1. The answers for multiple choice questions are on the quiz. Here is a sample solution for the coding question.

for all nodes, $v.dfs# = -1$
dfscounter = 0, finishCounter = 0
DFS(s)

function DFS($v$)
    $v.dfs# = dfscounter++$
    $v$ “discovered”
    $v$.finished = true
    $v$.finishTime = finishCounter++

    for each edge ($v$, $x$) do
        if $x.dfs# = -1$ then
            $x$ is undiscovered, so this is a tree edge
            label ($v$, $x$) as a tree edge.
            DFS($x$)
        else
            if $v$.finished == false then
                Since $x$ is still active, it is an ancestor of ours.
                label ($v$, $x$) as a back edge.
            else
                $x$ is not an ancestor of ours (since it is finished)
                if $v.dfs# > x.dfs#$ then
                    Since we arrived at $x$ first, this must be a cross edge
                    label ($v$, $x$) as a cross edge.
                else
                    $x$ is a descendant of ours, this must be a forward edge.
                    label ($v$, $x$) as a forward edge.
            end if
        end if
    end for
end function

2. Often there are multiple shortest paths between two nodes. Give a linear time algorithm for the following task:

Input: An undirected graph $G = (V, E)$ with unit edge lengths; nodes $u, v \in V$.
Output: The number of distinct shortest paths from $u$ to $v$.

Solution: We will modify Breadth First Search to create an algorithm for this problem. We will keep track of two extra variables for each node: $\text{distance}$ which is the distance from $u$ to that node, and $\text{numPaths}$ which is the number of paths of length $\text{distance}$ from $u$ to that node. The algorithm works on the following principle: every path of length $k$ from $u$ to $v$ can be broken up into a path of length $k - 1$ from $u$ to some neighbor of $v$ followed by an edge to
function NumShortestPaths(G, u, v)
    Let Q be an empty queue
    u.visited = true
    u.distance = 0
    u.numPaths = 1
    add u to Q
    while Q is not empty do
        curr = Q.dequeue()
        for each neighbor t of curr do
            if t.visited == false then
                t.visited = true
                t.distance = curr.distance + 1
                t.numPaths = curr.numPaths
                add t to Q
            else
                if t.distance == curr.distance + 1 then
                    t.numPaths += curr.numPaths
                end if
            end if
        end for
    end while
    return v.numPaths
end function
Thus if we can keep track of the number of shortest paths to all intermediate nodes, we can get the number of shortest paths to \( v \).

Proof of correctness:
First, note that for all vertices, \( t \), once \( t.\text{distance} \) is set, it is actually the distance from \( u \) to \( t \). Our algorithm runs a BFS (just with some extra variables) the \( \text{distance} \) variable is just the “layer” as described in the lecture videos. In those videos they prove that these are indeed the distances from \( u \) to every other vertex \( t \).

We also need the following lemma:
If the distance from \( u \) to \( t \) is \( k \), and there is a path, \( P \), of length \( k \) from \( u \) to \( t \), where \( s \) appears immediately before \( t \) on \( P \) then the distance from \( u \) to \( s \) is \( k - 1 \).

Proof: The portion of \( P \) from \( u \) to \( s \) gives a path of length \( k - 1 \) from \( u \) to \( s \) so the distance is at most \( k - 1 \). Suppose (for the sake of contradiction) that the length from \( u \) to \( s \) is less than \( k - 1 \). Then there is a path (call it \( Q \)) of length less than \( k - 1 \) from \( u \) to \( s \). Adding the edge from \( s \) to \( t \) to \( Q \) gives a path from \( u \) to \( t \) of length less than \( k \), but this contradicts the fact that the distance from \( u \) to \( t \) is \( k \). Thus the distance from \( u \) to \( s \) is exactly \( k - 1 \).

This lemma tells us that to find paths of length \( k \) to a vertex, \( u \), of distance \( k \), we only need to check the vertices at distance \( k - 1 \), and add up all the paths that can be extended by an edge to \( u \).

We are now ready to prove our main correctness result. We use induction to show: After every vertex at \( \text{distance} \) \( n - 1 \) has been removed from the queue and processed, if \( t \) is a vertex at \( \text{distance} \) \( n \) then \( t.\text{numPaths} \) is the number of paths of length \( n \) from \( u \) to \( t \).

Proof by induction on \( n \) (the distance from \( u \) to the current vertex)
Base Case: \( n = 0 \), \( u \) is the only vertex at \( \text{distance} \) \( 0 \). There is one path from \( u \) to itself (the path is simply the vertex \( u \) with no edges), and \( u \) has \( \text{numPaths} \) set to 1, so \( u.\text{numPaths} \) is correct.

Inductive Hypothesis: Suppose that for \( n = 0, 1, \ldots, k \) that if \( t.\text{distance} = n \) then \( t.\text{numPaths} \) is the number of paths of length \( n \) from \( u \) to \( t \).

Inductive Step: Suppose that \( t \) is a vertex where \( t.\text{distance} = k + 1 \). By our lemma, every path of length \( k + 1 \) from \( u \) to \( t \) can be broken into a path of length \( k \) from \( u \) to some vertex \( s \) (where the distance from \( u \) to \( s \) is \( k \)), and then an edge from \( s \) to \( t \). By inductive hypothesis, the \( \text{numPaths} \) variable stored in the vertices at distance \( k \) are correct. When we do the BFS, we process vertices in order of the \( \text{distance} \), so while processing the vertices at distance \( k \), we will find every edge from a vertex at distance \( k \) to \( t \), once we have processed all the vertices at distance \( k \), \( t.\text{numPaths} \) contains the sum of \( \text{numPaths} \) for all distance \( k \) neighbors of \( t \). By our lemma, all paths to \( t \) of the minimum length are of this form, so \( t.\text{numPaths} \) is indeed equal to the number of length \( k + 1 \) paths from \( u \) to \( t \).

We return \( v.\text{numPaths} \), which by our claim is the number of paths of length \( v.\text{distance} \) from \( u \) to \( v \). By our lemmas and claim, this is actually the number of shortest paths from \( u \) to \( v \).

Running Time: We just run a BFS with some extra constant work per vertex, so the running time is \( O(m + n) \).

3. Consider a directed graph on \( n \) vertices, where each vertex has exactly one outgoing edge. This graph consists of a collection of cycles as well as additional vertices that have paths to
the cycles, which we will call the branches. We define the weight of the cycle to be the total number of vertices that are either on the cycle or on branches that are connected to the cycle. Give pseudocode for a linear time algorithm that identifies all of the cycles and computes the length and weight of each cycle.

Solution In a directed graph, a weakly connected component are the connected components of the undirected graph you would get by ignoring the direction on all of the edges. In our situation a weakly connected component of the graph is exactly a cycle and its branches. We will use the following algorithm, first we separate the vertices into their weakly connected components, then we can actually calculate weights easily.

```plaintext
function Weights_of_Cycles(G)
    Let G' be the undirected version of G
    currComponent ← 0
    for each vertex in G' do
        Start a BFS in G' at u, whenever you mark a vertex as visited, also mark its component as currComponent
        currComponent++
    end for
    mark all vertices as unvisited
    Let Lengths[] be an array of size currComponents - 1, with every entry set to 0.
    Let Weights[] be an array of size currComponents - 1, with every entry set to 0.
    for each vertex in G do
        Start a DFS in G at u.
        whenever you mark a vertex, v, as visited, increment Weights[v.component]
        if (s, t) is the first back edge you see in a component. then
            ⊿ The path from t to s and the back edge (s, t) is the cycle for this component.
            Counting the number of vertices from t to s gives us the number of vertices on the cycle.
            counter ← 1
            current ← t
            while current ≠ s do
                current ← current.neighbor
                counter++
            end while
            Lengths[s.component] ← counter
        end if
    end for
end function
```

Proof of Correctness: Since every vertex has outdegree 1, no vertex can appear on a branch for more than one cycle, thus every vertex which can reach a cycle is on the cycle or on a branch for it. Thus if we do a BFS ignoring edge directions, a connected component (i.e. a weakly connected component of G) is exactly a cycle and its branches. Our BFS runs the component finding algorithm in undirected graphs (as described in the videos) so each component is a cycle with its branches. When we run a DFS starting in some branch, we will follow it until it reaches the cycle, and then follow along the cycle (since the graph has
outdegree-1 we have no choices, the DFS just follows the out-edge). When the DFS goes all the way around the cycle, we arrive at the first back edge (call it \((s, t)\)) in this component, which (since we are on a cycle) counting the length of the path from \(t\) to \(s\) gives the length of the cycle. Thus \(\text{Lengths}[]\) is correct.

Our secondary DFSs will visit every vertex (as all DFSs in for loops do), so by updating \(\text{Weights}[]\) each time we visit a vertex, we count every vertex in a component, which by our previous argument is equal to the weight of the cycle and branches. Thus \(\text{Weights}[]\) is correct as well.

Running time: We just run two DFSs with some extra constant work per vertex, and some extra work of counting the lengths of cycles (this can be done with their own DFSs, where each vertex in the graph is visited at most once.) In total we just run a constant number of DFSs, so the running time is \(O(m + n)\).

4. Implement your algorithm from the previous problem for finding the cycles in an out-degree one graph. Your algorithm should be designed to work on very large graphs, e.g., with \(n = 100,000,000\).

Write an input generator which creates completely random out-degree one graphs where each vertex points to another vertex chosen uniformly at random and run your program on inputs from your generator of various sizes.

You are free to write in any programming language you like. The quality of your algorithm may be graded, but the actual quality of the code will not be graded. The expectation is that you write the algorithmic code yourself – but you can use other code or libraries for supporting operations.

We will ask you to turn in your algorithmic code, including the code generating the random inputs.

Also please write up an answer to the following question: As the size of the problem increases – how does the number of cycles, and the length and the weight of the cycles change, when the input is a random graph with out-degree one? Provide data to support your answer.

**Solution:**

Here is a table of empirical results we got, averaged over ten instances for each size:

<table>
<thead>
<tr>
<th>Size</th>
<th>Number of Cycles</th>
<th>Maximum Weight</th>
<th>Minimum Weight</th>
<th>Average Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000 nodes</td>
<td>2.6</td>
<td>881.4</td>
<td>40.2</td>
<td>384.6</td>
</tr>
<tr>
<td>10,000 nodes</td>
<td>4.4</td>
<td>8,453.5</td>
<td>19.0</td>
<td>2,272.7</td>
</tr>
<tr>
<td>100,000 nodes</td>
<td>6.6</td>
<td>73,755.5</td>
<td>28.1</td>
<td>15,151.5</td>
</tr>
<tr>
<td>1,000,000 nodes</td>
<td>6.6</td>
<td>720,662.6</td>
<td>414.5</td>
<td>151,515.2</td>
</tr>
<tr>
<td>10,000,000 nodes</td>
<td>7.5</td>
<td>7,147,064.4</td>
<td>68.9</td>
<td>1,333,333.3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Size</th>
<th>Maximum Length</th>
<th>Minimum Length</th>
<th>Average Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000 nodes</td>
<td>35.0</td>
<td>3.7</td>
<td>10.9</td>
</tr>
<tr>
<td>10,000 nodes</td>
<td>104.9</td>
<td>3.5</td>
<td>17.2</td>
</tr>
<tr>
<td>100,000 nodes</td>
<td>273.8</td>
<td>2.7</td>
<td>66.9</td>
</tr>
<tr>
<td>1,000,000 nodes</td>
<td>711.1</td>
<td>8.0</td>
<td>228.1</td>
</tr>
<tr>
<td>10,000,000 nodes</td>
<td>1,622.3</td>
<td>3.5</td>
<td>361.3</td>
</tr>
</tbody>
</table>
The number of cycles appears to be increasing logarithmically with respect to the size of the graph (a constant increase every time we multiply the size by 10). Maximum and average weights appear to be increasing approximately linearly with the size of the graph. Minimum weight varies wildly with no apparent pattern.

Maximum and average length appear to be increasing linearly with the size of the graph as well (though at a lesser rate – approximately tripling whenever the graph size increases by a factor of 10.) The relationship seems less clear at the higher values. The minimum length seems approximately constant except for the outlier at 1,000,000 nodes.

5. Problem 3.15 on page 102 of [DPV].

\textbf{Solution} (a) To formulate this graph-theoretically, we first need a graph: let the set of vertices be the set of intersections in the town. If there is a one-way-street that takes you from intersection \( u \) to intersection \( v \) then add the edge from \( u \) to \( v \) to the graph. The question “can you navigate from every intersection to every other” asks whether there is a path in the graph between every pair of vertices. Equivalently, this asks if the graph contains only one strongly connected component.

We can easily solve this in linear time using the algorithm described in the videos for finding the strongly-connected-components of the graph and seeing if there is only one component or more than one. (The proof of correctness and running time justification for this algorithm are in the video)

(b) The weaker claim, that no matter where you go from city hall, you can always return, says that starting at city hall, you can never leave the strongly connected component city hall’s vertex is in (by definition, if we can arrive at a destination and from that destination return to city hall, the destination is in city hall’s strongly connected component. Conversely, if we arrive somewhere from city hall and cannot return they are not in the same strongly connected component). We can again solve this in linear time as follows: Find the strongly connected components of the graph, then start a BFS (or DFS) from city hall’s vertex. If we ever arrive at a vertex that is not in the same strongly connected component, the mayor’s claim is false. Otherwise it is true.

This algorithm is correct because BFS will find everything reachable from city hall, and (by definition of strongly connected component) we leave the strongly connected component if and only if there is a place reachable from city hall from which we cannot return to city hall.

This runs in linear time since it is just two linear time algorithms.

6. We have a connected, undirected graph \( G = (V, E) \), and a specific vertex \( u \in V \). Suppose that we compute a depth-first search tree rooted at \( u \), and obtain a tree \( T \) that includes all nodes of \( G \). Suppose we then compute a breadth-first search tree rooted at \( u \), and obtain the same tree \( T \). Prove that \( G = T \). (In other words, if \( T \) is both a depth-first search tree and a breadth-first search tree rooted at \( u \), then \( G \) cannot contain any edges that do not belong to \( T \).)

We will prove this by contrapositive: suppose that we have a graph which is not a tree, then if we start a BFS and DFS from some vertex \( u \) then we obtain different trees from the BFS and DFS.
Figure 1: Edges found by the DFS are shown in red. The dashed lines form the cycle through \( s \), which causes the contradiction

Since \( G \) is not a tree, there is a cycle in \( G \). Among the vertices in \( G \) which are part of a cycle, let \( v \) be one which is closest to \( u \). Case 1: \( v = u \) (i.e. the node we start from is part of a cycle). The BFS will take every edge incident to \( u \) as a tree edge. The DFS will go down some edge in a cycle containing \( u \) (possibly after going down some paths), it will continue along some cycle containing \( u \) (possibly diverting down other paths, but always returning to the cycle and taking these edges as tree edges), until it arrives at \( t \), a neighbor of \( u \). The DFS will view the edge from \( t \) to \( u \) as a back edge, and not take it. But the BFS took every edge adjacent to \( u \) as a tree edge, so these trees are different.

Case 2: \( v \neq u \). The BFS will arrive at \( v \) not having visited any other vertex on a cycle containing \( v \) (since \( v \) is the closest vertex to \( u \) that is on a cycle). Thus it will take both edges incident to \( v \) on the cycle. We claim the DFS will also arrive at \( v \) not having visited any other vertex along the cycle – suppose not, that the DFS arrives at some other vertex \( t \) before \( v \). Then follow the tree edges of the DFS from \( u \) to \( t \), from \( t \) go along their common cycle to \( v \) and go from \( v \) along the shortest path to \( u \), until you see some vertex \( s \) which is already visited. This forms a cycle in the graph involving \( s \) (see figure). But we found \( s \) by going backwards on the shortest path from \( v \) to \( u \), so \( s \) is closer to \( u \) than \( v \) is. This contradicts \( v \) being the closest vertex to \( u \) that is part of a cycle. Thus the DFS arrives at \( v \) having not visited anything else along the cycle. When it arrives at \( v \) it will take one of its incident edges along a cycle, after this any DFS will eventually arrive at the other neighbor of \( v \) along that cycle, and not take it since it will be a back edge. The BFS took this edge, so we again have that the trees are different.
Remark: The claim is false in the case of directed graphs (a DFS and BFS in a directed cycle, starting at the same location, can produce the same trees). The claim is also false if we allow the DFS and BFS to start at different locations (an example graph for this is a cycle with 3 nodes, with one additional node adjacent to a single node on the cycle).