CS4102 - ALGORITHMS - SPRING 2022

DUE FRIDAY, MARCH 4, 2022 AT 11:30 PM VIA GRADESCOPE

A_ADV(HZ9XS)

Collaboration Policy: You are encouraged to collaborate with up to 3 other students, but all work submitted must be your own *independently* written solution. List the computing ids of all of your collaborators in the `collabs` command at the top of the tex file. Do not share written notes, documents (including Google docs, Overleaf docs, discussion notes, PDFs), or code. Do not seek published or online solutions for any assignments. If you use any published or online resources (which may not include solutions) when completing this assignment, be sure to cite by naming the book etc. or listing a website's URL. Do not submit a solution that you are unable to explain orally to a member of the course staff. Any solutions that share similar text/code will be considered in breach of this policy. Please refer to the syllabus for a complete description of the collaboration policy.

Collaborators: list collaborators's computing IDs

Sources: Cormen, et al, Introduction to Algorithms. *(add others here)*

PROBLEM 1 *Bazinga!*

Theoretical Physicist Sheldon Cooper has decided to give up on String Theory in favor of researching Dark Matter. Unfortunately, his grant-funded position at Caltech is dependent on his continued work in String Theory, so he must search elsewhere. He applies and receives offers from MIT and Harvard. While money is no object to Sheldon, he wants to ensure he's paid fairly and that his offers are at least the median salary among the two schools' Physics departments. Therefore, he hires you to find the median salary across the two departments. Each school mantains a database of all of the salaries for that particular school, but there is no central database.

Each school has given you the ability to access their particular data by executing *queries*. For each query, you provide a particular database with a value k such that $1 \leq k \leq n$, and the database returns to you the k^{th} smallest salary in that school's Physics department.

You may assume that: each school has exactly n physicists (i.e. $2n$ total physicists across both schools), every salary is unique (i.e. no two physicists, regardless of school, have the same salary), and we define the *median* as the n^{th} highest salary across both schools.

1. Design an algorithm that finds the median salary across both schools in $\Theta(log(n))$ total queries.
   Solution:
   We define two databases as list A and list B. There are two cases: the size of the list could be even or odd.

   First, the size of the list is odd. we find the median salary of both lists. Then, compare two median salaries. If the median of list A is larger than the median of list B, we can conclude that only the smaller part of list A and the larger part of list B could be the potential median of the combined list. In contrast, if the median of list A is smaller than the median of list B, we can say that only the larger part of list A and the smaller part of list B could be potential median we'd like to find. In this way, we eliminate the range of the target for both lists by half, and we only need executing queries by 2 times.

   Second, the size of the list is even. If the median of list A is smaller than the median of list B, we eliminate the smaller part of list A and its current median and the larger part of list B. In contrast, if the median of list B is smaller than the median of list A, we eliminate the smaller part of list B with its current median and the larger part of list A. By doing that, we can keep both lists the same size after elimination.

After eliminating some parts of the list, we are looking for the median value of both new lists, and then compare them as we did in the first step. By keeping doing this, we will finally get two values, one in list A and the other in list B. We just need to pick the larger one and that is exactly the n^{th} highest value across both lists.

2. State the complete recurrence for your algorithm. You may put your $f(n)$ in big-theta notation. Show that the solution for your recurrence is $\Theta(log(n))$.
Solution:
base case: $T(1) = 1$

$$\begin{aligned} T(n) &= T(n/2) + 1, i = 1 \\ T(n/2) &= T(n/4) + 1 \\ T(n) &= T(n/4) + 2, i = 2 \\ T(n) &= T(n/(2^i)) + i \\ i &= log_2 n \\ T(n) &= T(1) + i = log_2 n + 1 \end{aligned}$$

Therefore $T(n) \in \Theta(logn)$

3. Prove that your algorithm above finds the correct answer. *Hint: Do induction on the size of the input.*
Solution:

Proof.
Base case:
for $n = 1$: according to our algorithm, when both list only has one value left, we come to the base case and pick the larger value to be the final median of two lists. Now, when $n = 1$, we have exactly one value in each list, so we juts need to compare them and pick the larger one. The result is the first highest value across both lists, which proves this algorithm works for $n = 1$.

Hypothesis:
Assume that $\forall n < n_0$, we can find the right answer for list of size n using my algorithm.

Inductive steps:
we need to prove that my algorithm also works for list with size n_0. Let's implement my algorithm on a list with size n_0.

If n_0 is odd, then we find the median salary of both lists and compare them to eliminate part of the list. Either the median of list A is larger than the median of list B or the median of list B is larger than the median of list A, we eliminate some parts of both lists according to our algorithm. So, We make the size of both lists smaller than n_0 after each list is eliminated by some parts, which is the case where $n < n_0$, and we can guarantee our algorithm is true for both sublists because of our hypothesis.

If n_0 is even, similar to the case where n_0 is odd. We do not care if the median of list A is larger than the median of list B. We just need to know that both lists' size will be smaller than n_0 after eliminating parts of the list after comparison. After we reduce their size below n_0, we can apply our algorithm successfully on the sublists with size $< n_0$ because of our hypothesis.

For either case, the size of the list will be smaller then n_0 after the first comparison and elimination. So, our algorithm works perfectly fine with lists of size n_0. And thus prove our correctness by induction.

□

PROBLEM 2 *Castle Hunter*

We are currently developing a new board game called *Castle Hunter*. This game works similarly to *Battleship*, except instead of trying to find your opponent's ships on a two dimensional board, you're trying to find and destroy a castle in your opponent's one dimensional board. Each player will decide the layout of their terrain, with castles placed on each hill. Specifically, each castle is placed such that they are higher than the surrounding area, i.e. they are on a local maximum, because hill tops are easier to defend. Each player's board will be a list of n floating point values. To guarantee that a local maximum exists somewhere in each player's list, we will force the first two elements in the list to be (in order) 0 and 1, and the last two elements to be (in order) 1 and 0.

To make progress, you name an index of your opponent's list, and she/he must respond with the value stored at that index (i.e., the altitude of the terrain). To win you must correctly identify that a particular index is a local maximum (the ends don't count), i.e., find one castle. An example board is shown in Figure 1. [We will require that all values in the list, excepting the first and last pairs, be unique.]

0	1	4	23	18	14	15	13	1	0
0	1	2	3	4	5	6	7	8	9

Figure 1: An example board of size $n = 10$. You win if you can identify any one local maximum (a castle); in this case both index 3 and index 6 are local maxima.

1. Devise a strategy which will guarantee that you can find a local maximum in your opponent's board using no more than $O(\log n)$ queries, prove your run time and correctness.
   Algorithm:
   We divide the list by two. Compare the last value in the left list and the first value in the right list. If the last value in the left list is larger than the first value in the right list, the left list must contain a local maximum. In contrast, if the last value in the left list is smaller than the first value in the right list, the right list must contain a local maximum. In this way, we can eliminate the range of list by half by just doing constant operation. After you find the half that might contain the local maximum, you just go to that sublist and do the same thing as the first step (divide and compare). Recursively doing that until there is only one value left in the sublist, which is exactly the local maximum we are looking for. And we will use no more than $O(\log n)$ queries.
   Run time prove:
   We divide the list by two each time and just recursively call one sublist with size $n/2$, the combine step is constant(which is 2 more specifically). So, the recurrence is $T(n) = T(n/2) + 2$. $a = 1, b = 2, f(n) = 2 \implies f(n) \in \Theta(n^{\log_b a})$. By using the master theorem, we can conclude that it belongs to case 2, where $T(n) \in \Theta(n^{\log_b a} log n) = log n$
   Correctness prove:
   In order for a list to have no local maximum, the list must be sorted. Considering this problem where we have 01 at the beginning and 10 at the end, there must be a local maximum because the whole list can not be sorted. When we divide the list by two, both sublists could

be sorted or not. If the sublist is not sorted, then there must be a local maximum in the sublist. If the sublist is sorted, there will be two situations. If the last element in the left list is larger than the first element in the right list, the last element in the left list is a local maximum. In the same way, if the first element in the right sublist is larger than the last element in the left sublist, the first element in the right list is a local maximum. whether the sublist is sorted or not, we could use the comparsion and the sublist containing the larger value must contain the local maximum.

2. Now show that $\Omega(\log n)$ queries are required by *any* algorithm (in the worst case). To do this, show that there is a way that your opponent could dynamically select values for each query as you ask them, rather than in advance (i.e. cheat, that scoundrel!) in such a way that $\Omega(\log n)$ queries are required by *any* guessing strategy you might use.
Solution:
We can prove this by using decision tree. To find the local maximum of a list, we need to keep comparing values in the list. For each comparison, we will have two possible path leading to the leaf of the tree. For the leaf, we have n leaf nodes in total, because we have exactly n possible locations for a local maximum to be at. So, considering the worst case, where each path is from the root to the leaf, we need $\Theta(\log n)$ run time.

PROBLEM 3 *Goldilocks and the n Bears*

BookWorld needs your help! Literary Detective Thursday Next is investigating the case of the mixed up porridge bowls. Mama and Papa Bear have called her to help "sort out" the mix-up caused by Goldilocks, who mixed up their n bear cubs' bowls of porridge (there are n bear cubs total and n bowls of porridge total). Each bear cub likes his/her porridge at a specific temperature, and thermometers haven't been invented in BookWorld at the time of this case. Since temperature is subjective (without thermometers), we can't ask the bears to compare themselves to one another directly. Similarly, since porridge can't talk, we can't ask the porridge to compare themselves to one another. Therefore, to match up each bear cub with their preferred bowl, Thursday Next must ask the cubs to check a specific bowl of porridge. After tasting a bowl of porridge, the cub will say one of "this porridge is too hot," "this porridge is too cold," or "this porridge is just right."

1. Give a *brute force* algorithm for matching up bears with their preferred bowls of porridge which performs $O(n^2)$ total "tastes." Prove that your algorithm is correct and that its running time is $O(n^2)$.
Algorithm:
We let the first bear taste the porridge bowls one by one, until it finds its preferred bowl. Then we let the second bear taste the porridge bowls one by one until it finds its preferred bowl. Keep doing that until all bears find their preferred bowl.
Correctness and running time prove:
Literally brute force algorithm can never fail. The first bear finds its favorite and the second bear finds its favorite until the n_{th} bear finds its favorite. So all bears are satisfied with the result. For the run time, we have n bears in total. And for each bear, we need at most n tastes to find its preferred bowl. n bears need n^2 tastes, which is $O(n^2)$ time complexity.

2. Give an *randomized* algorithm which matches bears with their preferred bowls of porridge and performs expected $O(n \log n)$ total "tastes." Prove that your algorithm is correct. Then, intuitively, but precisely, describe why the expected running time of your algorithm is $O(n \log n)$. *Hint: while this is not a sorting problem, your understanding of the sorts we've discussed in class may help when tackling this problem.*
Algorithm:
Put all porridge bowls on a list. Choose a random bear and let it taste all porridge bowls.

For porridge bowls that are too hot for the bear, we move them to the bottom of the list. Now the list composes three parts: porridge bowls that are too cold on the left of the list, a porridge bowl that is fit for the bear on the middle of the list, and porridge bowls that are too hot on the right of the list. Then, randomly choose a second bear and let it taste the porridge on the middle of the list. If the bear considers the porridge bowl to be too hot, the bear can find its porridge bowl on the left part of the list. If the bear considers the porridge bowl to be too cold, the bear can find its porridge bowl on the right part of the list. Then let the second bear do the same thing as our first bear, finding its preferred bowl and dividing the sublist into two parts where the left is colder and the right is hotter. Recursively doing this until the last chosen bear finds its preferred porridge bowl.

Correctness prove:

Although the temperature of the porridge bowl is relatively different for each bear, the temperature of the porridge bowl is relatively constant for a certain bear. In this way, we could use a single bear to determine whether the bowl is too hot or too cold. After we know the relative temperature for each porridge bowl, we could rearrange them as colder on the left and hotter on the right and preferred on the middle. The second chosen bear can use the porridge bowl on the middle as a pivot to find its preferred bowl. The bear will just need to taste the middle bowl and it can eliminate half of the porridge bowls that are not preferred. In this way, we can determine the preferred bowl for each bear one by one just as a binary tree.

Run time prove:

For the first bear, it takes n tastes to find its preferred porridge bowl and arrange the list. For the second bear, it takes $1 + n/2$ tastes to find its preferred porridge bowl and arrange the list, and we want to multiply it by two because the list is divided by two parts. For the third bear, it takes $2 + n/4$ tastes to find its preferred porridge bowl and arrange the list, and we want to multiply it by four because the list is divided by four parts... For the last bear, it takes $log_2(n/2)$ tastes to find its preferred porridge bowl, and we want to multiply it by $n/2$ because the list is divided by $n/2$ parts. When we sum them together, we have $n + 2(1 + n/2) + 4(2 + n/4) + 8(3 + n/8) + ... + 2^i(i + n/2^i)$. For each layer, we need about n tastes. Considering the length of the sum i (how many layers do we have) is $log_2 n$, the overall runtime would be $O(nlogn)$.