Problem 22.4-2 (30 points):
Give a linear-time algorithm that takes as input a directed acyclic graph G(V,E) and two vertices s and t, and returns the number of simple paths from s to t in G. For example, the directed acyclic graph of Figure 22.8 contains exactly four simple paths from vertex p to vertex v: pov, poryv, posryv, and psryv.
(Your algorithm needs only to count the simple paths, not list them.)

Solution:
Idea:
We will use additional variable paths for each node u ∈ V to store the count of number of simple paths from node u to v. For each node u, the number of paths will be given by summing all the number of paths from each of node u’s neighbors. As the graph is acyclic, there won’t be any partially completed paths. We will keep the v constant (equal to the destination node t) for all the recursive invocations until each recursion reaches the destination node where the count of paths will be 1(base case). We will invoke the function as COUNT_SIMPLE-PATHS (s, t).

Algorithm:
COUNT_SIMPLE-PATHS(u,v)
if u == v then
    return 1
else if u.paths != NIL then
    return u.paths
else
    for each x ∈ Adj[u] do
        u.paths += COUNT_SIMPLE-PATHS(x, v)
    end for
    return u.paths
end if

Complexity Analysis:
The total number of executions of the for loop in all the recursive invocations would be O(V+E).

Problem 22.3(40 points):
An Euler tour of a strongly 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.
a. Show that G has an Euler tour if and only if in-degree(v) = out-degree(v) for each vertex v belongs to V.
b. Describe an $O(E)$ time algorithm to find an Euler tour of $G$ if one exists. (Hint: Merge edge-disjoint cycles.)

**Solution:**

**a)**

*Part 1:* If $G$ has Euler tour then in-degree(v) = out-degree(v) for each vertex $v$ belongs to $V$.

As $G$ is strongly connected, we can decompose $G$ into multiple edge-disjoint cycles when combined will give the Euler tour. For each of these cycles, every vertex has to have an edge coming into it and an edge going out of it to complete the cycle. This implies that in-degree(v) = outdegree(v) for each vertex $v$ part of the cycle. Now, for the entire graph as we have an Euler tour, each disjoint cycle should be connected such that each cycle has an edge going out of it and an edge coming into it. Thus, we can say that for each vertex $v$ in $V$, in-degree(v) = out-degree(v).

*Part 2:* If in-degree(v) = out-degree(v) for each vertex $v$ belongs to $V$, then $G$ has Euler tour.

Let us start at a vertex $u$ and, via random traversal of edges, create a cycle. We know that once we take any edge entering a vertex $v \neq u$, we can find an edge leaving $v$ that we have not yet taken. Eventually, we 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 merge this cycle into the cycle we already know. That is, if the original cycle is $u, \ldots, v, w, \ldots, u$, and the new cycle is $v, x, \ldots, v$, then we can create the cycle $u, \ldots, v, x, \ldots, v, w, \ldots, u$. We continue this process of finding a vertex with an unvisited leaving edge on a visited cycle, visiting a cycle starting and ending at this vertex, and merging in the newly visited cycle, until we have visited every edge forming Euler tour. Thus, we can say that $G$ has Euler tour, if in-degree(v) = out-degree(v) for each vertex $v$ belongs to $V$.

**b)**

**Idea:**

Assuming $G$ is represent by adjacency lists, we make an additional copy of this adjacency list to track the edges yet to be explored. We will return the output i.e. Euler tour in the form of doubly linked list $T$. The algorithm uses the proof for part 2 in answer to part a) above. We keep on building $T$ by finding edge disjoint cycles and merging them in original doubly linked list.

We also maintain another singly linked list $L$ in which each list element consists of two parts:
1. a vertex v, and 2. Position pointer of v in T.
This is used while coalescing a newly built cycle into T.

Initially, L contains one vertex, which may be any vertex of G.

**Algorithm:**
**EULER-TOUR(G)**

T - an empty list
L – Tuple (any vertex v, NULL)
while L is not empty
    do remove (v, position-in-T) from L
    C = VISIT(v)
    if position-in-T = NULL then
        T = C
    else
        Coalesce C into T just before position-in-T
    endif
end while
return T

**VISIT(v)**

C - an empty sequence of vertices
u - v
while out-degree(u) > 0
    do let w be the first vertex in Adj[u]
       remove w from Adj[u]
       out-degree(u) -= 1
       add u onto the end of C
       if out-degree(u) > 0 then
           add (u, u’s position in T) to L
           u = w
       end if
end while
return C

**Time Complexity:**
Because we remove each edge from its adjacency list as it is visited, no edge is visited more than once. Hence, time complexity would be O(E).
**Problem 23.2-1 (30 points):**
Kruskal’s algorithm can return different spanning trees for the same input graph G, depending on how it breaks ties when the edges are sorted into order. Show that for each minimum spanning tree T of G, there is a way to sort the edges of G in Kruskal’s algorithm so that the algorithm returns T.

**Solution:**
Suppose we want to obtain MST as T using the Kruskal’s algorithm. We will first order the edges first in increasing order of their weights and in case of ties i.e. edges with same weights, we will give more priority to the edges part of T over those not part of T. Since we prioritize the edges in T, we will always pick them up over other edges part of other MST’s of G.