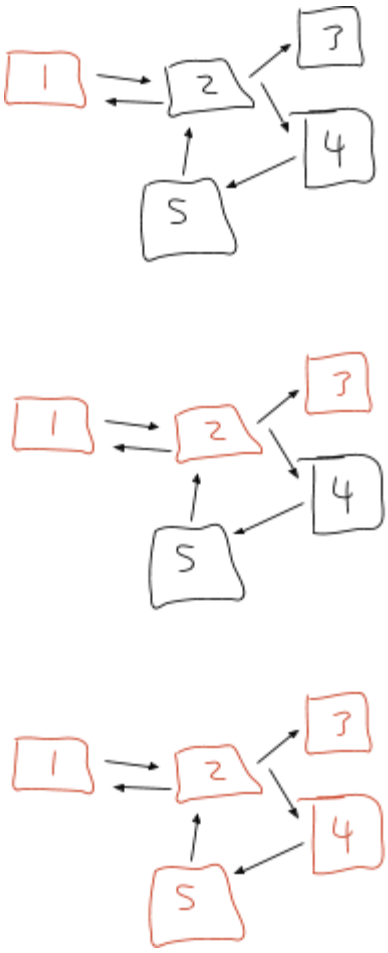
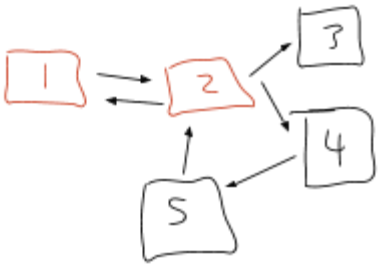
Problem 1

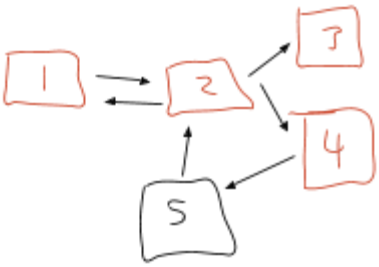
1. **Give pseudo code for an algorithm that takes as input [a directed, weighted graph, a source node s and a destination node d] and returns the average of the lengths of all of the paths leading from s to d.**

```
AVERAGE_DISTANCE(g, s, d)
v = array(0...g.length)
for i = 0...g.length
  v[i] = false

l = list()

DFS(g, s, d, v, l, 0)

return GET_AVERAGE(l)

DFS(g, s, d, v, l, c)
v[s] = true

if s == d
  l.add(c)
  v[s] = false
  return

for i...g.length
  if g[s][d] != 0
    if v[i] == false
      DFS(g, i, d, v, l, c + 1)

v[s] = false

GET_AVERAGE(l)
su = 0
ss = l.size

for i = 0...ss
  su = su + l[i]

a = su / ss

return a
```

2. **Give an illustration of your algorithm**







In DFS, I go deep down a path until I reach a "dead-end", then I retrace back only as far as to find another valid path to go down.

In my algorithm, whenever I reach the destination I was looking for, I snapshot how many edges I passed to get there, to be later aggregated then used to find the average.

3. **Do a time complexity analysis (big oh) of your algorithm.**

   O(N!). The time complexity here will be the same time complexity as counting the number of possible paths, which is shown to be O(N!) here: Count all possible paths between two vertices.

4. **Describe the choice(s) that you have made to make your algorithm run faster.**

   DFS makes the most sense for this algorithm.
   I keep track of the amount of edges we are processing in the recursion with the "c + 1". When the recursion has found, if it exists, the destination, then I store the amount of edges it took to get there in a dynamic list. You cannot do averages along the way, you must first have all the data items. Once I found all the lengths, I simply summed them up, then used that number to divide by the amount of paths found. This returns the average.

5. **Write Java code that implements your algorithm. The code should be added to the base that we have in the sample key for Asgn3.**

   See included files.

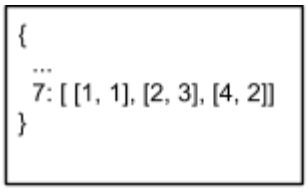
Problem 2

1. **Consider the hard problem of determining all the shortest round trips from a source node to every other node in a directed, weighted graph. Give pseudo code for an algorithm that takes as input [a directed, weighted graph and a source node s] and prints all the shortest round trips from the source node to every other node in a graph. Your algorithm must first pre-process the graph by removing the edges with highest weight (e.g, if the edge weights range from 1 to 7, then all the edges with a weight of 7 must be first removed from the graph before the round trips are computed). Assume that a round trip will still exist from the source to every other node after these edges are removed.**

```
SHORTEST_ROUNDTRIP_PRE_PROCESSED(g, s)
p = array(0...g.length)(0...g.length)

h = 0
l = {}

for i = 0...g.length
  for j = 0...g.length
    w = g[i][j]

    p[i][j] = w

    if w >= h
      h = w
      l[h].add([i,j])

for i = 0...l[h].length
  pa = l[h]
  p[pa[0]][pa[1]] = 0

for j = 1...p.length
  SHORTEST_ROUND_TRIP(p, s, j)

SHORTEST_ROUND_TRIP(G, s, d)
DIJKSTRA(G, s, d)
```

```
DIJKSTRA(G, d, s)

DIJKSTRA(G, S, D)
   for each vertex V in G
      distance[V] <- infinite
      parents[V] <- -1
      visited[V] <- false

   distance[S] <- 0
   parents[S] <- -1

   for i in 0...G.length
     u = FIND_NEXT_LEAST_WEIGHT(distance, visited)

     visited[u] = true

     for v in 0...G.length
       p <- distance[u] + G[u][v]
       if !visited[v] and p < distance[v]
         distance[v] = p
         parents[v] = u

    PRINT_SOLUTION(distance, visited)

PRINT_SOLUTION(distance, visited)
// Not provided here
```

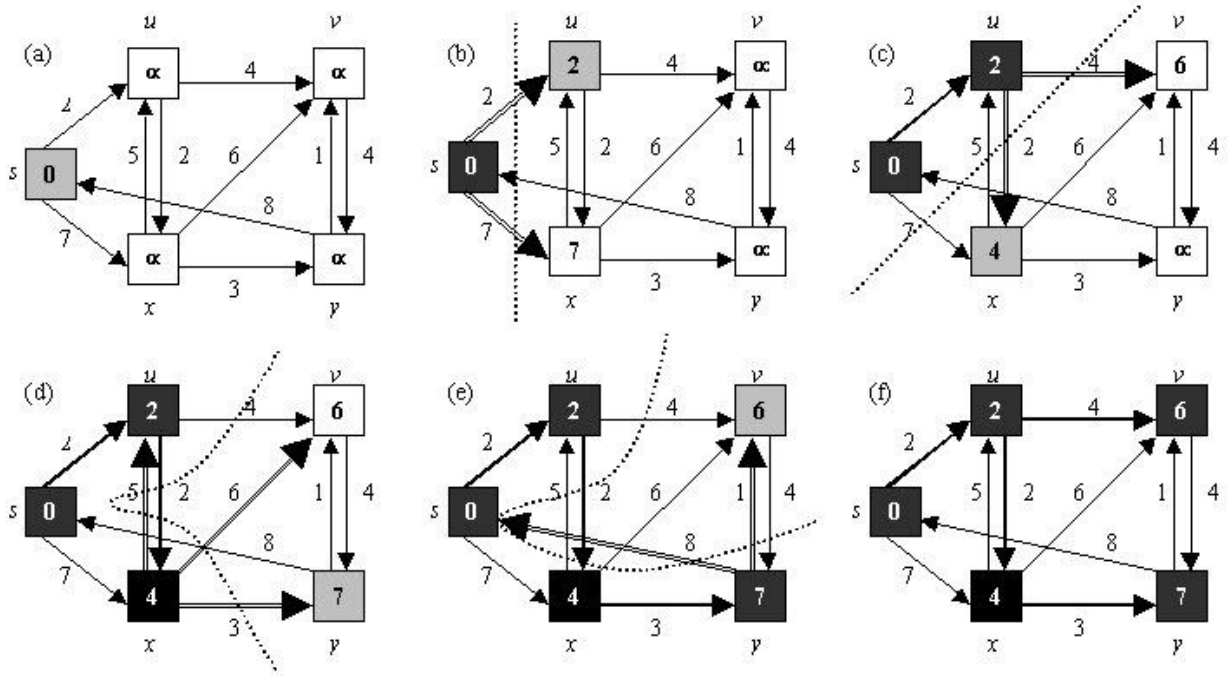
2. **Give a pictorial illustration of your algorithm**



1	2	3	4	5
5	7	3	4	2
2	1	3	7	1
4	1	3	0	5
0	0	7	1	2

→

1	2	3	4	5
5	0	3	4	2
2	1	3	0	1
4	1	3	0	5
0	0	0	1	2



(Source of Dijkstra illustration: Dijkstra's Algorithm)

3. **Do a time complexity analysis (big oh) of your algorithm.**

   The time complexity should be 2 Dijkstra algorithms, with the advantage that a number of max edges (M) have been removed. In order to find that max edge, I have to iterate through all the edges (E). The amount of work this makes is O(E(E-M)logV).

4. **Describe the choice(s) that you have made to make your algorithm run faster.**

   I did not work on the dijkstra implementation already provided. I can speak to the optimization piece of removing the largest edges.
   I made a copy of the original graph, only because it was unclear whether or not we could mutate the original input. Assuming that we couldn't (which seems like good practice), I went through and did a basic copy of the 2D array. I also paid attention to the highest weight found so far. Now, this next part is the part that I am proud of. I wanted to keep track of all the locations of the highest weights, so that we didn't have to iterate through the entire graph again to find them. I did this by using a HashMap in Java (noted as {} in pseudocode). I kept track of where the highest values were in this map, so that I could simply go back and just assign those locations to be "0". I didn't need to keep track of all the weight's locations, just the locations that are currently thought to be the highest. Once another highest weight was found, I only added to that list. This reduced an iteration through the entire graph.

5. **Write Java code that implements your algorithm. The code should be added to the base that we have in the sample key for Asgn3.**

   See included files