1. **BFS Algorithm**: We need to modify it such that it outputs the number of shortest paths from s to other nodes

BFS(s):

```
    Initialize L[0] to consist of the single element s
    Initialize Count[i] = 0 for all i not equal to s
    Set Count[s] = 1
    Initialize counterShortestPaths[i] = 0 for all i not equal to s
    Set counterShortestPaths[s] = 1
    Set Discovered[i] = true
    Set Discovered[s] = false for all i not equal to s
    Set the layer counter i = 0

    While L[i] is not empty
        For each node u ∈ L[i]
            For each v s.t. (u, v) is an edge
                If Discovered[v] = false
                    counterShortestPaths[v] += counterShortestPaths[u]
                    Add v to the list L[i+1]
            Endfor

            For each v s.t. (u, v) is an edge
                Discovered[v] = true
            Endfor

            Increment the layer counter i by one
        Enfor
    Endwhile
```

Justification of Time Complexity: The runtime is O(m) since it only visits each edge twice at the most and this is only if the edge connects two nodes that are in the same layer or level. Other than that, it is basically just the BFS algorithm except we keep an array to keep track of the number of shortest paths.

Proof of Correctness: the BFS algorithm goes through the graph and starts at some node. It then goes to the next "level" or "layer" of nodes which is said to be any nodes that are next to or adjacent to the starting node. Then after that, the following "levels" are the next adjacent nodes. At a certain iteration "x", BFS will find all nodes that are x steps away from the original node. Therefore, this will search through the entire graph and find the shortest path to all nodes that are reachable.

2.

a. We can determine if a node u is an articulation point by checking if either of its children have an L value >= its d-value. However, if the root node has only one child, then it will NOT be the articulation point.

b. **Algorithm down below**:

```
Global Initialization: $D[u] = -1$ for all node $u$ //d-values
Global Initialization: counter = 1
*Global Initialization: L[u]= −1 for all node $u$ //l-values
*Global Initialization: A= {} //articulation points
*Global Initialization: C //children map
DFS($s$)

Function DFS(u)
        $D[u]$ = counter
        currL = counter
        counter = counter + 1
        for each edge $(u, v) \in E$
                If $D[v] == -1$
                        // $(u, v)$ is a tree edge
                        C[u] += v
                        DFS(v)
                        *currL = min(currL, L[v])
                Else
                        // $(u, v)$ is a non-tree edge
                        *currL = min(currL, D[v])
        *L[u] = currL

        If (D[u] == 1) and (C[u] > 1) //where the root has only one child
                        return; //not an articulation point

        for each child $v$ in C[u]
                If (D[u] <= L[v])
                        A += u //u will be added to A, the set of articulation points.

End Function
```

Justification of Time Complexity: Since we already know that DFS is linear, all we are adding is the line that checks the minimum of the current L value which is constant time and would not change the runtime. Also, when we iterate through the edges, this is done in constant time for all the edges, so it would still be O(m) where m represents the edges.

Proof of Correctness: Suppose there exists nodes a, b, and c such that a is the parent of b and b is the parent of c. If the c's L-value >= the d-value of b, then that subtree has no alternate way of reaching the a other than through b.

3.

To compute a topological ordering of *G*:
Find a node *v* with no incoming edges and order it first
Delete *v* from *G*
Recursively compute a topological ordering of $G-\{v\}$ and append this order after *v*

Algorithm:

```
Global Initialization: B = {} //births
Global Initialization: D = {} //deaths
Global Initialization: E = {} //edges

For each person i
    let B[i] be their birth and D[i] be their death.
    E += (B[i], D[i])

For each fact F: (i, j)
    If (D[i] < B[j]) //if person i's death is before person j's death
        E += (D[i], B[j])
    Else
        E = E + (B[i], D[j]) + (B[j], D[i])

Let the graph G = B + D
Initialize L as a list
While (G is not empty)
    If there are no source nodes in G
        return "Not internally consistent"
    Else
        Let n be a source node of G
        Delete any edge from E that has n as a node
        G -= n
        L += n

Return L which is the topological ordering
```

Justification of Time Complexity: The while loop specifically is really just a topological sort which we already know has a runtime of O(m+n). The only steps we added were the initializations of the graph where we used O(n) time for the for-loop with each person i and O(n) time for the oth for loop.

Proof of Correctness: The only difference in this topological sort algorithm is the construction of the graph due to the two types of facts given. There will be three types of edges we have. The first one will be an edge from a person's birth to that same person's death. This is true because we know that a person's birth will always come before a person's death. Next, we have an edge from a person's death to another person's birth and that's because we know that (from the type of facts we are given) that a person's birth comes before another person's death. Finally, we have the fact that one person's life overlaps with another's. So we can represent this with an edge from a person's birth to another person's death as well as that other person's birth to a person's birth. And this is because we know that the first person's birth comes before the next person's death as well as vice versa.

4.

Algorithm:

```
Initialize I to be the list of intervals
Initialize X to represent the resulting set of numbers
Sort I by end point //O(nlogn) time

While I is not empty:
        Let (Li, Ri) be the first interval in I
        Remove (Li, Ri) from I
        Let x = Ri - 0.5
        X += x
        Delete any node (L, R) from I such that L < x

Return X
```

Correctness Justification:

- Suppose by contradiction that (L, R) is an interval such that there exists no element in X falling between L and R. If this (L, R) were the first interval in I, then we would be defining an x between L and R, so this can not be the case. Therefore, it must be true that (L, R) was deleted from I. Hence, we know that there is some x > L. Since this x is defined to be less than some Ri which is the endpoint of an interval ending before (L, R), then we also know that x < Ri < R. Therefore, there is some x such that L < x < R which is a contradiction.

Time Complexity Justification:

- We know that there are two steps to the interval scheduling greedy algorithm in general: sorting and then a linear scan. These should be in O(nlogn) for sorting and then O(n) for the linear scan. Therefore O(nlogn) + O(n) => O(nlogn). The only difference between that standard algorithm and ours is that we have a loop where we add a few operations to assist us within the while loop. However, all of these operations end up either constant time except for when we delete any node (L, R) from I such that L < x. This ends up being in O(logn) and since we are in a loop, the complexity for the loop is O(nlogn). In total, the time complexity is O(nlogn) + O(nlogn) => O(nlogn).