1. An (closed) *Euler tour* of a connected directed graph G = (V, E) is a cycle that traverses each edge of G exactly once, although it may visit a vertex more than once. Implement an O(E)-time algorithm to find an Euler tour of G if one exists.

Method :

A known fact : A graph G has an Euler tour if and only if in-degree(v) = out-degree(v) for each vertex v in V.

A simple cycle is a path in a graph that starts and ends at the same vertex without passing through the same vertex more than once. A complex cycle, is a cycle that passes through the same vertex more than once. We can easily decompose a complex cycle to a set of simple cycles by breaking up the cycle at those points where the cycle passes through the same vertex more than once.

Since an Euler tour is just a complex cycle it does not really matter which vertex v in the graph we start from. For each edge e that points away from vertex v we can construct one cycle that starts and ends at v. The same is also true for all the vertices that we visit during our exploration of one cycle. All we have to do now, is to decompose the graph into a proper set of edge-disjoint cycles and join them together in order to form an Euler tour. The easiest way to do this is recursively. We start at a random vertex and systematically follow each one of its edges while removing them from E. When our current vertex has no more out-edges we push it into the result stack and continue until E becomes empty.

```
Euler-Tour(G)
Initialize an empty result-stack S
Pick a random vertex v in V
    Euler-Tour-Recursion(G,v ,S )
Return S

Euler-Tour-Recursion(G,v ,S )
    For all edges e = (v, u) in E
        Remove e from E
        Euler-Tour-Recursion(G,u,S )
    EndFor
    Push v into the result-stack
Return S
```

It order to do an Euler tour we can pop vertices from S one at a time and follow the corresponding edges. While this recursion seems a bit complicated at first, all we need to do in order to analyze the running time of our algorithm is to realize that since we are removing one edge from E at each call, there can be at most |E| calls in total taking at most O(|E|) time.

2. In mathematics, and more specifically in graph theory, a tree is an undirected graph in which any two vertices are connected by exactly one path. In other words, any connected graph without simple cycles is a tree. The center of T is the node v in the tree that minimize the length of the longest path from v to any other node. Implement an O(n) time algorithm to compute the center of T .

Method:

We can find the center of the tree T using a very simple algorithm. We start with the full tree T and at each iteration we remove all leaves and the edges that connect each leaf to the rest of the tree (note that since this is a tree, there will be exactly one such edge for each leaf). The algorithm terminates when we are left with a single vertex and that vertex is the center of our tree. A number of implementation details need to be worked out in order for the algorithm to be efficient. Our data structure of choice will be a queue. The first step will be to add all leaf vertices to the queue, a step that can be performed in O(n) time by iterating through all the vertices and adding those with a single connected edge to the queue. From that point on all we need to do is: a) iteratively remove a leaf vertex v from the queue, b) remove vertex v and its corresponding edge e = (v, u) from the tree, c) If vertex u is now a leaf (has only one connected edge) add it to the queue, d) Check to see if a single vertex remains in the tree. If so return it as the center c, otherwise keep iterating. More formally our algorithm can be stated as follows:

```
Find-Center(T )
    For all v in V
        If v is a leaf add it to the queue
    End For
    While |V | > 1
        Remove a vertex v from the queue
        Remove vertex v and edge e from the tree
        Follow edge e = (v, u) to the next connected vertex u
        If u is a leaf add it to the queue
    End While
    Return the last element c in V
```

The first step iterates through all the vertices taking O(n) time. While a bit more subtle, we can show that the while loop will take O(n) time as well. Each vertex is added to the queue at most once and at each iteration of the while loop one vertex is removed from the queue. Therefore the while loop will execute less than n times. Thus we have proven that the total running time of our algorithm is O(n).