Problem B-1

a. Take a node x in tree T, which has a parent p and two children i and j. Because of the layout of a tree, of p, i, and j, none are adjacent. We can therefore color x 0 and p, i, and j 1.
   The same holds for p, i, and j, if we treat them as we did x; they are colored 1, and all of the adjacent nodes (none of which are touching) can then be colored 0.
b. A bipartite graph (1) is a graph which can be grouped into two disjoint sets, U and V, such that every edge connects a node in U with a node in V. We can simply color all nodes in U one color and all nodes in V another; any edge will have a different color on each end as a natural extension of being bipartite, so a bipartite graph is 2-colorable (2).
   Finally, even if every possible edge exists without making a bipartite graph into a graph that isn't bipartite, to get from any node U or V to any other node U or V respectively, we *must* traverse two edges – if our starting node is in U, we must travel to a node in V before we can go to a node in U, and vice versa. Therefore, any path starting and ending at the same node (a cycle) must traverse a number of edges that is a multiple of two – more simply, it must be even (3).
c. To color G, we must use at least d+1 colors – one color for the node with the maximum number of degrees, and one color for each node it's connected to.
   However, even if every other node has the same degree d, any one node will never be connected to more than d differently-colored nodes – and so we can color it with the last color left in our set of d+1 colors.

Exercise 22.1-1

This depends on the implementation of the list. If lists have a constant time to check the length, we simply check the length O(1) of each node in the list of nodes O(n), so it's linear. If checking the length requires us to step through it, then it's O(n*d), where d is the degree of the node with the most degrees, and d < n.

To check in-degrees, it will take O(n*d) (and O(n) space), as we need to check the list of every node's out-degrees and see which nodes are being pointed at.

Exercise 22.1-3

Adjacency list:

We can make a new list of length n (where n is the number of nodes), and traverse down the adjacency list. More precisely, a loop n ∈ G with an internal loop of u ∈ E(n), where for each node u we add n to node u in our new list. This would be O(n*d), where d is the degree of the node with the largest degree.

Adjacency matrix:

We can simply translate across the diagonal of the matrix (or rather, have a loop y within a loop x and switch matrix[x][y] with matrix[y][x], skipping cases where x == y). This would be O(n^2).

Exercise 22.2-2

U: 0, NIL	T, X, Y: 1, X	W: 2, T	S: 3, W	R: 4, S	V: 5, R

Exercise 22.3-2

Q: NIL	0,	15
S: Q	1,	6
V: S	2,	5
W: V	3,	4
T: Q	7,	14
X: T	8,	11
Z: X	9,	10
Y: T	12,	13
R: NIL	16,	19
U: R	17,	18

Exercise 22.3-12

In DFS(G), initialize cc as 0. Then, within the statement “if u.color == WHITE”, add cc += 1 before DFS-VISIT(G, u), and change that to DFS-VISIT(G, u, cc)

Then change DFS-VISIT(G, u) to DFS-VISIT(G, u, cc), and add the line “v.cc = cc” in the body of the “if v.color == WHITE” statement.