CS4102 - ALGORITHMS - SPRING 2020

DUE THURSDAY, FEBRUARY 6 AT 11P VIA COLLAB

H2(YL4DF)

Collaboration Policy: You are encouraged to collaborate with up to 4 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 them. 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: mw5ew

Sources: Cormen, et al, Introduction to Algorithms

PROBLEM 1 *Logs*

Prove that $\log n \in o(n^\varepsilon)$ for any positive constant $\varepsilon > 0$. That is, $\log n$ grows more slowly than any polynomial, even those with fractional powers, e.g., $\varepsilon = 0.00001$. (Note: you may find the limit definitions of the order-classes (CLRS, pp. 50-51) helpful for this problem.)

Proof. Assume ε is an arbitrary positive constant. By using L'Hospital's rule, we have

$$\lim_{n\to\infty} \frac{\log n}{n^\varepsilon} = \lim_{n\to\infty} \frac{\frac{1}{n \ln 2}}{\varepsilon n^{\varepsilon-1}} = \lim_{n\to\infty} \frac{1}{\varepsilon \cdot \ln 2 \cdot n^\varepsilon} = \frac{1}{\varepsilon \cdot \ln 2} \lim_{n\to\infty} \frac{1}{n^\varepsilon}. \tag{1}$$

Since ε is positive, n^ε converges to ∞ as n goes ∞, so the limit above converges to 0. By applying the limit definitions of the order-classes, we conclude $\log n \in o(n^\varepsilon)$. □

PROBLEM 2 *Master Theorem*

Prove a Θ asymptotic bound on the following recurrences using the Master Theorem. Be sure to state which case and why that case applies in your proof. If the Master Theorem does not apply, explain why it fails. (Note: you may need to use substitution before applying the Master Theorem.)

1. $T(n) = 6T(n/2) + n^2$

   Proof. We apply Case 1. $a = 6$, $b = 2$ and let $\varepsilon = 0.5$. Then we have $\log_b a - \varepsilon \approx 2.08$ and thus $f(n) = n^2 \in O(n^{\log_b a - \varepsilon})$. By Master Theorem, we conclude $T(n) \in \Theta(n^{\log_2 6})$. □

2. $T(n) = 27T(n/3) + n^3$

   Proof. We apply Case 2. $a = 27$ and $b = 3$. Then $\log_b a = 3$ and thus $f(n) = n^3 \in \Theta(n^{\log_b a})$. By Master Theorem, we conclude $T(n) \in \Theta(n^3 \log n)$. □

3. $T(n) = 4T(n/2) + n^2 \log n$

   Proof. The Master Theorem does not apply. $a = 4$ and $b = 2$, so $\log_b a = 2$. Clearly, $n^2 \log n \notin \Theta(n^{\log_b a})$. From Problem 1, we know that $\log n \in o(n^\varepsilon)$ for any positive constant $\varepsilon > 0$. This implies that for any positive constant $\varepsilon > 0$, $n^2 \log n \notin O(n^{\log_b a - \varepsilon})$ and $n^2 \log n \notin \Omega(n^{\log_b a + \varepsilon})$. Therefore, all 3 cases fail. □

4. $T(n) = 9T(n/3) + n^2\sqrt{n}$

   Proof. We apply Case 3. $a = 9$, $b = 3$ and $\varepsilon = 0.5$. Then $\log_b a + \varepsilon = 2.5$ and thus $f(n) = n^2\sqrt{n} \in \Omega(n^{\log_b a + \varepsilon})$. There exists $c = 0.6$, for all sufficiently large n, such that $9 \cdot (\frac{n}{3})^2\sqrt{\frac{n}{3}} \leq c \cdot n^2\sqrt{n}$. By Master Theorem, we conclude $T(n) \in \Theta(n^2\sqrt{n})$. □

5. $T(n) = 3T(n/9) + \sqrt{n}$

   Proof. We apply Case 2. $a = 3$ and $b = 9$. Then $\log_b a = \frac{1}{2}$ and thus $f(n) = \sqrt{n} \in \Theta(n^{\log_b a})$. By Master Theorem, we conclude $T(n) \in \Theta(\sqrt{n}\log n)$. □

6. $T(n) = 2T(\sqrt{n}) + 2$

   Proof. We apply substitution at first by letting $n = 2^m$. Then we rewrite the equation as $T(2^m) = 2T(2^{\frac{m}{2}}) + 2$. Then let $S(m) = T(2^m)$, so we have $S(m) = 2S(\frac{m}{2}) + 2$. We apply Case 1 on $S(m)$. It follows that $a = 2$ and $b = 2$. Let $\varepsilon = 0.5$. This implies $\log_b a - \varepsilon = 0.5$. Then we have $2 \in O(m^{\log_b a - \varepsilon})$, so $S(m) = \Theta(m)$ by Master Theorem. Then we substitute n back and thus we have $T(n) = \Theta(\log_2 n)$ □

PROBLEM 3 *Fast Exponentiation*

Given a pair of positive integers (a, n), give pseudo-code for a divide and conquer algorithm that computes a^n using only $O(\log n)$ multiplications. Prove that your algorithm is $\in O(\log n)$.

Pseudo-code for power (a, n):
If $n = 0$, then the algorithm returns 1;
If $n = 1$, then the algorithm returns a;
Declare a variable to store value of power $(a, \lfloor \frac{n}{2} \rfloor)$. Call it temp.
If n is a multiple of 2, then the algorithm returns temp · temp;
Else, the algorithm returns a · temp · temp;
The algorithm divides a by 2 and take 1 or 2 multiplications in each step, so we have

$$T(n) = T(\frac{n}{2}) + c \tag{2}$$

where c is 1 or 2. Then we use induction to prove $T(n) \leq c\log n \in O(\log n)$.
Proof. Base Case: $T(1) = 0 \leq c\log 1$.
Hypothesis: $\forall x_0 \leq n$, $T(n) \leq c\log n$.
Inductive Step:

$$\begin{aligned} T(x_0 + 1) &= T(\frac{x_0 + 1}{2}) + c && (3)\\ &\leq c\log_2 \frac{x_0 + 1}{2} + c && (4)\\ &\leq c\log_2(x_0 + 1) - c + c && (5)\\ &\leq c\log_2(x_0 + 1) && (6)\\ & && (7) \end{aligned}$$

Therefore, $T(n) \leq c\log n \in O(\log n)$. □

PROBLEM 4 *Castle Hunter*

We are planning 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; castles are placed on every hill. Specifically, each castle is placed such that they are higher than the surrounding area, i.e., they are on a local maximum. (After all, hilltops are easier to defend!) Each player's board will be a list of n floating point values corresponding to the elevation. 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 elevation of the terrain). To win you must correctly identify that a particular index is a local maximum (the first and last elements 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.

   Strategy:
   Step 1: Find the middle index of the board by $k = \lceil \frac{t-s}{2} \rceil$ where t is the last index and s is the first index. Then query the number at index k, $k-1$, and $k+1$ and compare.
   Step 2:
   Case 1: If the number at index k is a maximum among the three number, then we're done.
   Case 2: If the number at index $k-1$ is a maximum among the three number, then we let $t = k-1$ and repeat the strategy from Step 1.
   Case 3: If the number at index $k+1$ is a maximum among the three number, then we let $s = k+1$ and repeat the strategy from Step 1.
   For each time, we divide the board by 2 and perform 3 queries, so we have

   $$T(n) = T(\frac{n}{2}) + 3. \tag{8}$$

   Then we prove $T(n) \leq 3 \log n \in O(\log n)$ and correctness.

   Proof. Run time:
   Base Case: The board should at least contains 5 elements according to the requirement, so we use size 5 as our base case. When $n = 5$, we can finish the game by 3 or 6 queries, so $T(5) \leq 3 \log 5$.
   Hypothesis: $\forall x_0 \leq n$, $T(n) \leq 3 \log n$.

Inductive Step:

$$T(x_0+1) = T(\frac{x_0+1}{2}) + 3 \tag{9}$$
$$= 3\log(\frac{x_0+1}{2}) + 3 \tag{10}$$
$$= 3\log(x_0+1) - 3 + 3 \tag{11}$$
$$= 3\log(x_0+1) \tag{12}$$

Therefore, $T(n) \leq 3\log n \in O(\log n)$.
Correctness:
For Case 1, if the middle element is bigger than the other two, then our goal is fulfilled. For Case 2 and 3, they can be proved in the same way with opposite side. Without loss of generality, let's consider Case 2. When the left element (index $k-1$) is bigger, this implies that the left element (index $k-1$) could be a local maximum. If it is, then our strategy will work. If it is not, then its left element (index $k-2$) could be a local maximum. Following this idea, from the middle to the left, there is a growing-up trend. The trend will always end up, because the first two elements are 0 and 1 and no duplicate. As long as the trend ends up, there is a local maximum, so our strategy works for Case 2. Similarly, for Case 3, when the right element (index $k+1$) is bigger, this implies that the right element (index $k+1$) could be a local maximum. If it is, then our strategy will work. If it is not, then its right element (index $k+2$) could be a local maximum. Following this idea, from the middle to the right, there is a growing-up trend. The trend will always end up, because the last two elements are 1 and 0 and no duplicate. As long as the trend ends up, there is a local maximum, so our strategy works for Case 3. □

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.

   Proof. First, there is one castle in opponent's array. Then no matter what algorithm that we take, the opponent can always give a sequence consisting of three monotonic numbers. As we know, the array will be divided every time when we query, so the opponent can always put the castle in the longer sub-array by providing corresponding monotonicity. The monotonicity will lure us to reach the last number available in the array. This means we at least need $3\log n$ queries, so $\Omega(\log n)$ queries are required by *any* algorithm (in the worst case). □