Hermmy Wang
704978214
Discussion 1B

Homework 3

1.
(a)

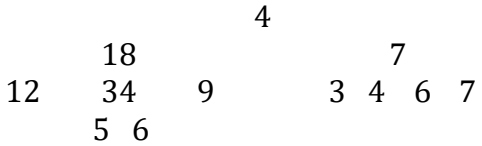
findNode(node root, node b)

```
if root is invalid or b is invalid:
        return false
if root is b:
        return true
for every child of root:
        if findNode(child, b)
                return true
return false
```

lowestCommonAncestor(node root, node u, node v)

```
if root, u, or v is invalid:
        return none
if not findNode(root, u) nor findNode(root, v):
        return none
for every child of root:
        if findNode(child, u) and findNode(child, v):
                return lowestCommonAncestor(child, u, v)
return root
```

Time complexity: O(|V|) ≈ O(|E|)
Suppose we have a tree:



...and we want to find the lowest common ancestor of 5 and 6, which is 34. Following the algorithm:

(1) Iterating the children of 4. From the first child 18, we can find 5 and 6.
(2) Iterating the children of 18. From the second child 34, we can find 5 and 6.
(3) Iterating the children of 34. From the first child 5, we can find 5 but not 6. Return 34.

If looking up an element in the tree is considered the cost of preprocessing and takes a constant operation, the time complexity is O(|V|). In the worst case, the algorithm will look up every node in the tree. The algorithm works as it recursively looks for common ancestors, until the children of the ancestor is not an ancestor anymore. The current ancestor is the lowest common ancestor.

2.

flight(n cities):

```
if cities are empty:
        return empty set

for j <-- 1 to n cities:
        for i <-- i to n cities:
                if i has a flight to j:
                        mark j as a candidate in the set
        if j is not marked:
                remove city j from cities

if nothing is removed
        return cities

return flight(cities)
```

Time complexity: $O(n^3)$

Consider the flights between cities as a matrix, where [i, j] = 1 means that there is a flight from city i to city j, 0<=i<n, 0<=j<n. If a city does not have any incoming flights, we remove the city from the set. Recursively, we would determine if a new set is valid after a city has been removed. If no more city needs to be removed, we have found the largest city set. Inside each procedure call, there is a two-level loop which takes $O(N^2)$. In the worst case, there are n cities that need to be removed, and the procedure will be called n times. The complexity is therefore $O(n^3)$.

3.

acyclic_helper(node, visited[]):

```
if node is terminal or node is visited:
        return node
else:
        mark node as visited
        for every neighbor from node:
                dfs(neighbor, visited[])
```

acyclic(starting_node, visited[]):

```
while curr_node is an unvisited vertex:

        mark curr_node as visited
        if acyclic_helper(curr_node, visited[]) returns a visited vertex:
                return false

        update curr_node


return true
```

Time complexity: $O(|V|) \approx O(|E|)$

We visit each node and each edge once. We mark nodes visited after every time we visit them; therefore, we will not repeat nodes or edges. In the worst case, we will visit every vertex, therefore the complexity is $O(|V|+|E|)$. If a graph contains a cycle, we will find a path that goes back to a node that we have already visited. acyclic_helper() will return as soon as it reaches a leaf node or goes back to a visited node. In the latter case, it means we have found a cycle, and the acyclic() function will return false.

4.

dfs_helper(source, visited[], order)

```
if source is invalid or visited
        return
mark source as visited
associate source to order
order <-- order + 1
for every neighbor of source:
        if the neighbor is unvisited
                dfs_helper(neighbor, visited[], ordering)
```

dfs(supersource)

```
set every node to unvisited state
order <-- 0

dfs_helper(supersource, visited[], order)
```

Time complexity: O(|V|) ≈ O(|E|)

In the depth first search, we traverse each node once and never repeat nodes. Hence, the time complexity is O(|V|). We first declare an array to initialize the visited state of each node to unvisited. Inside dfs_helper(), every time we enter the procedure, we mark the current node to visited, and assign an order to it. The super source node is the first node to be passed in, so the order is 0. And then we increment the order for the next node. From the current code, we visit every one of its unvisited neighbors. We would recursively call the procedure on its neighbors and assign orders to them, until every node is marked visited. Now we will have a number list from 0 to n-1, which corresponds to the order that the node has been visited.