CS4102 - ALGORITHMS - SPRING 2022

DUE FRIDAY, FEBRUARY 18, 2022 AT 11:30 PM VIA GRADESCOPE

A2BASIC(SOLUTIONS)

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 *Asymptotics*

1. Write a mathematical statement using the appropriate order-class to express "Algorithm A's worst-case $W(n)$ is quadratic."
   Solution: $W(n) \in \Theta(n^2)$. (Notes: In CS we often us $=$ instead of $\in$. Either is fine., though order-classes are sets. A $W(n)$ is a specific formula so we an be precise and use Θ instead of O or Ω.)

2. Write a mathematical statement using the appropriate order-class to express "Algorithm A's time-complexity $T(n)$ is never worse than cubic for any input."
   Solution: $T(n) \in O(n^3)$.

3. Write a statement using words and an appropriate order-class to express "It's not possible for an algorithm that solves problem P to succeed unless it does at least a cubic number of operations."
   Solution: Any algorithm that solves P will be $\Omega(n^3)$ (or something similar using Ω).

4. Prove or disprove the following statement: $n(\log n)^2 \in O\left(n^{1.5}(\log n)\right)$.

   Solution: True. We must show $\exists c > 0, n_0 > 0$ such that $\forall n > n_0, n(\log n)^2 \leq cn^{1.5} \log n$. Let $c = 1$ and $n_0 = 1$, then

   $$\begin{aligned} n(\log n)^2 &\leq cn^{1.5} \log n \\ n(\log n)^2 &\leq n^{1.5} \log n \\ n(\log n)(\log n) &\leq n^{1.5} \log n \\ (\log n)(\log n) &\leq n^{1/2} \log n \\ (\log n) &\leq n^{1/2}. \end{aligned}$$

   For all $n > 1$, $\log n \leq \sqrt{n}$ and therefore $n(\log n)^2 \in O(n^{1.5} \log n)$.

   Solution (Alternative): A second way to solve this is to use the limit definition for order classes:

if $f(n) \in O(g(n))$ then $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ where $c \leq \infty$.

$$\begin{aligned}
\lim_{n\to\infty} \frac{n(\log n)^2}{n^{1.5}\log n} &= \lim_{n\to\infty} \frac{\log n}{n^{0.5}} && \text{(cancel terms)} \\
&= \lim_{n\to\infty} \frac{k/n}{(1/2)n^{-0.5}} && \text{(apply L'Hopital's rule)} \\
&= \lim_{n\to\infty} \frac{2k}{n^{0.5}} && \text{(simplify)} \\
&= 0 && \text{(limit of a constant over function of n is zero)}
\end{aligned}$$

Therefore $n(\log n)^2 \in O(n^{1.5}\log n)$ (and in fact it's in "little-oh" and grows more slowly).

PROBLEM 2 *Basic Sorting*

1. In a few sentences, explain if changing the comparison done in mergesort's *merge()* function from $\leq$ to $<$ makes the sorting algorithm incorrect, and also whether it makes the sort unstable.
   Solution: This change will only make an impact when merge is is comparing elements in the sublists that are equal. In this case, either element can be moved without affecting correctness, so the algorithm is still correct. This does make this sort unstable, since the change will move the item in the second sublist before the element from the first sublist.

2. Which of the following are true about insertion sort and mergesort?

   (a) Insertion sort would run reasonably fast when the list is nearly in reverse-sorted order but with a few items out of order.

   (b) For small inputs we would still expect mergesort to run more quickly that insertion sort.

   (c) The lower-bounds argument that showed that sorts like insertion sort must be $\Omega(n^2)$ does not apply to mergesort because when a list item is moved in *merge()* it may un-do more than one inversion.

   (d) We say the cost of "dividing" in mergesort is 1 because we must do a constant amount of work to find the midpoint of the subproblem we're sorting.

   Solution: The 3rd item is true, and the others are false. The 1st is false because many items would have to move a great distance if the list is nearly in reverse-sorted order. (It would be true if we said in "forward" sorted order.) For the 2nd question, for small inputs we don't expect to see much difference in the run-times. For the 4th question, we count comparisons and so we'd say 0 are done during the divide step.

PROBLEM 3 *Recurrence Relations*

1. Reduce the following recurrence to its closed form (i.e. remove the recursive part of its defininiton) using the *unrolling method*.

$$T(n) = 3T(n/3) + n \text{ and } T(1) = 1$$

Be sure to show the general form of the recurrence in terms of how many times you've "un-rolled", as well as a formula for how many times you "un-roll" before getting to the base case.

Solution:

$$\begin{aligned} T(n) &= 3T\left(\frac{n}{3}\right) + n && \text{(count is } i = 1\text{)} \\ &= 3\left[3T\left(\frac{n}{9}\right) + \frac{n}{3}\right] + n = 9T\left(\frac{n}{9}\right) + 2n && \text{(count is } i = 2\text{)} \\ &= 9\left[3T\left(\frac{n}{27}\right) + \frac{n}{9}\right] + n = 27T\left(\frac{n}{27}\right) + 3n && \text{(count is } i = 3\text{)} \\ &= 3^i T\left(\frac{n}{3^i}\right) + in && \text{(after } i \text{ iterations)} \end{aligned}$$

The unrolling will continue until $n = 1$, so solving for i when $1 = \frac{n}{3^i}$ finds $i = \log_3 n$. Substituting this back into the general form shows:

$$T(n) = 3^{\log_3 n} T\left(\frac{n}{3^{\log_3 n}}\right) + (\log_3 n)n = n + n\log_3 n \in \Theta(n \log n)$$

(So the order class is just like mergesort's recurrence. Are you surprised? No, if you've studied the Master Theorem.)

2. Use the Master Theorem to find the order-class for this recurrence: $T(n) = 3T(n/2) + n\log n$. State which case applies, and if no case applies and the Master Theorem cannot be used, state that and explain why.
Solution: We're comparing $f(n) = n\log n$ to $n^{\log_2 3} = n^{1.585}$). At first glance it appears $f(n)$ grows more slowly, so let's try Case 1. That applies if $n\log n \in O(n^{1.585-\epsilon})$. If say $\epsilon = 0.1$ then this is true. Therefore Case 1 does apply, and $T(n) \in O(n^{\log_2 3})$.

3. Use the Master Theorem to find the order-class for this recurrence: $T(n) = 3T(n/4) + n\log n$. State which case applies, and if no case applies and the Master Theorem cannot be used, state that and explain why.
Solution: We're comparing $f(n) = n\log n$ to $n^{\log_4 3} = n^{0.79}$). (Even without a calculator, you know that $\log_4 3 < 1.0$.) At first glance it appears $f(n)$ grows more quickly, so let's try Case 3. That applies if $n\log n \in \Omega(n^{0.79+\epsilon})$. If ϵ is something small, say 0.1 then this is true.

   For Case 3 we do have to check the regularity condition and show $3(n/4)(\log(n/4)) \leq cn\log n$ for some $c < 1$. You can see that $0.75(n\log n - \log 4) \leq cn\log n$ when $c = 0.75$.

   Therefore Case 3 does apply, and $T(n) \in O(n\log n)$.

   Note: For the above two problems, we were able to find an ϵ to make a case apply. Study the example where this is not possible in CLRS pp. 95f.

4. Show you understand how to do a proof using the "guess and check" method and induction. Show that the following recurrence $\in O(n\log_2 n)$:

$$T(n) = 4T(n/4) + n \text{ and } T(1) = 1$$

   You can assume n is a power of 4.

   Hints: For the induction, you have to prove the relationship for a small value of n. You'll find $n = 1$ doesn't work, but you can show it holds for the next larger value of n. (Again,

assume n is a power of 4.) It's OK for the induction proof if the relationship holds for some small value of n even if it doesn't hold for $n = 1$.
Also, you'll need to guess a value for c. For this problem, the value of c is not anything strange or unusual. A small value will work, you will find it easiest to just keep c in your math calculations and when you get to the final step you can see what value of c makes your relationship true. (This problem is much easier than the example we did in class!)

Solution: To show $T(n) = 4T(n/4) + n \in O(n \log_2 n)$, we need to find a value c such that $T(n) \leq cn \log_2 n$ for all n greater than some n_0.

Let's first find a base case for our induction proof. We cannot find a c to make the relationship hold for $n = 1$: $T(1) = 1$ and $c \times 1 \times log_2 1 = 0$. But it does hold for $n = 4$: $T(4) = 4T(4/4) + 4 = 8$, which is $\leq c \times 4 \times \log_2 4$ for any $c \geq 1$.

For our induction hypothesis, we'll assume $\forall n \leq n_0, T(n) \leq cn \log_2 n$. So our goal is to show $T(n_0 + 1) \leq c(n_0 + 1) \log_2(n_0 + 1)$ for some value c. (We could guess at c right now, but let's keep it in the math and see what it needs to be at the very end.)

$$
\begin{aligned}
T(n_0+1) &= 4T\left(\frac{n_0+1}{4}\right) + (n_0+1) && \text{(apply recurrence)} \\
&\leq 4\left[c\left(\frac{n_0+1}{4}\right)\log_2\left(\frac{n_0+1}{4}\right)\right] + (n_0+1) && \text{(apply induction hypothesis)} \\
&= c(n_0+1)\log_2\left(\frac{n_0+1}{4}\right) + (n_0+1) && \text{(simplify)} \\
&= c(n_0+1)\left[\log_2(n_0+1) - \log_2 4\right] + (n_0+1) && \text{(some rules of logs)} \\
&= c(n_0+1)\log_2(n_0+1) - 2c(n_0+1) + (n_0+1) && \text{(simplify)} \\
&\leq c(n_0+1)\log_2(n_0+1) && \text{(for } c \geq \frac{1}{2}\text{)}
\end{aligned}
$$

Solution (Alternative): It's equally valid to state our induction hypothesis with a $<$ rather than a $\leq$, like this: $\forall n < n_0, T(n) \leq cn \log_2 n$. Then our goal is to show $T(n_0) \leq cn_0 \log_2(n_0)$ for some value c. The proof looks the same but you won't have all those "+1" and it may be easier to follow (and type). You may have learned to do induction proofs this way.

PROBLEM 4 *Divide and Conquer #1*

Write pseudo-code that implements a divide and conquer algorithm for the following problem. Given a list L of size n, find values of the largest and second largest items in the list. (Assume that L contains unique values.)

In your pseudo-code, you can indicate that a pair of values is returned by a function using Python-like syntax, if you wish. For example, a function $funky()$ that had this return statement:

```
return a, b
```

would could be used to assign a to x and b to y if called this way:

```
(x, y) = funky()
```

Solution: See code below. Note how the entire list is passed and *first* and *last* are used to control what subsection of the list is processed by a recursive call. This avoids creating sublists at cost $O(n)$ every time you make a recursive call.

```
function find_max_max2(L, first, last)
    mid = (first + last) / 2

    (max_left, max2_left) = find_max_max2(L, first, mid)
    (max_right, max2_right) = find_max_max2(L, mid+1. last)

    max_val = max(max_left, max_right)
    max2_val = max( max2_left, max2_right, min(max_left, max_right) )
    return (max_val, max2_val)
```

PROBLEM 5 *Divide and Conquer #2*

Conference Superstar. There is a CS conference with n attendees. One attendee is a "superstar" — she is new to the field and has written the top paper at the conference. She is the attendee whom all other attendees know, yet she knows no other attendee. Specifically, if attendee a_i is the superstar, then $\forall a_j \neq a_i$, $knows(a_j, a_i) == true$ and $knows(a_i, a_j) == false$. Other attendees may or may not know each other, as is true for "normal" meetings. Give a $O(n)$ algorithm which determines who the superstar is.

Hint: Compare pairs of attendees and try to eliminate one of them. Then you might want to do a swap for each comparison to make sure all attendees that have a certain property are together in one part of your list so you can recurse on just those.

Solution: Divide the list into two sublists. Iterate, comparing a person A from the first sublist with a person B in the second sublist. If A knows B, then A is not superstar. If A doesn't know B, then B is not the superstar. So we can eliminate one from each pair. If needed, do a swap to move the eliminated person into one of the sublists, so that after you've iterated over all pairs, one sublist contains only eliminated attendees. Then recurse on the other sublist. (Note: if the number of attendees at any step is not even, make sure the "leftover" attendee is included in the sublist that you recurse on (since you cannot eliminate them). You'll reach a base case of one attendee who has not been eliminated, and that must be the superstar.

The recurrence is $T(n) = T(n/2) + n/2$, which is $\Theta(n)$.

PROBLEM 6 *Gradescope Submission*

Submit a version of this `.tex` file to Gradescope with your solutions added. You should only submit your `.pdf` and `.tex` files.