Question 1

An undirected graph is said to be bipartite if all its vertices can be partitioned into two disjoint subsets X and Y so that every edge connects a vertex in X with a vertex in Y. Design a linear time, i.e., O(|V| + |E|), time algorithm to check if a graph is bipartite or not.

Make an array called colors, of size |V| where each vertex in the graph maps to a unique slot in the array. Initialize this array with null values.

```
Function isBipartite(vertex v, array colors, bool XorY) {
        If XorY is true:
                Declare a variable “current color” as X
        Else:
                Declare a variable “current color” as Y

        Set the current vertex’s slot in the colors array equal to the value of current color

        For each neighbor, u, of the current vertex, v:
                If neighbor u has been assigned a color value equal to the current color:
                        return false
                If neighbor u has not yet been assigned a color:
                        call isBipartite on the neighbor u, but pass the inverse of XorY, and return it’s value
        return true
}
```

Call the above function with a driver like so:

```
For each vertex v, in the graph:
        If v has not been visited:
                Call isBipartite(v, true)
                If function returns false, graph is not bipartite
If code reaches this point, then return true – the graph is bipartite
```

I approached this problem as a graph coloring problem. My algorithm alternatingly colors in nodes of the graph with two colors – we’ll call them X and Y. If any adjacent nodes are the same color, we know the graph is not bipartite, because there is a edge between two nodes that are supposed to belong to disjoint sets.

Question 2a

Prove that a non-empty DAG must have at least one source.

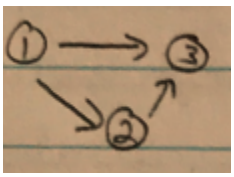
At least one source → at least one node without any incoming connections.

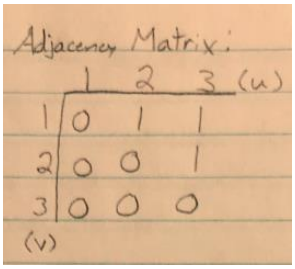
I'll prove this by contradiction. Assume that there exists a DAG such that there are no sources – all nodes have incoming connections. If a node has incoming connection, it can therefore be reached by another node in the graph. If all nodes have incoming connections, that means there are at least two nodes that can reach one another (they are mutually reachable). This is by definition a cycle, which violates the "acyclic" assertion of our initial statement. Thus, every DAG must have at least one source.

Question 2b

What is the time complexity of finding a source in a directed graph or to determine such a source does not exist if the graph is represented by its adjacency matrix? Describe the algorithm.

For an adjacency matrix where rows represent outgoing connections and columns represent incoming connections (e.g. the graph below would be represented by the matrix shown) the algorithm will take O(|V|^2) time.



Would be represented by:


Psuedocode / Explanation:

Create a hashmap equal of size equal to the number of vertices in the graph.

For every row, r, in the matrix:
    For every column, c, in the matrix:
        Check the cell[r][c] for an edge (in a matrix, 1's indicate edges and 0's indicate no edges)
        If there is an edge, mark the node that corresponds to the column as not a source in the hashmap created above (Code would look something like: hashmap_name[c] == false).

At the end of the loop, all nodes which cannot be sources will be marked false. Simply traverse the array to find all nodes which are in fact sources.

The algorithm is O(|V|^2) because we must check all potential connections in the matrix, even non existent ones.

Question 2c

What is the time complexity of finding a source in a directed graph or to determine such a source does not exist if the graph is represented by its adjacency list? Describe the algorithm.

For an adjacency list where every vertex has an associated list of outgoing edges, the algorithm is linear proportional to the size of the graph and runs in O(|V| + |E|) time.

Pseudocode / Explanation:

First create a hashmap of size equal to the number of vertices in the graph just like in part b.

For every vertex in the list:
    For each of that vertex's outgoing edges / neighbors:
        Mark each neighbor as false in the hashmap.

After the loop has run, the hashtable has marked all nodes which cannot be sources.

Simply traverse the hashtable structure to find all nodes which are sources.

The algorithm runs in linear time because the first loop simply checks every vertex and every edge once (|V| + |E|). The final traversal of the hashtable is a O(|V|) operation because it is of size |V|.

Question 3

Describe a linear time algorithm to compute the neighbor degree for each vertex in an undirected graph. The neighbor degree of a node x is defined as the sum of the degree of all of its neighbors.

Assume that the graph is stored as an adjacency list where each undirected edge between two arbitrary nodes x and y is represented as an outgoing edge from node x to y and an outgoing edge from node y to x.

Create a hash table of size |V| where every vertex in the graph maps to a unique slot in the data structure.

Loop through the vertices in the graph
    Count each vertex's edges and store the number in the appropriate slot of the hash table structure created above.

Loop through the vertices in the graph a second time,
    For every edge / neighbor of the current vertex,
        Look up the neighbor's degree from the hash table structure created above
        Sum the degrees of all neighbors to get the neighbor degree for the current vertex.

Both loops check every vertex and every edge exactly once, so together they reduce down to O(|V| + |E|).

Question 4a

Consider a directed graph that has a weight w(v) on each vertex v. Define the reachability weight of vertex v as follows:

$$r(v)=\max\{w(u) \mid u \text{ is reachable from } v\}$$

That is, the reachability weight of v is the largest weight that can be reached from v.

Now, assume the graph is a DAG. Describe a linear time algorithm to compute the reachability weight for all vertices.

The algorithm is a variation of the DFS method we discussed in class.

Psuedocode / Explanation:

Assume a setup for a typical DFS algorithm. We have a Boolean array that keeps track of which nodes have been visited and a method explore that takes a vertex as a parameter. Assume that every node has a weight property and a reachability weight property in addition to the list of its neighbors.

Now, we'll tweak explore a little bit to achieve the desired result. Explore can be written like so:

```
Explore(vertex v) {

        Mark v as visited.

        Set v's reachability weight to be the "regular" weight of v (r(v) = w(v)

        For each neighbor / edge of v (we'll call them u):
                If u has not been visited:
                        Explore(u)
                        If u's reachability weight is higher than v's current reachability weight:
                                Set v's reachability weight equal to u's reachability weight.
}
```

Lastly, we have a driver function like so:

```
For every vertex v in the graph:
        If v has not been visited:
                Explore(v)
```

The algorithm works by using the directed graph DFS method from class but assign reachability weight where we'd ordinarily set postorder numbering. This ensures that the highest weight percolates backwards into each node's reachability weight.

Question 4b

Consider a directed graph that has a weight w(v) on each vertex v. Define the reachability weight of vertex v as follows:

$$r(v)=\max\{w(u) \mid u \text{ is reachable from } v\}$$

That is, the reachability weight of v is the largest weight that can be reached from v.

Now, Assume that the graph is a general directed graph (with possible cycles). Describe a linear time algorithm to find the reachability weight for all vertices.

We know that every directed graph is a DAG across it's strongly connected components, so we'll deal with the cycles by first running the SCC algorithm described in class. This gives us all the SCC regions in linear time. We can modify the algorithm slightly so at the end, each SCC sink is mapped to a slot in a hashtable.

Next we'll calculate the max weight of each SCC by checking every node in a given SCC (starting with the sink then touching all others in a DFS like manner) and updating the maximum weight for that SCC as we go. We just need to check every vertex once, so this completes in O(|V|) time.

Now that we have the graph in SCC form, with a max weight for every SCC, we can run the algorithm from 4a – treating every SCC as an individual node.