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.)

   (a) $\theta(m)$

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**. What running time is required, in the worst case, to solve the computational problem stated above? (Assume that G has no parallel edges.)

   (a) **Answer:** $\theta(n^2)$

   Explanation:

   Quick Summary: When you run BFS/DFS on a list, you touch each vertex once and each edge at most twice $\longrightarrow$ time $O(m+n)$. Below is the detailed breakdown of why this is true:

   1. The "Contact List" Analogy
   Imagine the Adjacency List as a physical address book.

   - **Vertices (n):** The names of people in the book (Alice, Bob, Charlie...).
   - **Edges (m):** The phone numbers written next to each name.

   In an undirected graph, if Alice is friends with Bob, Bob's number is next to Alice's name, *and* Alice's number is next to Bob's name.

   2. Tracing the Algorithm (The "Touch")
   When BFS/DFS visits a vertex u, it iterates through neighbors:

```
for neighbor in Adj[u]:
    if neighbor is not visited:
        visit(neighbor)
```

   This is where the "touching" happens:

   - **Touch #1:** When processing **Alice**, we scan her list and see "Bob". We check if he is visited. We have "touched" edge (Alice, Bob) once.
   - **Touch #2:** Later, when processing **Bob**, we scan his list and see "Alice". We check her status. We have "touched" edge (Alice, Bob) a second time.

   Since the edge exists in only two places in memory, the code cannot iterate over it more than twice.

   3. The Mathematical Proof

   The total work is the sum of the degrees of all vertices.

   By the Handshaking Lemma, $\sum_{v \in V} \deg(v) = 2m$. Thus, Total Time = $O(n$ for vertices$) + O(2m$ for edges$) \longrightarrow O(n+m)$.

Regarding the Adjacency Matrix (The actual question):
For the lower bound, observe that you might need to look at every entry of the adjacency matrix (e.g., if it has only one "1" and the rest are zeroes). One easy way to prove the upper bound is to first build an adjacency list representation (in $\theta(n^2)$ time, with a single scan over the given adjacency matrix) and then run BFS or DFS as in the video lectures.

The adjacency list representation yields asymptotically faster BFS/DFS on sparse graphs, since even in the worst case

$$\Theta(m+n) = \Theta(n+n) = \Theta(n) \ < \ \Theta(n^2),$$

where the $\Theta(n^2)$ bound corresponds to scanning all rows in an $n \times n$ adjacency matrix.

Quick summary:

i. **Adjacency List:** BFS/DFS = $\Theta(m+n)$
ii. **Adjacency Matrix:** BFS/DFS = $\Theta(n^2)$

3. 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?

Relationship between Radius (r) and Diameter (d)

Definitions:

- Let d be the **diameter** of the graph: $d = \max_{s,t} \text{dist}(s,t)$.
- Let $l(s)$ be the maximum distance from vertex s to any other vertex: $l(s) = \max_t \text{dist}(s,t)$.
- Let r be the **radius** of the graph: $r = \min_s l(s)$.

Claim 1: $r \leq d$

Proof: By definition, $l(s)$ represents the distance from s to its farthest node. The radius r is simply the value $l(c)$ for some central node c. Since the diameter d is the maximum distance between *any* pair of nodes in the entire graph, no single shortest path (including the one defining r) can exceed d.

$$r = \min_s(\max_t \text{dist}(s,t)) \leq \max_{s,t} \text{dist}(s,t) = d$$

Thus, $\boxed{r \leq d}$.

Claim 2: $r \geq d/2$

Proof: Let u and v be the endpoints of a diametral path, such that $\text{dist}(u,v) = d$. Let c be a center vertex of the graph such that $l(c) = r$.

By the definition of radius, the distance from the center c to any node is at most r. Therefore:

$$\text{dist}(c,u) \leq r \quad \text{and} \quad \text{dist}(c,v) \leq r$$

By the Triangle Inequality, the shortest path between u and v cannot be longer than the path from u to c plus the path from c to v:

$$\text{dist}(u,v) \leq \text{dist}(u,c) + \text{dist}(c,v)$$

Substituting the known values:

$$d \leq r + r$$

$$d \leq 2r \implies \boxed{r \geq d/2}$$

Conclusion: The inequalities that always hold are $r \leq d$ and $r \geq d/2$.

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.

(a) **Answer:** Sometimes yes, sometimes no

Explanation: The DFS-based algorithm is **not** guaranteed to minimize the number of backward edges. There are two main reasons for this:

i. **Computational Complexity Argument:** The problem of finding an ordering that minimizes the number of backward edges is equivalent to the **Minimum Feedback Arc Set** problem. This problem is known to be **NP-hard**. Since the DFS-based topological sort runs in linear time $O(m+n)$, it cannot possibly solve an NP-hard problem optimally in all cases (unless $P = NP$).

ii. **Good example where DFS suceeds**
Consider a simple directed graph with edges: $(A, B), (B, C), (C, A)$.

- **Optimal Ordering:** The ordering A, B, C results in exactly **1 backward arc** (C, A).
- **DFS Run:** Suppose we run DFS starting at vertex A.
  - DFS visits $A \to B \to C$.
  - From C, the only neighbor is A, which is already on the recursion stack (gray).
  - DFS backtracks. The finish times would be ordered: C finishes first, then B, then A.
  - The resulting topological ordering (reverse finish times) is: $\mathbf{A}, \mathbf{B}, \mathbf{C}$.

  Let us count the backward arcs in this specific DFS-generated ordering (A, B, C):
  - (C, A): C comes after A. **Backward**.
  - $(A, B), (B, C)$: Both forward.

  This ordering has **1 backward arc**, which matches the optimal. In this case, DFS succeeded in finding the optimal ordering.

iii. **Counter-Example (using the graph provided):**
Consider the graph G with edges $(s, v), (s, w), (v, w), (v, t), (w, t), (t, s)$.

- **Optimal Ordering:** As stated in the problem, the ordering s, v, w, t results in exactly **1 backward arc** (t, s).
- **A "Bad" DFS Run:** Suppose we run DFS starting at vertex t.
  - DFS visits $t \to s \to v \to w$.
  - From w, the only neighbor is t, which is already on the recursion stack (gray).
  - DFS backtracks. The finish times would be ordered: w finishes first, then v, then s, then t.
  - The resulting topological ordering (reverse finish times) is: $\mathbf{t}, \mathbf{s}, \mathbf{v}, \mathbf{w}$.

  Let us count the backward arcs in this specific DFS-generated ordering (t, s, v, w):
  - (v, t): v comes after t. **Backward**.
  - (w, t): w comes after t. **Backward**.
  - $(t, s), (s, v), (s, w), (v, w)$: All forward.

  This ordering has **2 backward arcs**. Since $2 > 1$, the DFS algorithm failed to find the optimal ordering.

iv. **Explanation of the Counter Example**

- The Goal: We want an order tha respect the arrows as much as possible.
- The Optimal: The ordering s, v, w, t has only 1 backward edge (t, s). The cost is 1.
- DFS Failure: DFS is sensitive to the starting point. Starting at t leads to the ordering t, s, v, w, which has 2 backward edges: (v, t) and (w, t). The cost is 2.
- Conclusion: Since DFS can produce an ordering with more backward edges than the optimal, it does not always minimize backward edges.

5. On adding one extra edge to a directed graph G, the number of strongly connected components...?

(a) The graph might already be strongly connected.