CIS 121

Homework 03

Fei Liu

Recitation 201

Collaborators:

Q1.

Q2.

(a) In my explanation, when I say name A is less than name B, then that means that the first letter of name A comes before the first letter of name B in the alphabet, and vice versa if name B is less than name A. If the first letters are the same, then the second letter of the names will be checked until there is a difference. With that being said, here's how one iteration of the algorithm will work.

My algorithm will have two inputs: the name of the TA we're trying to find, denoted by x, and a string array containing the names of all n TAs, with the assumption that the names are in alphabetical order but the first TA is randomized. Let T_1 denote the name of the TA in the first position and $T_{n/2}$ denote the name of the TA in the middle position. The first thing the algorithm will do is check if $T_{n/2}$ is equal to x or if T_1 is equal to x (this is also the base case of my algorithm). As a code snippet, it looks like "if (A[n/2].equals(x) —— A[0].equals(x))", and if this is true, then the algorithm will return the respective position.

If not, then my algorithm will check a series of conditionals. After I explain what my algorithm will do if that conditional is true, I will provide the POC right afterwards.

1. If x is less than $T_{n/2}$ and greater than T_1, the algorithm will do a recursive call with first half of the array as the new array input. POC: In this case, the name has to be between the first position and the middle position, so the name is in the first half of the array.

2. If x is less than $T_{n/2}$ and less than T_1:

- 2.1: If T_1 is less than $T_{n/2}$, then the algorithm will do a recursive call on the second half of the array.
- 2.2: If T_1 is greater than $T_{n/2}$, then the algorithm will do a recursive call on the first half of the array.

POC: If x is less than $T_{n/2}$ and less than T_1, then that means that x is smaller than at least half of the names. We need one more piece of information before we can deduce which half of the array to do a recursive call on: which half of the array the smallest name is contained in. To explain this in greater detail, if the smallest name were in the first half of the array, then if x was in the second half of the array, then x is greater than $T_{n/2}$ and the condition of x being less than $T_{n/2}$ cannot be true, and if smallest name were in the second half of the array, and x was in the first half of the array, then x is greater than T_1 and the condition of x being less than T_1 cannot be true. So, x must be in the same half of the array as the smallest name. We can check which half of the array contains the smallest name by comparing T_1 and $T_{n/2}$. If T_1 is smaller than $T_{n/2}$, that means that the smallest name is in the second half of the array. The edge case here is when T_1 is actually the smallest name itself, and this case is not possible because the the conditional x is less than T_1 cannot be true, and this case is never entered. If T_1 is greater than $T_{n/2}$, that means that the smallest name is in the first half of the array.

3. If x is greater than $T_{n/2}$ and less than T_1, then the algorithm will do a recursive call on the second half of the array.

POC: x being less than T_1 tells me that x has to be before the first position, so we go to the last position and work our way back, but x being greater than $T_{n/2}$ tells me that x is to the right of the middle position, so x is in the second half of the array.

4. If x is greater than $T_{n/2}$ and greater than T_1:

- 4.1: If T_1 is greater than $T_{n/2}$, then the algorithm will do a recursive call on the first half of the array.
- 4.2: If T_1 is less than $T_{n/2}$, then the algorithm will do a recursive call on the second half of the array.

POC: If x is greater than $T_{n/2}$ and greater than T_1, then that means that x is greater than at least half of the names. We need one more piece of information before we can deduce which half of the array to do a recursive call on: which half of the array the smallest name is contained in. To explain this in greater detail, if the smallest name were in the first half of the array, then if x was in the second half of the array, then x is less than T_1 and the condition of x being greater than T_1 cannot be true, and if smallest name were in the second half of the array, and x was in the first half, then x is smaller than $T_{n/2}$ and the condition of x being greater than $T_{n/2}$ cannot be true. So, x must be in the same half of the array as the smallest name. We can check which half of the array contains the smallest name by comparing T_1 and $T_{n/2}$. If T_1 is smaller than $T_{n/2}$, that means that the smallest name is in the second half of the array. The edge case here is when T_1 is actually the smallest name itself, and in this case, x is still in the second half of the array, otherwise the condition of x being greater than both T_1 and $T_{n/2}$ is not met, and this case is not entered. If T_1 is greater than $T_{n/2}$, that means that the smallest name is in the first half of the array.

POC continuation: My algorithm will keep on executing recursive calls on the half containing x, until that x falls in the middle or first position of the input array.

Now I will be analyzing the run time of my algorithm. Every comparison made in my algorithm takes constant time, c. The code blocks inside the conditionals are simply recursive calls or a return statement, so if those take anytime, it's also constant time at worst. Now, I have to find out how many iterations my algorithm will perform in the worst-case. In the worst-case my algorithm will have to keep splitting the array into halves until the input is an array with one name and that name is x, and this is essentially what a binary search does, and in the worst-case, with an input array of size n, a binary search takes $lg(n)$ iterations to arrive at the answer. My algorithm also takes $lg(n)$ iterations to arrive at the answer in the worst-case, and if each iteration takes constant time, then my algorithm's run-time is $c * lg(n)$, which is big-theta($lg(n)$).

(b) My algorithm in (a) still meets the target run-time in the worst case. If in the case where there are no duplicates, my algorithm's worst case scenario is doing recursive calls until the input array is of size one. If in the case where there is a duplicate, then my algorithm's worse case scenario is doing recursive calls until the input array is of size two (both positions will be equal to x). If we started with arrays of the same size in both cases, the case where there can exist duplicate names in the array takes one iteration less than the case where there are no duplicate names. The worst case run-time of that case with duplicate names is $c * lg(n) - c$, which is still big-theta($lg(n)$).

Q3.

(a) My algorithm will have a variable called current that helps me keep track of which node I am currently on within the binary tree. "Current" starts by pointing to the root node of the binary tree. My algorithm will also initialize an empty queue and an empty stack, and then start by adding "current" (which is at the root node) to both the queue and the stack.

At this point, my algorithm enters a while loop, with the condition of looping while the queue is not empty. Here's what one iteration of the loop does:

1. If there exists a left sub tree, make current equal to the root node of that left sub tree ("current = current.left" or something similar). Else, if the stack is empty, then break out of the while loop. Else, pop a node from the stack, set current equal to that node. Check if "current" has a right sub tree. If it does, make current equal to the root node of the right sub tree, and if not, then break out of the while loop.
2. Dequeue the queue, and print the value of the node it returns. (Now it's empty).

3. If current has a right sub tree, push "current" onto the stack".
4. Enqueue "current" into the queue.

My algorithm is finished here. Now, I will provide a POC. My algorithm simply dequeues (and print) and enqueues whichever node it is on when in the while loop. So I have to make sure that 1) it traverses in the correct order and 2) the while loop breaks when there are no more nodes to be traversed.
1. My while loop checks for a left sub tree to traverse to, and if not then it will traverse to a right sub tree. Since my loop is only pushing a node onto the stack if that node has a right sub tree, I know at some future iteration, when the left sub tree gets exhausted of nodes, the first step in my loop will pop that node from the stack and then traverse the right sub tree. The nodes in the stack do not get re-visited, that is by step 2, "current" never points to it again when it's popped. Notice that higher level root nodes are also deeper into the stack, so as the nodes exhaust both their left and right sub trees, higher level nodes get popped (until the original root node is popped and the loop starts traversing the right half of the tree). This proves that my loop traverses the tree in the desired order, and since every node visited is enqeued, step 2 dequeues and prints, thus printing the node values in the desired order.
2. The stack in my algorithm keeps track of which nodes still have a right sub tree that my loop has not traversed yet. Once this stack becomes empty, the only possible paths are to traverse to the left sub tree, and if that doesn't exist, then this tells me that the entire binary tree is exhausted of both left and right sub trees to traverse through and then breaks out of the while loop.

Now I will analyze the space-complexity of my algorithm. Notice that the queue always contains one element at most, since in each iteration of the while loop, a node gets dequeued and then a node gets enqueued. Also, the stack will never have as many nodes as the total number of nodes in the binary tree because nodes without a right sub tree (this includes leaves) do not get pushed into the stack. Thus, in the worst case-scenario, if n is the size of the input binary tree, then my algorithm will take up at most $n - 1 + 1 = n$ memory space, which is O(n).
Now I will analyze the run time of my algorithm. The steps before the while loop all take constant time. Every if, else if, else statement and function in the while loop take constant time. Thus, to show that the algorithm runs in big-theta(n) time, I have to prove that the while loop only traverses each node once and once only. Let's say I am given a root node x. Step 3 only pushes x into the stack if it has a right sub-tree. This ensures that later in a future iteration, step 1 will eventually pop x and "current" becomes the root node of the right sub tree of x. x is never pushed into the stack again, because when every node in the right sub tree has been traversed, the next parent node of x with a right sub tree (if it exists) is the one getting popped from the stack, and "current" points to its right sub tree's root node. This keeps going until x has no more parent nodes with a right sub tree, and after traversing through x, "current" is never equal to x again. This along with the proof that the loop traverses through every node in the tree tells me that every node gets visited exactly once. Thus, the while loop loops n times, and the run time of my algorithm is $c_1 * n + c_2$, which is big-theta(n).

(b) My algorithm will have a variable called current that will keep track of which node in the tree I am currently on. I start by setting current equal to the root node of the input binary tree. I initialize an empty stack, and push "current" into it. Now, I enter a while loop that simply has "true" as a conditional:
1. If there exists a left sub tree, traverse to it and set "current" equal to the root node.
2. Else, if the stack is empty, exit the while loop. Else, pop a node from the stack, print its value, and set "current" equal to it. If "current" contains a right sub tree, then traverse to the right sub tree and set current equal to the root node, else, break out of the while loop.
3. If "current" contains a right sub tree, push "current" into the stack, else, print the value of "current".

This is the end of my algorithm, for my POC, I have to prove that it prints everything in the desired order. The first step of my algorithm traverses to the left-most node of the tree. In step 3, only nodes with a right sub tree get pushed into the stack (this is to prevent an infinite loop), so this ensures that after the left sub tree of a node is done printing, it traverses will traverse through the right. Also, nodes with no sub trees just get their values printed. Step 2 of my algorithm makes sure that the order of which node values get printed is everything in the left sub tree, then when the root node is popped from the stack, its value gets printed, and finally it traverses the right sub tree and prints everything there. This is the desired order from the problem. The logic by which the while loop breaks is the same as part (a).

Now I will analyze the space-complexity of my algorithm. My stack will always have less nodes than the total number of nodes in the input binary tree because nodes without a right sub tree are never pushed into the stack (including leaves). So, in the worst case scenario, my stack takes up only $n-1$ memory space, which is O(n).
Now I will analyze the run time of my algorithm. This analysis is the same as the analysis of my algorithm in part (a) since both algorithms use the while loop to traverse the binary tree in the exact same way. Since the while loop of my algorithm in part (a) takes linear time, the while loop of my algorithm in this question also takes linear time. Thus, my algorithm's run time is $c_1 * n + c_2$, which is big-theta(n).

Q4. So the input of the algorithm is a queue called B. There's also an empty queue Q and an empty stack S. I will also have a boolean variable called "isOddIndex" set to true. There will be a while loop that loops on the condition that B is not empty. This while loop is meant to separate the submissions in B into S and Q. Here's what one iteration of the while loop will do:
1. Dequeue a submission from B.
2. If "isOddIndex" is true, then I will push the submission into S, else I will enqueue the submission into Q.
3. Flip the value of "isOddIndex".

After this while loop exits, I reset "isOddIndex" to true. Now there will be another while loop that merges the submissions from Q and S back into B. This while loop loops on the condition that Q or S are non-empty. Here's what one iteration does:
1. If "isOddIndex" is true, then I will pop a submission from S, else I will dequeue a submission from Q.
2. Enqueue the submission into B.
3. Flip the value of "isOddIndex".

Finally, I return B.

I will now do a space-complexity analysis. The only actions that took space outside of the Q and S was the boolean variable "isOddIndex". This takes up one memory space, and that is O(1).
I will now do a run time analysis. Every step outside of both while loops take constant time. For both while loops every action inside the loops take constant time. If B is of size n, because the first while loop dequeues once every iteration, it will terminate in n iterations, making the first while loop's run time $c_1 n$. At this point Q and S are of size n when combined. Each iteration of the second while loop takes one submission away from either S or Q. Since the second while loop terminates when both Q and S are empty, and Q and S are of size n combined, the second while loop terminates after n iterations. Thus, the run time for the second while loop is $c_2 n$. Adding the run times of the two while loops and the constant from the actions outside the while loop, my algorithms run time is $c_1 n + c_2 n + c_3 = (c_1 + c_2)n + c_3$, which is O($n$).

Q5.

(a) I will analyze the expected run time of the algorithm. But first, let's analyze the expected run time of one iteration of the algorithm. Input size of T and L is n. The interesting part of the algorithm to analyze is the for loop since every step outside of that loop takes constant time. Let's denote the ith TA with T_i and the chosen location l. In the first iteration of the for loop, the chance that T_1 prefers l is $\frac{1}{n}$. In the next iteration of the loop, the chance that T_2 prefers l is $(\frac{n-1}{n})(\frac{1}{n-1})$, because the first TA must not prefer l, and the chance that that T_2 will prefer l is $\frac{1}{n-1}$ because there is one less TA to ask but the TA that prefers l is still among the remaining TAs. Following this pattern, the probability that T_i for $i \geq 2$ will prefer location l is $(\frac{n-1}{n})(\frac{n-2}{n-1})...(\frac{n-(i-1)}{n-(i-2)})(\frac{1}{n-i+1})$. The logic behind this probability is that the probability of T_1 not preferring l is $1 - Pr(T_1)$, the probability of the T_2 not preferring l is $1 - Pr(T_2)$, and so on, until the probability of T_{i-1} not preferring l is $1 - Pr(T_{i-1})$. This all needs to happen, so their probabilities are multiplied, and then finally, the ith TA needs to prefer l, so the final term to be multiplied is $\frac{1}{n-(i-1)}$. Notice that the numerator of one term can always cancel the denominator of the next term, so the simplified expression is $Pr(T_i) = \frac{1}{n}$ for $1 \leq i \leq n$.
Let X be a random variable that represents how many TAs are asked before a TA prefers l. Now, using the definition of expectation, the expectation of the run time of the loop, with n TAs, is:

$$E[L(n)] = E[X] = \sum_{i=1}^{n}(i * Pr(X = i)) = \sum_{i=1}^{n}(i * Pr(T_i))$$

In the previous paragraph, $Pr(T_i)) = 1/n$. So, the summation simplifies to:

$$\sum_{i=1}^{n}(i * \frac{1}{n}) = \frac{1}{n}\sum_{i=1}^{n} i = \frac{1}{n} * \frac{n(n+1)}{2} = \frac{n+1}{2}$$

.
The algorithm will run n times, each successive iteration will have one less TA and one less location to match. For $1 \leq i \leq n$, the ith iteration of the algorithm will have a run time of $\frac{i+1}{2}$. The total run time of the algorithm will be the summation of the run times of iteration 1 all the way two iteration n:

$$\sum_{i=1}^{n}\frac{i+1}{2} = \sum_{i=1}^{n}\frac{i}{2} + \frac{1}{2} = \frac{n}{2} + \frac{1}{2}\sum_{i=1}^{n} i = \frac{n}{2} + \frac{1}{2}(\frac{n(n+1)}{2}) = \frac{n^2+3n}{4}$$

The expected run time is big-theta(n^2).

I will now analyze the worst-case run time. In the worst case of each iteration of the algorithm, the for loop must iterate to the last TA, who will prefer l, every TA before the last TA must not prefer l. For the ith iteration of the algorithm, the run time of that iteration will then be i. So, the total run time of the algorithm is the sum of the run times of all iterations 1 through n:

$$\sum_{i=1}^{n} i = \frac{n^2+n}{2}$$

This is big-theta(n^2).

(b)

(c) The algorithm in part (a) still works because the only information that the algorithm acts upon is when the TA responds with the climate is just right. So, the lack of information telling us whether the TA thinks the climate is too cool or too tropical does not affect the algorithm.