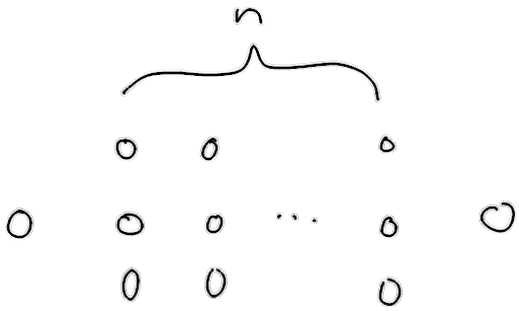
Table of Contents

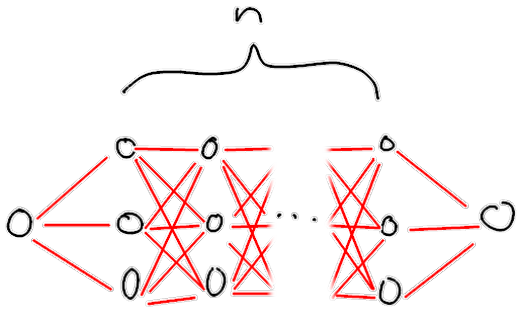
Question 2.

Let $n \in \mathbb{N}$ be arbitrary, choose $c = 3$, $k = 2$.

First, add node s and t
In between s and t, add $3n$ nodes in a block with length n, width 3.
This means we have $3n + 2$ nodes.



For every node in a column, connect it to every node in the next column with weight x.



Every node has 3 choices (except for the 2nd last column of nodes). Including the first node, this will mean there is

$$\overbrace{3 \times 3 \times \ldots \times 3}^{n} = 3^n$$

paths from s to t, all with weight $n \times x$. This means they are all the shortest path from s to t.

Since n is arbitrary, this means that for any $n \in \mathbb{N}$, there is an undirected graph of $3n + 2$ nodes with 3^n shortest path from the left-most to right-most node. ■

Question 3.

Suppose to the contrary that given a graph G with n verticies such that
for every $v \in V$, $\mathbf{deg}(v) \geq \frac{n}{2}$ and G has > 1 connected component.

Then G can be split into subgraphs $S_1, \cdots, S_k$ such that they are all 1 connected component.

Since all nodes have $\mathbf{deg}(v) \geq \frac{n}{2}$, then for all subgraphs, it must have $|S_i| \geq \frac{n}{2} + 1$ nodes.
This is because for a node to have $\mathbf{deg}(v) \geq \frac{n}{2}$, there must be that node, along with at least $\frac{n}{2}$ other nodes.

However, there are only n nodes in the graph G. It is not possible for all subgraphs to have more than $\frac{n}{2} + 1$ if we have more than 1 subgraph.
This is because if we have ≥ 2 subgraphs, then

$$\begin{aligned} |G| &= |S_1| + |S_2| + \ldots + |S_k| && \text{[by given]} \\ &\geq 2\left(\frac{n}{2} + 1\right) && \text{[at least 2 subgraphs]} \\ &= n + 2 && \text{[by algebra]} \end{aligned}$$

Which is a contradiction! There are only n nodes in the graph G.

$\therefore$ The supposition is wrong, and so G must be one connect component. ■

Question 4.

(a)

In order to check if there is a cycle in a directed graph, I would use DFS with backtracking to visit all nodes. If a node has no children, it would return. Else, add the current node to the stack, then search for more children. If the current node is already in the stack, that means there is a cycle.

```
bool hasCycle = False
Stack stack = {}

def traverse(Node root):
    if (root in stack): # cycle is found
        hasCycle = True
        break

    stack.push(node)

    for node in outgoingNodes: # call traverse on all children
        traverse(node)

    stack.pop()

traverse(root)
return hasCycle
```

The time complexity is $\mathcal{O}(V^2 + E)$. This is because there could be max V elements in the stack, and it takes $\mathcal{O}(|\text{Stack}|)$ to search the stack. We need to search the stack after every node, so we have $\mathcal{O}(V^2)$. We also have $\mathcal{O}(E)$ since we we need to call **`traverse`** on every edge.

In terms of space complexity, it would need to store the stack which could contain all nodes. So it would be $\mathcal{O}(V)$

(b)

In order to get a valid ordering of the verticies, I would just use the same algorithm as (a). However, I would also use a queue to store the valid ordering of the verticies. For every recursive call, I would add to the queue. If a cycle is found, I would then start deleting nodes from the bottom until I get to the duplicated node. That way, It would return a valid cycle instead.

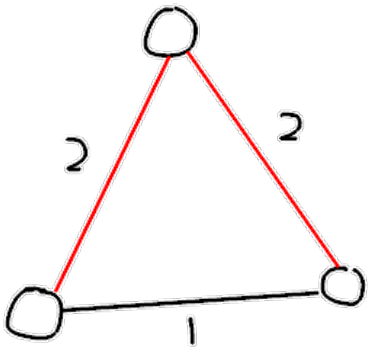
```
Stack stack = {}
Queue queue = {}

def traverse(Node root):
    if (root in stack): # cycle is found
        for node in queue: # remove elements until cyclic node is found
            if node != root:
                stack.remove(node)
            else:
                queue.enqueue(node)
                return

    stack.push(node)

    if (queue.first != queue.last): # add to queue if it's not a cycle
        queue.enqueue(node)

    for node in outgoingNodes:
        traverse(node)

    stack.pop()

traverse(root)
return queue
```

In terms of time complexity, it will be the same as (a). This is because it's the same algorithm, except we have a queue as well. **enqueue** takes $\mathcal{O}(1)$, and removing elements will take max of $\mathcal{O}(|\text{Queue}|)$. Both are lower than $\mathcal{O}(V^2)$, so the final complexity is $\mathcal{O}(V^2 + E)$

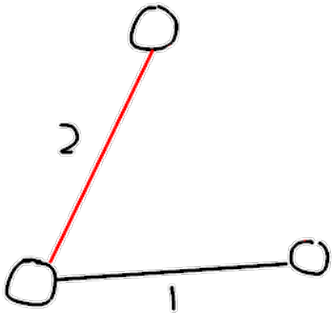
For space complexity, we will have both a queue and a stack with max V elements. Therefore, the space complexity is $\mathcal{O}(V)$.

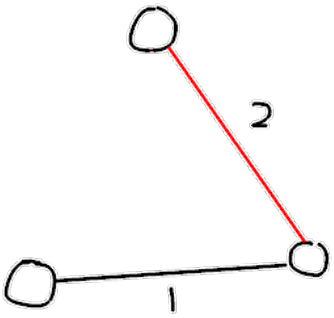
Question 5.

This is false. Let G be the following graph.



Let $T \subset E$ be the red edges.
T is a spanning tree because it connects all of the nodes together.
The two red edges also belong to the two spanning trees below.





However, T is not a minimum-cost spanning tree since the total weight of the edges is 4. The minimum-cost spanning tree for G has weight 3.