CSORW4231_001_2022_1 - ANALYSIS OF ALGORITHMS I

hw1

Benjamin Jenney

TOTAL POINTS

148 / 160

QUESTION 1

1 Problem 1 **20 / 20**

QUESTION 2

Problem 2 32 pts

2.1 (a) **7.5 / 8**

2.2 **6.5 / 8**

2.3 **8 / 8**

2.4 **8 / 8**

QUESTION 3

3 Problem 3 **14 / 25**

QUESTION 4

4 Problem 4 **24 / 28**

QUESTION 5

5 Problem 5 **30 / 30**

QUESTION 6

6 Problem 6 **30 / 25**

+ 5 *Point adjustment*

Problem 1

Design an algorithm that returns an index i s.t $A[i] = i$. A is sorted.

We can use a modified $BinarySearch$ where $mid = i$ and i is the target value.

Input: A sorted array A of n distinct integers
Output: An index i such that $A[i] = i$, if one exists; -1, otherwise.

$ModifiedBinarySearch(A, l, r)$

```
1: if l >= r then
2:     return -1
3: end if
4: i = l + (r − l)/2
5: if A[i] == i then
6:     return i
7: end if
8: if A[i] < i then
9:     return ModifiedBinarySearch(A, i + 1, r)
10: else
11:     return ModifiedBinarySearch(A, l, i)
12: end if
```

Correctness: 1. If $A[m = i]$ equals i. We return i and we are done. 2. If $a[i] < i$. We then search the subarray $[i+1..r]$. Since i could not be in $A[l..i]$ since every value in $A[l..i] < i$. We know this since A is sorted in ascending order. 3. If $a[m = i] > i$. Our algorithm will then search the new range A[l,i].

In both case 2 and 3 the subarrays will continue to shrink. Since i is the pivot it will take take on every value in the possible subarrays that can contain it. If by the time $A[l..i]$ or $A[i..r]$ have size one and $A[i]$ still does not $equali$. Then we know i did not exist in the A

Analysis: $T(n/2) + 1$ applying master theorem $a = 1, b = 2, k = 0$ so $a = b^k$ and $n^k = 1$ therefor $modifiedbinarysearch = O(logn)$

1 Problem 1 **20 / 20**

Problem 2

a) We can get the minimum and maximum element in the arrays in O(n) time and then take the absolute difference of both of them. This will always yield the maximum value because these elements are the furthest apart on the number line.

Input: An unsorted array of integers A
Output: x and y in A s.t. $|x - y|$ is a maximum

$MaximumDifference_1(A)$:

```
1:  max = min = A[1]
2:  for i = 1 to A.length do
3:      if A[i] < min then
4:          min = A[i]
5:      end if
6:      if A[i] > max then
7:          max = A[i]
8:      end if
9:  end for
10: return absolute(min − max)
```

Correctness: In the for loop we find the minimum, and maximum value in A. On line 1 we guess that the first element in A is both a minimum and maximum, we then go through each element in A checking whether it is smaller then our minimum guess or larger than our maximum guess, we then update our minimum and maximum guess accordingly. Since we do this for every element in A we are guaranteed that both min and max will contain the smallest and largest elements in A. Without loss of generality, the absolute difference of two values is the distance of the two values from each other on the number line. Since we are taking the absolute difference of the min and max values in A we are guaranteed they will be the two values farthest apart on the number line. Therefore, $|min - max|$ is the maximum difference.

Analysis: Line 1 and 10 are clearly constant time instructions. Inside the For loop we have 2 tests, and assignment instructions which again are constant. Since we can not guarantee any previous guess is the min/max we must check every element in A. This means that $MaximumDifference_1 = \Theta(n)$ in the worst case.

b) Since the Array A is sorted then we only have to take the absolute difference of the first and last values in A.

Input: A sorted array A of n integers.
Output: Elements $x, y \in A$ such $|x - y|$ is a maximum.

2.1 (a) **7.5 / 8**

Problem 2

a) We can get the minimum and maximum element in the arrays in O(n) time and then take the absolute difference of both of them. This will always yield the maximum value because these elements are the furthest apart on the number line.

Input: An unsorted array of integers A
Output: x and y in A s.t. $|x - y|$ is a maximum

$MaximumDifference_1(A)$:

```
1:  max = min = A[1]
2:  for i = 1 to A.length do
3:      if A[i] < min then
4:          min = A[i]
5:      end if
6:      if A[i] > max then
7:          max = A[i]
8:      end if
9:  end for
10: return absolute(min − max)
```

Correctness: In the for loop we find the minimum, and maximum value in A. On line 1 we guess that the first element in A is both a minimum and maximum, we then go through each element in A checking whether it is smaller then our minimum guess or larger than our maximum guess, we then update our minimum and maximum guess accordingly. Since we do this for every element in A we are guaranteed that both min and max will contain the smallest and largest elements in A. Without loss of generality, the absolute difference of two values is the distance of the two values from each other on the number line. Since we are taking the absolute difference of the min and max values in A we are guaranteed they will be the two values farthest apart on the number line. Therefore, $|min - max|$ is the maximum difference.

Analysis: Line 1 and 10 are clearly constant time instructions. Inside the For loop we have 2 tests, and assignment instructions which again are constant. Since we can not guarantee any previous guess is the min/max we must check every element in A. This means that $MaximumDifference_1 = \Theta(n)$ in the worst case.

b) Since the Array A is sorted then we only have to take the absolute difference of the first and last values in A.

Input: A sorted array A of n integers.
Output: Elements $x, y \in A$ such $|x - y|$ is a maximum.

$MaximumDifference_2(A)$:
1: **return** $absolute(A[1] - A[n])$

Correctness: By definition, since A is sorted, the first value of A will be the minimum value of A, and the last value in A will be the maximum. Furthermore, practically by definition, the minimum value and maximum value will be farthest apart from each other on the number line. Therefore the first and last values of A will have the greatest absolute difference.

Analysis: Obviously $MaximumDifference_2 = \Theta(1)$ in both the best and worst case.

c)

To find the minimum distance in the array A we can sort A and take the minimum distance between adjacent pairs i.e. $A[i], A[i+1]$.

Input: The unsorted array of integers A and its length n.
Output: The minimum absolute difference of $x, y \in A$.

$MinimumDifference_1$
1: $MergeSort(A, 0, n)$
2: $min = \infty$
3: **for** $i = 1, 2, ..., n-1$ **do**
4: **if** $min > absolute(A[i] - A[i+1])$ **then**
5: $min = absolute(A[i] - A[i+1])$
6: **end if**
7: **end for**
8: **return** min

Correctness: By sorting A we guarantee that for any iteration i, $A[i] < A[i+1] < A[i+2] < ... < A[n]$, as such $A[i]$ and $A[i+1]$ are closest to each other in one dimension for i from 1 to n. Before the first iteration of the For loop min is given a dummy infinitely large value, and $A[1] < A[2] < A[3] < ... < A[n]$. Before the ith iteration of the for loop min contains the smallest absolute difference of adjacent pairs up to $A[i]$. On the ith iteration if the distance from $A[i]$ to $A[i+1]$ is smaller than all adjacent pairs up to $A[i-1]$ then, as is clearly shown on lines 4 and 5, min will be assigned the new smallest distance. By the time $i = n - 1$ min will contain the smallest distance between adjacent pairs up to $A[n-1]$, and if $absolute(A[i = n-1] - A[i+1 = n]) < min$ then we know it is the smallest distance of all previous tested adjacent pairs in A, as such $absolute(A[i = n-1] - A[i+1 = n])$ would be the minimum difference.

Analysis: The $MergeSort$ procedure is unmodified from its implementation in class. The test in the for loop on line 3 runs n times. Inside the for loop is a

2.2 **6.5 / 8**

$MaximumDifference_2(A)$:
1: **return** $absolute(A[1] - A[n])$

Correctness: By definition, since A is sorted, the first value of A will be the minimum value of A, and the last value in A will be the maximum. Furthermore, practically by definition, the minimum value and maximum value will be farthest apart from each other on the number line. Therefore the first and last values of A will have the greatest absolute difference.

Analysis: Obviously $MaximumDifference_2 = \Theta(1)$ in both the best and worst case.

c)

To find the minimum distance in the array A we can sort A and take the minimum distance between adjacent pairs i.e. $A[i], A[i+1]$.

Input: The unsorted array of integers A and its length n.
Output: The minimum absolute difference of $x, y \in A$.

$MinimumDifference_1$
1: $MergeSort(A, 0, n)$
2: $min = \infty$
3: **for** $i = 1, 2, ..., n-1$ **do**
4: **if** $min > absolute(A[i] - A[i+1])$ **then**
5: $min = absolute(A[i] - A[i+1])$
6: **end if**
7: **end for**
8: **return** min

Correctness: By sorting A we guarantee that for any iteration i, $A[i] < A[i+1] < A[i+2] < ... < A[n]$, as such $A[i]$ and $A[i+1]$ are closest to each other in one dimension for i from 1 to n. Before the first iteration of the For loop min is given a dummy infinitely large value, and $A[1] < A[2] < A[3] < ... < A[n]$. Before the ith iteration of the for loop min contains the smallest absolute difference of adjacent pairs up to $A[i]$. On the ith iteration if the distance from $A[i]$ to $A[i+1]$ is smaller than all adjacent pairs up to $A[i-1]$ then, as is clearly shown on lines 4 and 5, min will be assigned the new smallest distance. By the time $i = n - 1$ min will contain the smallest distance between adjacent pairs up to $A[n-1]$, and if $absolute(A[i = n-1] - A[i+1 = n]) < min$ then we know it is the smallest distance of all previous tested adjacent pairs in A, as such $absolute(A[i = n-1] - A[i+1 = n])$ would be the minimum difference.

Analysis: The $MergeSort$ procedure is unmodified from its implementation in class. The test in the for loop on line 3 runs n times. Inside the for loop is a

comparison, and an assignment, which are called $n-1$ times. Therefore we have in the worst case: $\Theta(nlgn)+\Sigma_{i=1}^{n}1+\Sigma_{i=1}^{n-1}2+C = \Theta(nlgn)+n+2(n-1)+C = \Theta(nlgn)+3n+C$. Clearly, anything on the order $\Theta(nlgn)$ dominates $3n+C$ and as such in the worst case $MinimumDifference_1 = \Theta(nlgn)$.

d)

Since A is already sorted, to find the minimum distance in the array A, we can take the minimum distance between adjacent pairs in A i.e. $A[i], A[i+1]$.

Input: The unsorted array of integers A and its length n.
Output: The minimum absolute difference of $x, y \in A$.

$MinimumDifference_2$

```
1: min = ∞
2: for i = 1, 2, ..., n − 1 do
3:     if min > absolute(A[i] − A[i + 1]) then
4:         min = absolute(A[i] − A[i + 1])
5:     end if
6: end for
7: return min
```

Correctness: A is already sorted. Refer to the explanation in problem $2c$.

Analysis: The test in the for loop on line 3 runs n times. Inside the for loop is a comparison, and an assignment, which are called $n-1$ times. Outside of the for loop we have two constant time calls, let C be some small constant. $MinimumDifference_2 = \Sigma_{i=1}^{n}1+\Sigma_{i=1}^{n-1}2+C = n+2(n-1)+C = 3n+C = \Theta(n)$ in the worst case, since we have to visit all adjacent pairs in A.

2.3 **8 / 8**

comparison, and an assignment, which are called $n-1$ times. Therefore we have in the worst case: $\Theta(nlgn)+\Sigma_{i=1}^{n}1+\Sigma_{i=1}^{n-1}2+C = \Theta(nlgn)+n+2(n-1)+C = \Theta(nlgn)+3n+C$. Clearly, anything on the order $\Theta(nlgn)$ dominates $3n+C$ and as such in the worst case $MinimumDifference_1 = \Theta(nlgn)$.

d)

Since A is already sorted, to find the minimum distance in the array A, we can take the minimum distance between adjacent pairs in A i.e. $A[i], A[i+1]$.

Input: The unsorted array of integers A and its length n.
Output: The minimum absolute difference of $x, y \in A$.

$MinimumDifference_2$

```
1: min = ∞
2: for i = 1, 2, ..., n − 1 do
3:     if min > absolute(A[i] − A[i + 1]) then
4:         min = absolute(A[i] − A[i + 1])
5:     end if
6: end for
7: return min
```

Correctness: A is already sorted. Refer to the explanation in problem $2c$.

Analysis: The test in the for loop on line 3 runs n times. Inside the for loop is a comparison, and an assignment, which are called $n-1$ times. Outside of the for loop we have two constant time calls, let C be some small constant. $MinimumDifference_2 = \Sigma_{i=1}^{n}1+\Sigma_{i=1}^{n-1}2+C = n+2(n-1)+C = 3n+C = \Theta(n)$ in the worst case, since we have to visit all adjacent pairs in A.

2.4 **8 / 8**

Problem 3

We can use a divide and conquer technique. keeping track of the majority element for both the left and right side.

Input: The Array A, l the low partition and r the high partition
Output: the majority element

$MajorityElement(A, l, r)$

1: **if** $l >= r$ **then**
2: **return** $A[l]$
3: **end if**
4: $m = l + (r - l)/2$
5: $candidateL = prob3(A, l, m)$
6: $candidateR = prob3(A, m + 1, r)$
7: $countL = 0$
8: $countR = 0$
9: **if** $candidateR == candidateL$ **then**
10: **return** $candidateL$
11: **for** $inrange(l, r)$ **do**
12: **if** $candidateL == A[i]$ **then**
13: $countL+ = 1$
14: **end if**
15: **end for**
16: **for do**$iinrange(l, r)$
17: **if** $candidateR == A[i]$ **then**
18: $countR+ = 1$
19: **end if**
20: **end for**
21: **if** $countL > countR$ **then**
22: **return** $candidateL$
23: **if** $countR > countL$ **then**
24: **return** $candidateR$

Correctness: If a majority exists in the array, it must exist in either the left or right halves as well. We then collect these majorities and check the entire array A to verify that they are in fact majorities.

Analysis:There are two recursive calls and $O(n)$ work being done outside the recursive calls so $T(n) = 2T(n/2) + O(n)$. By the master theorem this is $O(nlogn)$.

3 Problem 3 **14 / 25**

Problem 4

N.B.: I have heard conflicting opinions from T.A.'s as to how to correctly interpret this problem. The first answer directly below I believe is the desired one so feel free to skip the alternative if you are satisfied.

To find a local minimum in the binary tree T we store the root of the current subtree T and compare it with its children. We either return the root outright, if the root is smaller than its children, or we continue visiting subtrees depending on which of the children has the smallest associated value, we continue this until we reach a root who's associated value is smaller than its children, or we reach a node who has no right child, this last subtree either has no children or one child, and that child is a leaf since the tree is complete, and therefore there are no longer subtrees to explore. We will move on a depth first path using a simple tree traversal.

Input: The root v_{root} of the complete binary tree T. Output: A node in T which is a local minimum.

$LocalMinimum(v_{root}, minimum)$

1: $minimum = v_{root}.x$
2: **if** v_{root} has no children **then**
3: **return** v_{root}
4: **end if**
5: **if** v_{root} has no right child **then** ▷ since T is complete v_{root} may have one child
6: **if** $minimum > u_{left}.x$ **then**
7: **return** u_{left}
8: **else**
9: **return** v_{root}
10: **end if**
11: **end if**
12: **if** $minimum > u_{left}.x$ **then**
13: **return** $LocalMinumum(u_{left}, minimum)$
14: **else if** $minimum > u_{right}.x$ **then**
15: **return** $LocalMinumum(u_{right}, minimum)$
16: **end if**=0

Correctness:
Loop invariant: v_{root} always contains the smallest value associated with nodes visited along the path.

Base case If a subtree has only one node, it is trivially a local minimum so we store the root and immediately return it. This is true before the first recursion maintaining the loop invariant.

Maintenance: Let i enumerate the recursive steps taken by $LocalMinimum$

up to and including the current recursive step. Before the ith recursion *minimum* contains the associated value of u_{i-1} where u_{i-1} is the smallest neighbor of v_{i-1} maintaining the loop invariant. (if v_{i-1} was smaller than it's children then there would not be an ith recursion since we would have returned v_{i-1}). On the ith recursion u_{i-1} becomes the root of the current subtree, v_i, and we check if v_i is smaller than its children. If v_i is smaller than both its children then v_i is the local minimum and we return it outright. If v_i is not smaller than its children we will continue the path by visiting the child with the smallest associated value maintaining the loop invariant.

Termination: Since the tree is finite in the number of nodes, we will terminate once we inevitably reach a leaf. If we travel from root to leaf then the associated value of the leaf will, by consequence of the logic described in maintenance, be smaller than all other nodes previously visited on the path and therefore will be a local minimum which we return immediately since the leaf is a subtree with only one node (check base case), thus maintaining the loop invariant.

Alternative interpretation of the problem statement:

Here we can do a simple tree traversal DFS comparing each value associated with a node for each node in each sub-tree in T. We can find the local minimums by visiting each node, checking if the node has a child, and then comparing the root of that sub-tree with it's child or children for the minimum value of the sub-tree. Since we are guaranteed a complete tree we know that if a root has a child it must have at least a left child. So we have two base base cases: in the case of a complete tree where a node has only one child we have a valid sub-tree so we collect the minimum and return *minimums*. In the case that the last visited node has no children, then we know we have checked all sub-trees and simply return *minimums*.

Input: A complete binary tree with $n = 2^d - 1$ nodes, and an array *minimums* of size $2^{d-1} + 1$ since the leaves are not sub-trees.
Output: An array of the local minimums.

$LocalMinimums(v_{root}, minimums)$

```
1: Let minimums be an empty array of size 2^{d-1} + 1
2: if v_root has no children then
3:     return minimums
4: end if
5: if v_root has one child then
6:     if v_left.x < v_root.x then
7:         minimums.append(v_left)
8:     else
9:         minimums.append(v_root)
```

10: **end if**
11: **return** *minimums*
12: **end if**
13: **if** $v_{left}.x < v_{right}.x$ and $v_{left}.x < v_{root}.x$ **then**
14: $minimums.append(v_{left})$
15: **else if** $v_{right}.x < v_{left}.x$ and $v_{right}.x < v_{root}.x$ **then**
16: $minimums.append(v_{right})$
17: **else**
18: $minimums.append(v_{root})$
19: **end if**
20: $LocalMinimums(v_{left}, minimums)$
21: $LocalMinimums(v_{right}, minimums)$

Correctness
Base cases:

1. If $v \in T$ has no children v is trivially the local minimum.

2. If v has one child we append the minimum to *minimums* and return it.

Upkeep: Before the ith recursion *minimums* contains minimum nodes of the $1, 2, ..., i-1$ sub-trees. On the ith recursion we compare v^i_{root}, v^i_{left} and v^i_{right} obtaining the minimum in that sub-tree. If the ith sub-tree has less than 3 nodes then it is the last sub-tree to be explored since we are given a complete binary tree and we proceed as described in our base cases.
Termination: We terminate when there are no more sub-trees to explore returning the local minimums in T as specified. Analysis: $2T(n-1) + c = 2 * ((n-1) + (n-2) + (n-3) + ... + 1) + O(1) =$

4 Problem 4 **24 / 28**

Problem 5

Explanation: This is a reach-ability graph problem. However if we are to go from the Trace array to a graph structure we have to be careful with how we go about making the graph. We can make pairs from the tuples which will form the individual vertex labels, so the the tuples $(C_i, C_j, t_k), (C_i, C_l, t_p)$ become the vertices $(C_i, t_k), (C_j, t_k), (C_l, t_k)$. If a computer can infect another computer at time t_k then the relationship between them is bi-directional since the communication between any number of computers at that time can go any way, for instance both $(C_i, t_k), (C_j, t_k)$ and (C_l, t_k) will have a bi-directional relationship with two edges between each other. However, since C_i could be infected at t_k, and then again communicate at some later time, say t_p, we must assume that C_i is still infected at time t_p so the vertices $(C_i, t_k), (C_i, t_p)$ will only have one edge between them in the direction of time. If we build the graph accordingly we can then run a breadth first search on the graph finding both our start and target s, and t on the provided $Query$
Implementation: The difficulty is in assigning the correct number of edges when there is a relationship between two computers at different times in linear time. For each trace (C_i, C_j, t_k) we know there will be two edges representing the bi-directional relationship between $(C_i, t_k), (C_j, t_k)$. However, we also have to remember the last time C_i (or any computer) communicated so we can create a single edge between when that computer last communicated and it's later communication. In order to do this we can label each computer with a number for indexing into an array of linked lists, called mem below, and appending vertices that contain that computers number. For example, $mem[i = C_i] =\rightarrow (C_i, t_k) \rightarrow (C_i, t_p)$. Then if C_i communicates again at a time later than t_p, say t_q, we can retrieve its last communication and create an edge between (C_i, t_p) and (C_i, t_q). By managing mem in this way for each computer in the trace we can always retrieve the vertex representing that computers last communication as well as all previous vertices associated with that computer. In parallel with managing mem, we can maintain an adjacency list that will represent the graph, adding edges for each trace and probing mem when appropriate. Similarly the adjacency list can be implemented with linked lists.

Input: a Linked List representation of an adjacency list adj of size $n = 2 * trace.length$, a mem array of linked lists of size $n = 2 * trace.length$ (since $2 * trace.length$ is the maximum number of vertices that can be parsed from $trace$), the pair of queries $Query$, and the array of triples $trace$. Output: 'YES' if there is a path from and to the queried computers with in the range of time $[x, y]$. 'NO' if not. $BuildGraphFromTrace(adj, mem, trace)$

1: **for** each tr in $trace$ **do**
2: $v = (tr[1], tr[3])$
3: $u = (tr[2], tr[3])$
4: $adj[v].append(u)$
5: $adj[u].append(v)$
6: **if** $mem[v.comp_{num}]$ is not empty **then**

```
        if mem[v.comp_num].get_last() is not equal to v then last_v = mem[v.comp_num].get_last()
            adj[last_v].append(v)
            mem[v.comp_num].append(v)
        end if
    end if
    if mem[u.comp_num] is not empty then
        if mem[u.comp_num].get_last() is not equal to u then last_u = mem[u.comp_num].get_last()
            adj[last_u].append(u)
            mem[v.comp_num].append(v)
        end if
    end if
    if mem[v.comp_num] is empty then
        mem[v.comp_num].append(v)
    end if
    if mem[u.comp_num] is empty then
        mem[u.comp_num].append(u)
    end if
end for
```

```
BFSVirus(adj, mem, trace, Query)
BuildGraphFromTrace(adj, mem, trace)
s = Query[1]
t = Query[2]
x = s.x
y = t.y
sComp_list = mem[s.comp_num]
tComp_list = mem[t.comp_num]
tmp = sComp_list.head
while tmp.v.time <= x do
    tmp = tmp.next
end while
s = tmp.v
tmp = tComp_list.tail
while tmp.v.time >= y do
    tmp = tmp.next
end while
t = tmp.v
Let q be an empty Queue
s.visited = 1
while q is not empty do
    v = q.dequeue()
    for each u in adj[v] do
        if u.visited == 0 then
            q.enqueue(u)
            if u matches t then
```

```
26:                 print 'YES'
27:                 return 'YES'
28:             end if
29:             u.visited = 1
30:         end if
31:     end for
32: end while
33: print 'NO'
34: return 'NO'
```

Correctness: If Computer C_i in the query could infect C_j in the time bounded by $x < y$ then there must be a path from C_i to C_j where $s \in adj$ such that $s.time$ is nearest to but not greater than x and a $t \in adj$ such that $t.time$ is nearest to but not less than y. By going through the list and finding such vertices s, t, and then running a BFS we are guaranteed to find that path if it exists. Analysis: Building the graph as one can see only required constant time tests, assignments, and constant time list operations, therefor $BuildGraphFromTrace = O(len(Trace))$. $BFSVirus$ includes some walks along linked lists on lines 9-10 each requiring $O(n)$ time where n is the number of vertices in the worst case. We then run a BFS from lines 20 - 31 requiring $O(m + n)$. All other instructions are constant time. All together we have $O(n) + O(n) + O(m + n) + O(len(trace))$ and since $O(m + n) > O(len(trace))$ we can say that $BFSVirus$ is on the order $O(m + n)$.

5 Problem 5 **30 / 30**

Problem 6

Input: An adjacency representation of a directed graph *adj*
Output: A source node from which all other nodes are reachable

The algorithm:

1. Call $DFS(adj)$ and use a stack to keep finish time order. we can do this by modifying the search utility:
   $Search(adj)$

```
1: visited[u] = 1
2: for v in adj[u] do
3:     if visited[v] == 0 then
4:         search(v)
5:     end if
6: end for
7: stack.append(u)
```

2. reverse the adjacency list:
   $Reverse(adj)$

```
revAdj = new adjacency list same size as adj
for i in adj do
    for j in adj[i] do
        revAdj[j].append(i)
    end for
end for
return revAdj
```

3. run a DFS that takes advantage of the ordered stack and collect SCC components in a list:
   $RevDFS(revAdj)$

```
components = []
for i in range(len(visited) do
    visited[i] = 0
end for
while len(stack) > 0 do
    u = stack.pop()
    if  thenvisited[u] == 0
        scc = []
        revsearch(revAdj, scc, u)
        components.append(scc)
    end if
end while
return components
```

   $revsearch(revAdj, scc, u)$

```
visited[u] = 1
scc.append(u)
for v in revAdj[u] do
    if  thenvisited[v] == 0
        revsearch(revAdj, scc, v)
    end if
end for
```

4. We can then construct the meta graph from the original adjacency list *adj*. We will associate the first element from each component in the list *components* with the corresponding nodes in the connected forest generated by the search on lines 8-12 replacing every label in the forest with its representative in *components*. This will give the SCC in *adj* a common label which represents the whole SCC. This will be the label of the SCC that we will associate with every node in *adj* that belongs to the SCC and we will do this for each SCC. We can then check in constant time if we have switched common labels in the case we have an edge from one SCC to the other. We will store these meta nodes and edges in a new adjacency list *adjscc*. This new meta graph will be guaranteed to be a DAG and therefore it will be easy to find if there is a vertex $s \in V$ from which all other vertices can be reached:

```
createMetaGraph(revAdj, adjscc, commonlabels, stack, adj)
 1: components = revDFS(revAdj, stack)
 2: for component in components do
 3:     commonlabel = component[0]
 4:     for u in component do
 5:         commonlabels[u] = commonlabel
 6:     end for
 7: end for
 8: for u in adj do:
 9:     for v in adj[u] do:
10:         v = commonlabels[v]
11:         u = commonlabels[u]
12:     end for
13:     if commonlabels[v]! = commonlabels[u] then
14:         adjscc[v].append(u)
15:     end if
16: end for
17: return adjscc
```

5. Finally we can count in degrees of each vertex of the meta graph *adjscc*. If more than one meta vertex has $indeg(0)$ we know that no vertex exist that can reach every other vertex. We can do this by applying one more search to *adjscc* taking $O(m + n)$ time.

Correctness: By pushing to a stack in $search(u)$ on line 7, from the first $DFS(adj)$ call, we ensure that we will only push a vertex the moment it clocks in its finish time leaving the last to finish on the top of the stack. This is the only change we made to $DFS(adj)$ and it is a constant time operation. In step 2 we reverse the the adjacency list adj this is a straight forward implementation that clearly $(i, j) \in E$ to $(j, i) \in E$. Since we do this only once for each vertex and each edge we maintain $O(m + n)$. In step 3 we call $revDFS(revAdj)$. This is a fairly straight forward DFS accept that we search vertices the reversed graph $revAdj$ in the last finished first order maintained by the stack. By doing this we process sink by sink ensuring each forest collect by components is a strongly connected component. The correctness of steps 1, 2 and 3 are further supported by proofs given in the lecture slides. In step 4 we create a meta graph from the correctly gathered strongly connected components in $components$. We generate a shared label for each SCCs in adj by performing a basic search through adj replacing each vertex in the forest generated by the search with a common label, thus consolidating each SCC into one meta vertex. The test on line 13 ensures that each meta vertex with an outgoing edge gets an outgoing edge. Finally in step 5 with the Meta graph complete its just a simple matter of checking if there exists more than one meta vertex with $indeg(v) = 0$ this can be done in linear time with another simple search through $adjscc$. If it is the case that more than one vertex has 0 in degrees then we know that there exists no vertex in the meta graph of adj since the meta graph representation of adj, $adjscc$ is guaranteed to be a DAG. However, if there is only one vertex with $indeg(v) = 0$ then we know there does exist a vertex $s \in V$ from which all other vertices are reachable since again $adjscc$ is a DAG. Analysis: in step one we do a basic DFS with only an addition of a constant time instruction of appending to the stack, this is an $O(m + n)$ procedure. in step 2 we reverse the graph adj clearly this is an $O(m + n)$ procedure. In step 3 we perform another DFS with only the addition of some 3 or 4 constant time instructions in managing the components, again this is an $O(m + n)$ procedure. In step 4 we go through each of the vertices in $components$ which in the worst case is $O(n)$, we then do a simple search through through the graph adj with 2 constant instructions on line 10 and 11 which is $O(m+n)$ the test on line 13 and 14 are both constant time. Finally in step 5 we would do one last search through $adjscc$ which is in the worst case $O(m + n)$. Putting it all together we have $O(m+n)+O(m+n)+O(m+n)+O(n)+O(m+n)$ which can be simplified to just $O(m + n)$.

6 Problem 6 **30 / 25**

+ 5 *Point adjustment*