Problem 1

Description: The given problem is to try to find the minimum spanning tree and in addition to that we are given a set of points F that are to be part of the solution. To solve this problem, I used the Kruskal's algorithm to get the minimum spanning tree, while making sure not to create any cycles and choosing the nodes with the least weight and keeping track of the elements in set F that are to be part of the solution.

PseudoCode:

```
1. runKrusKal(Graph g)
    a. mergeSort( g.edgeList, 0, g.numEd - 1)
    b. UnionFind unionFind = new UnionFind( g.numEd - 1, g.numVert )
    c. for every edge in g:
        i. if edge.inSetF:  // is part of set F
            1. if  unionFind.get(edge.u) is same as unionFind.get(edge.v):
                a. print(-1)
        ii. unionFind.union(edge.u, edge.v)
        iii. unionFind.sum += edge.weight
    d. for every edge e in edgeList:
        i. if (!e.inSetF) :
            1. if unionFind.get(e.u) not equal to unionFind.get(e.v):
                a. unionFind.union(e.u, e.v)
                b. unionFind.sum += e.weight
    e. print( unionFind.sum )
```

Run time analysis:

- Run time for mergeSort is O (n log n)
- for loop at line c is of O(n) where n is the maximum number of edges in the given graph.
  - The unionFind.get() is O(n)
  - unionFind.union() is O(n)
- for loop at line d. is of O(n) where n is the number of edges.
  - The unionFind.get() is O(n)
  - unionFind.union() is O(n)

Total runtime is given by O(n log n) + O(2 n^2) + O(2 n^2)

which is: **O(2 n^2)**

Problem 2

Description: The given problem was to figure out whether there exists a negative cycle in a directed graph. This problem is solved using the bellmanFord's algorithm to detect any negative cycles in the graph. The graph is represented as an adjacency list using the Graph class and the algorithm is run for all the vertices until we detect any negative cycle.

Pseudocode:

```
1. static boolean isNegCycleBellmanFord(Graph graph, int src) :
     a. int V = graph.V // number of vertices of the graph
     b. int E = graph.E    // number of edges of the graph
     c. int[] dist = new int[V]
     d. for all the vertices in the graph initialize the distance to maxValue of integer
     e. dist[src] = 0
     f. for  i=1 to V-1:
          i. for j=0 to E:
               1. int u = graph.edge[j].src
               2. int v = graph.edge[j].dest
               3. int weight = graph.edge[j].weight
               4. if dist[u] is not max value and dist[u]+weight<dist[v]:
                    a. dist[v] = dist[u] + weight;
     g. for i = 0 to E:
          i. int u = graph.edge[i].src
         ii. int v = graph.edge[i].dest
        iii. int weight = graph.edge[i].weight
         iv. if (dist[u] is not maxValue and dist[u] + weight < dist[v]:
               1. return true
     h. return false
```

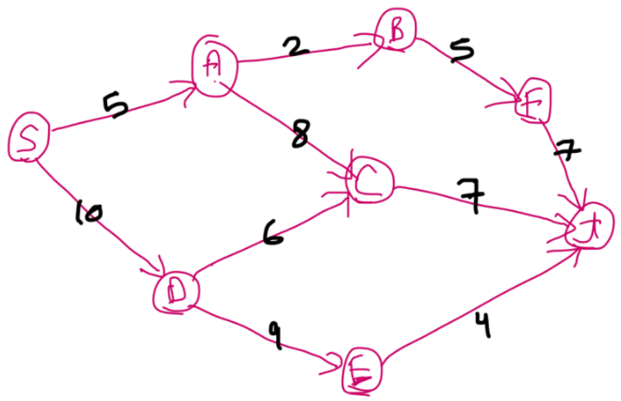
Run time analysis:

- Initializing the distance array is O(n)
- For loop at line f is O(n) where n is the number of vertices
  - Another for loop at line f(i) is O(n) where n is the maximum number of edges there can be.
- For loop at line g is O(n) where n is the edges.

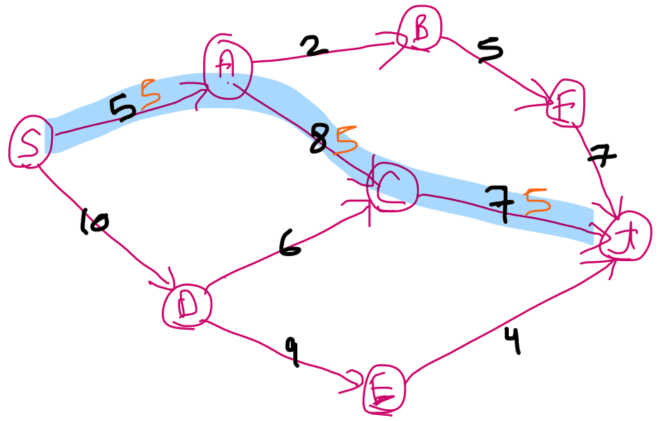
Total time complexity is given by O(n^2)

Problem 3

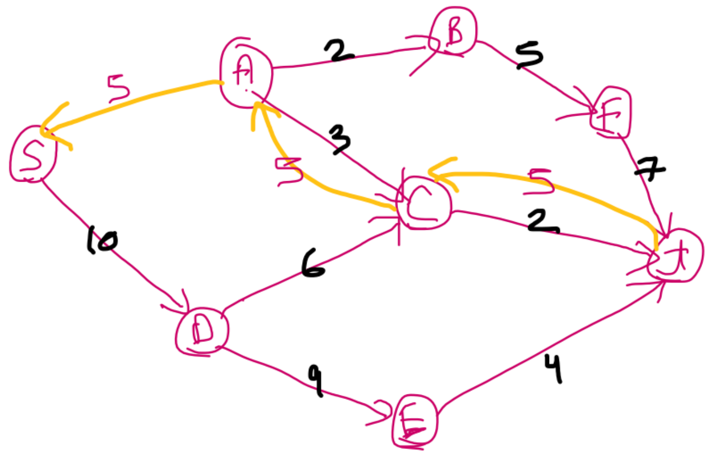
Initial graph:



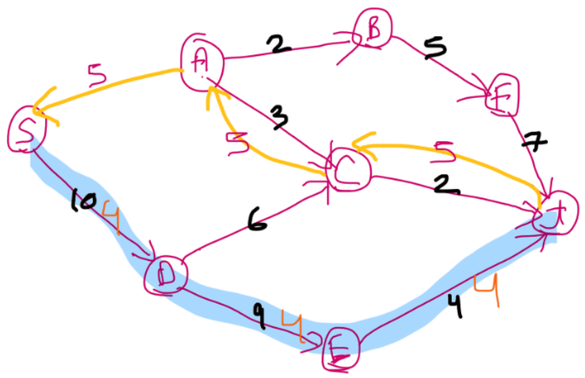
After running BFS for the first iteration we choose the path, s-A-C-t, and we push 5 units.



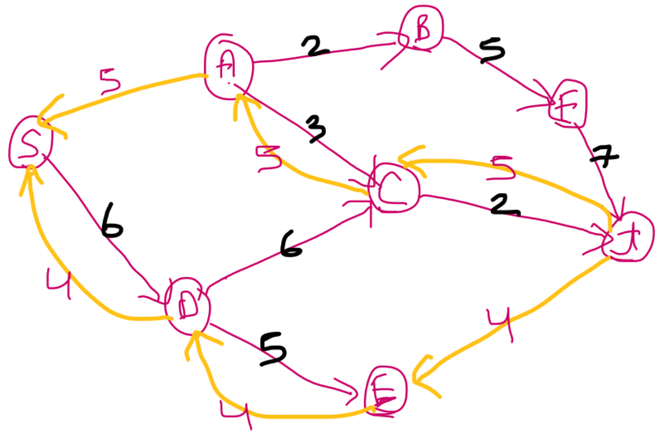
Residual graph is represented by the pink lines, and the backward edges are represented by the yellow lines with arrows pointing backwards.



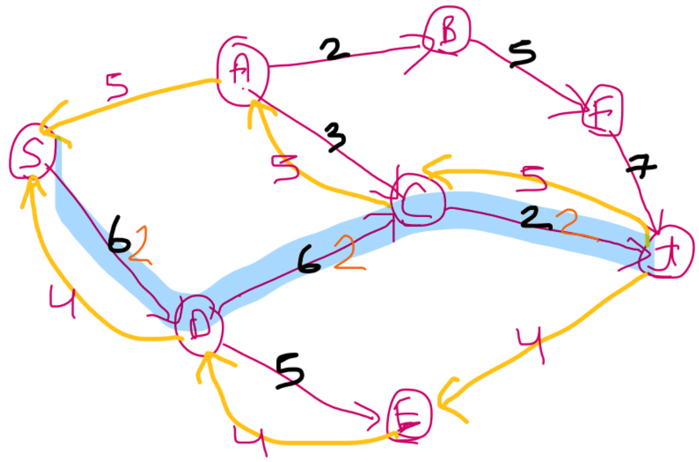
For the next iteration, we choose the path s-D-E-t, and we push 4 units.



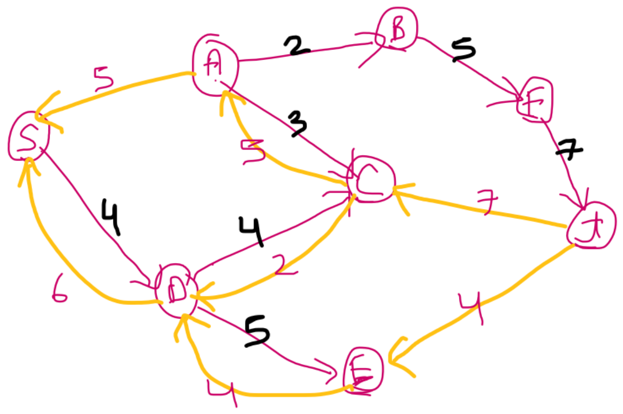
The residual graph is again represented by the pink lines, and we have backward edges represented by yellow lines from t to s.



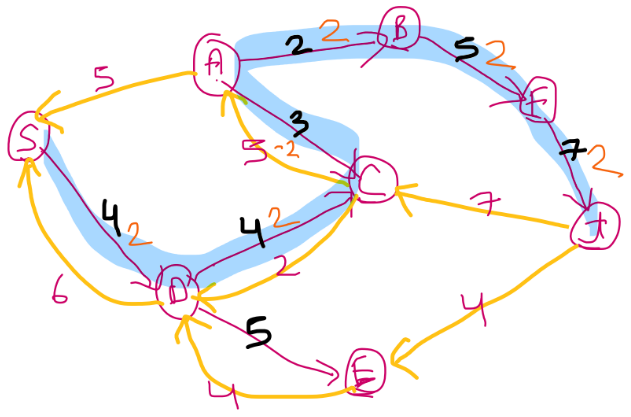
For the next iteration, we ran BFS and took the path from s-D-C-t, which is the current available shortest path, and we push 2 units.



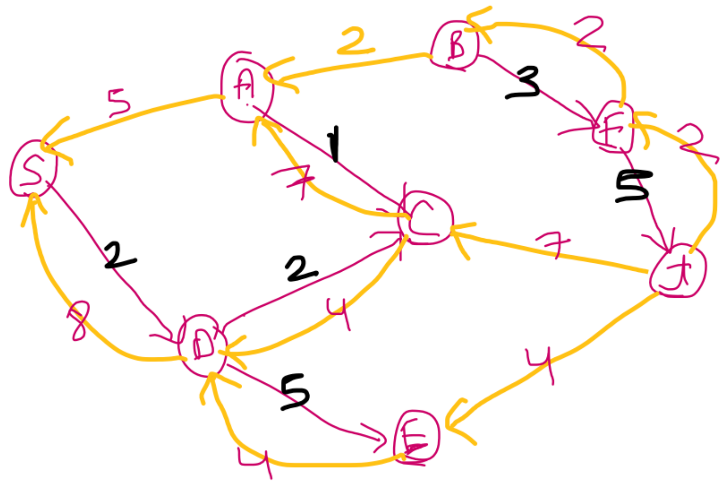
Again, the residual graph is represented by pink lines, and the backward edges are represented by yellow lines from t-s.



The next augmented path available includes a backward edge, which is s-D-C-A-B-F-t, and we push 2 units.



The residual graph is represented by the pink lines and as you can see there is no path remaining from s-t, and there are remaining backward edges, that represent the flow is complete and we cannot push more.



Problem 4

Problem P – Hamiltonian path: Given is an unweighted undirected graph G and two vertices s and t. Does there exist a path from s to t that goes through every vertex exactly once?

Problem Q1 – Longest path: Given is an unweighted undirected graph G and two vertices s and t. Find the length of the longest path from s to t. The path can go through every vertex at most once.

Problem Q2 – Shortest path with negative weights: Given is a weighted undirected graph G with arbitrary weights (positive, negative, or zeros) and two vertices s and t. Find the length of the shortest path from s to t. The path can go through every vertex at most once.

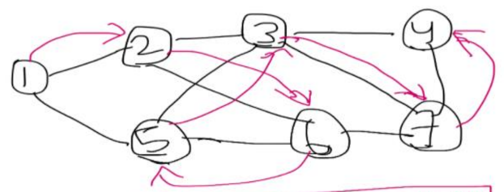
a. Reduce Problem P to Q1

Given a graph **G(n,e)**, where **n** is the number of vertices, and **e** is the number of edges, we have a starting point **s**, and an ending point **t,** we have to show that if there exists a Hamiltonian path or not. Hamiltonian path, is a path that starts from vertex **s** and ends at vertex **t** and covers all the vertices **n.** We are given another problem Q1, in which we have to find the longest path from a starting point s to an ending point t in the graph, in which every vertex can be visited once.

To reduce the problem P to Q1, we can use the same graph as the input and the same starting and ending point as the input i.e. G1=G, s1=s, t1=t.

Assuming that we can compute the longest path of the graph G1, and we get the length l as the output, we can say that there exists a Hamiltonian path from s to t, in graph G, if the length l is same as n-1, i.e. the length of the longest path should be equal to the number of vertices-1 in the graph G.

Following is an example in which we have a graph G, which is same as G1, and if we find the length of the longest path from s1=1, to t1= 4, we get a length of 6 everytime. Here n=7, and length l= n-1, hence there exists a Hamiltonian path in the given graph.



$\boxed{\text{Hpath} = 1 \to 2 \to 6 \to 5 \to 3 \to 7 \to 4}$

which is also the longest path of

$\boxed{\text{length} = 6}$

b. Reduce problem P to Q2

For this problem, we are given Q2 which is solvable by using AQ2. In Q2, we have a graph G2, and a starting point s2, and an ending point t2, and we can calculate the shortest path from s2-t2. G2 is a weighted undirected graph, that can have +ve, -ve or 0 weights.

We are given problem P, in which we have a graph G(n,e) where n is the number of vertices and e is the number of edges, and also we have a starting point s, an ending point t.

To check whether there exists a Hamiltonian path, we can run the following pseudocode, which is in polynomial time:

```
1. For every edge (u,v) in G:
    a. We given the edge a weight of -1
    b. We calculate the shortest path, of graph G2, which has the same starting point s2,
       and ending point t2 using the AQ2 algorithm.
    c. If the shortest path is –(n-1)
        i. Then there exists a Hamiltonian path in graph G
```

If there are no paths of length –(n-1) then there is no Hamiltonian path in the graph G.