Response to Assignment 6 Question 2
Jingbo Yang (Write) Muzi Zhao (Read)

Part (a) We will prove the statement by contradiction.
Assume G is a non-empty directed acyclic graph (DAG) that does not have a source. By definition, a source in a graph is a node with in-degree 0, meaning there are no edges directed into it. If G has no source, then every node in G must have an in-degree of at least 1.
Now, pick an arbitrary node v_1 in G. Since v_1 has in-degree ≥ 1, there exists at least one other node v_2 such that there is an edge $v_2 \to v_1$. Similarly, since v_2 also has in-degree ≥ 1, there must be another node v_3 such that there is an edge $v_3 \to v_2$. Repeating this process indefinitely, we generate a sequence of nodes $v_1, v_2, v_3, \ldots$, where each node points to the previous one.
Since G is finite (it contains a finite number of nodes), this process must eventually revisit a previously visited node. In other words, there exists some node v_k such that $v_k \to v_j$ for some earlier node v_j in the sequence. This implies the existence of a cycle in G.
However, this contradicts the assumption that G is acyclic. Therefore, our assumption that G has no source must be false.
Hence, every non-empty directed acyclic graph G must have at least one source.

$\square$

Part (b)
We need to prove both directions of the "if and only if" statement.
Direction 1: Acyclic $\implies$ Source-Removal Leads to Empty Graph
Proof: Assume G is acyclic. We need to show that repeatedly applying the source-removal operation results in an empty graph.
By part (a), any acyclic digraph has at least one source. Removing a source does not create any new cycles, so the remaining subgraph is still acyclic and has at least one source. Repeating this process until all nodes are removed yields an empty graph.
Direction 2: Source-Removal Leads to Empty Graph $\implies$ Acyclic
Proof: Assume the source-removal operation results in an empty graph. We show G is acyclic by contradiction. Suppose G contains a cycle. Then, no node in the cycle can be a source since every node in the cycle has incoming edges. The source-removal operation would halt before the graph becomes empty, contradicting our assumption. Thus, G cannot have a cycle and must be acyclic.
We have shown: Acyclic $\iff$ Source-Removal Leads to Empty Graph

$\square$

part(c)

```
1.   def is_acyclic(graph):
2.       $n = \text{len(graph)}$
3.       $in\_degree = [0] \times n$
4.       for $u$ in range($n$):
5.           for $v$ in graph[$u$]:
6.               $in\_degree[v] += 1$
```

```
7.        $S = [u$ **for** $u$ **in** range$(n)$ **if** $in\_degree[u] == 0]$
8.        $processed\_nodes = 0$
9.        **while** $S$:
10.           $u = S.pop()$
11.           $processed\_nodes+ = 1$
12.           **for** $v$ **in** graph$[u]$:
13.               $in\_degree[v]- = 1$
14.               **if** $in\_degree[v] == 0$:
15.                   $S.append(v)$
16.        **return** $processed\_nodes == n$
```

We first compute the in-degree of each node, initializing an array where each element represents the in-degree. Then, we update the in-degree values based on the adjacency list of the graph. Next, we create a list S containing nodes with in-degree 0 (source nodes). We also initialize `processed_nodes` to count the number of nodes processed during the source-removal operation. While S is not empty, we remove a source node u from S (using `pop()`), increment `processed_nodes`, and "remove" u from the graph by decreasing the in-degree of its neighbors by 1. If the in-degree of a neighbor v becomes 0, we add v to S as a new source node. Finally, we check if `processed_nodes` equals n (the total number of nodes). If true, the graph is acyclic by part (b); otherwise, it contains a cycle. This algorithm works since it strictly relies on the fact that in an acyclic graph, there will always be at least one node with no incoming edges, as shown in the previous part we proved. By repeatedly removing source nodes and updating in-degrees, we effectively perform a topological sort. Then, we use the conclusion from part (b) to verify if the graph is acyclic.

Part (d)
The time complexity is $O(n + m)$ because:
Lines 1–3: These lines take constant time. Time complexity: $O(1)$.
Lines 5–7: The outer loop iterates n times. The inner loop iterates over all neighbors of u. The total number of iterations across all nodes u is equal to the total number of edges m, since each edge (u, v) contributes exactly once to the in-degree count. Thus, in total, we have Time complexity: $O(n + m)$.
Line 9: Since `len(S)` $= n$, we iterate exactly n steps. Time complexity: $O(n)$.
Line 11: Variable assignment takes constant step. Time complexity: $O(1)$.
Lines 13–19: The `while` loop iterates at most n times since there will be at most n nodes in S. The inner `for` loop iterates over all outgoing edges for the node u popped out by `u = S.pop()`. The total number of iterations across all nodes is equal to the total number of edges m, as each edge (u, v) is processed exactly once when u is removed from S. The operation `in_degree[v] -= 1` takes $O(1)$ time. The operation `S.append(v)` takes $O(1)$ time. Thus, the runtime of Lines 13–19 is $O(n + m)$.
Line 20: This line takes constant time. Time complexity: $O(1)$.
Overall Complexity:

$$3O(1) + O(n + m) + O(n + m) + O(n) = O(n + m)$$