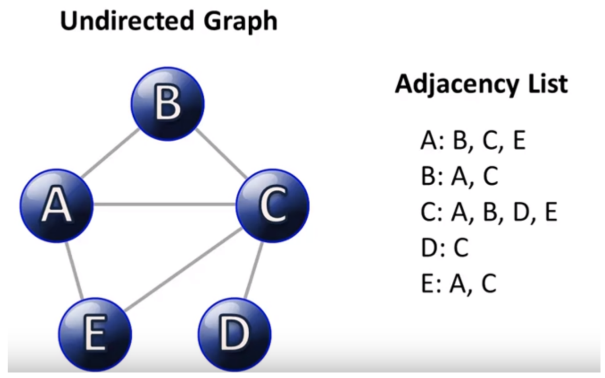
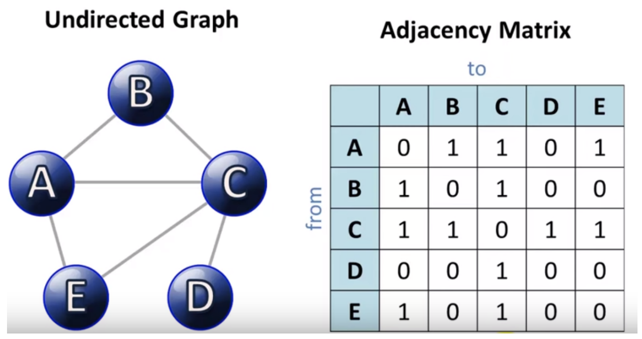
Question 1.
Given an adjacency-list representation of a directed graph, where each vertex maintains an array of its outgoing edges (but *not* its incoming edges), how long does it take, in the worst case, to compute the in-degree of a given vertex? As usual, we use n and m to denote the number of vertices and edges, respectively, of the given graph. Also, let k denote the maximum in-degree of a vertex. (Recall that the in-degree of a vertex is the number of edges that enter it.)
The in-degree of a vertex is the number of edges that enter it, therefore one must count **all** the edges in the graph, otherwise there is a risk that an edge not counted contributed to the in-degree for the vertex. If the graph has m edges, then the lower bound is O(m). There is also an upper bound of O(m), because when only need to read all the edges (m) and keep track of the in-degree count.
O(m)

Question 2.
Consider the following problem: given an undirected graph G with n vertices and m edges, and two vertices s and t, does there exist at least one s-t path?
If G is given in its adjacency list representation, then the above problem can be solved in $O(m + n)$ time, using BFS or DFS. (Make sure you see why this is true.)
Suppose instead that G is given in its adjacency *matrix* representation. What running time is required, in the worst case, to solve the computational problem stated above? (Assume that GG has no parallel edges.)
Check out the following adjacency list then matrix.



A: B, C, E
B: A, C
C: A, B, D, E
D: C
E: A, C



	A	B	C	D	E
A	0	1	1	0	1
B	1	0	1	0	0
C	1	1	0	1	1
D	0	0	1	0	0
E	1	0	1	0	0

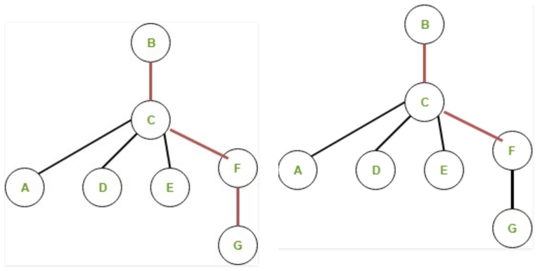
Say we want to find the edges adjacent to the vertex A. To do this we have to traverse the whole array of nodes (A,B,C,D,E) which is of length n. Upper bound would be to use this matrix and build an adjacency list (with the edges), then use BFS or DFS with this.
$O(n^2)$

Question 3.
This problem explores the relationship between two definitions about graph distances. In this problem, we consider only graphs that are undirected and connected. The diameter of a graph is the maximum, over all choices of vertices s and t, of the shortest-path distance between s and t. (Recall the shortest-path distance between s and t is the fewest number of edges in an s-t path.)

Next, for a vertex s, let $l(s)$ denote the maximum, over all vertices t, of the shortest-path distance between s and t. The radius of a graph is the minimum of $l(s)$ over all choices of the vertex s.
Which of the following inequalities always hold (i.e., in every undirected connected graph) for the radius r and the diameter d? [Select all that apply.]
Distance: the distance between two vertices u and v in a graph G is is defined as the length of a shortest path from u to v and is denoted by $d(u,v)$.
Some distance rules:

1. $d(u,v) \geq 0$ and if $d(u,v) = 0 \Leftrightarrow u = v$
2. $d(u,v) = d(v,u)$
3. $d(u,w) \leq d(u,v) + d(v,w)$

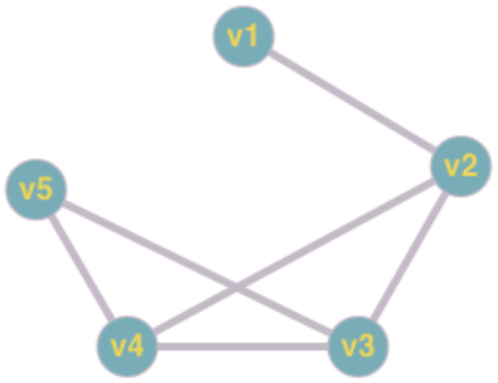
Eccentricity: The maximum distance between a given vertex v to any other vertex u in the graph. (recall distance is shortest path).
$e(v) = max\{d(u,v) \mid u,v \in G\}$



Diameter: Maximum eccentricity. →
$diam(G) = max\{e(v) \mid v \in G\}$

Radius: Minimum eccentricity. →
$rad(G) = min\{e(v) \mid v \in G\}$

Example of eccentricity:



v	e(v)
v1	3
v2	2
v3	2
v4	2
v5	3

diam(G) = 3, rad(G) = 2

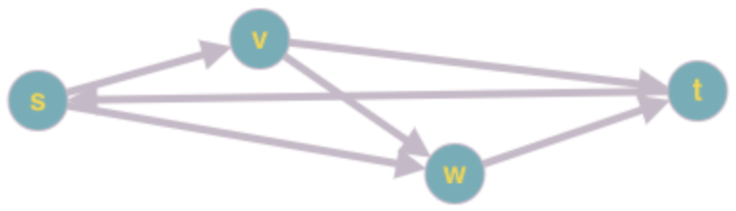
Center: The set of all such vertices for which $e(v) = rad(G)$ make up the center of G.
Periphery: The set of all such vertices for which $e(v) = diam(G)$ make up the periphery of G.

We can prove $rad(G) \leq diam(G) \leq 2 * rad(G)$
For the $diam(G) \leq 2 * rad(G)$portion, consider two vertices u and v on the periphery. Consider the vertex w in the center. Then by the distance law,

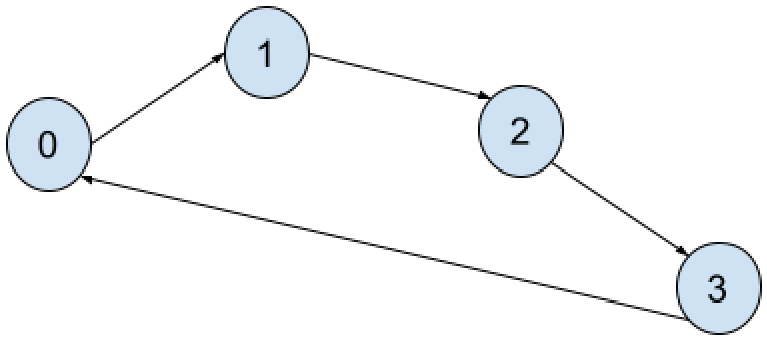
$d(u, v) \leq d(u, w) + d(w, v)$
$\leq 2 * e(w)$
So, $diam(G) \leq 2 * rad(G)$
$r \leq d, r \geq d/2$

Question 4.
Consider our algorithm for computing a topological ordering that is based on depth-first search (i.e., NOT the "straightforward solution"). Suppose we run this algorithm on a graph G that is NOT directed acyclic. Obviously it won't compute a topological order (since none exist). Does it compute an ordering that minimizes the number of edges that go backward?

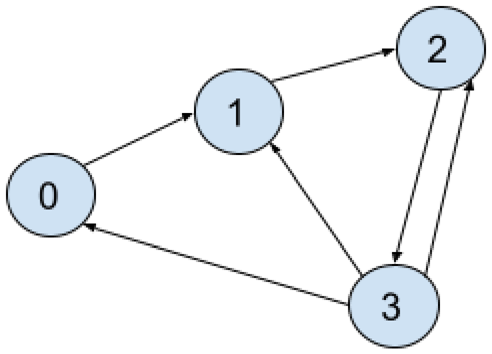
For example, consider the four-node graph with the six directed edges (s,v),(s,w),(v,w),(v,t),(w,t),(t,s). Suppose the vertices are ordered s,v,w,t. Then there is one backwards arc, the (t,s) arc. No ordering of the vertices has zero backwards arcs, and some have more than one.
The graph will look like the following:



By running DFS from vertex *s*, there is only one backwards edge *(t, s)*. But if we run DFS from t, the ordering is w, v, s, t with all the edges pointing backward.
Another example:



If we start DFS from 0, we get the order 0 → 1 → 2 → 3 , with ONE backward arc (3 → 0), which is optimal!
But consider:



If we DFS from 0, we get 0 → 1 → 2 → 3 with THREE backwards arcs (3 → 0, 3 → 1, 3 → 2)
But if we use the order 3 → 0 → 1 → 2, then we only have ONE backward arc (2 → 3)
Sometimes yes, sometimes no

Question 5.
On adding one extra edge to a directed graph G, the number of strongly connected components...?
Consider the directed path 1 → 2 → 3 → 4; there are four strongly connected components, one for each vertex. If we add an edge 4 → 1, then the four strongly connected components merge into one. On the other hand, adding edge 1 → 2 doesn't change anything.
…could remain the same (for some graphs G)