1)
Algorithm description in plain English:

First, pick any starting node s on the graph. Initialize an empty list of nodes that represents the current path of the algorithm and add s to it, then perform a recursive depth first search while adding nodes to the path list as they are encountered. Make sure that the list only contains the nodes of the current DFS path and none from other paths. If an edge is encountered that leads to a node already in the path (also make sure this node is not the current node's parent), return the path list but only the repeated node, the nodes encountered after the repeated node, and then the repeated node again. If the graph is not connected, then you may need to call the algorithm again on different components of the graph to ensure that every node and edge is visited.

Proof of Correctness:

- Claim (Loop invariant): At the start of every recursion of the DFS, the path list will show the current path that the algorithm is taking.
- Initialization: Just before the first DFS recursion, the DFS algorithm is called on a starting node s, which also is added to the path list and thus the current path is correct.
- Maintenance: As the DFS algorithm encounters a new node, it is appended to the end of the path. Because the DFS algorithm is called recursively, there may be different path lists for different function calls, ensuring that any path will not contain nodes from other paths. Thus the invariant holds from iteration to iteration.
- Termination: The DFS recursion ends when either an edge to a previously encountered node in the path has been found (in which case the cyclic path is returned), or when every node and edge in the graph has been checked without a cycle being found.

Complexity Analysis:

The algorithm is essentially a depth first search with minor adjustments added that occur in constant time. Depth first search is known to have a runtime of $O(m + n)$ since it iterates through every node and every edge. Therefore, this algorithm also has a runtime of $O(m + n)$.

2) The claim is true.
Proof: Assume that the claim is false, and G is an unconnected graph with n nodes, where n is an even number and every node of G has a degree of at least n/2. Because G is unconnected, it must be made up of at least 2 different connected components where each node in each component has a degree of at least n/2. In order for a component to meet this condition, the component must contain at least ((n/2)+1) nodes, so that any/all of the nodes will have n/2 neighbors. However, this means that another component of G can have, at most, ((n/2)-1) nodes. It would be impossible for a single node in this component to have at least n/2 neighbors. Therefore, an unconnected graph with the conditions stated previously cannot exist and the claim is true.

3) The running time of the BFS algorithm would change from $O(m+n)$ to $O(n^2)$, where n is the amount of nodes in the input graph. This is because when a node is encountered using an adjacency list, the BFS algorithm will only iterate through all of its neighbors. The amount of

iterations depends on the degree of the current node, meaning this step occurs in O(m) time. However with the use of an adjacency matrix, the BFS algorithm will need to iterate through every other node in the graph to find neighbors, for every node encountered. The iterations will always remain the same regardless of the degree of the current node, meaning this step occurs in O(n) time for every node, and therefore the total runtime at the end of the algorithm will be $O(n^2)$.

4) The problem can be modeled as a graph problem by constructing a rooted "decision tree". The root node will contain a tuple or list that represents the original state of the containers (0/10 gallons, 7/7 gallons, 4/4 gallons). From there, the vertices of the node will represent choosing to pour water from any container x to any other container y, meaning there will be at most 6 different vertices to represent the 6 different possible decisions (if pouring from an empty container or pouring to a full container, the decision will not be completed). The child nodes will represent the state of the containers after the decision is completed. Continue branching downwards until a node is found with either a 2/7- or 2/4- gallon container. To ensure the algorithm does not keep repeating infinitely, you could terminate node states that have been previously discovered. If all nodes end up terminated and the solution is never found, then the solution does not exist.

5)
Algorithm description in plain English:

First, initialize a set of nodes for every person's birth date, and a set of nodes for every person's death date. Create an edge directed from every person's birth node to their respective death nodes. If person P_i died before person P_j was born, create an edge directed from P_i's death node to P_j's birth node. If the life spans of P_i and P_j overlapped at least partially, create an edge directed from P_j's birth node to P_i's death node. Then, attempt to organize a topological ordering by finding a node with no incoming edges and ordering it first. Delete that node from the graph, then recursively perform the first step. If the topological ordering is successfully made, then the proposed dates of birth and death can be anything as long as they follow the arrangement of the topological ordering. Otherwise if the topological ordering cannot be successfully made, then no such dates can exist and the facts collected by the ethnographers are not internally consistent.

Proof of Correctness:

- Claim (Loop invariant): At the start of every iteration, every directed edge (x, y) in the topological ordering will have x at an earlier date than y.
- Initialization: The topological ordering begins with a node that has no incoming edges, if one exists. This cannot lead to a cycle since there is no path from a future node to this one.
- Maintenance: At every iteration of the algorithm, only nodes with no incoming edges are added to the end of the topological ordering, ensuring that no edges will be directed backwards and no cycles can be formed. Therefore, it is impossible to find an edge (x, y) in the topological ordering where y is an earlier date than x. Thus the iteration holds from iteration to iteration.

- Termination: The iteration ends when either all the nodes have been added to the topological ordering, or when a node with no incoming edges cannot be found (in which case, a topological ordering cannot exist).

Complexity Analysis:

Initializing nodes for every birth and death date occurs in O(m) time where m is the amount of dates and also the amount of nodes. Creating edges occurs in O(n) time where n is the amount of facts collected by the ethnographers and also the amount of edges. As we have learned previously, the algorithm to create a topological ordering requires an O(m + n) runtime. Therefore, this algorithm also has a runtime of O(m + n).