Vertex {
    private String name;
    private ArrayList<Edge> edges = new ArrayList<Edge>();

    public Vertex(String name) {
        this.name = name;
    }

    public void addEdge(Vertex dst, int weight) {
        Edge tmp = new Edge(this, dst, weight);
        edges.add(tmp);
    }

    public String getName() {
        return this.name;
    }

    public ArrayList<Edge> getEdges() {
        return edges;
    }

    @Override
    public int hashCode() {
        return Objects.hash(edges, name);
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Vertex other = (Vertex) obj;
        return Objects.equals(edges, other.edges) && Objects.equals(name, other.name);
    }
}
```

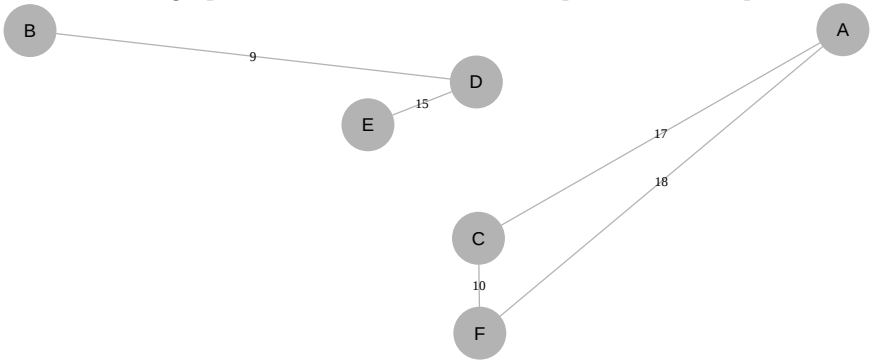
```
public class Graph {
    private HashMap<String, Vertex> vertices = new HashMap<String, Vertex>();

    public Graph(String filename) {
        FileReader fr;
        try {
            fr = new FileReader(filename);
        } catch (FileNotFoundException fne) {
            System.err.println("File not found!");
            return;
        }

        Scanner inp = new Scanner(fr);
        while (inp.hasNextLine()) {
            String line = inp.nextLine();
            String[] vals = line.split(",");
            if (vals.length != 3) {
                System.err.println("Invalid file format");
                System.err.print(line);
                return;
            }
            String src = vals[0];
            String dst = vals[1];
            int w = Integer.parseInt(vals[2]);

            addVertex(src);
            addVertex(dst);
            addEdge(src, dst, w);
            addEdge(dst, src, w);
        }
        inp.close();
    }

    public void addVertex(String vertexName) {
        if (!vertices.containsKey(vertexName)) {
            Vertex tmp = new Vertex(vertexName);
            vertices.put(vertexName, tmp);
        }
    }

    public void addEdge(String src, String dst, int weight) {
        Vertex srcV = vertices.get(src);
        Vertex dstV = vertices.get(dst);

        srcV.addEdge(dstV, weight);
    }

    /**
     * Determine if this graph is connected, meaning that there is a path from every
     * node to every other node.
     *
     * @param start The node to start the search at
     * @return true if the graph is connected and false otherwise
     */
    public boolean isConnected(String start) {
        // You will write this code
    }

    public static void main(String[] args) {
        Graph g = new Graph("graph.txt");
        System.out.println(g.isConnected("A"));
    }
}
```

We say that a graph is connected if there is a path from every node to every other node. (This does not mean that there is an edge between every pair of nodes, it means that there is some path between every pair of nodes.)

For example, the following graph is connected:




However, this graph is not connected. For example, there is no path from B to A:




An algorithm to determine whether or not a graph is connected is as follows:

1. Create a *visited* list to store which nodes you have already visited.
2. Create a *stack* to store the nodes you have seen, but haven't visited yet.
3. Choose a starting node and add it to the *stack* and the *visited* list.
4. As long as there are nodes on the stack...
   (a) Remove the node from the top of the stack
   (b) For every node that this node has an edge to, if it is not on the visited list then add it to *stack* and *visited*.
5. Ensure that you have visited every node from the original graph.

You should think carefully about what this algorithm is doing and why it works.

Write the following method in the `Graph` class. For full credit, your solution should be better than $O(N^2)$ where N is the number of vertices in the graph.

```
/**
 * Determine if this graph is connected, meaning that there is a path from every
 * node to every other node.
 *
 * @param start The node to start the search at
 * @return true if the graph is connected and false otherwise
 */
public boolean isConnected(String start) {
```