1.

(a)

Possible:

Here, there are no directed paths of length 2 or higher because all directed paths have the length of 1: (1 to 4), (1 to 2), (3 to 2), (3 to 4), (5 to 4)

This was possible because there were no two or more nodes that had a path between node with only indegree paths and node with only outdegree paths. The nodes 2 and 4 consisted of only indegree paths. If the nodes 4 and 2 are connected, it would break the case and there would exist a direct path of length 2. This is because they are both nodes with only indegree paths. If there is a directed path between them, one of nodes 2 and 4 must have an outdegree path, which allows a directed path of length 2 to occur. Similarly, nodes 1,3, and 5 are all nodes with only outdegree paths. If any of these three nodes were connected, it would cause directed paths of length 2 to exist.

In a big picture, generally, there are no directed paths of length 2 or higher if there does not exist a cycle between odd number of nodes. I will prove this by proof by contradiction. Let’s say that odd number of nodes that form a cycle can create a scenario where there are no directed paths of length 2 or higher. First of all, cycle in a graph means that you can return back to the initial node after following the paths given. If there are odd number of nodes (2n+1 nodes) in a cycle, it means that there must exist at least two paths for each node in order to fulfill the definition of cycle (since you have to enter from one path and leave by another path). If there must be at least two paths for each node, we need to balance the number of nodes that only have outdegree paths and indegree paths so that they can fulfill the requirement. Let’s say there are n nodes with only outdegree paths and n+1 nodes with only indegree paths. In this case, the n nodes that only have outdegree paths should only be connected to the n+1 nodes with only indegree paths to have no directed path of length 2 or higher to exist. In order for the nodes to be a cycle, all nodes have to be connected to at least two nodes, which means that n+1 nodes with only indegree paths should have at least 2n+2 edges. There are n nodes with outdegree paths so if you think in an undirected path, the path from node A to B is same as B to A, so 2n edges can be connected with the n nodes with only outdegree paths. However, there exists 2 edges that must exist. These edges are edges that must be connected from 2 nodes with only indegree paths from the n+1 nodes. If two nodes with indegree paths are connected, at least one of them is no longer a node with only indegree path when you add direction, which breaks the definition. Similarily, if you declare n+1 nodes with only outdegree paths and n nodes with only indegree paths, two of the n+1 nodes with only outdegree paths must be connected to each other, which makes one of them to include an indegree path, breaking the definition. Therefore, there cannot exist no directed paths of length 2 or higher within a cycle of odd number of nodes.

Impossible:

Here, there is a directed path of length 2. (3 to 1 to 2), (3 to 1 to 4).

This case was impossible due to the cycle of odd number of nodes. In this graph, if you take out the directions and view it in an undirected graph, there exists a cycle between nodes 1,3, and 4 and nodes 1,2, and 3, which allows a directed path of length 2 or more to occur. Specifically, in this example, we only added one path between nodes 1 and 3 from the previous possible example. However, since two nodes that should only have outdegree paths are connected, one of them must break the law and should include an indegree path, which causes a directed path of length 2 to occur.

(b)

Pseudocode:

possibleOrImpossible(undirectedGraph G):

node s 🡨 choose any node to begin with from G

Number 🡨 1

Queue q 🡨 (s, Number)

while q is not empty:

(node, Number) = Q.dequeue()

for neigh in neighbors[node]:

if neigh.node exists in Q:

if neigh.Number equals Number:

return “Impossible”

else:

continue

if (Number equals 1):

neigh.parent 🡨 node

else:

node.parent 🡨 neigh

if (Number equals 1):

Number 🡨 2

else:

Number 🡨 1

q.enqueue(neigh, Number)

return “Possible”

Explanation of the pseudocode:

Here, I begin with a random node from the tree and give a variable called Number the value of 1. Then, I insert the node and the number with it to the queue. While the queue is not empty, it checks for neighbors, which are nodes that only require path 1 between the nodes. If the neighbor already exists inside the Queue and the neighbor’s number equals to the number of the node, it means that the directed paths of length 2 or higher occurs, so it returns impossible. If the neighbor already exists inside the Queue but does not have the number variable match, then we simply skip that case by saying “continue.” However, in the other case, it moves on to the next part. If the number assigned equals to 1, node becomes neigh’s parent. If not, neigh is the parent of node. This if-else statement allows to give directions from the undirected graph G. Then, if the number at the node given was 1, it changes to 2 but if the number was not 1, it changes back to 1. Afterwards, you insert the node neighbor with the changed number to the queue. If this while loop ends, it means that there are no directed paths of length 2 so it returns “possible.”

Proof of the pseudocode:

From part (a), I explained that the case of directed paths of length 2 or higher occurs if the undirected graph that the user provides is an undirected graph in which any part of the undirected graph, there exists a cycle between any odd number of nodes. For example, if there are five nodes a-e, and they form a cycle, that undirected graph must have directed paths of length 2 or higher when you convert it into directed graph.

We first start from a random node s and put it into queue q. The variable Number simply refers to whether this current node is a node with indegree paths or outdegree paths. If the Number is set to 1, it means that the node with Number 1 is a parent node, which only consists of indegree paths and if the Number is set to 2 (not 1), it is a child node, which only consists of outdegree paths. Then, until the queue is empty, we run the following code inside the while loop. Inside the while loop, we take the first priority node out with the number and check the neighbors. If the neighbor already exists in queue, it is fine. However, if the neighbor that exists in queue has the same number as the node, there is a problem. As explained before, the number simply tells whether the node consists of only outdegree paths or indegree paths. If two nodes that are neighbors have same number values, it means that both of them either contain only indegree paths or outdegree paths. However, if they are neighbors that have directed path, one of them should point towards the other node, meaning that one should have only an indegree path and other should have an outdegree path. Therefore, if they have the same number values, it means that no undirected path of length 2 or higher is impossible, so we return “impossible.” In other words, lets say Nodes A and B are neighbors to each other but both contain outdegree paths only in order to match the requirement. However, if they are neighbors in a directed graph, one of them should point at each other. For example, A should point at B. Then, node A now has an indegree path, which breaks the requirement of this problem.

However, if the number is not the same, we skip that neighbor value by continue and move on to the next neighbor. (we skip by using “continue” because we do not need to touch the node that already exists inside the queue) If the neighbor node does not exist inside queue, we move on and depending on the number, set the neighbor’s parent to node or set node’s parent to neighbor. After doing so, we have to change the value of the number to continue with the pattern of parent and child. If the neighbor’s node is the parent of the node, then the neighbor’s node should be the parent of all of its neighbor’s node to continue with the pattern of keeping only indegree paths. Similarily, if the neighbor’s node is the child of the node, then the neighbor’s node should be the child of all of its neighbor’s node to continue with the pattern of keeping outdegree paths. Therefore, we need to adjust the value of variable Number for every neighbor nodes. Once the while loop runs until the queue is empty, the task is done and since there were no problems regarding the requirement, we return “Possible,” indicating that there are no directed paths of length 2 or more.

Runtime/Space:

The setting of variables are just constant time c. I have a while loop that runs until the queue is empty, which is O(n) since the queue runs for every node in the graph. Now, inside the queue, there are again constant runtime values c, which we can skip in computing big O. There is a for loop that checks each neighbor by going inside the adjacency list, which depends on O(degree(node)). Since we cannot specify the degree of the node, we leave it as it is. Therefore, the runtime is O(n\*maxDegreeOfN).

The space it requires is O(m+n) where m is the number of edges and n is the number of nodes, according to the space required for the adjacency list from the textbook.

Source: Even though putting the number to specify the direction of the path was my idea, I generally got the idea of how to code from the lecture on Breadth Search Algorithm that we went over in class that calculated the distance from the initial node. It is from slide 1\_30\_BFS.pdf from piazza that was published on January 30th, 2020 by Professor Dora. Similarly, I calculated the runtime by referring back to the slides.

2.

(a)

In this problem, every node is basically one piece of work that makes the skyscraper. In other words, the job of installing windows, painting the walls, installing doors, and etc. are all nodes. The edges, which connect the nodes so that there exists a path, are simply connected in a way so that once step 1 is completed, you can follow the path to complete the next step. The important function required in the path is the direction. It is impossible to build windows before you build the walls. Therefore, the directions indicate the order of the events. In addition, if two jobs can be done at the same time, there should not exist a path between them since no direction is require among them if two tasks can be done simultaneously. However, just like the situation where you need to have the walls built to have the windows built, building walls node can have an arrow pointing at the building windows node. Once all jobs of the child node are completed (by completed, it means that the task written at the node is done and the direction path to the parent node was done), then the parent node can successfully complete the task written on the node. In summary, nodes are the tasks, edges connect the nodes(tasks) so that you can start other task once completing one, and the directions tell the order of the tasks that should be done.

(b)

Pseudocode:

currentIndegreeOfNodes(AdjacentListOfDirectedGraph list):

length 🡨 number of nodes in list

randomNumber 🡨 generate random integer from 0 to length – 1

for each adjacent edge in list[randomNumber]:

remove edge (u, randomNode) #u here stands for neighbor nodes

remove list[randomNumber] # remove the actual node

returnArray 🡨 array of size length - 1

for each node v in list:

Consider each edge (v, neighbor node of v)

returnArray[neighbor node of v] 🡨 returnArray[neighbor node of v] + 1

return returnArray

Explanation of Pseudocode:

Here, I began by declaring a variable length that includes the number of nodes in the directed graph and randomNumber that generates a random integer from 0 to length -1 (since list is based off of index, I need to subtract 1 from length). The question specifically said we are removing nodes and adjacent edges, so I let the random generator decide which node to delete. Then, the for loop deletes all the neighbor edges of the node that is located at the index of randomNumber of the list. After removing the neighbor edges, now we delete the actual node itself. Then, we make an empty array of size length-1 (again, arrays are based off of index, so 1 should be subtracted from length). Then, for each node v in list, take each edge that makes a path to its neighbor node. Within the index of the neighbor node, you add 1 to it to. After the for loop ends, return the array.

Proof of Pseudocode:

Here, I first choose a random node that exists in the adjacent list form of directed graph and delete all the edges and the node itself. After the node and the edges are deleted, I look at each node starting from the top of the adjacent list. In adjacent list of directed graph, each node contains the information of where its outdegree path is headed to.

0 (1:5), (4:9), (7:8)

1 (2:12), (3:15), (7:4)

2 (3:3), (6:11)

3 (6:9)

4 (5:2), (6:20), (7:5)

5 (2:1), (6:13)

6

7 (2:7), (5:6)

This is an example of adjacent list from the lecture slide 02\_04\_graphs.pdf. Here it tells that from node 0, there is path to node 1, 4, and 9. (The number after the colon is simply the weight of the edge, which is unimportant in this question). Therefore, since each node tells to which node their outdegree paths are headed to, it means that the node connected has an indegree of 1. Therefore, at the index of the array that I created prior to the for loop, I add one to it, indicating that there is one more indegree of the node. After going through each node in the list, we simply return the array that we created, which tells the number of indegrees of each node after deleting nodes and its adjacent edges.

The important task here is to prove that the runtime is O(m+n) which I will explain below at the runtime section. This section was just proving that my pseudocode works.

Runtime:

Setting the variables at the beginning is all constant time c, which is meaningless in calculating big oh notation. Therefore, the portions that affect the big oh notation are the two for loops that I run throughout my program. The first for loop simply looked at the edges of a random node that we wanted to delete. Since it is a deletion of edges of one node, it can also be represented as a constant time. The second for loop looked at all the nodes. According to the textbook (page 91), if node u has few neighbors, nodes can take less time than O(n) time and we looked at all the nodes, which is O(sigma addition of all nodes). By definition, sigma addition of all nodes that take less than O(n) time equals 2m, which is O(2m), which is O(m). Finally, we took O(n) time to change the array that we wanted to return. Therefore O(m) + O(n) = O(m+n).

(c)

Pseudocode:

orderOfTask(AdjacentListOfDirectedGraph list):

length 🡨 number of nodes in the list

array 🡨 an array of size length – 1

answerArray 🡨 an arry of size length – 1

number 🡨 0

for each node u in list:

Consider each edge (v, neighbor node of v)

array[neighbor node of v] 🡨 array[neighbor node of v] + 1

while(answerArray[length-1] equals null):

for each x in len(array) - 1:

if (array[x] == 0):

if(array[x] already exists in answerArray):

continue #skip this iteration

answerArray[number] 🡨 list[x]

number 🡨 number + 1

for each adjacent edge in list[x]:

remove edge (u, x) #u here stands for neighbor nodes array[u] 🡨 array[u] - 1

return answerArray

Explanation of Pseudocode:

First, we get the length of the list and store it inn the variable length and create two arrays of same size, one is the array that will be the answer that will be returned and one is the array that will record the number of indegrees to each node. Then, there is also a variable called number that will be used to move the index of the answerArray. First, we fill the array that records the number of indegrees to each node. Afterwards, in the while loop that runs until the answerArray is full, we look at each index of the array that records the number of indegrees and for index that is 0, unless that index already exists in the answer array, we put it at the answer array and add 1 to the index of the answer array so that we do not keep putting the node at the same index (and create infinite loop). Then, we delete all the edges of the node that we inserted in the answerArray and erase 1 to all values in the array that were connected with the node that we inserted in the answerArray. Finally, we then return the answer Array after the while loop is over.

Proof of Pseudocode:

The explanation of variables is explained above. As explained in part (b), the adjacent list form of directed graph contains the information regarding the outdegree path of each node. We first take the node that has zero indegrees, which means that the task at the node can be done without anything done prior to the task. In other words, there are no requirements for that node. After the node is completed in day 0, we have to find the next work that can be done after the first task is done. To find this, we have to delete the node and the edges connected with the node. Once we delete them, we can find the next task that does not require anything else but the first task, which is finding the node of zero indegrees. Even though I wrote my function with loops, this concept is similar to the idea of recursion. Basically, you repeat this step until there are no more edges for the entire list, which means that all nodes are completed and are inserted in the answerArray that will be returned. However, in between the step of finding each node with zero indegrees, we need to delete the number of indegrees that is stored in the variable array. This is done while you delete the edge. When the edges of a node that consists of zero indegrees and at least one outdegree disappear, we can simply subtract 1 from the array of which the index number of the array was connected with the node that its edges deleted. For example, if node A has direction towards node B and if the edge that connects node A to node B disappears, the indegree of B should decrease by 1. Such subtraction can be done because it we know that node A (or the node A that had its edges deleted) only has outdegrees and if node A had outdegrees, the node that it was connected to must be indegrees of those nodes. After we repeat these steps until the while loop breaks, our job is done.

Runtime:

Assigning variables and its values all take constant time c, which are unsignificant when calculating big oh notation. There are two loops, including the first for loop and the nested loop of while loop and for loop. The first for loop is simply from the pseudocode from part(b) of my homework assignment, which I declared was O(m+n). Now, at the second loop (the nested loop), the while loop runs until all indexes of answerArray is not 0 or not null. AnswerArray has the size equal to the number of nodes, giving O(n). Inside the while loop, there exists a for loop, which runs until you check every element of array, is also O(n) since the array has same size as answerArray, which is O(n). Inside the for loop, there is another for loop that deletes the edges and decreases the indegree values by 1 in array. Since it depends on the number of edges that the node that has zero indegrees has, it is simply a constant. So, the final big oh is O(m+n) + O(n) \* O(n) = O(m+n) + O(n^2) = O(n^2 + m + n) = O(n^2 + m).