COMP2123 - Assignment 1

UID: 500557727

Problem 1.

This algorithm first runs a for loop to check every second element in the array for evenness. This first loop runs n/2 times or (n-1)/2 times depending on whether there are an even or odd number of elements in array A. It checks evenness by using the mod operation, which we've been told has O(1) time, and the possible assignment of b to false would also take O(1) time. Therefore, since the loop runs n/2 times or n-1/2 times and each loop takes O(1) time, this loop takes O(n) time.

Similarly, the second for loop checks every odd element for oddness. It similarly uses the mod operation (O(1)) and can assign b to false O(1), hence each loop is also in O(1) time. Added up, the loop takes O(n) time.

The algorithm hence uses O(n) time as seen in the addition of the times of these for loops: O(n) + O(n) = 2O(n) = O(n).

Problem 2.

The way I've approached this problem is to use the push and pop operations to manage another stack ("minStack") which holds the minimums of the original stack ("createStack") as we add and remove elements off it. The getMinimum operation simply has to retrieve the (t)th element of the "minStack" to retrieve the minimum of "createStack".

a) Size()- No change from the slides. Return one more than the index of the top element.

isEmpty()- return the Boolean result to: createStack.size() == 0 .

Push(e)- if you push an element, you should add one to 't' and assign the element to the 't'th element of the array as shown in the class slides. Next, to account for the getMinimum operation, you should push this element onto minStack as well ONLY if it's less than or the same as the top element of minStack.

Pop()- if you use pop on an array, you should minus one from 't' and return the element that you just popped, as shown in the slides. For the getMinimum operation, you should remove that element from minStack only if it's equal to the top element of the stack.

getMinimum()- now that Push and Pop have been working on minStack for us, we can simply retrieve the minimum element of createStack as it is now the top element of minStack. Of course, if isEmpty() returns true, then getMinimum should return null.

b) I will be arguing for and analysing my data structure/operations in the order that I've presented them above:

Size()- This will always return N since t is saved as the index of the top-most element. Since arrays start counting indices at 0, adding 1 to t returns the total size of the array.

isEmpty()- this operation will only return true if the size of the array is 0, meaning that it has identified the empty array. If the array is any other size, it will return false.

Push(e)- This operation will assign e to the 't'th element of the array as required. It also supports getMinimum() by pushing e onto minStack if it's lower than the element at the top of minStack. Importantly, it also pushes e if it's the same as the top of minStack, so in the case that one of these 'e's is popped, minStack still saves the other instance of e in createStack.

Pop()- pop functions by pushing back the top element of createStack, making the original 't'th element inaccessible. It also supports getMinimum by looking into minStack and removing the top element if it's the same for both stacks.

getMinimum()- getMinimum functions by returning the top of minStack, which should be the minimum of createStack. We can be sure that it's the minimum of createStack since elements are only pushed if they're lower than every other element in the stack. This is implied since elements are only pushed onto minStack if they're lower than the element right before it, meaning that, by induction, if we had one element in the stack and the next element is lower than it, it would be pushed to the top. In this base case, when a third element lower than the second appears, it will be lower than the first element as well since the second element is lower than the first. This relationship will continue and appear as: first > second> third > fourth> fifth> …>k. Then, when the ‘k+1’th element is pushed onto the stack, it will be lower than k, which is lower than all the previous elements.

The pop operation accounts for the situation where an element from minStack is removed from createStack. Pop will only remove any elements from minStack if createStack.pop() returns the top element of minStack. This is because the top element of minStack is both the minimum of createStack and the most recently added minimum of createStack. Since pop() removes the top element of the stack, that number will either be the current minimum or higher than it– it will never be a number higher than createStack.pop() since it would never have been added if it was. That means that element will never refer to an element lower than the top element of minStack, the minimum number pushed and kept in the stack. Having this top element not be the minimum would contradict the fact proven above - that all elements in minStack are in ascending order, hence the getMinimum() operations functions as required.

c) Each of these operations has the same space complexity: *O(N)*. The array will always use a space of size O(N) no matter which operation is used. Of course, if push() is called on an array at full capacity, I could choose whether to move the elements to an array of double the size (2N). However, even if this is the desired error handling, each operation will continue to use the size N (originally 2N).

In terms of time complexity, as is shown in the slides, the size() and isEmpty() operations are in O(1) time since they simply make one comparison which are in O(1) time (adding one to a variable and returning a Boolean respectively). getMinimum() is similar since it simply retrieves the ‘t’th element of an array without looping through it, hence it is also in O(1) time.

The pop() and push(e) operations are similar in their analysis. They have several steps which take O(1) time, but there are few of these steps, they are never looped, and they have no correlation with the length of the array/s. In pop(), we decrement ‘t’ once, then return the ‘t+1’th element. Each of these takes O(1) time. To account for getMinimum, we also remove that element IF it’s also the top element of the minStack. So pop() takes O(1) + O(1) + (sometimes) O(1) = O(1). Similarly, push increments t once and assigns the ‘t’th element to e. Then it checks if e is less than the current minimum and pushes it onto minStack if it is. So the original push operation takes 2O(1), and the Boolean we’ve added takes O(1) time, which can lead us to call push(e) again on minStack. The push operation hence takes 6O(1) = O(1).

Problem 3.

a) My algorithm operates on two key observations:

1. Any lighthouse whose paper aeroplane will never hit another lighthouse has a height that is the maximum of all the lighthouses before it. This is when iterating from index 0 to index n-1 in the given array.
2. For all other lighthouses that will eventually hit another lighthouse, we can check it against its relevant lighthouses (the ones to the left of it) if we use reverse-iteration. As I iterate through these lighthouses from right to left, I can check each height against the leftmost element that hasn't been assigned an index in the answer array ('index_hit'). The first height that is higher than that one will be the height that all previous lighthouses will eventually hit. Therefore, I can assign this lighthouse's index to the previous lighthouse's places in the index_hit.

My algorithm first initialises an array of length n-1. This will be the index_hit. I also need an initial maximum (curr_max) that is negative infinity. Next, I would use a for loop to act on my first observation and iterate through array A, using an if statement to check if the current element of A is higher than curr_max. If it is, then that lighthouse's index in index_hit would be assigned the value of -1.

For the rest of the lighthouses that *will* eventually hit another lighthouse, I would use another for loop to iterate through A in reverse. I also need variables curr_elem and curr_ind, initialised as A[0] and 0 respectively, which keep track of which element/s I'm trying to find the hit index for. Inside the for loop, I would first check this element against curr_elem. If this element is not higher than curr_elem, then I should reassign it to curr_elem. If this element is higher than curr_elem, then I should reassign and iterate through all the indices between this element's index - 1 and curr_ind to assign all previous element's positions in index_hit the index of the lighthouse they all would've just hit. I would also have an if statement to check to make sure that the element of index_hit I'm about to reassign is not already -1 (and hence must never hit anything). Lastly, I reassign curr_elem and curr_ind to the next element whose index isn't -1 so I can start comparing maximums of lighthouse sections again. This is important because my algorithm works by checking for the maximum of small sections of the lighthouses and assigning all previous (not -1) elements to its index in index_hit.

b) I will first prove that my observations are correct. For the first observation that lighthouses whose airplanes never hit anything must be the cumulative maximum, this fact follow as lighthouse lighthouse 'i' can only send an airplane to lighthouse 'k' if i < k AND A[k] < A[i]. Furthermore, there cannot be another lighthouse 'j' between 'i' and 'k' for which A[k] < A[j]. Hence, it follows that, for i, j, m…k, A[k] > … > A[m] > A[j] > A[i] i.e. A[k] is the maximum in the set i to k. I can prove that this is true in my first for loop by using an invariant: that curr_max will always be the maximum of the set i to k where i < k. This is clear as curr_max starts from negative infinity, a number which will be immediately reassigned to the element at A[0] (since that lighthouse can never reach any others) and then updated again only if A[k] is greater than the curr_max, which is the greatest number of all numbers before k. Hence, I correctly assign -1 to all lighthouses that will never reach another lighthouse (the set never_reach).

Now I need to prove my second observation, which actually follows from the first. Since I've assigned -1 to all elements in never_reach, I can assume that all other lighthouses' airplanes can reach one of the lighthouses to their left. Formally, for each A[m] not belonging to never_reach, there exists A[l], l < m, where A[l] > A[m]. So, if I start from some element in the array and compare each lower element, I must eventually reach an element that has a higher value than it. If m > t>…> l (adjacent lighthouses) and A[m] > A[t],.. and A[m] < A[l] (all lighthouses left of m have lower heights than m except l), then all lighthouses between m and l will hit l. To prove that my second for loop achieves this, I can prove by contradiction. If I assume that A[l] is not actually greater than A[m], or any of the lighthouses between them, then that would contradict the initial fact A[m] > A[t], or A[t] > A[h] (where h < t). All lighthouses between m and l (not including the ones that never hit anything, as we've already accounted for those) must have heights less than A[l], and will therefore hit it. Hence my algorithm correctly assigns the index of the lighthouses that airplanes will hit.

c) I first initialise index_hit and curr_max, which each take O(1) time. Next, I iterate through all of the elements in A and perform an O(1) if statement which checks if the current element is higher than the curr_max. If it is, then I perform two more O(1) time operations: to reassign curr_max to the current value and to assign the corresponding element in index_hit to -1. In the worst case, I would perform 3 O(1) operations in a for loop which runs n times, hence so far we have O(1) + O(1) + O(3n) = O(n).

The next steps are to initialise curr_elem, and curr_ind, both equalling O(1) and adding up to O(1). The next for loop is a bit more complicated: the Boolean checking whether the element we're on is bigger than curr_elem is in O(1), and the else statement reassigning curr_elem if it isn't is also in O(1). Also in the if statement is the for loop assigning this element's index to all the positions of the elements between curr_ind and this elment's index – 1. In the worst case, the inner for loop can only run a maximum of n times (O(n)). Then there's an if statement checking that the next element isn't -1 (O(1)) and reassigning curr_elem and curr_ind (2 O(1)). So, with the previous calculations, these all add to O(n) + O(1) + O(1) + O(n) + O(1) + 2 O(1) = O(n).