**Problem 1**

The traversal steps will be:

1. G -> E -> B -> A



1. Back track to B
2. B -> C -> F



1. Back track to F->C -> B -> E
2. E -> D



All nodes are visited now, the sequence of nodes visited is:

G -> E -> B -> A -> B -> C -> F -> C -> B -> E -> D

The edge classification is show below (T- Tree, B-Back, F-Forward, C-Cross)



**Problem 2**

1. Explore node one edge away from I



1. Explore node two edge away from I



1. Explore nodes 3 edges away from I



1. Explore nodes 4 edges away from I



The sequence of nodes visited is: I,F,H,E,G,B,C,D,A

**Problem 3**

**Part 1**

1. Initial graph, all nodes have D value as ∞, except for S



1. S comes into C, edges (S,a), (S,b) and (S,c) get relaxed



1. b comes into C as it has the min D, edges (b,d), (b,a) and (b,c) are relaxed



1. a comes into C as it has min D, no edges to relax



1. d comes into C as it has min D, no edge relaxation



1. c comes into C, since it is the min D, relax edge (c,e)



1. Finally, the last remaining node e comes into the cloud



**Part 2**

The shortest path from S to every node:

1. b : S -> b
2. c : S -> b -> c
3. d : S -> b -> d
4. a : S -> b -> a
5. e : S -> b -> c -> e

**Problem 4**

**Part 1**

1. Initial graph



1. The node a is pulled into the cloud, the minimum weight edge is (a,c)



1. The node c gets pulled into the cloud and the edge (c,b) has the minimum weight



1. The node b gets pulled into the cloud and the minimum weight edge (b,d) is explored



1. The node d gets pulled into the cloud and the edge (d,g) is explored



1. The node g is pulled into the cloud and the min weight edge node (d,f) is explored



1. The node f is pulled into the cloud and the min weight edge (f,e) is explored



1. The final node e is pulled into the cloud



**Part 2**

The minimum spanning tree generated is show below the set is:

((a,c),(c,b),(b,d),(d,g),(d,f),(f,e))



**Problem 5**

*Note: Please change the value of the variable FRIEND\_INPUT in the class Hw6\_P5 to the file location of the friend input file before running the code*

**Data structures used**

The code uses a few data structures to achieve its functionality:

1. HashMap<String,HashSet<String>> friendMap

This map read the input friend list and keeps a mapping between friends and is used to populate the adjacency matrix.

1. HashSet<String> uniquePeople

This hashset is used to hold a unique list of people this is later converted to the people label array.

1. ArrayList<String> peopleLable

This is an array of alphabetically sorted names of the people, this is created from the uniquePeople set and is used for printing the adjacency matrix and also in traversing it.

1. int[][] adjacencyMatrix

This 2d array holds the friend relationships between people.

1. HashMap<String,Integer> personPos

This holds the mapping between a name and the array index in the peopleLable arraylist

**Learnings**

The adjacency matrix serves as an easy and intuitive way to represent a graph, here the row and column indexes represent the nodes while the value at the intersection of the nodes represents the edge weight.

The advantage of using an adjacency matrix other than it being intuitive is it has a O(1) time complexity for checking relationship between nodes, adding relationships and updating them.

However, the disadvantage would be that it has a space complexity of O(n2) where n is the number of nodes, thus a sparse graph with many nodes but very few edges between them will take up the same space as a dense graph. Another disadvantage would be if new nodes need to be added we will have to create a new larger adjacency matrix and do a copy from the older one to the new one, so in scenarios where your graph is continuously expanding like social network graphs where new users are continuously getting added this datastructure may not be an efficient data structure to hold the information.