**CS 180** Homework 4

**Problem 1**

This algorithm iteratively uses a stack to compare two people at a time and end up with a final candidate left in the stack to be the celebrity.

findCelebrity(people)

if people.size == 1 or people.size == 0

return // size is nil or there is only one person

stack<people> st

push every person in people to stack

while st.size != 1

person1 = st.pop

person2 = st.pop  
 if person1 knows person2

st.push(person2)

else

st.push(person1)

candidate = st.pop

for person in people

if candidate knows person

return null

return candidate

**Problem 2**

1. A Depth First Search by definition creates a tree that goes from nodes processed last to nodes processed first. It uses a stack to store visited nodes and pops the last visited node to process. This results in the parent always being processed after the children, and this applies for a directed graph as well. The postorder number is assigned when the node is colored black, which is when it gets processed. Therefore, since children are processed before parents, they will always have a lower number than their parent, resulting in a reverse topological sort when you run a DFS postorder traversal on a DAG.
2. We can prove this again by the definition of DFS. If we’re running a postorder numbering on a graph, the first node, A, that we visit in a strongly connected component will be the last one numbered and therefore have a higher number than any other node in the component. The DFS would start at the graph and hit A first. Since it is a strongly connected component, all nodes are reachable by one another within it so the DFS will go through every node until it hits a leaf node or a node it has already visited. From here, it will number the last node as 1 and backtrack. The first node, A, can be thought of as a root and since it is at the top of the DFS tree, it will be numbered last in postorder.
3. If A and B are two strongly connected components in a directed graph G and there’s an edge from A to B, then the maximum post order number assigned in A will be higher than the maximum in B. There are two cases that arise from this. One, if DFS visits A first and then B, the nodes in A will be visited first and so they will be marked later in the postorder numbering, therefore yielding a higher number. The second case is if DFS visits B before A. Since the directed edge goes from A to B, then the nodes of A are unreachable and this upholds the theorem because A will be marked later after B is finished.

**Problem 3**

1. One way to prove this is to use bipartiteness. The claim states that if an edge on a directed strongly connected graph has same colored endpoints, then the graph has an odd cycle. The definition of a bipartite graph means that the nodes can be two-colored such that no two adjacent ones have the same color. A bipartite graph also will never have an odd cycle. This means that if you have the same situation as the question’s claim where you run into an already colored node and it has the same color as the previously visited node, then the graph is not bipartite and it must have an odd cycle.

We can also prove it using set theory. We can assume a graph that has an odd cycle C, that is composed of n nodes numbered 1 to n where n connects to 1. For a bipartite graph, the vertices can be split into two sets where there is at least one edge from one set to the other. We can split the vertices of C by their numbers, even or odd. If we distribute them into the even / odd sets, we get node 1 in the odd set, node 2 in the even set, etc. Eventually we would get to node n and place it in the odd set. However node n has an edge to another node in the same set, node 1 and no edge to the even set. Therefore, this graph cannot be bipartite and so it will not be correctly colored with two colors.

1. We can consider the following graph. It is a weakly connected graph because the node 4 only has an indegree of 1 and no outdegree so it cannot reach any other node. If we start at the node labeled 1 and run a DFS with two-coloring, we would go to 2 (color it black), 3 and then hit a dead end at 4 (color it black). Then we would backtrack to 2 which has another path going to 5 and color that white. From 5 we would run into a previously colored node, which in a strongly connected graph would indicate an odd cycle. However, there are no odd cycles in this graph.



1. The theory behind detecting odd edges in directed graphs is to look at the back edges and their coloring in a DFS 2-color traversal. The back edges can be identified by their post order number, where they satisfy the inequality postorder(u) < postorder(v) when the back edge is from u to v.

stack s

run BFS on the graph to label all vertices with distance from some vertex s and their parent

for every edge (u, v) of the graph

if d(u) == d(v)

while u != v

output v

s.push(w)

v = v.parent

w = w.parent

output v

while !s.empty

w = s.pop

output w

exit

**Problem 4**

1. For this situation, we want to find the shortest path in the whole graph while maintaining sufficient amount of energy. For this reason, we only consider the charging stations and the destination as endpoints from the source and run Dijkstra’s on that set.

push the source, destination and every charging station onto a graph

for every item in the graph

for every other item in the graph

if the distance between item and other item is <= 200

use Dijkstra’s algorithm to find the shortest path from item to other item

add this edge to the graph with the weight being the distance

run Dijsktra’s algorithm on the whole graph starting at the source to find shortest path to the destination

1. For this variation of the problem, we want to minimize the amount of stops we take at charging stations. Instead of finding the shortest path between the source and each station, we now will do a greedy algorithm to find the longest path. By efficiently using most of our fuel before recharging, we will minimize the amount of time we spend stopping.

current = source // start at Los Angeles

distance[stations] = {INT\_MAX} // store distance to each station, set each to large number

while (current != destination)

if distance to destination <= 200

go there, we’re done

return

for every charging station

use Dijkstra’s algorithm to calculate the shortest path to each

if calculated distance > distance[station]