Assignment 5

Question 1

The best upper bound for T(n)/n in the sorted list L is 5/2. Using the accounting method, I will charge Insert 5/2 and OutputAndReduce 0. For m calls to Insert, the total actual cost is m and the total amortized cost is 5m/2, which means there is 5m/2 - m = 3m/2 left as credit. This means that for each item x inserted into A, x has 3/2 credit on it. The actual cost of a call to OutputAndReduce at this point would cost m, since there are m x's in A. Since there is 3m/2 left as credit, and 3m/2 > m, we clearly have enough credit to complete the call to OutputAndReduce. We are now left with 3m/2 - m = m/2 as credit and the size of A is now $\lfloor m/3 \rfloor$ and m/2 > $\lfloor m/3 \rfloor$ so we clearly have enough credit left to make another call to OutputAndReduce and we have enough credit to make an unlimited number of consecutive calls to OutputAndReduce at this point. Since Insert costs 5/2 and OutputAndReduce costs 0, it is clear that T(n)/n can never be greater than 5/2. If I were to charge Insert 2 and OutputAndReduce 0, then m calls to Insert would result in a total amortized cost of 2m, which means there would be 2m - m = m left as credit. The actual cost of a call to OutputAndReduce at this point would cost m, which means after this call we will have used all our remaining credit. The problem with this is that if $\lfloor m/3 \rfloor$ > 0, we do not have enough credit to support another call to OutputAndReduce, so 2 cannot be the best upper bound for T(n)/n and neither can any value in L that is smaller than 2. Therefore, 5/2 is the best upper bound for T(n)/n for any sequence of n of the two operations Insert and OutputAndReduce.

Question 2

a)

```
Supersource-Finder(A)
1 for i = 1 to n
2     if Find-Set(i) == NIL
3         for j = 1 to n
4             if (i != j)
5                 Make-Set((i, j))
6                 if (A[i, j] == 0)
7                     Make-Set(i)
8                     break
9                 else if (Find-Set((j, i)) == NIL)
10                    if (A[j, i] == 1)
11                        Make-Set(i)
12                        if (Find-Set(j) == NIL)
13                            Make-Set(j)
14                        break
15                    else
16                        if (Find-Set(j) == NIL)
17                            Make-Set(j)
18        if Find-Set(i) == NIL
19            return i
20 return NIL
```

My algorithm iterates through each row A that has not already been proven to not be a supersource. The algorithm keeps track of which nodes are not supersources by checking if the node belongs to a disjoint set. Only nodes that have been proven to not be supersources belong to disjoint sets. If A[i, j] == 0 then a set gets created for i because i is not supersource and the nested for loop is broken. If A[i, j] == 1, then we need to find the value at A[j, i]. The algorithm limits the amount of accesses to the matrix by checking first if Find-Set((j, i)) == NIL because if it is, then A[j, i] has not yet been visited by the first for loop. If A[j, i] == 1, then the algorithm calls Make-Set(i) and if Find-Set(j) == NIL, it calls Make-Set(j) as well and the nested for loop is broken, this is because i and j have both been proven to not be supersources since these two nodes point to each other. If A[j, i] == 0, then i can still be a supersource but if Find-Set(j) == NIL then Make-Set(j) is called since j can no longer be a supersource since it does not point to i. If Find-Set((j, i)) != NIL, then this means A[j, i] has already been accessed and A[j, i] == 0. This is because if A[j, i] == 1, then the algorithm would never visit the row representing the node i because i would have already been proven to not be a supersource when checking for j since we already know that A[i, j] == 1. Finally, after visiting all the columns of row i once each, if Find-Set(i) == NIL, then i is returned since the algorithm has proven that i is a supersource. If the algorithm has proven that there are no supersources in G, then NIL is returned.

b)

My algorithm accesses A 2m + n times in the worst-case if there is at least one edge. The worst-case scenario is if the final row of A does not represent a supersource and all of the edges in G reside in the final row and the rest of the elements in the matrix are 0. This means the algorithm would access only one element from each row prior to the final row plus the last element in the final row that proves it is not a supersource, this results in a total of n accesses. The last row could be accessed m times and in this case the other m rows which this row points to would also be accessed one time so there are 2m total accesses. This results in a total of 2m + n accesses. If the graph has no edges, then there are exactly n accesses to A. Therefore, in any graph with m > 0, there will never be more than 2m + n accesses to A.

c)

```
Supersource-Finder(L)
for u = 1 to n
    if Find-Set(u) == NIL
        neighbours = 0
        current = L[u]
        if (current == NIL)
            Make-Set(u)
            for (i = 1 to n)
                if u != i
                    Make-Set((u, i))
                    Union((0, 0), (u, i))
        while current != NIL
            v = current->info
            Make-Set((u, v))
            Union((1, 1), (u, v))
            neighbours++
            if (Find-Set(v) == NIL)
                Make-Set(v)
            if (Find-Set((v, u)) == NIL)
                temp = L[v]
                while temp != NIL
                    x = temp->info
                    if (x == u)
                        Make-Set(u)
                    if (Find-Set(x) == NIL)
                        Make-Set(x)
                    Make-Set((v, x))
                    Union((1, 1), (v, x))
                    temp = temp->link
                for j = 1 to n
                    if (v != j && Find-Set((v, j)) == NIL)
                        Make-Set((v, j))
                        Union((0, 0), (v, j))
            current = current->link
        for k = 1 to n
            if (u != k && Find-Set((u, k)) == NIL)
                Make-Set((u, k))
                Union((0, 0), (u, k))
        if (Find-Set(u) == NIL && neighbours == n - 1)
            return u
    return NIL
```

My algorithm iterates through every linked list in L until a supersource is found. The algorithm keeps track of which nodes are not supersources by checking if the node belongs to a disjoint set. Only nodes that have been proven to not be supersources belong to disjoint sets. If L[u] is empty, this means that u is not a supersource and a set is created for (u, i) for all i from 1 to n, except when i == u, and Union((0, 0), (u, i)) is called since these edges are non-existent. The nested while loop iterates over the neighbours (v) of u and a set is created for (u, v) and Union((1, 1), (u, v)) is called since this edge is existent. Since u points to its neighbours, each neighbour (v) node is proven to not be a supersource. The algorithm then checks if (v, u) belongs to a disjoint set, if it does then the edge (v, u) is non-existent because if (v, u) was existent then u would belong to a disjoint set and would not be undergoing the current steps in the algorithm. If (v, u) does not belong to a disjoint set, then another nested while loop iterates over all the neighbours of v, and if u is one of these neighbours then u is proven to not be a supersource since there is an edge that goes into it. For each neighbour (x) of v visited, Make-Set(x) is called if x does not already belong to a disjoint set since x is clearly not supersource since there is an edge (v, x). Make-Set((v, x)) and Union((1, 1), (v, x)) is also called since (v, x) is clearly an existent edge, and Make-Set((v, j)) and Union((0, 0), (v, j)) are called for the non-existent neighbours of v. Keeping track of all the edges that are existent and non-existent ensures that no linked-list will have to be traversed more than once. If after all the neighbours of u have been visited, u does not belong to a disjoint set and u is adjacent to n - 1 nodes, this means that u is a supersource since it points to every node and no node points to it. If no supersource is found, NIL is returned.

My algorithm accesses the adjacency lists m times in the worst-case. Since there are only m edges, there can only be a maximum of m accesses to the adjacency lists and this occurs only when the final node from 1 to n is a supersource or there is no supersource at all. By lines 16-22, all of the neighbours of v are visited, which means that for every node adjacent to u, the neighbours of those nodes are all visited. Therefore, if no supersource is found by the nth node, then there will be a total of m accesses to the adjacency lists.

Question 3

a)

My algorithm will start with a for loop that iterates n times. Let L denote the adjacency lists of G. On iteration u of the for loop, there is a nested while loop that iterates over every vertex v in the linked-list at L[u]. I then represent the edge (u, v) as (u, v) if $u < v$ or as (v, u) if $u \geq v$. I then check if the edge (u, v) is undirected by calling Find-Set((u, v)) and if NIL is returned, then the edge is undirected, otherwise, the edge has already been directed and the algorithm moves on to the next edge. If the edge is undirected, then the algorithm calls Make-Set((u, v)) and determines which vertex to direct the edge towards. Find-Set(u) is called and if NIL is returned then Make-Set(u) is called and "v to u" is printed. If NIL is not returned, then the algorithm prints "u to v" and Find-Set(v) is called and if NIL is returned then Make-Set(v) is called.

The for loop iterates exactly n times and the nested while loop iterates at most n - 1 times per iteration of the for loop (if each vertex has an edge with every other vertex). Make-Set and Find-Set run in O(1) time complexity and the rest of the algorithm within the nested while loop also takes constant time. The algorithm clearly iterates over every edge given by the adjacency list and since the graph is undirected, each edge in G is represented twice by the adjacency list, so there are 2m total iterations of the algorithm. Therefore, given the worst-case scenario, $n(n - 1) = n^2 - n = 2m < 2m + n \in O(m + n)$, which satisfies the precondition of the worst-case running time of my algorithm.

The algorithm iterates once for each edge, and if two edges contain the same vertices, then they are represented the same way, this is to ensure that the same edge is not directed twice. Once an edge becomes directed, Make-Set is called for that edge, only edges that are directed belong to a set, otherwise the edge does not belong to a set. This allows the algorithm to check whether an edge has yet to be directed or not. For each vertex in a given edge, the algorithm checks to if the vertex belongs to a set, if not this means no edge points in its direction and the algorithm calls Make-Set on that vertex and directs the edge in its direction. If the vertex already belongs to a set, then the edge points in the direction of the other vertex, and adds that vertex to a set if it does not already belong to one. This guarantees that at most one vertex will not have an edge pointed in its direction, the reason being because every vertex has at least one edge since the graph is connected.

The pseudo-code for the algorithm is as follows,

```
Graph-Orientation(L)
1 for u = 1 to n
2       current = L[u]
3       while v != NIL
4               v = current->info
5               if u < v
6                       (u, v) = (u, v)
7               if $u \geq v$
8                       (u, v) = (v, u)
9               if Find-Set((u, v)) == NIL
10                      Make-Set((u, v))
11                      if Find-Set(u) == NIL
12                              Make-Set(u)
13                              print "v to u"
14                      else
```

```
            print “u to v”
            if Find-Set(v) == NIL
                Make-Set(v)
    current = current->link
```

b)

G is a connected graph so, if m < n there are more vertices than edges and since each edge can only point in one direction, one vertex will not have an edge pointed in its direction. Therefore, if there is a way to orient G such that every vertex of G has at least one edge going into it, then there has to be at least one edge per vertex, which means $m \geq n$.

c)

The algorithm for this problem is the same algorithm I provided in part a) of this question.

Since the algorithm is the same, the running time of the algorithm remains $O(m + n)$ and the justification is the same as well.

Since $m \geq n$, there is at least one edge per vertex. By the if statements in the algorithm, an edge (u, v) goes into u if no edges go into it already, and into v if at least one edge already goes into u. Since there is at least one edge per vertex and the edge direction always prioritizes towards the vertex of the edge with no edges directed towards it, each vertex will have at least one edge going into it.