Charles Zhang

Disc. 1D
Fri. 10am - 1pm
Song

1)

- Algorithm:
  - // Assuming the jar will break at some point in the ladder
  - Set an initial starting point *s* at the bottom of the ladder
  - While there are jars remaining:
    - *n* is equal to the current height of the target area
    - Calculate a partition size *m* = ceil($n^{k-1/k}$), which leaves $n^{1/k}$ partitions to be checked
    - Set initial height *h* equal to *m* + *s*
    - While there is still a full-size partition to check:
      - Drop the current jar
      - If the jar breaks, return to the last partition's starting point and break from the loop
      - If the jar remains intact, add *m* to the height *h*
    - Set the new size of the target area to *m*
    - Set the new starting point *s* to the current height *h*
  - Return the rung below the current height, as the current height is the earliest rung where the jar broke, therefore the previous rung must be the highest safe rung
- Proof:
  - In order for the algorithm to find the highest safe rung, the algorithm must have cleared every single rung below the result
  - The algorithm moves upwards from the bottom of the ladder in partitions
  - In the final partition, the algorithm is guaranteed to be checking "partitions" of size 1 from the bottom upwards until it finds the first rung in which the jar breaks
  - Therefore, within this final partition, every single rung below the result is guaranteed to have been checked
  - Since the algorithm checks both the starting point and ending point of each partition it's presented with, we can think of the previous level of partitions as "rungs"
  - As a result, the same logic used for the final partition can then be extended to apply to the larger partitions
  - In any given partition, the algorithm is guaranteed to be clearing partitions from the bottom upwards until it finds the first partition in which the jar breaks
  - Therefore, within any given partition, every single partition below the suspect partition is guaranteed to have been cleared
  - By extension, every single rung below the final result is guaranteed to have been cleared
- Time Complexity: O($n^{1/k}$)
- Time Complexity Proof:

- The initial number of partitions is $n^{1/k}$, which implies there will be partitions of size $n^{k-1/k}$ initially
- This tells us that our next target area will be of size $n^{k-1/k}$
- Since moving to the next partition also implies breaking a jar, k will be reduced by 1
- This means that the number of partitions in the next step of the algorithm can be accurately represented by the total size of the target area divided by the size of each partition
- The size of each partition is represented by the target area raised to the power of $k - 1 / k$ where k is the number of jars remaining
    - It follows that the next section's partition size will be equal to $(n^{k-1/k})^{k-2/k-1}$, or $n^{k-2/k}$
- Mathematically, this means the number of partitions is equal to $n^{k-1/k} / n^{k-2/k}$, or $n^{1/k}$
- Inductively, this applies to every following partition
- Therefore, the number of partitions to search will always be $n^{1/k}$
- This means that the number of searches will be $kn^{1/k}$
    - Since k is a much smaller value than n generally, we can reduce this to think of k as a constant, leaving us with an $O(n^{1/k})$ solution
- The function $f_k(n) = n^{1/k}$ satisfies the limit condition provided by the problem, as any function $n^{1/k}$ is larger at all values of n than $n^{1/k+1}$

2)

- Base Cases:
  - A tree with only a root node:
    - # of leaves: 1
    - # of nodes with 2 children: 0
    - 0 = 1 - 1 → correct
  - A tree with a root node with a single child node:
    - # of leaves: 1
    - # of nodes with 2 children: 0
    - 0 = 1 - 1 → correct
  - A tree with a root node with 2 child nodes:
    - # of leaves: 2
    - # of nodes with 2 children: 1
    - 1 = 2 - 1 → correct
- Induction:
  - Assume: # of leaf nodes = 1 + # of nodes with 2 children for a tree with *k* nodes
  - Prove: # of leaf nodes = 1 + # of nodes with 2 children for a tree with *k* + 1 nodes
    - There are 2 options for adding a node:
      - A node can be added to a node with no child nodes
        - The node with no child nodes, which used to be a leaf, is no longer considered a leaf
        - The new node is a leaf
        - The number of leaves and nodes with 2 children didn't change
      - A node can be added to a node with 1 child node
        - The node with 1 child node now has 2 child nodes
        - The new node is a leaf
        - The number of leaves and nodes with 2 children both increased by 1
    - It is impossible to change either the number of leaves or the number of nodes with 2 children without changing the other by the same amount
    - By induction, it's been proved that a tree must have exactly 1 more leaf than nodes with 2 child nodes

3)

- Proof:
  - The claim is true
  - Assume it isn't true, and there can be 2 separate connected components (the minimum requirement to prove the claim false)
  - Take a user/node n_1
  - According to the problem statement, n_1 must be connected to at least n/2 nodes, which means the connected component containing n_1 contains, at minimum, n/2 + 1 nodes
  - Using the same logic for a user/node n_2 in the other connected component, we can say that n_2's connected component must also have at least n/2 + 1 nodes
  - In order for these connected components to be completely separate, every node in each connected component must be distinct
  - Therefore, for this situation to occur, there must be (n/2 + 1) + (n/2 + 1) = n + 2 nodes in the graph
  - This contradicts the problem statement, as there can only be n nodes in the graph → proved by contradiction

4)

- Algorithm:
  - For each node:
    - Initialize a variable to hold the number of paths to that node as 0
  - Start from node *v* as the root with the number of paths to *v* set as 1
  - While current node is not *w* and there are still nodes to explore:
    - Increment each child node's number of paths by the number of paths to the current node and enqueue them to be explored later
    - Move onto the next queued node
  - Return the number of short paths
- Proof:
  - For this algorithm to break, the algorithm must have not found the shortest path to *w*, or it must not investigate every shortest path to *w*
    - First case:
      - There must be some node *x*, which has a path from *v* to *x*, such that the path is shorter than *x*'s level on the BFS tree
      - This implies that the short path is strictly less than the path found using BFS
      - Taking some node *y*, which precedes *x* in the BFS tree, we can then say that for this condition to be true, the BFS distance to *x* must be strictly greater than the BFS distance to *y* + 1
        - There are 3 possibilities during the exploration of node *y*
          - *x* is not discovered:
            - *x* will then be discovered as *y* is explored, leading its distance to be equal to the distance of *y* + 1 → contradiction
            - *x* was already explored, which, by the definition of the BFS tree, means that the distance to *x* is less than or equal to the distance to *y* → contradiction
            - *x* was discovered, but not explored, which means that it was discovered by some other node which is equidistant from the root to *y*, which means the distance to *x* is less than or equal to the distance to *y* + 1
          - BFS must find the shortest path
    - Second case:
      - Base Case:
        - The root node is pre-assigned a value of 1
        - Each of the root's children's values are incremented by the root's value
        - Since only the root has been investigated, there is only 1 path to each of its children
        - Base case successful

        - Induction:
            - Assume: the algorithm works for some amount of nodes *n*
            - Prove: the algorithm works for some amount of nodes *n* + 1
                - If the algorithm works for some nodes *n*, then all leaf nodes currently contain the correct number of paths to them
                - A new connection is formed
                - By addition principle, the number of paths to the destination node must be increased by the number of paths to the origin node
                - The algorithm works correctly for a new node insertion
- Time Complexity: O(m + n)
- Time Complexity Proof:
    - The first loop of the algorithm is guaranteed to only visit each node/edge 1 time, performing constant time operations on each node, resulting in an m + n runtime
    - The second loop of the algorithm is guaranteed to visit each edge only once
    - The combination of runtimes from these 2 loops can be reduced to O(m + n)

5)

- Algorithm:
    - Pick a number to remove from the sequence, regardless of answer of black box, you will be able to remove 1 number from your search
    - Check the initial sequence of numbers
    - If there is a subset that has a sum of *k*
        - While there is still a number *n* in the sequence that has not been checked
            - Pick a number that has not been checked and remove it from the sequence
            - If the box still outputs YES, that number is guaranteed to not be in the target subset
                - Remove the number from the sequence permanently
            - If the box outputs NO, then you know for a fact that number must be in the target subset
                - Place the value back into the sequence
                - Mark that value as “checked”
        - Return the final sequence
    - Otherwise
        - Return some failure condition
- Proof:
    - For this algorithm to work, it must be able to find the final subset of numbers that add up to *k*, if such a subset exists
    - There are 2 possibilities:
        - The subset exists
            - From here, every time we pick a number to temporarily remove, we get a deterministic answer whether or not it is in the subset
            - Therefore, since we analyze each number in this way, by the end of the algorithm, we will know exactly what numbers are in the subset and which are not
        - The subset doesn’t exist
            - This is caught by the outer if-statement, and will send the program to some failure condition
- Time Complexity: O(n)
- Time Complexity Proof:
    - Each of the *n* numbers in the set are run through the blackbox
    - If the blackbox outputs YES, the number is removed from consideration
    - If the blackbox outputs NO, the number is marked as checked, and the while loop only checks non-checked numbers
    - As a result, it’s impossible for more than *n* numbers to be run through the blackbox, since, regardless of output, each run through the blackbox will feature a new value that has a deterministic outcome

6a)

- Algorithm:
  - Set left bound to the beginning of the array
  - Set right bound to *n*
  - While the left bound is less than the right bound
    - Average the left and right indexes to calculate a middle index
    - If the middle index is equal to the value at the middle index minus 1
      - Move the left bound to the element after the middle index
    - Otherwise
      - Move the right bound to the middle index
  - Return the index of the left bound plus 1
- Proof:
  - In order for the algorithm to break, one of 3 cases must happen:
    - The missing number was passed over at the end:
      - For this to happen, all values checked would pass the conditional, as the missing value would be *n* + 1
      - This would mean that the left bound would continuously move right
      - In both even and odd cases, this results in the left bound overlapping with the right bound before the right bound ever shifts
      - Since the right boundary is initialized to *n*, the value *n* + 1 is returned, contradicting the case statement
    - The missing number was passed over at the beginning:
      - For this to happen, all values checked would fail the conditional, as the missing value would be 1
      - This would mean that the right bound would continuously move left
      - In both odd and even cases, this results in the right bound overlapping with the left before the left boundary ever shifts
      - Since the left boundary is initialized at 0, the value 1 is returned, contradicting the case statement
    - The missing number was passed over in the middle:
      - In order for this to occur, one of 2 things must've occurred:
        - The left boundary was shifted too far right:
          - In order for this to happen, the previous middle index must have passed the conditional
          - The only way for this to have passed is if the missing value was located on the right of the boundary
          - The case statement is contradicted
        - The right boundary was shifted too far left:
          - In order for this to have happened, the previous middle index must have failed the conditional

        - The only way for this to have happened is if the missing value was either on or to the left of the boundary
        - The case statement is contradicted
- Time Complexity: O(log n)
- Time Complexity Proof:
    - Each run through, the algorithm eliminates half of the remaining choices from consideration
    - As a result, the algorithm can address 2^n elements given *n* time
    - Therefore, the time complexity is O(log n)

6b)

- Algorithm:
    - Initialize an array *temp* of size *n* + 2 with all 0s
    - For each value in the given array:
        - Use the value as an index into *temp*
        - Set the indexed value in *temp* to 1
    - For each value in *temp* starting from index 1:
        - If the value is 0, return the index of the value
- Proof:
    - In order for this algorithm not to work, one of 3 cases must happen
    - The missing value must have been marked with a 1 in *temp*:
        - The only way for a value to have been marked with a 1 in *temp* is for the value to have been present in the given array
        - By definition this means the value is not missing, which contradicts the case statement
    - A non-missing value must have not been marked with a 1 in *temp*:
        - The only way for the algorithm to have missed a value while running through the given array is for that value to not be in the array
        - By that definition, the value that has been marked with a 1 in *temp* is a missing value
        - Either there are multiple missing values, which contradicts the problem statement, or this was actually the missing value, which contradicts the case statement
    - All values in *temp* were marked with a 1:
        - In this case, the assumption is that the missing value was not marked with a 1, as that would be caught by the first case
        - The only way for these conditions to be true is if the missing value was 0 or less or greater than *n* + 1
        - This is a contradiction of the problem statement, which says that the array only contains values between 1 and *n* + 1, inclusive.
- Time Complexity: O(n)
- Time Complexity Proof:

- The algorithm runs through each element of the provided list once and performs 2 array accesses for each one, resulting in an O(n) runtime
- It then runs through another *n* elements in a second array, performing another array access for a total O(2n) runtime
- O(2n) can simply be reduced to O(n)