Problem # 1

It is impossible for a comparison sorting tree to have more than $n!$ leaf nodes. Each leaf node repxresents a reordering of the original elements in the provided list. There are $n!$ different ways to order n elements and therefore there will be exactly $n!$ leaf nodes. There is no information provided about the initial ordering of the elements and therefore each reodering has equal probability of $\frac{1}{n!}$. Asking what happens when there are more than $n!$ leaves is misleading because this is not possible.

Problem # 2

Proof by contraposition. Assume that a graph G is unconnected. This means that the nodes of G can be partitioned into two subsets S_1, S_2 such that there is no edge that connects any node of S_1 to any node of S_2. Consider the sizes of S_1 and S_2. It is clear that one of them must contain $n/2$ or fewer nodes. Select any arbitrary node in this subset. It can at most be connected to every other node in the subset. This means that its degree is at most $n/2 - 1$. Thus, in any unconnected graph with n nodes there will be a node with degree less than $n/2$. Thus by contraposition if every node in a graph has degree of at least $n/2$ then the graph is connected.

Problem # 3

Begin with the standard depth first search algorithm as discussed in class. Modify it so that it timestamps each node v at its time of discovery and at the time when all of the nodes in v's adjacency list have been examined, i.e. v has been finished. We can now classify the edges using this information. An edge (u, v) is a descendant edge if the discovery time of u is earlier than the discovery time of v which is earlier than the finish time of u which is earlier than the finish time of v. The edge is a cross edge if the discovery time of v is less than the finish time of v which is less than the discovery time of u which is less than the finish time of u.

Problem # 4

The first task is to construct the graph G that we will be operating on. In our graph edges will represent flights and vertices will represent departure and destination locations. For example, if node u stores the information NYC and 5PM and node v stores DC and 6PM and there is an edge connecting the two nodes then this indicates that there is a flight leaving NYC at 5 that will land in DC at 6. First set up nodes and edges in this way for all of the required flights. Next we need to determine when it is possible for a single plane to be used for multiple flights. Consider the start node and end node for any two

of the required flights. We will add an edge from the end node of the first flight to the start of the second if it is possible for a single plane to do both of the flights. There are two cases when this is possible. If the locations are the same at the two nodes and the takeoff is less than m minutes after the landing then the edge is added. If the cities are different and the takeoff is less than $2m + l$ after the landing then the edge is added. For each edge representing a required flight we will set a capacity of one because there's no need for it to happen multiple times and a lower bound of one because it must happen at least once. The optional flights will have a capacity of one and a lower bound of zero. We want to determine if there is a feasible circulation in this network. In order to reduce this problem to a max flow problem as discussed in class we add a super-source node and a super-sink node. The source node has an edge connecting to the starting node of each required flight that has a capacity of 1 and a lower bound of 0. This represents that a plane can begin the day at any location. The destination of each required flight has an edge than connects to the super-sink with capacity 1 and lower bound 0. Then we can assign demands of -k to the super-source, k to the super-sink, and 0 to every other node in the graph. Finally an edge connecting the source to the sink must be added with a lower bound of 0 and a maximum capacity of k to reflect the fact that we may not need to use all k planes for the required flights. Next we modify G to remove lower bounds as decscribed in class so that the max flow algorithm can handle the graph. Alternatively we can use a max flow implementation that accounts for lower bounds on its own. In either case we then run Ford-Fulkerson on the graph. If the max flow in G is equal to k then we know that there is a feasible circulation and the flights can be completed with k or fewer planes.

Problem # 5

The first step is to construct a new graph that represents all of the possible states of the system. First number all of the nodes in the original graph 1 through n. Then nodes in the new graph will be tuples that represent valid locations of the two robots. For example, one node might store the information (4, 9) indicating that robot one is on node 4 and robot two is at node 9. Next we need to add edges to our new graph. We insert these edges between nodes when it is possible to reach one state from the other by moving one robot. So the node (x, y) would connect to node (x, z) only if there is an edge between nodes y and z in the original graph. Note that one of the elements of the tuple must remain the same because a single state change consists of one robot moving once. Then simply run a shortest path algorithm on the resulting graph with the node storing (s_1, s_2) as the start node and (d_1, d_2) as the finish node. If there is not path then the problem is not solvable, otherwise the schedule can be reconstructed by examining the states along the calculated shortest path. The number of valid states for r robots is maximized when the V nodes in the initial graph are divided equally into r isolated components that are fully connected within themselves. This is because this arrangement minimizes the number of

conflicts between robots. There are $(V/r)^r$ possible states. For each of these states we want to know how many adjacent states there are. In the worst case each of the r robots can move to any of the adjacent $(V/r - 1)$ nodes. Thus the number of edges in the new graph is $(V/r)^r(V/r - 1)r$. Thus the algorithm has a run time of $O(V^2 + V^3)$ in the case of two robots. This can be generalized for r robots to $O\left(\left(\frac{V}{r}\right)^r + \left(\frac{V}{r}\right)^r \left(\frac{V}{r} - 1\right) r\right)$.