Problem 1 (30 points, Problem 22.2-7) There are two types of professional wrestlers: “babyfaces” (“good guys”) and “heels” (“bad guys”). Between any pair of professional wrestlers, there may or may not be a rivalry. Suppose we have n professional wrestlers and we have a list of r pairs of wrestlers for which there are rivalries. Give an O(n + r)-time algorithm that determines whether it is possible to designate some of the wrestlers as babyfaces and the remainder as heels such that each rivalry is between a babyface and a heel. If it is possible to perform such a designation, your algorithm should produce it.

Solution. We create a graph by representing the wrestlers as vertices and the rivalries as edges. Then, choose one vertex at random. Let this vertex be the root and a “good guy” (It can be a “bad guy”. It’s your choice). We run the breadth-first search (BFS) from the root and keep updating their distances from the root. All vertices with odd distances will be “bad guys” and with even distances will be “good guys”. If you find two adjacent vertices being allocated to the same group, then return false. The time complexity of this algorithm is O(n + r), which is the same as that of BFS.

Problem 2 (30 points, Problem 22.4-2) 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.

Solution. We will use additional variable paths for each node u ∈ V to store the 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. We will invoke the function as COUNT_SIMPLE_PATHS(s, t).

Pseudocode:

Algorithm 1 Problem 22.4-2: COUNT_SIMPLE_PATHS
Input: G = (V, E), s, t
Output: s.paths
1: if s == t then
2: return 1
3: else if s.paths ≠ NIL then
4: return s.paths
5: else
6: for each (s, u) ∈ E do
7: let s.paths+ = COUNT_SIMPLE_PATHS(u, t)
8: end for
9: return s.paths
10: end if
Problem 3 (40 points, Problem 22-3) 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 $\text{in-degree}(v) = \text{out-degree}(v)$ 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.  

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

As $G$ has Euler tour and the Euler tour is made up of multiple edge-disjoint cycles. 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 $\text{in-degree}(v) = \text{out-degree}(v)$ 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 $V$, $\text{in-degree}(v) = \text{out-degree}(v)$.

Part 2: If $\text{in-degree}(v) = \text{out-degree}(v)$ 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 unvisited 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 $u, v, w, ..., u$, and the new cycle is $v, x, ..., v$, then we can create the cycle $u, v, x, ..., u$. 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 $\text{in-degree}(v) = \text{out-degree}(v)$ for each vertex $v$ belongs to $V$.

b) The algorithm is as follows:

Pseudocode:

The time complexity is $O(|E|)$. 
Algorithm 2 Problem 22-3: Euler-Tour

Input: $G = (V, E)$

Output: $T$

1: for each vertex $v \in V$ do
2:     if in-degree($u$) $\neq$ out-degree($v$) then
3:         return "NO EULER TOUR"
4:     end if
5: end for
6: start from an arbitrary vertex $u \in V$ and find a cycle $C$ without repetitive edges
7: let $T = C$
8: remove edges of $C$ from $E$
9: while $E \neq \emptyset$ do
10:    find a vertex $u \in T$ such that there is an edge $(u, v) \in E$
11:    find a cycle $C$ starting with $(u, v)$ without repetitive edges in $G = (V, E)$
12:    remove edges of $C$ from $E$
13:    let $T = T \cup C$
14: end while
15: return $T$