- Please note that one of the main goals of this course is to design efficient algorithms. So, there are points for efficiency even though we may not explicitly state this in the question.
- Unless otherwise mentioned, assume that graphs are given in adjacency list representation.
- In the lectures, we have discussed an $O(V + E)$ algorithm for finding all the SCCs of a directed graph. We can extend this algorithm to design an algorithm `CreateMetaGraph(`G`)` that outputs the meta-graph of a given directed graph in $O(V + E)$ time. For this homework, you may use `CreateMetaGraph(`G`)` as a sub-routine.
- The other instructions are the same as in Homework-0.

There are 6 questions for a total of 100 points.

---

1. An undirected graph is said to be *connected* iff every pair of vertices in the graph are reachable from one another. Prove the following statement:

   Any connected undirected graph with n nodes has at least $(n-1)$ edges.

   We will prove the statement using Mathematical Induction. The first step in such a proof is to define the propositional function. Fortunately for this problem, this is already given in the statement of the claim.

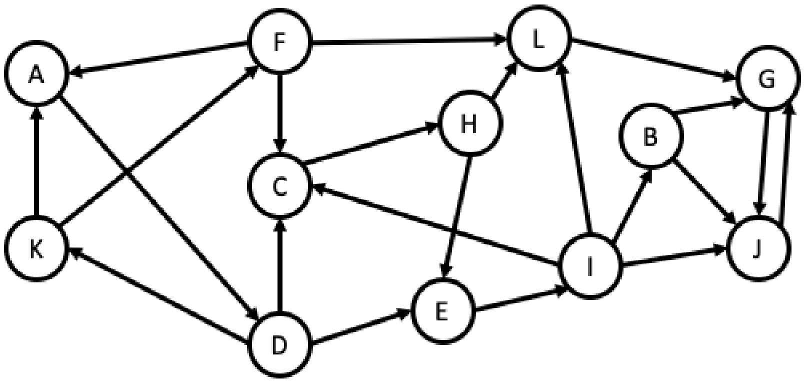
   $P(n)$: Any connected undirected graph with n nodes has at least $(n-1)$ edges.

   The base case is simple. $P(1)$ holds since any connected graph with 1 node having at least 0 edges, is indeed true. For the inductive step, we assume that $P(1), P(2), ..., P(k)$ holds for an arbitrary $k \geq 1$, and then we will show that $P(k+1)$ holds. Consider any connected graph G with $(k+1)$ nodes ~~and k edges~~. You are asked to complete the argument by doing the following case analysis:

   (a) (5 points) Show that if the degrees of all nodes in G is at least 2, then G has at least k edges.
   Answer 1a)
   Let the number of edges in G be E. We know that sum of degrees of all nodes in an undirected graph = 2*(number of edges in the graph).
   Hence, 2*E = sum of degrees of all nodes in G.
   Since all nodes have degree >= 2,
   sum of degrees of all nodes >= 2*(number of nodes in G)
   Hence, 2*E >= 2*(k+1)
   So, E >= k+1 and we can say E >= k.
   Hence proven that if the degrees of all nodes in G is at least 2, then G has at least k edges.

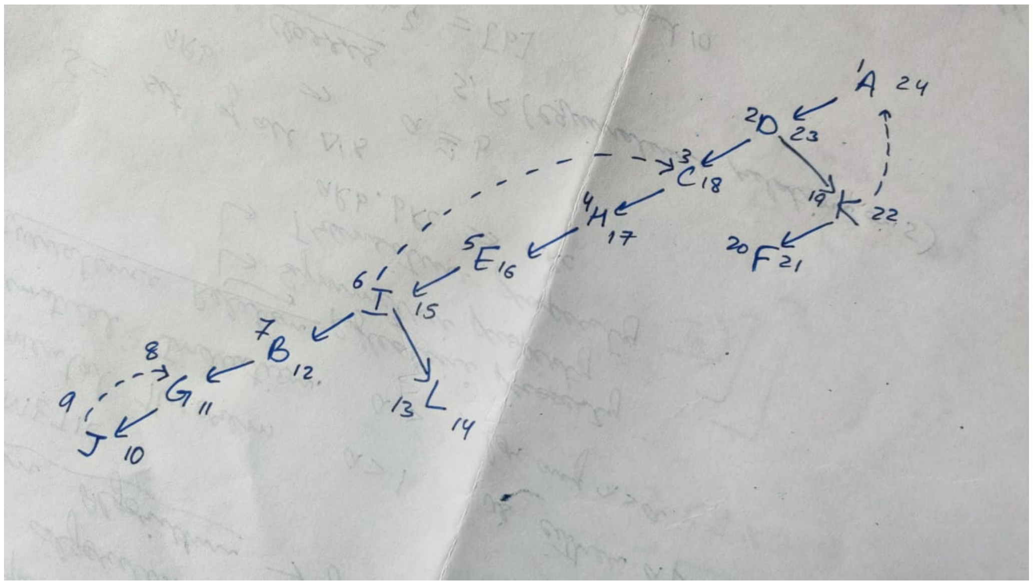
   (b) (5 points) Consider the case where there exists a node v with degree 1 in G. In this case, consider the graph G' obtained from G by removing vertex v and its edge. Now use the induction assumption on G' to conclude that G has at least k edges.
   Answer 1b)
   There is a node v in G such that degree of v is 1. So it must be connected to only one other vertex (say u) in G. Now we remove v from G to form G'. G' has all edges of G except the (u,v) edge. Hence, G' (having k nodes) will still remain connected. Since P(k) holds true according to the induction hypothesis, G' will have minimum (k-1) edges.
   Now to reconstruct G from G', we add node v and the edge (u,v). So G has (k+1) nodes and (k-1)+1 = k edges minimum.
   Hence Proved that if there is node of degree 1 in connected graph G (having total k+1 nodes), it will have minimum k edges
   Hence, P(k+1) holds true under the assumption of P(1), P(2), ... , P(k) being true.
   So, P(n) is true for all n >= 1.

2. Consider the following directed graph and answer the questions that follow:



(a) (1 point) Is the graph a DAG?
NO.
We can spot a cycle A − > D − > K. Hence the graph is not acyclic.

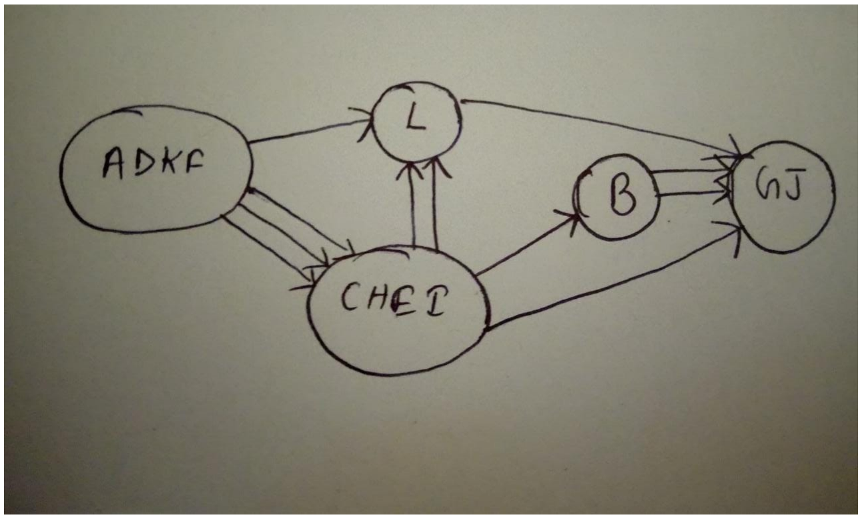
(b) (2 points) How many SCCs does this graph have?
DFS Tree for this graph (also denoting pre and post numbers during traversal):
Arranging the nodes in descending order of post numbers : A D K F C H E I L B G J



Let G' be the transpose graph of G ie the direction of all edges in G are reversed.
Now, to obtain SCC's of G, we can do DFS traversal of G' in order A D K F C H E I L B G J, because that will give us SCC's of G', which are same as that of G (SCC's of a graph are same as that of the transpose graph).
So, the strongly connected componenets obtained are :
1. A D K F
2. C H E I
3. L
4. B
5. G J

This graph has 5 strongly connected components

(c) (1 point) How many source SCCs does this graph have?
Drawing the meta graph of the given graph :



So, from the meta-graph, we can see that there is a single source SCC which is A,D,K,F.

(d) (2 points) What is the distance of node B from the A?
4 units
The SCC containing A can be connected in 2 steps to the SCC containing B.
One more edge is needed from A to D and another from C to I so that A - D - C - I - B forms the shortest path from A to B.
The distance between the node B from node A is 4 units.

(e) (2 points) Suppose we run the DFS algorithm on the graph exploring nodes in alphabetical order. Given this, what is the pre-number of vertex F?
The pre-number of F is 20, as shown in the figure above.

(f) (2 points) Suppose we run the DFS algorithm on the graph exploring nodes in alphabetical order. Given this, what is the post-number of vertex J?
The post-number of J is 10, as shown in the figure above.

3. (20 points) Suppose a degree program consists of n mandatory courses. The *prerequisite graph* G has a node for each course, and an edge from course u to course v if and only if u is a prerequisite for v. Design an algorithm that takes as input the adjacency list of the prerequisite graph G and outputs the minimum number of quarters necessary to complete the program. You may assume that there is no limit on the number of courses a student can take in one quarter. Analyse running time and give proof of correctness of your algorithm.

Answer 3)

Main Idea: We can see that the prerequisite graph G is a DAG, since no cycles can exist in the current scenario (a path from node A to node B means that course A has to be completed before course B. If the presence of a cycle makes a path from B to A too, it will be possible to do both courses). To find the minimum number of quarters needed we need to identify the courses to be done in each quarter. The source nodes of the graph do not have any prerequisites and can all be done in the first quarter.Using this idea, we can remove the courses done in the previous quarters and choose the source nodes of the resulting graph to be done in the current semester, till all the courses are completed.
The algorithm is given as

```
1) Initialize the CurrQuarterList as empty (This will store the list of course nodes to be done in
the current semester)
2) Initialize the array indeg[u] with 0s for all vertices of G and call findIndegree on G
procedure findIndegree(G):
    - for all edges (u,v) in G':
        - indeg[v] = indeg[v] + 1
3) for all vertices in G:
    if(indeg[v] == 0):
        add v to CurrQuarterList
4) Run the procedure findMinQuarters
procedure findMinQuarters(G,CurrQuarterList):
    NumQuarters =0
    if (CurrQuarterList is empty):
        return 0
    while(CurrQuarterList != empty):
        NumQuarters = NumQuarters + 1
        NextQuarterList = empty     \\(stores the list of courses for the next quarter)
        course = eject(CurrQuarterList)
        for all edges (course, v) in G:
            indeg[v] = indeg[v] - 1
            if(indeg[v] == 0):
                add v to NextQuarterList
        CurrQuarterList = NextQuarterList
    return NumQuarters
```

The answer returned by findMinQuarters is the minimum number of quarters required to complete all the courses.

Running Time: The time taken to find indegree of all vertices is O(V+E). In the procedure findMinQuarters is every vertex is present in the CurrQuarterList exactly once and is expanded to its outgoing neighbours. This is O(1+outdegree(v)) which is O(V+E). Therefore the total time complexity is O(V+E).

Proof of Correctness:

Proof by Induction: Induction on the number of nodes in the graph, n.

Base Case: For n=1, the graph has one node with indegree = 0. Our algorithm returns 1 which is the minimum number of quarters needed to complete it. Hence true.

Induction Hypothesis: Assume that the above algorithm computes the minimum number of quarters required correctly for all DAGs with upto k nodes, $k \geq 2$.

Induction Step: Consider a graph G with n = k+1. Since G is a DAG it has atleast one sink node, say v. Let us remove this sink node and the corresponding edges to it from the graph to give G'. G' has k nodes and the Hypothesis holds on it and let the minimum number of quarters in G' be l. Consider the following cases-

Case 1. If v is an outgoing neighbour of a node in the last QuarterList of G'.
This implies that the answer must be l+1. According to the algorithm for graph G, there will be an extra iteration with QuarterList = [v]. This will output the NumQuarters as (l+1).

Case 2. If v is not a neighbour of a node in the last QuarterList of G'.
This implies that the answer must be same as G', i,e l. According to the algorithm for graph G, the vertex

v will be added to the QuarterList after its parent's QuarterList, which cannot be the last QuarterList of G' as stated above. This implies that no new QuarterLists were made in G over G'. Therefore the output NumQuarters will remain l.
Hence proved.

4. A particular video game involves walking along some path in a map that can be represented as a directed graph $G = (V, E)$. At every node in the graph, there is a single bag of coins that can be collected on visiting that node for the first time. The amount of money in the bag at node v is given by $c(v) > 0$. The goal is to find what is the maximum amount of money that you can collect if you start walking from a given node $s \in V$. The path along which you travel need not be a simple path.

   Design an algorithm for this problem. You are given a directed graph $G = (V, E)$ in adjacency list representation and a start node $s \in V$ as input. Also given as input is a matrix C, where $C[u] = c(u)$. Your algorithm should return the maximum amount of money that is possible to collect when starting from s.

   (a) (10 points) Give a linear time algorithm that works for DAG's.

   (b) (10 points) Extend this to a linear time algorithm that works for any directed graph. (*Hint: Consider making use of the meta-graph of the given graph.*)

   Give running time analysis and proof of correctness for both parts.
   Answer 4a)

   Main Idea. Given G is a DAG. DAGs do not contain non-simple paths. Let the start node be s.The variable money[v] for a vertex v stores the maximum amount of money collected from a path from s to v. So the maximum amount of money that can be collected in the whole graph is the max(money[v]) for all vertices v reachable from s. The Algorithm is given as
   (1) Initialize $parent(v)$ = dummy vertex d with c(d) = 0 for all $v \in V$
   (2) Initialize $money(v) = 0$ for all $v \in V$
   (3) Initialize $maxamount = 0$, this will store the maximum amount of money collected out of all possible paths traversed so far,in every iteration
   (3)Run the following procedure $explore(G, v)$ on the source node s.

```
    procedure $explore(G, v)$:
    $money(v) = money(parent(v)) + c(v))$
    $maxamount = max(maxamount, money(v))$
    $for\ all\ outgoing\ neighbours\ u\ of\ v$:
        $if\ (money(parent(u)) < money(v))$:
            $parent(u) = v$
        $explore(G, u)$
```

   - Maximum amount of money collected = $maxamount$

```
procedure **calcmax(G,s):**
    for all v ∈ vertices in G:
        parent[v] = dummyvertex
        money[v] = 0
    maxamount = 0
    money[dummyvertex] = 0
    return explore(G,s)
```

   Running Time Complexity = Time complexity of explore (on node s)
   = O(|V| + |E|)

   Proof of Correctness:-

   Proof by induction on the number of nodes (n) in graph G

Define **P(n)** :- given a DAG with n nodes, calcmax(G,s) returns the max amount of money we can collect starting from any given node s.
Also, for all nodes v ∈ G, money[v] is the maximum possible amount of money which can be collected on the path from s to v (if v is reachable from s).

Claim : P(n) is true for all n >= 1.

Base Case :
P(1) :- G has only 1 node and hence we can run the algorithm only on the single node in G (let's call it s).
explore(G,s) will set money[s] = $0 + C(s)$
maxamount = max(0,money[s]) = C(s)
So the answer returned by the algorithm is C(s) which is logically correct since we cannot move from s to any other node, so the maximum possible amount of money we can collect is C(s)

Induction Hypothesis :-
Let P(1), P(2), ... , P(k) be true. So for any graph G with <= k nodes, the algorithm returns the max amount of money which can be collected starting from any node in G.

Induction step :-
Graph G has k+1 nodes and for any node, say s in G we have to find the maximum money which we can collect starting from s
By checking the outgoing neighbors list of s, there can be 2 cases:-

Case1 : s is a sink node
There are no outgoing edges from s, so inside explore(G,s): money[s] = parent[s] + C(s) = C(s)
maxamount = max(maxamount,money[s]) = C(s)
No further recursive calls since s does not have any outgoing neighbours.
So here we have G with (k+1) nodes and calcmax(G,s) returns C(s) which is the maximum amount of money we can collect starting from s, which is a sink node.
Also, the only vertex reachable from s, that is s itself, money[s] is the exact maximum possible amount of money we can collect from s to s.
Hence, P(k+1) is true in this case.

Case2 : s is not a sink node
From G, we derive G' = G - v where v is a sink node in G (since G is a DAG, there must be atleast one sink node)
Now, G' will also be a DAG.
Say s is a node in G' having total of k nodes.
From Induction Hypothesis we know that calcmax(G',s) returns the max amount of money we can collect starting from any given node s in G'. Also, for all nodes x ∈ G', money[x] is the maximum possible amount of money which can be collected on the path from s to x (if x is reachable from s).
On adding v and the corresponding edges back to G', there will be vertices u_1, u_2, ... , u_m with m >= 1 such that there is an edge from each of them to v in graph G originally.
From Induction Hypothesis we know that money[u_i] is the max amount which can be collected along any path from s to u_i.
So v is going to be visited m times explore(G,u_i) is called. Since v is a sink, no further recursive calls will be made after v.
At the end of the procedure, parent[v] will be u_p such that money[u_p] is maximum among all the u_i's.
So, money[v] = money[u_p] + C(v) and
maxamount = max(maxamount,money[v]).

So, if money[v] is more than the maxamount calculated so far, answer returned by the procedure will be money[v] corresponding to the that path from s to v on which we collected maximum amount of money. Otherwise the answer returned will be money[v'] where v' is some other sink node in G.
So, for all k+1 vertices in G, money[x] is the maximum possible amount of money which can be collected on the path from s to x (if x is reachable from s).
Hence ,P(k+1) is true in this case as well.

Hence, it is proven that P(n) holds true for all n >= 1.
Hence, the correctness of the algorithm is proven.

Time Complexity is O(V+E) since we visit every vertex and every edge recursively.

Answer 4b)
Main Idea. For any general directed Graph we can make use of the meta graph which is a DAG.
So the modified algorithm will be :

```
procedure $explore(G, v)$:
$money(v) = money(parent(v)) + c(v))$
$maxamount = max(maxamount, money(v))$
$for\ all\ outgoing\ neighbours\ u\ of\ v$:
$if\ (money(parent(u)) < money(v))$:
$parent(u) = v$
$explore(G, u)$
- Maximum amount of money collected = $maxamount$
procedure **calcmax(G,s)**:
    Create a meta graph of G and call it M
    M = CreateMetaGraph(G)
    Let the SCC containing s be S
    for all v ∈ nodes in M:
parent[v] = dummyvertex
money[v] = 0
C[v] = sum of C[u] of all u (nodes of G) ∈ v (node of M and an SCC of G)
    maxamount = 0
    money[dummyvertex] = 0
    return explore(M,S)
```

Proof of Correctness :
Proof by construction
We have proven in part a that calling the explore(G,s) routine on any DAG, G returns the maximum amount of money which can be collected in G starting from node s.
Meta graph M of G is also a DAG.
Within each SCC which is a part of M, every node is reachable from every other node. So, we claim that starting at any node, and restricting our traversal to the particular SCC, the max amount which can be collected is the sum of coins placed at every node.
This argument is true because starting from any node in a SCC, we can traverse all other nodes. We might need to traverse some nodes more than once, but since the coins are collected only for the first visit, the amount is added to the money collected only once. Hence, the C[v] of every node v in M is set to be equal to sum of all coins in the corresponding SCC.

Calling explore(M,S) will return the max amount of money which we can collect by traversing M and starting at node S, where S is the SCC in meta graph containing given starting node, s. This amount is

also equal to the max amount which can be collected starting from node s in graph G.
This argument is correct because when we take any 2 vertices s_1 and s_2 in G such that s_2 is reachable from s_1, there can be 2 cases:

1. Both are in same SCC in M
In this case, as written above the max amount collected will be along the simple/ non-simple path from $s-1$ to s_2 where we can visit all vertices in the corresponding SCC atleast once and collect coins from there. We cannot have a path from s_1 to s_2 containing nodes outside the SCC because that will cause M to be cyclic which it is characteristically not.

2. Both are in distinct SCC's in M
Since s_2 is reachable from s_1, the two SCC's say S_1 and S_2 containing s_1 and s_2, there must be a path ((it will be a simple path since M is a DAG)) from S_1 to S_2 in M.
money[S_2] will be equal to the maximum amount which can be collected from travelling from S_1 to S_2 after the complete run of explore(M,S_1).
money[S_2] will also be equal to the maximum amount which can be collected by travelling from any node in S_1 to any node in S_2 since we will visit the maximum possible number of nodes within each SCC on the path and also the explore() routine chooses the path which fetches the max amount.

Running time = Running time time to create meta graph M + Running time of calcmax on M
= O(V+E) + O(V'+E') where V,E are number of nodes, edges in G and V',E' are nodes, edges in M respectively
since V' <= V and E' <= E,
Running Time of this algorithm is **O(V+E)**.

5. Given a directed graph $G = (V, E)$ that is not a strongly connected graph, you have to determine if there exists a pair of vertices $u, v \in V$ such that the graph $G' = (V, E \cup \{(u, v)\})$ is strongly connected. In other words, you have to determine whether there exists a pair of vertices $u, v \in V$ such that adding a directed edge from u to v in G converts it into a strongly connected graph. Design an algorithm for this problem. Your algorithm should output "yes" if such an edge exists and "no" otherwise.
   (a) (10 points) Give a linear time algorithm that works for DAG's.
   (b) (10 points) Extend this to a linear time algorithm that works for any directed graph. (*<u>Hint</u>: Consider making use of the meta-graph of the given graph.*)

Give running time analysis and proof of correctness for both parts. Answer 5a)
Main Idea: We have learnt that a DAG has at least one source and one sink node. So, first of all, we will find the source and sink nodes of our DAG. If there are more than one source node, or more than one sink node, then we say that there exists no edge (u,v), such that after joining the edge, the graph will become strongly connected.
If there exists a single source and a single sink node in our DAG, we will join the edge from our sink node to our source node and check if our graph is strongly connected or not. If yes, we report true, otherwise false.
The algorithm is as follows:
edgeExists(G):

```
- initialize the arrays indeg[u], outdeg[u] with 0s for all vertices
- for all edges (u,v) in G:
    - indeg[v] = indeg[v] + 1
    - outdeg[u] = outdeg[u] + 1
- initialize the source and sink node to null
- for all v in G:
    - check if indegree is 0:
        - if source is null:
            - source = v
        - else:
            - return **false**
    - check if outdegree is 0:
        - if sink is null:
            - sink = v
        - else:
            - return **false**
- add the edge (sink,source)
- check if number of strongly connected components is 1
- if yes:
    - return **true**
- else:
    - return **false**
```

We can prove the correctness of the idea stated above as follows:
Part 1: If there exists an edge in a DAG, which when connected constructs a strongly connected graph, then it is the edge from sink to source.
Let us assume it is some other edge, which when connected, forms a strongly connected graph if possible. But such a case is never possible, because if the source node is not present in the edge or the sink node is not present in the edge or both, then there exists a node in the graph, to which there are no incoming edges, or from which there are no outgoing edges. Then, its safe to say that there exists no path from sink node to any other node, and there exists no path from any other node to the source node, which contradicts the fact that the graph is strongly connected. Hence, proved.

Part 2: DAG having more than one source or more than one sink won't have an edge, which when connected, makes the graph strongly connected.
Proof by contraction: Let us assume there exists a DAG, having more than one source or sink, and there exists a edge, which when connected, makes the graph strongly connected.
From the proof of part 1, it is safe to say that the edge should be from a sink to a source, and one edge can connect one sink to a source at maximum. And, as the graph contains more than one source or sink node, there will be a node present to which there are no paths to any other nodes, or from which there are no paths from any other nodes, which contradicts the fact that the graph is strongly connected. Hence, proved.

Running Time: The running time to find if there are multiple source and sink nodes in the graph is O(V+E). And, after joining the edge from sink to source node, to check the number of strongly connected components, it takes O(V+E), running time. Therefore, the total running time of the algorithm is O(V+E).

Answer 5b)
Main Idea: The idea is to make a meta-graph from our directed graph, which will be a DAG, and then we will apply the idea discussed in 5a, because the nodes of the DAG formed are already strongly connected. If the edge makes the DAG strongly connected, then the whole graph will be strongly connected.

The algorithm for execution is discussed below:
edgeExists(G):

```
    - G' = CreateMetaGraph(G)
    - initialize the arrays indeg[u], outdeg[u] with 0s for all vertices of G'
    - for all edges (u,v) in G':
        - indeg[v] = indeg[v] + 1
        - outdeg[u] = outdeg[u] + 1
- initialize the source and sink node to null
- for all v in G':
    - check if indegree is 0:
        - if source is null:
            - source = v
        - else:
            - return **false**
    - check if outdegree is 0:
        - if sink is null:
            - sink = v
        - else:
            - return **false**
- add the edge (sink,source)
- check if number of strongly connected components of G' is 1
- if yes:
    - return **true**
- else:
    - return **false**
```

We can prove the correctness of the idea stated above as follows:
Part 1: If the meta-graph of a directed graph G is strongly connected, it implies that the graph G is strongly connected.
Proof by contradiction: Let us assume there exists a graph G, such that the meta-graph of G, G' is strongly connected, but the graph G is not strongly connected.
The graph G, not being strongly connected implies there exists at least two vertices u and v such that there does not exist a path from v to u or from u to v. Then, this will certainly imply that the meta-graph of G will have two nodes, one containing u and other containing v, say u' and v'.
Now, since the meta-graph G' is strongly connected, it implies that there exists a path from u' to v' and from v' to u', through edges between nodes (a,b) and (c,d) respectively, a and d being in u', and b and c being in v'.
Since we know that u' and v' are strongly connected, we can say that there exists a path from u to a and d, and from a and d to u. Same could be said for v, b and c. Now, since there is an edge from a to b and from c to d, it implies that there exists an edge from u to v and from v to u. This implies that our graph G is strongly connected, which is a contradiction. Hence, proved.

Part 2: DAG cannot be connected by adding a single edge if it has more than one source node or more sink node. If DAG has a single source and a single sink node, then the graph might be connected only after adding an edge from sink to source.
This is proved in part (a) already.

Running Time: The running time to create the meta-graph is O(V+E), and to find if there are multiple source and sink nodes in the graph is O(V'+E'). And, after joining the edge from sink to source node, to check the number of strongly connected components, it takes O(V'+E'), running time. Therefore, the total running time of the algorithm is O(V+E)+O(V'+E')+O(V'+E')=O(V+E) (V'<V and E'<E).

6. (20 points) Let us call any directed graph $G = (V, E)$ *one-way-connected* iff for all pair of vertices u and v at least one of the following holds:

(a) there is a path from vertex u to v,

(b) there is a path from vertex v to u.

Design an algorithm to check if a given directed graph is one-way-connected. Give running time analysis and proof of correctness of your algorithm.
Answer:
Main Idea: First, thinking of a DAG, we claim that a DAG is not one-way connected if it has two source nodes(as there won't be a path from one source node to another). So, we will keep removing the source node and report false as soon as we discover more than one source node, otherwise true.
Now, for a general directed graph, we can convert the graph into it's meta-graph, which is definitely going to be a DAG. And, as the strongly connected components in the directed graph, are one way connected too, we just need to prove that the meta-graph, which is a DAG, is a one-way connected graph, which we will do using above idea.
The pseudo code is as follows:

```
isOneWayConnected(G):
    - G' = CreateMetaGraph(G)
    - initialize an empty array indeg[u] with 0 for all vertices
    - for all edges (u,v) in G':
        - indeg[v] = indeg[v] + 1
    - initialize source node to null
    - for all v in G':
        - check if its a source, i.e. indegree is 0
            - if source is null:
                - set source node to v
            - else:
                - return false
    - while number of vertices greater than 1:
        - initialize node nextsource to null
        - for each edge in adjacency list of source (source, v):
            - indeg[v] = indeg[v] - 1
            - check if indeg[v]==0:
                - check if nextsource=null:
                    - set nextsource to v
                - else:
                    - return false
        - set source to nextsource
    - return true
```

Our algorithm is based on the correctness of the proposition that a DAG can be one-way connected iff it just has one source node. We can prove the correctness of the algorithm as follows:

Part 1: DAG is one way connected implies it just has one-source node
We can solve this part using contradiction.
Let us assume we have a one-way connected DAG, with two source nodes, say v and v', and the definition of source nodes is that there are no indegrees to v and v'.
This directly implies that, starting from v or v', there is no path to the node v' and v respectively, as none of them has an incoming edge to complete the path, which contradicts the fact that the graph is one-way connected.
Therefore, a DAG is one-way connected if it has just one source node.
Part 2: P(n) : The algorithm tells us correctly if our graph is one-way connected or not.
We can solve this part using induction.
Basic Step: P(1) gives true, which is true because a single vertex can be said as one-way connected graph, in the same was as strongly connected graph.

Inductive step: We assume that P(1), P(2)....P(k) holds.
We will show that P(k+1) is true. According to our algorithm, we check if given graph has more than one source node, if it has, then false according to part 1. If not, we remove one source node, and the remaining graph is of k nodes, and our hypothesis was that P(k) is true, which implies P(k+1) is true. Hence, proved.

Running time: The runtime to get a meta-graph is O(V+E). Array initialize is O(V), indegree assignment is O(E), finding initial source is O(V). Now, for that while loop, we can see that overall the loop just goes over all the vertices one by one, and keeps removing the edges. In the worst case scenario, which is when the algorithm returns true, the loop keeps removing all the vertices and edges, until only one vertex remains, which means that the loop is O(V+E). Therefore, our net running time of the algorithm is O(V+E).