Jonathan Tso
CS 401
HW 2

1. a) True — m find n union (m log n)

Consider that the union is the connection of 2 trees at the root. Then, every time we do a union, we are effectively cutting the number of disjointed sets in half. This imitates log n behavior. Because we did not change structure via compression, we will perform the equal m find operations.

b) True — If every weight is unique, then there must be a spanning tree that contains the smallest weights possible. Because of this, we can argue that if you would modify this MST, then it is no longer the smallest possible weights so you are no longer part of the unique MST.

c) True — If we consider that all MST's have lightest edge e, then we can look at any spanning tree without e. If we say an MST exists without e, then we can arbitrarily remove any edge and add e in, making another MST that does include e and costs less than it is a real MST.

2. We can apply the reece-delete methodolgy. However, instead of deleting the maximum weighted edges, we delete the minimum weight, if it does not cause a disconnect. This way, we retain all the largest values and still have a tree that spans.

3. This is the 2 colorable problem. We can approach this similarly to BFS.

1. Choose an arbitrary node and color it red.
2. Expand out for each leaving edge and color the new vertex blue.
3. For every new blue edge, go to its outgoing edges and look at the next vertex. If it is not already colored color it red and continue. If it is blue, we have determined it is not possible.
4. Continue until all the vertices are colored. If we reach the end, it can be 2-colorable.

This runs in $O(V+E)$ because we run into each vertex and edge only once, so we never have to do more. We can implement this by using a queue since we have an adjacency matrix.

4. Consider the following algorithm:

1) Sort the jobs in terms of deadlines in increasing order
2) Next, sort the jobs in terms of penalty, in decreasing order
3) Look at the first job. It should be with the earliest deadline. Look at the penalties for any such job and pick the

Choose an arbitrary vertex and color it red.

2. Expand out for each leaving edge and color the new vertex blue.
3. For every new blue edge, go to its outgoing edges and look at the next vertex. If it is not already colored ~~or is red~~ color it red and continue. If it is blue, we have determined it is not possible.
4. Continue until all the vertices are colored. If we reach the end, it can be 2-colorable.

This runs in $O(V+E)$ because we run into each vertex and edge only once, so we never have to do more. We can implement this by using a queue since we have an adjacency matrix.

4. Consider the following algorithm:
   1) Sort the jobs in terms of deadlines in increasing order
   2) Next, sort the jobs in terms of penalty, in decreasing order
   3) Look at the first job. It should be with the earliest deadline. Look at the penalties for any such job and pick the one with the greatest penalty (we get rid of the biggest issue).
   4) Repeat step 3 until we run out of our time slot.

The runtime will be $O(n \log n)$ due to our sorts, and that steps 3 and 4 should be $O(n)$ since we only run through the list of jobs.

This minimizes penalty because for every time slot, we choose the jobs with the earliest deadlines and of those, we eliminate the jobs with the biggest penalties.

5. Let us consider a graph that is already existing with d vertices. These d vertices are the n natural numbers. Now, let us make an array of these vertices. This takes $O(n)$ time. Sorting this to be in decreasing order will take $O(n \log n)$ time.

Now, consider that we take the largest natural number node d and then remove its edges from the graph. We can take all the vertices that used it as a connecting edge and create a new edge for each to preserve the graph. Then, add d back, but have every vertex have an edge to it. Thus, we can create a graph that will have d degrees. This can be done for all natural numbers and will take $O(n \log n)$ for the sorting.

6. Consider that S_1 and S_2 are in 2 arrays, where S_2 is the shorter,

1) For the 1st index in S_2, go from the beginning of S_1 until you find a match. If you don't, return false.
2) If you found a match, go to the next index in S_2 and go through S_1 until you find a match. If you do not, repeat step 1 but from where you left off in S_1.
3) Repeat 2 until you can confirm that all of S_2 is in S_1. Then return true, else false if you cannot find a match based on S_1.

The worst case runtime is $O(n^2)$ in case the word of S_2 is at the end of S_1, but there are multiple beginnings of it inside S_1.

7. Restriction, has n+8 edges, meaning that to get an MST, we must break any cycle (so we need to remove 9 edges) Then, we know that since all costs are distinct, there are 9 maximal edges we can remove.

1) Run BFS on the graph starting on an arbitrary node.
2) If BFS determines that there is a cycle, find and delete the edge with the highest cost. If not, go to 4.
3) Repeat 1 and 2 for a total of 9 times, or until BFS has determined no cycle.
4) End algorithm since there is no cycle, so we have an MST.

This will run in O(n) time because we are only running BFS which is linear to the total edges and weights. Also, we do this to a max of 9 times, so at worst it is 9n which is O(n).