CS180 Solution 2

1. a Eulerian cycle of a graph G is a path which starts and ends on the same vertex and traverses every edge in the graph exactly once.

   (a) We have seen that an undirected connected graph $G = (V, E)$ such that all the vertices have even degrees has a Eulerian cycle. Give an $O(|E|)$ time algorithm to find it.

   Solution:

   We know that a Eulerian path (tour) can be decomposed by some simple cycle in G. Moreover, we know that starting from a vertex and arbitrarily traverse edges will eventually return to the starting vertex, which is a simple cycle. Hence, we can decompose the graph by these simple cycles and then connected these cycles to form a Eulerian tour. Here is the idea of the algorithm.

   Let us start at a vertex u and, via arbitrary traversal of edges, create a cycle and get back to vertex u, and if there are still edges leaving u that we have not taken, we can continue the cycle. Eventually, we get back to vertex u and there are no untaken edges leaving u. If we have visited every edge in the graph G, we are done. Otherwise, since G is connected, there must be some unvisited edge leaving a vertex, say v, on the cycle. We can traverse a new cycle starting at v, visiting only previously unvisited edges, and we can splice this cycle into the cycle we already know. That is, if the original cycle is $(u, ..., v, w, ..., u)$, and the new cycle is $(v, x, ..., v)$, then we can create the cycle $(u, ..., v, x, ..., v, w, ..., u)$. We continue this process until we have visited every edge.

```
EULERTOUR(G)
  T ← {}    //an empty list
  L ← (v)   //v is an arbitrary vertex
  while L is not empty
      remove u from L
      C ← get a simple cycle of u
      remove C from G and add a non-0 degree vertex from C into L
      if T = {}
          T ← C
      else
          splice C into T from u
  return T
```

   To see the algorithm takes $O(|E|)$ time, observe that because we remove each edge as it is visited, no edge is visited more than once. Since each edge is visited at some time, the number of vertices we processed in L is at most $|E|$. Hence, the total running time is $O(|E|)$.

   (b) A directed graph is strongly connected if every vertex is reachable from every other vertex. Assume that for every vertex in a directed graph $G = (V, E)$ its in-degree equals its outdegree, and G is strongly connected. Prove that G has a Eulerian cycle and give an $O(E)$ time algorithm to find it.

   Solution:

   We can apply the same strategy in 1(a). Pick one node v_i as the start, and then randomly pick one neighbor v_j to form the first step of the path. Since for each node, its in-degree is

equal to its out-degree. Then, there must be an out-going edge for v_j too. We repeat this progress until we meet v_i, which means we find one cycle. If there are still edges left, we repeatedly identify cycles. At last, we concatenate these cycles to form the Eulerian cycle given that the graph is strongly connected. Suppose we can't concatenate these cycles, it means that we won't reach some vertices from others. It contradicts with the definition of the strongly connected graph. In the above solution, we visit each edge once. So the time complexity is $O(E)$.

2. A celebrity among n persons is someone who is known by everyone but does not know anyone. Equivalently, given a directed graph $G = (V, E)$ with n vertices, a directed edge from v_i to v_j represents person i knows person j, a celebrity vertex is the vertex with no outgoing edge and $n-1$ incoming edges. In the class, we have seen an $O(n)$ recursive algorithm that finds whether celebrity exists or not and it does returns it. The graph G is represented by an $n \times n$ adjacency matrix M, which is a $(0,1)$-matrix such that $M[i,j] = 1$ if and only if there is a directed edge from v_i to v_j. Give an iterative $O(n)$ time algorithm to find the celebrity vertex in G, or output none if no one is.

   Solution:

```
FindCelebrity(M)
  c ← 1        # candidate
  for i ← 2 to n
      if M[c, i] = 1        # Candidate knows person i
          c ← i
  if IsCelebrity(c)
      return c
  else
      return None
```

```
IsCelebrity(c)
  for i ← 1 to n
      if i ≠ c and (M[i, c] = 0 or M[c, i] = 1)
          return False
  return True
```

3. Given a undirected tree T, the diameter of a tree is the number of edges in the longest path in the tree. Design an algorithm that find the diameter of the tree in $O(n)$ time where n is number of the nodes in the tree.

   Solution:

   BFS Version:

   We start the first BFS at any leaf x, and use it to find a furthest (leaf) node y from x. Then start the second BFS at y and use it to find a furthest leaf z from y. Return the number of rounds between y and z as the diameter. The way this works is that the first BFS finds one endpoint and the second finds the other. To prove this, we just need to prove that the first BFS has y as an endpoint of some longest path, because if we start a BFS from a longest-path endpoint y then any furthest node z from y will be the other endpoint of some longest path.

   We prove it by contradiction. If y is not the endpoint of some longest path, then there is some longest path ab for some leaves a and b. There are two cases:

(a) The path xy intersects the path ab at some node v. It must be that the distance vy is at least the distance vb, since our BFS picked y as a furthest leaf from x. But then avy is also a longest path, a contradiction.

(b) The path xy does not intersect the path ab. In this case, since we're working on a tree, there is a single node v on the ab path and a single node w on the xy path such that every path from x or y to a or b passes through v and w. Since y is a furthest leaf from x, the path $xwvb$ is not longer than the path xwy, so the path wvb is not longer than the path wy, and the path vb is not longer than wy. But then the distance $avwy$ is more than the distance avb, so $avwy$ is a longer path than ab, a contradiction

Recursive Version 1:

```
DIAMETER(T)
  if T has only one edge
      return 1
  if T is empty
      return 0
  Remove all leaves, get a reduced tree T'
  d ← 2+ Diameter (T')
  return d
```

Recursive version 2:

```
TREEDIAMETER(v)
  if v has no child
      return depth ← 0, diamter ← 0
  else
      for each child c_i:
          depth_i, diameter_i ← TREEDIAMETER(c_i)
          depth ← the largest d_i + 1
          diameter ← max { the largest diamter_i
                         { the largest depth_i + the second largest depth_j + 2   (if v has multiple children)
                         { depth
      return depth, diameter
```

The algorithm runs in $O(n)$ time since the function TREEDIAMETER is called on every node once and the depth and diameter are saved for every node.

4. Let $K_n = (V, E)$ be a complete undirected graph with n vertices (namely, every two vertices are connected), and let n be an even number. A spanning tree of G is a *connected* subgraph of G that contains all vertices in G and no cycles. Design a recursive algorithm that given the graph K_n, *partitions* the set of edges E into $n/2$ distinct subsets $E_1, E_2, ..., E_{n/2}$, such that for every subset E_i, the subgraph $G_i = (V, E_i)$ is a spanning tree of K_n.

   (Hint: Solve the problem recursively by removing two nodes and their edges in K_n. From the output of recursive call, you get $(n-2)/2$ spanning trees for K_{n-2}. Extend those trees to be spanning trees for K_n and then construct a new spanning tree to complete the job. PS: A collection

of sets $S_1, \ldots, S_k$ is a partition of S, if each S_i is a subset of S, no two subsets have a non-empty intersection, and the union of all the subsets is S.)

Solution:

We first prove it can be down by induction.

The base case $n = 2$ is trivial. Assume the statement is true for $n = 2k$. We need to show that it is true for $n+2 = 2k+2$. By induction hypothesis, there exists a spanning tree partition $E_1, ..., E_k$ of the complete graph on the first n vertices. We add two more vertices A and B, and label all the edges connecting A and B to the first n vertices by by $A_1, A_2, A_3, ..A_n$ and $B_1, B_2, B_3, ..B_n$ respectively (A_j denotes the edge between node A and j) and let AB be the edge connecting A and B. We show how to partition the complete graph on $n+2$ vertices into $k+1$ spanning trees as follows.

We extend every spanning tree E_i on the first n vertices by adding two edges A_i and B_{n-i}. In other words, we define a spanning tree E_i' for the graph on $n+2$ vertices as $E_i' = E_i \cup \{A_i, B_{n-i}\}$. This allows us to define k spanning tree $E_1', \ldots, E_k'$ on the graph with $n+2$ vertices. It's easy to see these spanning trees are all disjoint and no two of them share an edge. We still need to construct one more spanning tree using the remaining edges. The spanning trees $E_1', \ldots, E_k'$have used edges $A_1, \ldots, A_k$ and $B_{k+1}, \ldots B_{2k}$. The remaining edges are $A_{k+1}, \ldots, A_{2k}$ and $B_1, \ldots B_k$ and AB. But these remaining edges form a spanning tree. Therefore we have shown how to decompose a graph on $n+2$ vertices into $k+1$ spanning trees.

The recursive algorithm is an inverse of the previous inductive proof.

```
DECOMPOSE(K_2n)
if n = 1
   return the only edge in K_2n.
Choose two arbitrary vertices A and B, let K_2n-2 denote the graph of K_2n after
    removing A and B and their edges A_1,A_2,A_3,..A_n,B_1,B_2,B_3,..B_n and AB (defined in the proof above)
E_1, ..., E_n-1 ← DECOMPOSE(K_2n-2)
for i ← 1 to n-1
   E_i' ← E_i ∪ {A_i, B_n-i}
E_n' ← {A_k+1, ..., A_2k, B_1, ... B_k} ∪ {AB}
return E_1', ..., E_n'
```