Faith Twardzik TA: Goldstein, O. Time: 2-3:50 pm, Section: 1F

Question 8b)

- Using the algorithm that the professor introduced in class, with k=2 jars, we designed a set-up with $\sqrt{n}$ partitions and $\sqrt{n}$ steps per partition. Using this set-up, we determined that if each partition was at a multiple of $\sqrt{n}$, and if the egg (jar in this case) broke at some multiple $x\sqrt{n}$, then we would continue dropping the egg at $(x - 1)\sqrt{n}$ and continue until we get to $x\sqrt{n} - 1$. Essentially, this would give us a worst-case runtime of $O(2\sqrt{n})$, which is just $O(\sqrt{n})$, because the first egg would be dropped at most $\sqrt{n}$ times, and then the second egg would be dropped at most $\sqrt{n}$ times (steps) within the partition we found.
    - For this problem, however, we are looking at the general case of k jars. So, we can generalize the above strategy. Since we determined the 2-jar case to have a worst-case try number of $2n^{1/2}$, we can generalize the worst-case try number for k jars to be $2kn^{1/k}$ times.
        - This means our partition sizes (and step sizes) will be at each subsequent multiple of the floor of $n^{(k-1)/k}$, which we go on to prove using induction.
        - The base case is the 1-jar case (k=1), and this jar will be dropped at most (worst case) $2(n / n^{(k-1)/k})$ times. This is because n is the last step, and we are using $n^{(k-1)/k}$ partitions, so this will in turn give us the number of times the jar will be dropped at most, which simplifies down to $2kn^{1/k}$ where k in this case is 1, and so the number of times is 2n, which is what we would expect
        - Using $2kn^{1/k}$ as our induction assumption, we then attempt to use this for the case where we have k – 1 jars, where the equation will be:
            - $2(k-1)n^{1/k}$
            - Since this is less than or equal to the upper bound of $2kn^{1/k}$, then we know for the k – 2 , k – 3, and so on cases, the worst-case number of tries will also be less than $2kn^{1/k}$, and so we have determined that this is the worst-case number of tries for any k number of jars

Question 5)

- We use proof by induction to show that any binary tree has a number of nodes with two children equal to one less than the number of leaves (leaves – 1 = nodes with two children)
    - First, we define the base case:
        - We have a tree with one node, in which case this node is both the root and a leaf, and since the number of leaves = 1 and the number of nodes with two children = 0, the above equation is satisfied (1 – 1 = 0)
        - Next, we define a scenario in which we have a tree called Tree1, with a number of leaves equal to L_{Tree1} and a number of nodes with two children equal to N_{Tree1}. We also have a subtree within Tree1, called Subtree1, with a leaf Leaf that is not the root of the tree, in which case we can assume that Leaf has some parent Parent.
        - By induction, we are going to look at a subtree of Subtree1 with one less leaf and show that for this subtree, the number of nodes with two children is one less than the number of leaves as well
        - First, we get rid of the leaf Leaf, resulting in the new subtree Subtree2, and we split this problem into two different outcomes:

- Parent only had one child (Leaf), and now has no children, in which case $L_{Subtree1} = L_{Subtree2}$ and $N_{Subtree1} = N_{Subtree2}$. This satisfies our relation, because neither the number of nodes with two children is changing (Parent only had one child to begin with) nor is the number of leaves changing (now Parent becomes a leaf, in place of Leaf). Since we assumed the relation was true of Subtree1, since the outcome for Subtree2 is the same, the relation is satisfied.
- The second outcome is where Leaf is deleted, but Parent had two children to begin with, so now Parent still has one child. In this case, the subtree has one less leaf (because Leaf was deleted, and Parent does NOT become a leaf), and one less node with two children (because Parent now has one child only).
    - So, $L_{Subtree1} = L_{Subtree2} - 1$ and $N_{Subtree1} = N_{Subtree2} – 1$, and because we assumed the relation was satisfied for Subtree1, if L and N for Subtree2 are both decreased by the same amount (1), then by induction we have proven the relation holds for Subtree2 as well

Question 7)

- We use a proof by contradiction to prove that the claim is true and graph G must be connected if it has an even n nodes where each node has at least a degree of $n/2$.
    - First assume for the sake of contradiction that this graph G with the above properties is not connected.
        - This means that there are at least two separate components of the graph that are connected by themselves (but not to one another, because then the entire graph would be connected)
    - If we have at least 2 connected components, then the smallest of these can have at most $n/2$ nodes (or less) and the largest of the two will have at least $n/2$ nodes (or more)
    - Then, pick an arbitrary node in the smallest connected component, node a
    - Based on our assumptions, node a can have at most $n/2 - 1$ incident edges (based on the fact that any undirected graph has edges equal to the total number of nodes - 1)
    - However, this is a contradiction, because the problem stipulates that every node must have at least $n/2$ incident edges.
        - Since $n/2 - 1 < n/2$, we have a contradiction, so the claim must be true

Question 10)

- We know that we can use one of the two search methods implemented in Chapter 3 to determine the number of shortest paths from v to w, or from v to any node, if v is taken to be the root (beginning node) of the search.
    - Here, we will use a Breadth-First Search, because we know that a BFS will not only determine all the nodes that a certain node such as v can reach, but also the shortest paths to them, given the fact that:
        - For each j greater than or equal to 1, layer L_j will have all of the nodes a distance of j from node v, where there is a path from v to w if and only if w is in one of these layers
        - Knowing this, we implement the BFS algorithm given on page 90 of the textbook, where the first layer contains the node v only
            - Then, we define SP_w to be the shortest path from v to w
                - As the base case, every node w that is in the first layer will have an SP_w value of 1, because there is only one edge: (v, w)
                - For all other nodes w on some layer other than the first layer, say Layer k, the shortest v-w path will go to the layer before k (Layer k-1) at some node (x), and then jump one more edge, from that node (x) to the node w on Layer k (along edge (x, w))
                - So, the SP_w will be the SP_x (the summation of all of the shortest paths to a node in Layer k-1 that also has an edge leading to w in Layer k)
                - This was our induction step; we can do this again by saying that the shortest path to x (SP_x) is the sum of all the shortest paths to a node in Layer k-2 that also has an edge leading to x in Layer k-1, and so forth until we return to our root v
                    - So, this satisfies the induction hypothesis
            - Because we know that a BFS algorithm's runtime is O(m + n) (we can justify this by recalling that O(n) time is necessary to set up the lists and manage the Discovered nodes array, while the time spent in the For loop is O(m), because 2m edges will be considered, and with the added summation calculation (which will be the number of incident edges to w, giving us O(m)), then the runtime will remain at O(m + n).

Question 5)

This is simple, in the sense that we can do the following:

- Input the entire sequence of real numbers (plus the integer k) into the black box
    - If the algorithm returns with "NO" then we are done; there is no subset of the sequence of real numbers that sums to k
    - If the algorithm returns with "YES", then input the sequence of real numbers **without** the first number in the sequence
        - If the algorithm returns "NO", then we are done; the subset that sums up to k is the sequence with the first number
        - If the algorithm returns "YES", then we again delete the first number in the new subsequence, and run it through the black box. We keep running this and the

previous bullet point until the algorithm returns “NO”, at which point we have found our subset, which is the current subsequence plus the last number that was removed

Question 6)

a)

Here we will be using the binary search algorithm and splitting our search size into two with every iteration of our search. The logic is such;

- if we choose the number at the middle index in our array and that number is **greater** than the middle number in our sequence from 1 to n+1,
    - then the missing number must be in the left half of the array (because we have one less lower number on the left side, so the “middle index” number is one higher than it should be)
        - and we half the left side and do the same search,
    - otherwise it is on the right side and we do the same search with the right side
- So, the algorithm is as follows:
    - Given array p, choose the number at index p[n/2]. If p[n/2] = n/2 + 1 (this will work for even or odd values of n), then split the array in half and search only the left half
        - Else, split the array in half and search only the right half
    - Continue until you have recursed down to the missing number

This algorithm has a time complexity of O(logn), because we know that binary search requires this time complexity, given that:

- After an arbitrary k number of “probes” the size of the search space will be at most $(1/2)^k n$, where when k = logn the search space becomes constant. Since a constant fraction (1/2) of the search space is being decreased after every run of this search, then this is an O(logn) time complexity

b)

Here, since we have no pre-determined sorting, the runtime will have to be O(n), because we will, in the worst-case scenario, have to search through all n numbers in order to find which is missing – if we do not search through some subset of n, we have no way of knowing what values were in that subset, and if our determined missing value was, in fact, missing.

So, instead, we follow this algorithm:

- Initialize a temporary array Temp of size n with every element equal to 0 (this will require O(n) runtime)
- Iterate through the array p that was given, and for each element,
    - Set Temp[p[i]] = 1
        - this denotes that the value p[i] is in the array p and so is **not** the missing value (the runtime for iterating through the entire array p will be O(n), because we visit all n elements, and the setting of the Temp array’s value is constant O(1))
- After we have visited all of p’s entries, iterate through all of Temp’s elements
    - If Temp[i] = 1, continue
    - If Temp[i] = 0, then this value was missing in the array p, and return this missing value

- The last step incurs a worst-case runtime of O(n), because it is possible that we will have to iterate through all of Temp, until its last element, to find the missing number

Therefore, simplifying the runtimes, the time complexity comes out to be O(n).