1. Written by: Crystal Yip; Read by: Xingyu Wang, Anna Lieu
Smallest upper bound: $\mathbf{\frac{5}{2}}$
Suppose we charge each insert $\frac{5}{2}$:

1. Charge 1 for the actual insert
2. Charge 3/2 as credit for potential output calls

Proof: Suppose k = m in the sequence of operations
The most expensive preceding sequence of operations on k = m is to call OUTPUTANDREDUCE() m times
This sequence would cost about

$$\left\lfloor \frac{k}{3^0} \right\rfloor + \left\lfloor \frac{k}{3^1} \right\rfloor + ... + \left\lfloor \frac{k}{3^m} \right\rfloor$$

$$\sum_{i=0}^{m} \left\lfloor \frac{k}{3^i} \right\rfloor < \sum_{i=0}^{\infty} \left\lfloor \frac{k}{3^i} \right\rfloor < \sum_{i=0}^{\infty} \frac{k}{3^i} = \text{k}*\frac{3}{2}$$

Since we have stored $\frac{3}{2}$ in each number, we have k$*\frac{3}{2}$ credit, which is enough to cover m calls to OUTPUTANDREDUCE()

2. Written by: Xingyu Wang; Read by: Crystal Yip, Anna Lieu
(a)

```
procedure FIND_SUPERSOURCE($A$):
1:  $skip \leftarrow 1$
2:  $i \leftarrow 1, j \leftarrow 1$
3:  $visited \leftarrow [\ ]$
4:  while True do
5:      if $A[i,j] = 1$ then
6:          $j \leftarrow j + 1$
7:          $skip \leftarrow skip + 1$
8:      else            //$A[i,j] = 0$
9:          if $i = j$ then
10:             $j \leftarrow j + 1$
11:         else
12:             $i \leftarrow i + skip$
13:             $skip \leftarrow 1$
14:     if $j > n$ then
15:         $j = 1$
16:     if $i > n$ then
17:         return NIL
18:     if (i, j) is in $visited$
19:         Check if the column $i$ is all 0
20:         if line 19 then
21:             return $i$
22:         else
23:             return NIL
24:     else
25:         add $(i, j)$ to $visited$
```

Proof of correctness: (Note: $A[u, v]$ represents an edge from u to v, in that particular direction)
Assume the vertices in G is $V = \{1, 2, ..., n\}$.
For a vertex u to be a supersource, it has to have an edge to every other vertices. So the row of u should be all 1s, except at $A[u, u]$. It also has to have no edge from any vertices to u, so the column of u must be all 0s.

Denote the row number to be i and the column number to be j. We start from the top-left corner $A[1,1]$.
Line 5-7: If the item is 1, there is an edge from i to u. We move to its right to keep checking, and increment skip, because the column i cannot be all 0s, it cannot be a supersource. We will skip it if we are on j's row.
Line 8-13: If the item is 0, it means there is no edge from i to u, or it means $i = u$. In the first case, there is a 0 in $i-th$ row, so i cannot be a supersource. We go check the next row. In the second case, it does not matter, so we keep checking this row by going to the slot to its right.
Line 14-15: If the new j is out of bounds of A, then we go back to check the beginning of that row. This is because since we moved right, the previous slot is a 1, so it might have a chance to be a supersource.
Line 16-17: If the new i is out of bounds of A, then there is no supersource in this graph. The reason is that this is caused by moving downwards from the previous slot, which must have been a 0. So, that row could not have been a supersource. Then we will skip *skip* rows, reasons explained above, and that brought us to the end of the matrix, so none of the vertices could have been a supersource.
Line 18-23: We keep track of all the slots visited. The only reason that we can get back to a slot that was visited before is because we were checking a row, because only moving right allows going back to the beginning of a row, whilst going down does not. So we checked an entire row, each time moving right. This means this row is full of 1s (except at $A[i,i]$). So i is a source. Line 19 checks the second part of supersource definition. If it passes we return i. Otherwise, since this whole row is 1 (except at $A[i,i]$), this rules out the possibility that any other vertices are supersources, because there is at least one 1 in their columns. So we return.
Line 24-25: We keep searching, and add this current edge to *visited* for use in line 18-23 in the future.
(b)
Since the algorithm will definitely move the current slot by one slot, to either right or down every iteration, without turning back (until we find a potential row of supersource), we will have to access it $n + n = 2n$ times in the worst case, with each n representing the length of a row and a column respectively.
Once we find a 1, we will start moving right (and possibly back to the beginning of the row), and check the entire row. That takes n access.
If this node is a source, then we also check its column, which takes n access.
So in total it takes $4n$ accesses.

(c)
```
procedure FIND_SUPERSOURCE_2($L$):
1:    $candidates$ = []
2:    for $v$ = 1... $n$:
3:        if $L[v]$.size = $n$:
4:            add $v$ to $candidates$
5:    if $candidates$ is not empty:
6:        for $u$ = 1... n:
7:            for $edge$ = L[u]:
8:                if $edge$ in $candidates$:
9:                    delete $edge$ from $candidates$
10:    if $candidates$ is empty:
11:        return NIL
12:    return $candidates$[0]
```
Proof of correctness:
Augment the Adjacency List L to keep track of the size of each Linked list in it. Then we keep track of all the sources in this graph in *candidates*, then eliminate the ones that has an edge to them. i.e., the ones that appears in other nodes' Linked List of neighbors. If there is any candidate left, it is the supersource.
(d)
The **for** loop on line 2 accesses all the nodes in the Adjacency List L. That takes up n accesses.
The **for** loop on line 6 accesses all the edges in L to see if it appears in *candidates*. This takes up m accesses.
Altogether, it takes $n + m$ access.

3. Written by: Anna Lieu; Read by: Xingyu Wang, Crystal Yip

(a)

```
procedure ORIENT_GRAPH($G$, S):
1.      directed_edges_of_G = []
2.      for each u in G:
3.          u.colour = white; u.children = empty list of size n // child of u is inserted by its key
4.      s.colour = grey
5.      Q = $\emptyset$
6.      while Q = $\emptyset$
7.          u = Q.pop
8.          for each v $\in$ $G$ adj[u]:
9.              if v.children[u.key] = NIL
10.                 u.children[v.key] = v
11.                 directed_edges_of_G.add(u,v)
12.             if v.colour == white:
13.                 v.colour = grey
14.                 Q.add(v)
15.      return directed_edges_of_G
```

Since G is connected, there is a path between any 2 vertices in V
line 2-3: iterates n times, since goes through each vertex in G
Say K_i = number of adjacent vertices to node i
line 6-14: overall iterates 2m times because
2m = $2|E|$ = $K_1 + K_2 + ... + K_n$, Since the for loop iterates K_i times for each u and G is undirected (each edge counted twice)
Thus, worst case is 2m + n = $\in \mathcal{O}(m+n)$
This algorithm assure every vertex except root S has an incoming edge. Since an incoming edge is created for every adjacent v of root S, and they are then added to Q, then an incoming edge is created for each adj v for those vertices (if a directed edge doesn't already exist), and so on, until all vertices are given an incoming edge.

(b)
m = $|E|$, n = $|V|$
Assume: we can orient G so each vertex has an incoming edge - then
$\forall$ v $\in$ V, $\exists$ u $\in$ V s.t. p[v] = u in directed G
Proof: m $\geq$ n
Assume by contradiction, m $<$ n (*)
In a minimally connected graph (tree), m = n-1
In G, every v $\in$ V has an incoming edge, so
m $\geq$ n - at least n edges
m $\geq$ n $>$ n-1 - we have a contradiction. Thus m $\geq$ n.

(c)
V = {1, 2, ..., n}, n = num vertices, m = num edges. Assume m $\geq$ n, which means G has a cycle.
OrientGraph(G, s):

```
1   DirectedEdges (DE) = [ ]
2   for each vertex u in G
3       u.color = white;
4       u.children = empty list of size n // each child is inserted by its key value as index
5       u.parents = [ ]
6       u.DE_indices = empty list of size n // stores index using child's key where (u, child) is stored in DE
7   s.color = gray
8   Q = empty queue
9   Q.add(s)
10  while Q is not empty
11      u = Q.pop
```

```
12      for each vertex v $\in$ G.Adj[u]
13          if (v.color == white)
14              u.children[v.key] = v
15              v.parents.add(u)
16              v.color = gray
17              Q.add(v)
18              DirectedEdges.add((u,v))
19              u.DE_indices[v.key] = DirectedEdges.size
20          else if (v.children[u.key] == nil and u.children[v.key] == nil)
21              v.children[u.key] = u
22              u.parents.add(v)
23              DirectedEdges.add((v,u))
24              temp = u
25              while (s.parents.size == 0)
26                  new_child = temp.parents[1]
27                  temp.parents[1] = nil
28                  new_child.children[temp.key] = nil
29                  temp.children[new_child.key] = new_child
30                  new_child.parents.add(temp)
31                  DE_index = new_child.DE_indices[temp.key]
32                  new_child.DE_indices[temp.key] = nil
33                  DirectedEdges[DE_index] = (temp, new_child)
34                  temp.DE_indices[new_child.key] = DE_index
35                  temp = new_child
36  return DirectedEdges
```

lines 2-6: each vertex in G in initialized attributes. This takes **n** time
lines 10-35: Outer while loop iterates n times because each vertex in V is added to queue Q only once, ie, when vertex.color is white. When a vertex is added to Q, vertex.color is changed to gray (**lines 16-17**)
line 12-35: For each iteration of a vertex u, the for loop visits only each vertex v adjacent to vertex u so it iterates k_i = number of adjacent vertices to u. This is equivalent to visiting all edges (u,v). So summing up all for-loop iterations for all n vertices, $k_1 + k_2 + ... + k_n$ = **2m**. It is twice number of edges, since G is undirected and each vertex pair is counted twice.
line 13-19: if statement applies only when v is encountered for the first time, so there are no directed edges going into v. Then make u the parent of v. This is constant time.
line 20-35: else if statement applies when v has already been encountered, and there are no directed edges between u and v. In this case, v is a vertex in a cycle. Since v is gray, then v has a parent (ie. incoming edge). So then make v the parent of u.
If root s does not yet have an incoming edge, then inner while loop is entered. It will reverse the edges in the path connecting u to root s. This creates an incoming edge to root s. Every other vertex in G will also have incoming edge(s) since every new vertex is assigned a parent in **13-19**, and in **20-35**, v already has an incoming edge. In **25-35**, u (with incoming edge of v), reverses its edge with another parent, and edges keep reversing until root s. This assures every vertex is the path still has an incoming edge.
line 25-35: inner while-loop occurs only once in function call, since after exiting, then while statement will always be false. Variable temp is first assigned to be vertex u (**line 24**). Then new_child is assigned the 1st parent of temp (**line 26**). new_child cannot be vertex v since u had a parent before adding v to u.parents. The directed edge is removed (**line 27-28**) and the direction reversed (**line 29-30**).
line 31-34: DE_index represents the index at which (new_child, temp) is stored in list DirectedEdges. The old edge is removed from DirectedEdges (constant time) and the new reversed edge is stored in same position. The last iteration will reverse the edge between root s and temp. It stops since s.parents is no longer empty.
Each inner while-loop reverses an edge which takes constant time, and it iterates at most **m** times, since each edge reversal can occur most m times.
Line 2-6 is n time. The total of outer while-loop (**line 10-35**) will take 2m + m = 3m time. 2m is from

visiting all undirected edges in G and at most m edge reversals is from inner while-loop (**line 25-35**). Thus worst case is 3m + n = $\in \mathcal{O}(m+n)$