CS4102 - ALGORITHMS - SPRING 2022

DUE FRIDAY, FEBRUARY 18, 2022 AT 11:30 PM VIA GRADESCOPE A2BASIC(RTG5XKH)

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: ra9ha

Sources: Cormen, et al, Introduction to Algorithms.

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: $W(n) \in \mathcal{O}(n^2)$

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: $T(n) \in \mathcal{O}(n^3)$

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: *A solves P and A's time complexity is given by* $T(n)$. $T(n) \in \Omega(n^3)$

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

   Solution:

   Proof. Let $c = 1, n_0 = 1$.
   Then, $n_0(\log n_0)^2 = (1)(log1)^2 = 0$,
   $c(n_0^{1.5}(\log n_0)) = (1)(\log 1) = 0$
   $0 \leq 0$.
   The proposition $n(\log n)^2 \leq n^{1.5}(\log n)$ can be simplified as follows:

   $$n(\log n)(\log n) \leq n^{1.5}(\log n)$$
   $$\log n \leq \sqrt{n}$$

   $\forall n \geq 1, \log n < \sqrt{n} \implies n(\log n)^2 \leq n^{1.5}(\log n)$.
   Thus, $n(\log n)^2 \in O\left(n^{1.5}(\log n)\right)$. □

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: *No, changing this comparison would not make the merge sort algorithm incorrect. The comparison would still correctly identify when the current element in the left subarray was less than that in the right and place it in the sorted array accordingly. However, this change would make the merge sort algorithm unstable, meaning that elements with the same value in the original array could be placed in a different order than they appeared in the original array. This is because by changing the $\leq$ to a $<$ would make it so that, in case when the current element of the left and right subarray are equal, the if statement containing the $<$ would not execute, so the else statement would execute and the element in the right subarray would be added to the new array next. This would swap the order in which the two equal elements originally appeared.*

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.

   Solution: *False*

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

   Solution: *False. We would expect insertion sort to be faster than merge sort for small inputs because the overhead of dividing the data and merging it is proportionally large for these small inputs which will cause merge sort to run relatively slowly.*

   (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.

   Solution: *True*

   (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: *True. The cost of dividing for mergesort is constant because the algorithm doesn't do any actual work (i.e. comparisons) in the divide stage.*

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 "unrolled", as well as a formula for how many times you "un-roll" before getting to the base case.

Solution:

$$\begin{aligned} T(n) &= 3\left(3T\left(\frac{n}{9}\right) + \frac{n}{3}\right) + n \\ &= 9T\left(\frac{n}{9}\right) + 2n \\ T(n) &= 9\left(3T\left(\frac{n}{27}\right) + \frac{n}{9}\right) + 2n \\ &= 27T\left(\frac{n}{27}\right) + 3n \end{aligned}$$

General Case: let i = 1 for the original recurrence relation, i = 2 for the first unroll, i = 3 for the second and so on. Then, the equation can be expressed in terms of i for i unrolls:

$$3^i T\left(\frac{n}{3^i}\right) + i(n)$$

To reach the base case, we must find i and n such that $n = 3^i$. Rearranging, we get $i = \log_3 n$. We can plug this into the general case relation to get:

$$\begin{aligned} T(n) &= 3^{\log_3 n} T\left(\frac{n}{3^{\log_3 n}}\right) + \log_3 n(n) \\ &= nT(1) + n\log_3 n \\ &= n + n\log_3 n \end{aligned}$$

Thus, $T(n) \in \Theta(n \log n)$.

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: *Using the master theorem format, we get:*

$$\begin{aligned} a = 3, b = 2, f(n) = n\log n. \\ log_2 3 = 1.585. \\ n\log n \in \mathcal{O}(n^{1.5}) \end{aligned}$$

(we used $\varepsilon \approx .085$)
Thus, the first case of the master theorem applies.
Conclusion: $T(n) \in \Theta(n^{1.585})$

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: *Using the master theorem format, we get:*

$$a = 3, b = 4, f(n) = n\log n.$$
$$log_4 3 = .79.$$
$$n\log n \in \Omega(n)$$

(we used $\varepsilon \approx .21$ for the constant to make things simple)
Thus, the third case of the master theorem applies.
The first condition is true, we just need to check the regularity condition.
Show $af(\frac{n}{b}) \leq cf(n)$ for some constant c:

$$3(\frac{n}{4})log(\frac{n}{4}) \leq cnlogn$$
$$\frac{3n}{4}(logn - log4) \leq clogn$$
$$\frac{3}{4}(nlogn) - 6n/4 \leq cnlogn$$

Thus, we can choose $c = .75$, which will make the $nlogn$ terms on each side equal so they will cancel out. And since the $-6n/4$ is negative, the left side will always be smaller. So the regularity condition holds.
Therefore, $T(n) \in \Theta(nlogn)$

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:

Goal: $T(n) \leq cn \log_2 n \in \mathcal{O}(n \log n)$

Base Case: $T(4) = 8 \leq cn \log_2 n = 8c$,
Holds for any $c \geq 1$.

Hypothesis: $\forall n \leq n_0$, $T(n) \leq cnlog_2 n$.

Show $T(n_0 + 1) \leq c(n_0 + 1) \log_2 (n_0 + 1)$:

$$\begin{aligned}
T(n_0+1) &= 4T(\frac{n_0+1}{4}) + n_0 + 1 \\
&\leq 4\left(c(\frac{n_0+1}{4})log_2(\frac{n_0+1}{4})\right) + n_0 + 1 \\
&= c(n_0+1)log_2(\frac{n_0+1}{4}) + n_0 + 1 \\
&= c(n_0+1)\left(\log_2(n_0+1) - \log_2(4)\right) + n_0 + 1 \\
&= c(n_0+1)\left(\log_2(n_0+1) - 2\right) + n_0 + 1 \\
&= c(n_0+1)\left(\log_2(n_0+1)\right) - 2c(n_0+1) + n_0 + 1
\end{aligned}$$

Now, if we choose $c = 1/2$ and plug it into the above equation, we find that:

$$T(n_0 + 1) = c(n_0 + 1) \log_2 (n_0 + 1)$$

Thus the goal holds.
Therefore $T(n) \in \mathcal{O}(n \log n)$.

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:

```
#assume 0 is passed into start and n − 1 into end
#assume length of list is >= 2

function largest2(L, start, end):

if start - end == 1:
  if L[start] > L[end]:
    return [L[start], L[end]]
  else:
    return [L[end], L[start]]

mid = (start + end) // 2;

if start == end:
  return [L[start], INT.MINVALUE]

largest2r = largest2[L, mid + 1, end]
largest2l = largest2[L, start, mid]

if largest2r[0] > largest2l[0]:
  if largest2r[1] > largest21[0]:
    return [largest2r[0], largest2r[1]]
  else:
    return [largest2r[0], largest2l[0]]
else:
  if largest2l[1] > largest2r[0]:
    return [largest21[0], largest21[1]]
  else:
    return [largest21[0], largest2r[0]]
```

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: We will implement a recursive algorithm

Base case: 1 attendee left in the list, this attendee is the superstar

Recursive case: split list into 2 halves, and pair up the first attendee in the first half of the list to the first attendee in the second half of the list and so on (for example, in a 4 person list indices 0 and 2 would be paired up, and indices, 1 and 3 would be paired). For an odd number of attendees, we will have one extra attendee in the first "half" of the list and will pair the rest as usual. For each pair of attendees a and b, if a knows b and b doesn't know a, swap their positions in the list. If there is an odd number of attendees, one attendee will not be paired and we will do nothing and leave them in the first half of the list. After iterating through once with this swapping process, potential candidates for the superstar will all be in the first half of the list. We will then clear all attendees in the second half of the list and repeat the process with the resulting list.

Time complexity: In this algorithm, we do 2 comparisons on each pair and split the list into 2 until there is only one attendee left. Thus, the time complexity for an n-attendee list can be given as $T(n) = T\left(\frac{n}{2}\right) + n$. Using the master theorem, $a = 1, b = 2, f(n) = n$. $n^{\log_b a} = n^0 = 1$. Case 3 applies, so $T(n) \in \mathcal{O}(n)$.

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.