Faith Twardzik Orpaz Goldstein, 1F, 2-3:50 pm

Exercise 14, pg. 195

a) This is an interval scheduling problem, so we will implement a greedy algorithm as follows:
    - Sort an array of start-times *s* and their corresponding processes from lowest to highest
    - Sort an array of end-times *e* and their corresponding processes from lowest to highest
    - Initialize a queue
    - Initialize an array *checked* to keep track of all processes that have been checked already
    - While there are still processes that have not yet been added to *checked* (not yet been checked by *status_check*) (so while the size of *checked* is less than the number of processes)
        - Traverse the start times. Take the first start time in *s* and compare it to the first end time in *e*. If it is less than the first end time, add it to the queue.
        - Continue traversing through the start times until you reach a start time greater than the first/current end time
        - At this point, insert a status check right before the first/current end time
        - Unload the queue of start times and add their corresponding processes to the array *checked*
        - Move the current end time to the next end time in array *e*

    The way this algorithm works, is it runs a while loop while there are still processes that haven't been checked (added to *checked*) and for each start time, it compares that time to the first/current end time. If it is less, then it adds the start time to the queue and does the same for the next start time. When a start time is found that is greater, we know we need a status-check, so we insert one right before the current end time, and then we make the next end time the current end time, add all of the processes in our queue to our *checked* array, and restart the while loop, starting with the next start time in our array *s*, and going until *checked* includes all of the processes.

    Time Complexity:

    To sort the start times and the end times, we have a runtime of O(nlogn) for both sorts. Then, we traverse the start times a maximum of n (where n is the number of processes) times in order to status check all n processes, which results in an O(n) runtime, with only O(1) comparisons/actions done inside of the loop. So, the max runtime for the algorithm is O(nlogn).

    Proof of Correctness:

    It must be true that all processes have been checked by *status_check*, because the while loop runs until this is true. Now we prove that, as the problem states, the algorithm produces the smallest set of times as possible to invoke *status_check.* We do this by comparing the set of times produced by this algorithm (set *S*) with the set of times produced by an algorithm that we assume for contradiction's sake produces a smaller set (set *S1*) and using induction:

    - For our base case, note that the first status_check in *S1* can be before or at the first status_check of *S* but NOT after, because *S* schedules a status_check exactly before the first finishing time
    - Given the induction hypothesis, assume that there are the same number of status checks in *S* as in *S1* at a certain time *t.* We say that there are *m* status checks already in *S1* when we get to the finishing time corresponding to the *m*-th status check in *S.*

- Now we prove this holds true during the induction step, for the *m+1* status check:
    - Given by the algorithm, the to-be-checked process that is resulting in our *m+1* status check has not been checked by any of the status checks before now, because its corresponding start time would have been unloaded from the queue and added to *checked*
    - So, it must be true that *S1* must also include a status check for this process somewhere in the time interval between the *m* status check of *S* and the *m+1* status check of *S* in order to check the unchecked process
    - While this *m+1* status check for *S1* may be before that of *S*, we know by the inductive assumption that it must occur before or at the *m+1* status check of *S*, and so we see that the status checks in *S1* CANNOT be less than those in *S*. Our algorithm produces a minimum set of status checks.

b)

- We have our set *S* of status checks, and we prove that the statement provided is TRUE. There must be a set of k* times at which you can run *status_check* so that some invocation occurs during the execution of each sensitive process. Basically, we are just looking to prove that the size of *S* (the number of status checks provided by our algorithm) equals k*
    - We are already given that k* is less than or equal to the size of *S* (the problem states that *status_check* needs to be invoked at least k times
    - Now we prove that the size of *S* is also less than or equal to k*. We do this by finding the biggest set of non-overlapping, disjoint sensitive processes. But, note that we did find this set through the implementation of our algorithm, because we inserted a status check only in the case that the end time (and process) that we were looking at was not checked by any of the other status checks, nor the ones moving forward, and so it is disjoint. This gives us the set of disjoint processes, so the size of *S* disjoint processes equals k*, the disjoint processes.

Exercise 18, pg. 197

Since we are trying to find the shortest travel route from a starting point to a destination, this will just be an application of the shortest path problem, in which we will use a modified version of Dijkstra's algorithm, with the catch that we are not *just* looking for the shortest time from the starting point to the destination, but also the shortest *route.* To do this, we will have to store an additional value when we choose a node to add to our explored nodes – not just the distance to that node, but also the node right before it in the shortest path to it. Then, when we arrive at the destination node, we can traverse backwards using these previous nodes, to get the route back to the starting node. The following is the algorithm, which is the same as Dijkstra's algorithm on pg. 138 of the textbook, except with the modification outlined above:

- Let *S* be the set of explored nodes
    - For each u in *S*, we store the shortest time to get to that node *t(u)* as well as the previous node on the shortest-time path to u, *p(u)*
- Initially S = {s} and *d(s)* = 0

- While *S* does not equal *V* (the set of explored nodes does not equal all of the nodes in the graph)
    - Choose a node *v* that is NOT in *S* and that has at least one edge from *S* for which the time to u, *t(u)* plus the time from *u* to *v* is a minimum
    - Add node *v* to the explored nodes *S* and store the new *t(v)* since it will be smaller than the old one – also store *u* as *v*'s previous node, *p(v) = u*
- EndWhile

Time Complexity: Since we will be using a heap-based priority queue implementation, and since the ChangeKey operations, which must be considered for every edge out of node *u* when node *u* is added to the set of explored nodes, is an O(logn) operation, and since there are *E* edges in the directed graph, we have a total runtime of O(*E*logn).

Proof of Correctness:

This is basically just a reiteration of the proof in the book on pages 139-140, since it will be the same regardless of our slight modification to Dijkstra's algorithm:

- We are proving, using induction, that for each *u* in *S*, the path *P(u)* generated by the algorithm is the shortest-time starting-node to *u* path
- Our base case, in which the size of *S* is one, holds because *S = {s}* and the time to *s* from *s* is zero *t(u) = 0*
- We assume the induction hypothesis, in which the size of *S* equals *k* also holds
- The inductive step begins with *S*, which now has a size of *k+1* nodes, where we added node *v*, and (*u, v)* is the last edge on the path from *s* to *v*, *P(v)*
- We know from the induction hypothesis that *P(u)* is the shortest-time *s*-*u* path for every node *u* in *S*
- Now we consider any other *s*-*v* path and show that it is at least as long as our path *P(v)*, proving that our algorithm does in fact generate *P(v)* as the shortest-time possible path
- Let *y* be the first node on *P* (the other *s*-*v* path) that isn't in *S*, and let *x in S* be the previous node to *y*
- We know that *P* is already at least as long as *P(v)* when it leaves *S*, and we see that our algorithm must have considered adding node *y* in iteration *k+1* via *(x, y)* but decided NOT to do this, instead adding *v* – this means that there is no path from *s* to *y* through *x* which is shorter than our generated path *P(v)*
- Since we have a sub-path *P*, then we know this must be at least as long as *P(v)*, and according to our induction hypothesis, this proves that the full path *P* is also at least as long as *P(v)*. Since this is true, and the shortest times on the path *P(v)* correspond to nodes/locations along a route, we know that the route *P(v)* must also be a minimum

Exercise 5, pg. 248

We are attempting to return all of the visible lines, and to be visible means that at one point, there is an x-coordinate for which the line is uppermost (where uppermost is defined as having a y-coordinate which is greater than the y-coordinates of all of the other lines at this point), so we are looking for all of the lines that are uppermost at some point (x-coordinate) in the graph. We use a divide-and-conquer method to do this.

The base case for this scenario is the case where we have less than or equal to three lines. We know that the first and last of these lines will be visible, and if there is an intersection point between first and second earlier on the x-axis than the intersection point between the first and third line, then the second line will be visible at some point as well.

- Sort the lines from smallest to greatest slope
- Divide the lines into two groups, partitioned by the number *middle,* which equals *n/2.* We will recursively find the visible lines in each of these subgroups. Also find all of the intersection points of the lines in each of these groups, but ONLY the intersection points between sequential lines (intersection point *i* is between line *L(i)* and line *L(i+1)*).
- It is true that for this list of intersection points, the x-coordinates will be increasing, because if we have two visible lines that intersect, then the one that is first in our list (has a smaller slope) will be visible (uppermost) to the left of the one that is next in our list (has a larger slope)
- Since we have recursively discovered both the visible lines and the sequential intersection points of the first half of the lines (the first group), we do the same for the second half (the second group in our division).
- Now, we must "merge" the two groups. We have the visible lines and intersection points isolated to those two groups, but we must merge them to find the visible lines in the complete set.
- Based on the logic of our base case, the first line *L1* will be visible, and so will the last line *Ln.*
- First, using the merge operation outlined in mergesort, which has a runtime of O(n) and uses a point-by-point comparison, we merge the two sorted lists of intersection points from the first and second groups into one sorted list of intersection points, sorted by order of smallest to highest x-coordinate
- For every x-value, *k,* we find the line in our first group (*G1)* that is uppermost at this x-value, and the line in our second group (*G2*) that is uppermost at this x-value. Here we will look at an intersection point – the absolute smallest value where the *G2* line that is uppermost is above the *G1* line that is uppermost. Right after this, the two lines intersect at a point *i.* So, the first line from *G1 (L(G1))* is uppermost in the region to the left of *i* and the second line from *G2 (L(G2))* is uppermost in the region to the right of *i.*
- This means that our final set of visible lines will correspond to *L1,* up to *L(G1),* then *(L(G2),* up to *Ln,* and the intersection points follow the same pattern (the one with the smallest x-coordinate from *G1,* up to that in *G1* right before *i,* then *i,* then that in *G2* right after *i,* up to the one with the largest x-coordinate, from *G2).*
- Now, we can recursively do this until we have a full list of visible lines based on all of the lines in the graph

Time Complexity:

This algorithm has a time complexity of O(nlogn). O(logn) stems from the splitting of the list of lines into successive halves as in a binary search, plus the O(n) time for merging the sorted lists of intersection points into one list of intersection points during each recursive call.

Proof of Correctness:

This problem does not ask for a proof of correctness.

Exercise 6, pg. 248

We are attempting to find a local minimum in the binary tree *T*, where a local minimum node has a label which is less than the labels of all nodes joined to this node by an edge. We can do this using the following algorithm:

- Start at the root node of the binary tree, *s*, and if the value of *s* is less than the values of each of its two children, then the root node is a local minimum, and we return *s* and terminate
- Else, arbitrarily choose one of the two smaller children and
    - Probe the two children of this node. If its value is less than the value of each of its two children (and since we know its value is less than the value of its parent), we have found a local minimum, and we return this node.
        - Else, we do not have a local minimum yet. Choose one of the node's children which has a value smaller than its value, and do this again
        - If we reach the end of the tree and have yet to return a local minimum, then the leaf node we end at will be a local minimum (it has no child values to check, and we know its value must be less than its parent's value), so we return the leaf node

Time Complexity: In this way, we will be choosing one child branch to explore at each node, and will at a maximum, traverse down one route from the starting node to a leaf node. Since the number of nodes that can be reached divides in half after each movement to a new child, and since we know a binary tree traversal is O(logn), the time complexity of this algorithm is also O(logn).

Proof of Correctness: There are several different scenarios for a potential local minimum returned, and we prove that each scenario results in a local minimum.

a) The local minimum returned is the root node. We know this must be a local minimum because we compared it to its two children, and only return it if it has a lower value
b) We have returned a non-leaf local minimum. This must be a minimum because it is less than its parent (it was only selected previously because it was a child of its parent with a *smaller* value), and we compared its value to its two children to ensure that it had a smaller value than both
c) We return a leaf node. There are no children to compare against, and its value must be less than its parents', because it was selected in the last iteration precisely *because* it had a smaller value than its parent

Question 5

We can first note that finding the index of the smallest element in our shifted array gives us *k*, the number of times the array has been shifted – this is because the smallest element was naturally the first element in our unshifted, sorted array, so its index in the shifted array will be equal to the number of times the array was circularly shifted. Therefore, all we have to do is find the smallest element in the shifted array. We could do this in linear time by traversing the array and keeping track of the minimum value encountered as well as its index, but this is inefficient. Instead, we can implement a binary search algorithm that is much more efficient, as follows:

- Select the first element in the array. If the element to the left of it is greater, then we have found the smallest element, and return 0 for the value of *k* (the array has not been shifted)
- Else, select the (n/2)-th element (the middle element of the array) and compare it with the first element. If it is greater than the first element, then the smallest element must be in the right half

of the array. If it is less than the first element, then the smallest element must be in the left half of the array

- Recursively do the same comparison on either the right or left half of the array, depending on if the middle element was greater than or less than the first element, respectively. When we reach a sub-array size of 1, we know that we have the smallest element
- Return the smallest element's index, which is equal to *k*

The intuition for this algorithm is to find the smallest element by using a divide-and-conquer approach, splitting up the array into half during each iteration. This works because when we compare the middle element to the first element and it is less, then we know that the array must have wrapped around so that a larger element is at the beginning, and the smallest element is in-between it and the middle element. By the same logic, if the middle element is greater than the first element, then the smallest element must be on the right side. It is this intuition that allows us to split the array into two after every comparison.

Time Complexity: Because this is a binary search (with trivial O(1)) comparisons in-between, and since the runtime of a binary search is O(logn), the runtime of this algorithm is O(logn).

Proof of Correctness:

We do this using induction, to prove that the smallest element is within the smaller and smaller sub-arrays (knowing that eventually we will end with a sub-array of size 1).

- The base case is the first iteration on the entire array. We know that the smallest element must be in the entire array, so the base case holds
- We assume the induction hypothesis, that on the *k-th* iteration, the smallest element is in the sub-array that is selected (either left or right)
- We prove that after the *k+1-th* iteration, the smallest element is still in the sub-array chosen, and we do this by splitting it up into two cases:
    - The left sub-array is chosen. In this case, the middle element is less than the first element, which means that the middle element has NOT wrapped around yet, and so the smallest element (which will be to the left of this larger element), must be in the left sub-array
    - The right sub-array is chosen. In this case, the middle element is greater than the first element, which means that the middle element HAS wrapped around, so the smallest element will be to the right of this larger element, and must be in the right sub-array
    - For both iterations we see that the smallest element is isolated in the next smaller sub-array that is iterated over, and since the index of the smallest element equals *k*, the algorithm correctly produces the value of *k*

Exercise 6

We will use a divide-and-conquer method. The algorithm is as follows:

- Pair up each of the *d* sorted arrays of integers so that there are *d/2* pairs. Do this by assigning the first two sorted arrays to a pair, the next two sorted arrays to a pair, and so on until no sorted arrays are left
    - Merge together the sorted arrays in each pair. To do this, do a point-by-point array comparison of the first element in the first array in the pair and the first element in the second array in the pair and insert the smaller of the two into the growing merged array.

Then compare the element not selected and the next element in the array whose element was selected. Do this until there are no elements left in either of the sorted arrays.

- After the first pairing & merging procedure, we will now have a total of *d/2* sorted arrays. Recursively do the previous steps again: pair up all of the sorted arrays (so there are now *d/4* pairs), merge together the pairs to get *d/4* sorted arrays.
- Repeat this until we end with *d/d* sorted arrays (we will end with one sorted list)

Time Complexity: The time complexity is O(nlogd). The merging of the elements in each pair has a linear runtime of O(a + b), where a is the length of the first array in the pair and b is the length of the second array in the pair. The splitting of all of the sorted arrays into successively half the number of arrays (*d/2*, then *d/4*, then *d/8* merged lists) is an O(logd) operation, so overall the algorithm has an O(nlogd) runtime, where n is the total number of ALL elements in all of the sorted arrays.

Proof of Correctness:

We prove that this algorithm produces a fully sorted array by assuming that it does not produce a fully sorted array (there is at least one element *v* which is not in the correct position, with its left element less than it and its right element greater than it) and proving by contradiction. We use induction the following way:

- For the base case, we have *d* arrays, which we are given are already sorted
- Assume, according to the inductive hypothesis, that after the *k-th* iteration, there are $d/2^k$ lists, all of which are sorted
- We prove that in the *k+1-th* iteration, there are $d/2^{k+1}$ lists, all of which are sorted
- During this iteration, we split up the arrays into pairs, and then merge them. Since the textbook proves that the merge portion of mergesort is correct (when we make a point-by-point comparison between elements and always select the smaller one, it is impossible to have a larger element before a smaller one in each of these sub-arrays), and since the arrays from the *k-th* iteration are known to be sorted, then the arrays in the *k+1-th* iteration must be as well