Hermmy Wang
704978214
Discussion 1B

Homework 2

1.
(a)

```
findEulerianCycle(vertex v):
    if all edges are visited
        return current_cycle

    find edges from v until forming a simple cycle
    mark them as visited

    u <-- a vertex in the cycle has an adjacent unvisited edge

    current_cycle <-- current_cycle + path from v to u + findEulerianCycle(u) +
        path from u back to v
```

Time complexity: O(E)
We mark edges as we visit them and look for a vertex in each cycle; we never return to the visited edges again. Assume that marking edges take 1 operation and checking whether a vertex has unvisited edge is constant, the total operations would be E. Therefore, the time complexity is O(E). This algorithm keeps looking for simple cycles in the graph until exhausting all edges. The simple cycle found each time might not cover the entire graph; hence, we need to recursively look for new cycles for the vertex in the current cycle that still have unvisited adjacent edges. The path is represented as a sequence of edges. Since we search for simple cycles, there are always two paths between the starting vertex *v* and the vertex which has an adjacent unvisited edge *u*. From *v*, the path would first reach *u*, follow the cycle, and then return to *u*. After all the edges are visited, we have found a Eulerian cycle.

(b)
Given that G is strongly connected directed graph, and each vertex's in-degree equals its out-degree, we would like to prove that G has a Eulerian Cycle.
Proof by induction:
Base: A graph with two vertices and two edges connected each other in two directions contains a Eulerian cycle, going back and forth between these two vertices.
Induction: Assume G = {V, E} contains a Eulerian cycle. Since G is strongly connected, adding one vertex would require this new vertex to connect to every old vertex. It would take V edges. in other direction, and another V edges in the other direction. For G' = {V+1, E+2V}, we may visit the 2V new edges by going to and returning from the (V+1)th vertex on every unvisited vertex when tracing through the original Eulerian cycle. Therefore, we would get a new Eulerian cycle.

```
findEulerianCycle(vertex v):
        if all edges are visited
                return current_cycle
        u <-- an adjacent vertex to v such that the two edges back and forth between
them are unvisited
        mark the edges as visited
        current_cycle <-- current_cycle + edge from v to u + findEulerianCycle(u) +
                edge from u to v
```

Time complexity: O(E)
Similar to part(a), marking edges take 1 operation, checking vertices is constant, and we only visit each edge once; therefore, the time complexity is O(E). The algorithm is similar to part(a):
Step 1: Every time we encounter a pair of edges going between the current vertex and another vertex, we first reach to the other vertex in one direction.
Step 2: Before returning to the current vertex and closing the loop, we call the procedure again with the other vertex as the current vertex, recursively looking for pairs of edges until no more edges are unvisited.
Step 3: Then, we may start closing the cycles until we return to the very first vertex.

2.

lookForCeleb(matrix n)

```
celeb <-- 1
for i <-- 1 to n:
if i is equal to celeb
    continue to next i
else if celeb > n
    break out of the loop
else if i is directed to celeb
    remove i
else if celeb is directed to i
    remove celeb
    celeb <-- celeb+1
if celeb > n
    return none
return celeb
```

Time complexity: O(n)
The algorithm only needs to check all n members once. A celebrity is someone who doesn't know anyone, but everyone knows. Therefore, an edge can only go to the celebrity node, but not going out from it. We first guess that *celeb* is a celebrity. If i knows *celeb*, then i is not a celebrity; if i does not know *celeb*, then *celeb* is not a celebrity. We will then pick a new *celeb.* After the iteration, *celeb* is the celebrity; if we run out of candidates, then there are no celebrities.

3.

```
height(current_node, diameter):
    if current_node is a terminal node
        return 0

    for every child node of current_node:
        child_height <-- height(current_node->child, diameter)

    max1 <-- maximum child height
    max2 <-- second maximum child height

    diameter <-- maximum between the old diameter and (max1 + max2)

    return 1 + max1
```

Time complexity: O(n)
Every time we encounter a parent node, we initiate procedure calls on its children. There are in total n-1 children and a parent and we visit each node once; therefore, the time complexity is O(n). Diameter is the longest path length in a tree. This algorithm finds the heights of the tree by recursively going into the children of each node and adding every time it descends one level. For each node, we find the highest two heights originated from that node, and their sum is the current longest path. The variable *diameter* is used to store the diameter value for comparison. We compare this path length with the previous longest path. If the current path length is longer, *diameter* is updated, with the highest node is the current node. When every node is visited, we would have the maximum path length, which is the diameter of the tree.

4.

spanningTrees(K_n)

```
if n < 2
      return no spanning tree

remove two nodes v and u and their edges from $K_n$ to form $K_{n-2}$
tree_set <-- spanningTrees($K_{n-2}$)

for every spanning tree in tree_set
      tree <-- connect v and u to $K_{n-2}$ spanning trees
      tree_set <-- tree_set + tree

mark v as visited
while there exist edges not in tree_new nor in tree_set and unvisited vertices
      tree_new <-- tree_new + connect any vertex in tree_new to an unvisited
vertices u via an unused edge e
      u <-- visited
      e <-- used

reset all vertices to unvisited

tree_set <-- tree_set + tree_new
return tree_set
```

Time complexity: O(nlogn)
T(n) = T(n/2) + (n/2)-1+(n-1)+n=T(n/2)+(5/2)n-2 ==> O(nlogn)

There are (n/2)-1 spanning trees in K_{n-2} and (n-2) spanning trees in K_n; therefore, for every two nodes added, there is one more spanning tree. The algorithm contains 3 steps:

(1) extends the spanning trees in K_{n-2} to form (n/2)-1 spanning trees in K_n
(2) constructs a new spanning tree using unused edges:
- start from a node that was removed earlier
- follow an edge on the node that was not used to construct other spanning trees
- set the tree as the current new spanning tree
- set this edge as used and the other vertex as visited
- repeatedly, connect unvisited vertices adjacent to the current spanning tree via unused edges

(3) resets all vertices, so that they are unvisited when the next procedure calls
(4) adds the new spanning tree to the tree set

Now, there are n/2 spanning trees in the set of spanning trees. Each spanning tree in the set does not overlap each other in edges.