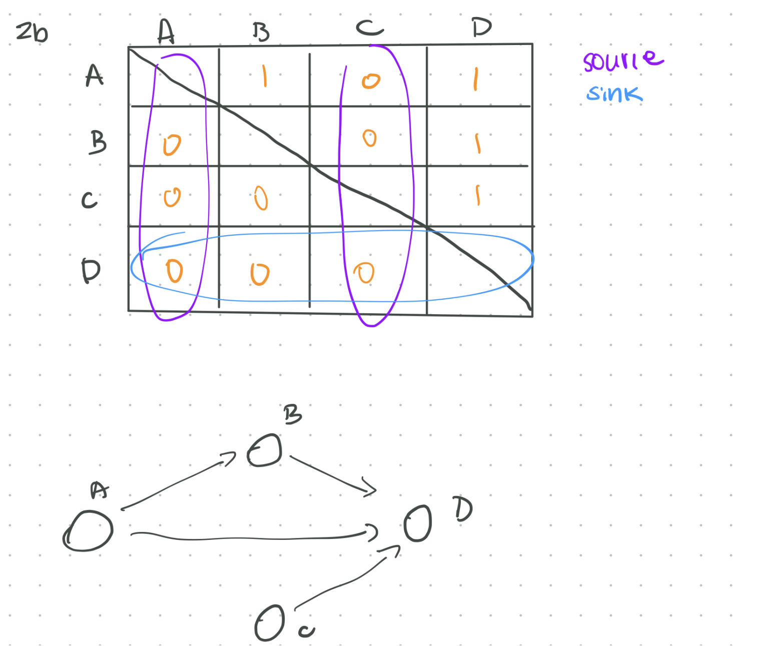
1. For any graph $G = (V, E)$, we can use an adjacency list to model it. By using depth first search, we can go through the entire graph. A sample algorithm would be to call DFS for each node, then you would look at all the nodes adjacent to that node, if they aren't marked (you can do a 1, 0 to mark them), then the algorithm would mark that node with the opposite of what your value is (if you are 0, then your adjancent node will be 1). If the algorithm completed the DFS algorithms and returns without an adjancent node being marked the same value as itself, then it is a bipartite graph.
This achieves the $O(|V|+|E|)$ runtime because in DFS, we go through each node, and for each node, we go through all the edges for that node. In the end, we go through all the vertices and all the edges, therefore achieving this total runtime.

2. Answer the following questions:

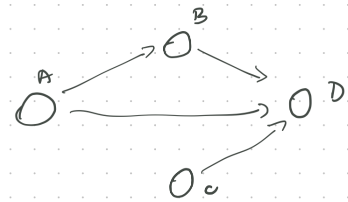
   (a) A DAG is a direct graph with no cycles. In a graph without any cycles, there must exist a node no indegree. Therefore, there is a source in any non-empty DAG.

   (b) In an adjacency matrix, we can consider whether a node i is connected with a node j depending on whether or not the value of $A_{ij} = 0, 1$. If it is 1, then node i is connected to node j. To find it, we would have to traverse through all the nodes, and within each node, we have to check whether or not its connected with all other nodes. We will traverse through the entire matrix, and determine a source node if there exist a node with the value 0 for any i connected to it. Since we must traverse the entire matrix, the total runtime will the $O(|V|^2)$, or $\boxed{\mathbf{O(n^2)}}$. Below is a visualization of the algorithm.



   (c) Here, we must go through all the nodes as well. If a node is a source node, it cannot appear in any linked list belonging to any of the n nodes, therefore we have to check all the edges as well. This will equal to $O(|V|+|E|)$, or $\boxed{\mathbf{O(n+m)}}$. Below is a visualization of the algorithm.

2c



A → [B] → [D] : B = true   D=true

B → [D] : D is already true

C → [D] : D is already true

D

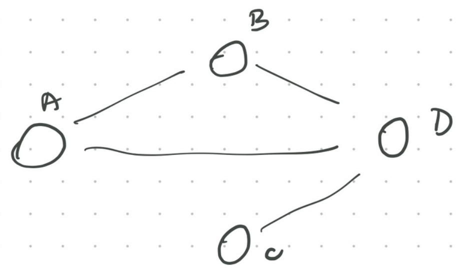
A and C are the only vertices that are false, therefore they are source nodes

3. Assume that our graph is represented with an adjacency list, that means that for each node, there is a linked list that represents all the nodes that are connected to it. We can simply loop through each vertex in $O(|V|)$ time, or $O(n)$, and in each vertex, we initiate a variable $sum = 0$, and add 1 for each element in the linked list. The sum will be stored in a dictionary where

```
neighors = { node: # of neighbors }
```

Then the algorithm will once loop all vertices x again, and for each neighbor n of x, it will search for the value of n in the dictionary and add it to the neighbor degree of x. This totals to be $\boxed{\mathbf{O(2 \times (m+n))}}$ since we are looping through all the vertices and all the edges twice in total. Below is a visual representation of the following algorithm:

3. consider the following undirected graph



represented by an adjacency list, we have the following

A : $\underbrace{B \to D}_{2}$

B : $\underbrace{A \to D}_{2}$

C : $\underbrace{D}_{1}$

D : $\underbrace{A \to B \to C}_{3}$

1. loop through each vertex
2. sum of the total amount of neighbors for that vertex
3. create dictionary

   [A:2, B:2, C:1, D:3]

4. loop through each vertex again
5. for each neighbor, get the # of edges from our dictionary

Example:

A: B -> D => from dictionary: B=2, D=3

∴ sum of neighbor degree for vertex A = 5

4. Here