Ex 1 (CLRS 22.2-7)

This is a modified version of the two coloring problem, so we can use a similar algorithm and color vertices of the graph of rivalries by two colors, "babyface" and "heel".

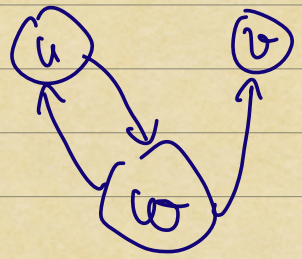
Coloring is proper when no two babyfaces and no two heels have a rivalry.

To two color, we implement a breadth first search of each connected component to get the d values for each vertex.

We can give all odd d values one color (ex heel) and even ones other color (babyface). We know that no other coloring will succeed where this one fails since if we have any other coloring we would have a vertex v that has the same color as v.p and since v and v.p must have different parities for their d values.

We now know that there is no better coloring so only thing we need to do is check each edge to see if the coloring is valid. If each edge works, it is possible to find a designation, if a single edge fails, then it is impossible. BFS took $O(n+r)$ time and checking edges took $O(r)$ time, total is $O(n+r)$.

Ex2 (CLRS 22.3-8)



Lets consider a graph with 3 vertices (u, v, w) and 3 edges (w,u) (u,w) and (w,v) shown in picture to the left. Suppose DFS first explores w, and that w's adjacency list has u before v. Now we discover u, which has w as the only adjacent vertex, but w is already visited, so u finishes. Since v is not yet a descendant of u and u is finished, v can never be descendant of u even though $u.d < v.d$.

Ex3 (CLRS 22.5-6)

First, we create a component graph in $O(V+E)$ time and we label each node with its component as we go:

For each vertex, we will give it an entry SCC, so that $v.SCC$ denotes the strongly connected component (vertex in the component graph) that v belongs to. Then, for each edge (u,v) in the original graph we add an edge from $u.SCC$ to $v.SCC$ if one does

not already exist. This only takes $O(|V|+|E|)$ time, then we need just constant amount for checking the existence in component graph and adding one if need be.

We also create a list for each component which contains the vertices in that component by forming an array A such that $A[i]$ contains a list of vertices in the ith connected component.

Now run DFS again and for each edge, check weather or not it connects 2 different components. If it doesn't delete it. If it does, determine weather it is the first edge connecting them, if not delete it. This can be done in constant time per edge since we can store the component edges into in a $k \times k$ matrix where k is number of connected components. Thus, runtime is $O(V+E)$.

Here, only edges we have are a minimal number which connect the k distinct connected components. Now we need to place edges within the connected components in the minimal way. The fewest edges which can be used to create a connected component with n vertices is n, and this is done with a cycle.

For each component, let $v_1, v_2, \ldots, v_n$ be the vertices in that component. We find them by using the array A created earlier. Add the edges $(v_1, v_2), (v_2, v_3), \ldots, (v_n, v_1)$. This is a linear number of vertices so total runtime is $O(V+E)$.