Charles Zhang

Disc. 1D
Fri. 10am - 12pm
Song

1a)

- Algorithm: // Assuming start/end times are organized into pairs
  - Sort the pairs of times by start time, from earliest to latest
  - Initialize a list *res* to hold time of invocations of status_checks
  - While there are still pairs in the original set:
    - Take the end time of the first pair and store it in a variable *end*
    - Initialize a variable *cur* to the start time of the first pair
    - Delete the first pair
    - While the next pair's start time in the original set is less than *end* and the original set is not empty:
      - Set *cur* equal to the pair's start time
      - Delete the pair
    - Append *cur* to *res*
  - Return *res*
- Proof:
  - Assume there exists an algorithm that finds a smaller set of times to invoke status_check
  - The size of this set, *i*, must be strictly less than the size of my set, *j*
  - Assume both algorithms pick the same first *n* times to invoke status_check, but differ at the *n* + 1st time
  - By the definition of my algorithm, I will find all possible start times of other processes that overlap with the current process' end time
    - This translates to finding the maximum number of possible overlaps for the current interval
    - This will result in the maximum number of intervals being deleted from this addition to *j*
  - Since the algorithms differ at this time, the other algorithm must find a time different from mine
    - Regardless of what this time is, it will either delete as many intervals as mine, or less intervals than mine, since my algorithm finds the maximum number of overlaps
  - As a result, the next step of the algorithm will begin with me having less intervals or the same number of intervals as the other algorithm
  - If we continue using this logic for each step until the algorithm finishes execution, the other algorithm will finish execution on the same step as me or after
  - Since each step adds 1 time to both *i* and *j*, *i* must be greater than or equal to *j*
  - This is a contradiction of the original statement, as *i* >= *j*
  - Proof by contradiction complete
- Time Complexity: O(*n* log *n*)

- Time Complexity Proof:
    - The initial sort of the start/end times will take O(n log n) time
    - The while loop will run and access each pair exactly once, resulting in an O(n) runtime
    - The overall runtime is therefore O(n log n)

1b)

- The statement is **true**
- Proof:
    - Base Case:
        - As stated in the problem, if there are a total of k^* non-overlapping sensitive processes, there must be k^* calls to status_check
        - This is because there will be no point where a single status_check can handle multiple processes, since no overlap occurs
        - As a result, there is a set of k^* status_check calls that handles k^* non-overlapping sensitive processes
        - Base case passed
    - Inductive step:
        - Assume the statement is true for n sensitive processes
        - Prove the statement is true for n + 1 sensitive processes
        - There are 2 distinct cases for adding a sensitive process to the set:
            - You add a non-overlapping sensitive process
                - This would require an extra call to status_check to handle the new process, so there would be k^* + 1 calls
                - However, by definition, k^* would increase by 1 because an additional non-overlapping process creates a new maximum number of processes that aren't running at the same time
                - Since k^* + 1 == k^* + 1, the statement would still hold true in this case
            - You add a sensitive process that overlaps with an existing sensitive process
                - Since the new process overlaps with an existing one, the value of k^* would stay the same
                - This new process, x, would have to be handled by a status_check
                - However, the process that x overlaps with, y, could share the status_check that handled it with x since they overlap, as long as the status_check runs within the period of time they overlap in
                - Since the value of k^* didn't change, and no extra calls to status_check are needed, the statement still holds true
        - Inductive step passed
    - Proof by induction complete

2)

- Algorithm:
    - Initialize a heap structure *h* to track minimums
    - Initialize a set to hold all finalized vertices *f*
    - Initialize a set to hold all non-finalized vertices *s*
    - Initialize a list *L* to hold the final path
    - Place the starting point into *f*, paired with the value 0 and a pointer to itself, and place the starting point into a variable *cur*
    - For all other stops:
        - Insert the stop into *s*, paired with some value representing infinity and the null pointer
    - While *cur* is not equal to the destination node:
        - For each of *s*'s outgoing edges:
            - Follow the edge to the destination and use the $f_e(t)$ function to calculate a weight for the edge
            - Add the calculated weight to *cur*'s paired time value
            - If the newly calculated time value is less than the destination's current time value:
                - Assign the destination with the new time value and a pointer to *cur*
            - Update the heap structure to reflect new possible minimum
            - Delete the selected edge
        - Pick the minimum value stop from the heap and delete it from the heap
        - Move the selected stop from *s* into *f*
        - Set the selected stop as *cur*
    - While *cur* is not equal to the starting point:
        - Add *cur* to *L*
        - Set *cur* equal to the node paired with *cur*
    - Reverse *L*
    - Return *L*
- Proof:
    - Base Case:
        - There are no intermediate stops
        - The algorithm will calculate the $f_e(t)$ from the start to the end
        - The time value of the end will be updated with the correct time
        - *cur* will iterate to be equal to the destination, stopping the loop
        - The algorithm will trace back from the destination to the start and reverse the path, generating the path from the start node to the destination
        - Base case passed
    - Inductive Step:
        - Note the function $f_e(t)$ for any stop is monotonically increasing, so it is always best to reach a stop as early as possible
        - Assume the function has executed correctly for the first *i* stops
        - Prove the function executes correctly for the *i* + 1st stop

                - The current stop is guaranteed to be the earliest stop that has not already been analyzed
                - All of the current stop's neighbors will have their time values calculated
                - Problem statement guarantees that it is impossible to travel back in time
                - As a result, the next shortest path to an unexplored stop can be travelled, as the logic of the algorithm guarantees there is no shorter way to that stop
                - The current node will be marked as explored
                - The next node will be accessed from the heap and updated accordingly
            - Inductive step passed
        - Proof by induction complete
- Time Complexity:
    - O(e log e)
- Time Complexity Proof:
    - All of the initialization takes O(n) time to place all stops in their appropriate sets
        - The graph is connected, so this can be written as approximately O(e)
    - The while loop accesses each edge in the graph once
        - For each edge, a possible heap update occurs, which takes O(log e) time
    - The loop therefore takes O(e log e) time
    - Accessing the minimum of a heap takes O(1) time
    - Following the path back to the origin takes at worst O(e) time
    - Reversing the list takes at worst O(e) time
    - This all simplifies to O(e log e) time

3)

- Algorithm: // not divide and conquer, but still an O(n log n) solution
  - Sort the lines by slope, from smallest a_i to largest
  - If n is less than or equal to 2:
    - Return the set of lines
  - Initialize a list *res* to hold all visible lines
    - Insert L_1 and L_2 into *res*
  - Initialize a list *i* to hold intersections
    - Insert the x-coordinate of the intersection of L_1 and L_2 into *i*
  - For all n lines after L_2:
    - Calculate the intersection of L_1 and the current line
    - While the calculated intersection has a smaller x-component than the last element of *i*, and *i* is not empty:
      - Delete the last element of *i*
      - Delete the last element of *res*
    - Insert the current line into *res*
    - Insert the calculated intersection's x-component into *i*
  - Return *res*
- Proof:
  - The largest and smallest sloped lines in the solution must be visible
    - The smallest slope must be uppermost on the left side of any intersections
    - The largest slope must be uppermost on the right side of any intersections
  - Any other line that is visible must intersect the smallest sloped line before the largest sloped line does (on the x-axis)
    - If the line intersects after, by virtue of the line having a smaller slope than the largest sloped line, it will be hidden from view
  - Using these facts, we can begin a proof by induction
  - Base case:
    - Consider the case where there are 3 lines (2 lines and less is handled by an edge case handler)
    - Since the algorithm sorts the lines by slope, we know $L_1 < L_2 < L_3$
    - The algorithm will begin examining the 3rd line
    - There are 2 possibilities:
      - L_2 becomes hidden by L_3:
        - This means that L_2 intersects L_1 after L_3 does
        - Since L_2 is responsible for the last intersection in the list *i*, the while loop will catch this and delete L_2 and its intersection from their corresponding lists
        - L_3 will be inserted into *res*, which now contains L_1 and L_3
        - The algorithm returns with only visible lines
      - L_2 is still visible:
        - This means that L_2 intersects L_1 before L_3 does

                - Since L_2 is responsible for the last intersection in the list *i*, the while loop will pass over this
                - L_3 will be inserted into *res*, which now contains L_1, L_2, and L_3
                - The algorithm returns with only visible lines
            - Base case passed
        - Inductive Step:
            - Assume the algorithm only contains visible lines until this step
            - The next line, L_i, will be analyzed
                - Due to sorting, L_i is guaranteed to have the largest slope so far
            - There are 2 possibilities:
                - L_i causes lines currently in the solution to be hidden:
                    - This means that whatever lines removed must have intersected with L_1 after L_i does
                    - When the algorithm checks the last element of *i*, it will detect this and proceed with the deletion of the last element
                        - This will continue until the conditional from above is no longer true
                        - When this condition becomes false, that implies there are no more intersections after the L_1, L_i intersection
                        - This means there are no more lines in *res* that are hidden
                    - L_i will be added to *res*
                    - This is the correct behavior
                - L_i removes no lines from the solution:
                    - This means that all lines currently in the solution intersect with L_1 before L_i does
                    - When the algorithm checks the last element of *i*, it will decide to pass over the while loop
                    - L_i will be added to *res* without removing any lines from *res*
                    - This is the correct behavior
            - Inductive step passed
        - Proof by induction complete
- Time Complexity: O(*n* log *n*)
- Time Complexity Proof:
    - Sorting takes O(*n* log *n*) time
    - All initialization is O(1) time
    - The loop loops through each line once, which takes O(*n*) time
    - Within the loop, we perform a series of constant time operations
    - The while loop can be analyzed as taking O(*n*) time in the worst case scenario, but over the course of the entire algorithm, it can only run a total of *n* times, as each line can only be deleted once

        - This results in the while loop contributing a total of n operations to the algorithm's runtime in the worst case
    - In total, this means the for loop runs in $O(n)$ time
    - This means the algorithm runs in $O(n \log n)$ time

4)

- Algorithm:
  - If *T* has children:
    - Probe *T* and its children
    - If left child is smaller than others:
      - Return the result of recursively calling this function on the left subtree
    - Else if right child is smaller than others:
      - Return the result of recursively calling this function on the right subtree
    - Else:
      - Return *T*
  - Return *T*
- Proof:
  - There are 2 return conditions:
    - *T* is less than both of its children
    - *T* is a leaf node
  - Both return conditions are inherently accurate as long as *T* is less than its parent node
  - Assume that *T* is greater than its parent node
  - *T* could be one of 2 things:
    - The root node:
      - Since *T* has no parent node, it is smaller than its “parent” by vacuous truth
      - This directly contradicts our assumption
    - A child node:
      - In order for *T* to be at this step of the algorithm, it must have been passed into a recursive call in a previous step
      - However, in order to be passed into a recursive call, *T* must have been smaller than its parent node
      - This directly contradicts our assumption
  - Both situations lead to a contradiction
  - Proof by contradiction complete
- Time Complexity: O(log *n*)
- Time Complexity Proof:
  - Each call of the function utilizes 3 probes
  - Every time the function is called, it moves 1 level lower on *T*
  - As a result, in the worst case scenario, there are a total of 3*d* probes called
  - Since $n = 2^d - 1$ by definition, we can say that *d* is approximately log *n*
  - Therefore, there are approximately 3 log *n* probes called, which is O(log *n*) time

5)

- Algorithm:
    - Initialize a variable *left* to 0
    - Initialize a variable *right* to the size of the array
    - While *left* is less than *right*:
        - If the value at index *left* is less than the value at index *right*:
            - Return *left*
        - Calculate an index *mid* using floor([*left* + *right*] / 2)
        - If the value at index *mid* is greater or equal to the value at index *left*:
            - Set *left* to *mid* + 1
        - Else:
            - Set *right* to *mid*
    - Return *left*
- Proof:
    - The size of the shift can be modelled as the index of the minimum element by basic reasoning
    - Therefore, the goal of the algorithm is to find the index of the minimum element
    - Since the array is sorted, the minimum element is the only element where the preceding element is larger than itself
        - The only exception to this is if the minimum element is the first element and has no previous element, but this is implicitly handled in the first if-statement (explained below)
    - Base Case:
        - Take an array of 1 element
        - Regardless of the number of shifts, the 1 element will be in the same position, so there have really been 0 shifts
        - *left* will immediately equal *right* upon initialization
        - The loop will never execute
        - 0 will be returned
        - Base case passed
    - Inductive Step:
        - Assume the algorithm has successfully executed until the array is of size *n*
        - The algorithm will now do one of 3 things:
            - It will find that the leftmost index's value is less than the rightmost index's value
                - This can only happen if the leftmost index holds the minimum element
                - The algorithm will return the leftmost index
                - This is the correct behavior by the logic above
            - The algorithm will decide to move the left boundary to the right of the midpoint
                - This will only happen if the value at the midpoint is greater than the leftmost value

                - Since the middle value is greater, that means the subarray from the left to the middle is non-decreasing, which implies the minimum element cannot be contained within that subarray
                - By that logic, the minimum element must be in the right subarray
                - This is the correct behavior
            - The algorithm will move the right boundary to the midpoint
                - This will only happen if the value at the midpoint is less than or equal to the leftmost value
                - This implies that at some point in the array between the left index and the middle, a decrease happened
                - We know that the only way for a decrease to happen is when we switch from the maximum element to the minimum element
                - Therefore, the minimum element must be in the left subarray
                - This is the correct behavior
            - All 3 situations exhibit the correct behavior
        - Inductive step passed
    - Proof by induction complete
- Time Complexity: O(log n)
- Time Complexity Proof:
    - We can use the equation $T(n) = T(n / 2) + C$
        - Binary search divides the problem in half by only analyzing the left or right subarray in the next step, resulting in a $T(n / 2)$ term
        - Before trimming the search area, we must perform a series of constant time operations, resulting in the C term
    - Expanding this equation out, we see it is equal to $T(n) = T(n / 2^i) + Ci$
    - We know the search time for an array of size 1 is O(1), therefore we can use $n / 2^i = 1$
        - This results in $i = \log n$
    - Plugging this in, we get $T(n) = 1 + C \log n$
    - This reduces down to O(log n)

6)

- Algorithm:
    - If *d* is 1 or less:
        - Return the one array
    - Place the result of a recursive call with *d* / 2 into an array *left*
    - Place the result of a recursive call with *d* / 2 into an array *right*
    - Initialize an array *res*
        - Initialize a pointer *p* at the beginning of *left*
        - Initialize a pointer *q* at the beginning of *right*
        - While *p* and *q* are both not past the end of their respective arrays:
            - If the value *p* is pointing at is smaller than the value *q*'s pointing at:
                - Place the value pointed to by *p* into *res*
                - Move *p* to the next index of *left*
            - Else:
                - Place the value pointed to by *q* into *res*
                - Move *q* to the next index of *right*
        - If *p* is not at the end of *left*:
            - Append the remainder of *left* to *res*
        - If *q* is not at the end of *right*:
            - Append the remainder of *right* to *res*
    - Return *res*
- Proof:
    - Base Case:
        - *d* = 2 (any smaller value is handled by the function's base case)
        - *left* and *right* end up holding the arrays
        - They will be merged by the same pointer logic found in merge sort
            - Each sorted array will have a pointer traverse them, comparing the 2 values
            - The smaller of the 2 values is placed into the final array
            - This guarantees all values in the final array are less than or equal to the values remaining in the original arrays, and that the values in the final array are in a non-decreasing order
        - The if statements at the end catch the remaining elements in the longer array, if it exists
        - The final array is returned
        - Base case passed
    - Inductive Step:
        - Assume the algorithm has worked correctly up until this point
        - We now have 2 sorted arrays of length n_1 and n_2 (there may be more than 2 arrays remaining in our execution, we don't care)
        - These 2 arrays will be merged by the logic detailed above
        - The final array will be returned, therefore the merge works correctly for any 2 arbitrary arrays in the problem

        - When the final array is returned, it becomes one of the original 2 arrays found above due to the recursive call
        - Inductive step passed
    - Proof by induction complete
- Time Complexity: O(n log d)
- Time Complexity Proof:
    - This question is the same as merge sort
    - Therefore, the recurrence relation we must solve is $T(d) = 2T(d / 2) + Cn$
        - Each step takes 2 times the time to perform the step on a set of half the size (d / 2), plus the time of the merge step (Cn)
    - Expanding this, we find that it is equal to $T(d) = 2^iT(d / 2^i) + iCn$
    - We know sorting a d = 1 set takes O(1) time, so we can use $d / 2^i = 1$
        - This results in i = log d
    - Plugging this in, we get $T(d) = d * 1+ (\log d) * Cn$
    - This simplifies to O(n log d)