1 Problem 1

Algorithm 1: *GreedyInfluenceMaximization*(*G*, *activationProbability*, *trials*)

Input: Graph *G* – an undirected graph, Float *activationProbability* – the probability that a node will spread its influence to a given adjacent node, Int *trials* – the number of trials to run for each node in calculating its influence

Output: A dictionary object containing nodes as keys with values being the calculated influence of each node

```
influenceAvg ← new Dictionary;
foreach Node n in G.nodes do
    influenceAvg[n] ← 0;
foreach Node n in G.nodes do
    for _ ← 0 to trials do
        Q ← new Queue;
        Q.enqueue(n);
        visited ← empty Set;
        while Q is not empty do
            curr ← Q.dequeue();
            add curr to visited;
            foreach Node n1 adjacent to n in G do
                rand ← Random float between 0 and 1;
                if rand < activationProbability and n1 is not in Q and n1 is not in visited then
                    Q.enqueue(n1);
        influenceAvg[n] ← influenceAvg[n] + |visited|;
    influenceAvg[n] ← influenceAvg[n]/trials;
return influenceAvg;
```

Algorithm 2: *HighDegreeHeuristicInfluenceMaximization*(*G*)

Input: Graph *G* – an undirected graph

Output: A dictionary object containing nodes as keys with values being the calculated influence of each node

```
influence ← new Dictionary;
foreach Node n in G.nodes do
    influence[n] ← G.degree(n);
return influence;
```

2 Problem 2

Algorithm 3: *GirvanNewman*(*G*, *modBound*)

Input: Graph G – an undirected graph, Float *modBound* – a bound for the modularity so the algorithm knows when it is complete

Output: An undirected graph composed of all vertices and some edges of G, likely with more connected components representing communities

```
G2 ← copy(G);
connectedComponents ← DetectConnectedComponents(G, G.edges);
while Modularity(G, connectedComponents) < modBound do
    betweennessCentrality ← CalculateEdgeBetweennessCentrality(G2);
    maxEdge ← edge with max betweenness centrality in G2;
    Remove maxEdge from G2;
    foreach nodeSet ∈ connectedComponents do
        if maxEdge.node1 ∈ nodeSet then
            currComponent ← nodeSet;
            break;
    newComponents ← DetectConnectedComponents(G, currComponent);
    if newComponents.length == 2 then
        Remove currComponent from connectedComponents;
        Add newComponents to connectedComponents;
return G2;
```

Algorithm 4: *DetectConnectedComponents*(*G*, *allNodes*)

Input: Graph G – an undirected graph, Node Set *allNodes* – a set of nodes of G to split into sets of nodes representing connected components

Output: A list of sets of nodes in G, each set representing one connected component

```
components ← empty list;
while allNodes is not empty do
    visited ← empty set;
    stack ← arbitrary element removed from allNodes inserted into an empty stack;
    while stack is not empty do
        currNode ← stack.pop();
        add currNode to visited;
        Remove currNode from allNodes;
        foreach Node n adjacent to currNode in G do
            if n is not in stack and n is not in visited then
                stack.push(n);
    append visited to components;
return components;
```

Algorithm 5: $Modularity(G, components)$

Input: Graph G – an undirected graph, List of Sets *components* – a list of sets of nodes representing the communities to use for G in the modularity calculation

Output: Integer – the modularity of G

```
sum ← 0;
m ← number of edges in G;
foreach Node n1 in G.nodes do
    foreach Node n2 in G.nodes do
        if n1 is in the same component as n2 then
            A ← 1 if G has edge (n1, n2);
            d1 ← degree of n1;
            d2 ← degree of n2;
            sum ← sum + A − ((d1 ∗ d2)/(2 ∗ m));
return sum/(2 ∗ m);
```

Algorithm 6: CalculateEdgeBetweennessCentrality(G)

Input: Graph G – an undirected graph

Output: Dictionary containing the the edges and their related betweenness centrality value

```
EBC ← empty dictionary;
foreach Edge e in G.edges do
    EBC[e] ← 0;
foreach Node n in G.nodes do
    distanceDict ← new Dictionary;
    distanceDict[n] ← 0;
    queue ← new Queue;
    queue.enqueue(n);
    visited ← empty set;
    while queue is not empty do
        currNode ← queue.dequeue();
        add currNode to visited;
        foreach Node u adjacent to currNode in G do
            if u not in visited and u not in queue then
                distanceDict[u] ← distanceDict[currNode] + 1;
                queue.enqueue(u);
    numShortestPaths ← new Dictionary;
    foreach Node u in G.nodes do
        numShortestPaths[u] ← 0;
    numShortestPaths[n] ← 1;
    visited ← empty set;
    distances ← sort list of (key, value) pairs in distanceDict by value;
    foreach (key, d) in distances do
        foreach Node pred adjacent to key in G do
            if distanceDict[pred] == d - 1 then
                numShortestPaths[key] ← numShortestPaths[key] + numShortestPaths[pred];
    edgeWeights ←new Dictionary;
    foreach Edge e in G do
        edgeWeights[e] ← 0;
    foreach (key, d) in distances do
        sumIncoming ← 0;
        foreach Node succ adjacent to key in G do
            if distanceDict[succ] == d + 1 then
                sumIncoming ← sumIncoming + edgeWeights[(key, succ)];
        foreach Node pred adjacent to key in G do
            if distanceDict[pred] == d − 1 then
                edgeWeights[(n, pred)] ←
                  (1 + sumIncoming) ∗ (numShortestPaths[pred]/numShortestPaths[key]);
    foreach Edge e in edgeWeights do
        EBC[e] ← EBC[e] + edgeWeights[e];
foreach Edge e in G.edges do
    EBC[e] ← EBC[e]/2;
return EBC;
```