CS4102 - ALGORITHMS - SPRING 2022

DUE FRIDAY, FEBRUARY 18, 2022 AT 11:30 PM VIA GRADESCOPE **A2BASIC(SSB3VK)**

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: none

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

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: *There exists no solution to problem P that takes less than a cubic number of operations... Or A solution to problem P's operational complexity(T(n)) is never better than cubic for any input.* $T(n) \in \Omega(n^3)$

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

   Solution: *The simplest way to go about this is to use the limit definition seen in Equation 1.*

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \tag{1}$$

Where $f(n) = n(\log n)^2$ *and* $g(n) = ((n^{1.5}(\log n))$. *The conditionals to prove big O is that the limit defined in Equation 1 is less than infinity.*

$$\lim_{n\to\infty} \frac{n(\log n)^2}{((n^{1.5}(\log n))} \tag{2}$$

$$= \lim_{n\to\infty} \frac{(\log n)}{((n^{1/2})} \tag{3}$$

All we need to do now is show that $n^{1/2}$ *will be larger than logn after a certain value of n. We know that* $\sqrt{1} - log1 > 0$ *which is one part of the problem. The second part is to show that* $sqrtn - logn$ *is increasing. (For convenience we will use the natural log.*

$$h(n) = \sqrt{n} - logn \tag{4}$$

$$h'(n) = \frac{1}{2\sqrt{n}} - \frac{1}{n} \tag{5}$$

In the above function, we know that $\sqrt{n}$ grows at a rate slower than n making the left term of $h'(n)$ grow larger than the right indicating that $h'(n)$ will be positive, leading to the conclusion that h(n) is increasing. To be more specific, the left term overtakes the right term after $n = 4$

Now we can go back to Equation 3. Because we know that $\sqrt{n}$ will eventually exceed and grow larger than the logn term, we can conclude that the limit tends to 0. This satisfies the originally stated condition that the limit has to be less than infinity. Therefore, we can conclude that $n(\log n)^2 \in O\left(n^{1.5}(\log n)\right)$.

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: *Changing the comparison within mergesort's merge() function will not make the sorting algorithm incorrect. The way we have it set up is that if a pair of values across arrays is not $\leq$ then we default to adding the term from the second array. In the case of equivalent pairs of values, then the difference between $\leq$ and $<$ is irrelevant as both values are the same and will be sorted properly.*

   Now for the question of stability, changing from a $\leq$ to a $<$ will definitely affect the stability of the algorithm. In the merge() method, this change would enable equal elements from the right side of the merge to be placed behind an equivalent element from the left side of the merge. This implies that the stability of the algorithm is compromised when this change is executed.

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: *The reverse-sorted list is the scenario which produces the worst-possible run time for insertion sort.*

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

   Solution: *Insertion sort would probably be faster.*

   (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: *This is true. Consider the case that we are performing a merge() with two lists where the left input consists of values that are strictly greater than the values in the right input. Every iteration of the merge() function will resolve inversions equal to the number of values of the left input. This is why mergesort is faster than insertion sort, the potential to resolve more than one inversion per iteration.*

   (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: *This is true because we only need to split the arrays in half (or near half), an operation that takes a constant amount of time.*

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: *Before I go around plugging things in, I want to compute the equations for T(n/3) and T(n/9) because these will be useful later.*

$$\begin{aligned} T(n/3) &= 3T(\frac{n/3}{3}) + n/3 \\ &= 3T(n/9) + n/3 \end{aligned}$$

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

Now, I can keep plug in T operating on subsequently smaller portions of n into the spaces that the original T(n) function has

$$\begin{aligned} T(n) &= 3T(n/3) + n && (1) \\ &= 3(T(n/3)) + n && (2) \\ &= 3(3T(n/9) + n/3) + n && (3) \\ &= 9T(n/9) + n + n && (4) \\ &= 9T(n/9) + 2n && (5) \\ &= 9(T(n/9)) + 2n && (6) \\ &= 9(3T(n/27) + n/9) + 2n && (7) \\ &= 27T(n/27) + n + 2n && (8) \\ &= 27T(n/27) + 3n && (9) \end{aligned}$$

By plugging in the T(n/3) and T(n/9) found in the prior equations into equations 3 and 7, we can find a pattern. This pattern is most noticeable when comparing equations 1, 5, and 9:

$$\begin{aligned} T(n) &= 3T(n/3) + n && (1) \\ &= 9T(n/9) + 2n && (2) \\ &= 27T(n/27) + 3n && (3) \end{aligned}$$

If we assume that i = 1 on the first equation, we can generalize this pattern to:

$$T(n) = 3^i T(\frac{n}{3^i}) + in \quad (1)$$

Now that we've established a pattern, we have to look ahead to see where we will stop this pattern. I know that we stop at T(1) or whenever $\frac{n}{3^i} = 1$. *Now we can solve for n:*

$$T(\frac{n}{3^i}) = T(1) \tag{2}$$
$$\frac{n}{3^i} = 1 \tag{3}$$
$$n = 3^i \tag{4}$$
$$log_3(n) = log_3(3^i) \tag{5}$$
$$log_3(n) = log_3(3)i \tag{6}$$
$$log_3(n) = i \tag{7}$$

Now, we can make the proper substitutions into the generalized equation for i and T to get:

$$T(n) = 3^{log_3(n)}T(1) + log_3(n)n \tag{1}$$

And if we do some simplifications and use the fact that $T(n) = 1$, *we can get:*

$$T(n) = n * T(1) + log_3(n)n \tag{2}$$
$$= n * 1 + log_3(n)n \tag{3}$$
$$= n + log_3(n)n \tag{4}$$

We find that through unrolling, our recurrence's closed form resembles something close to an $nlog(n)$ *runtime algorithm.*

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: *Case 1 of the Master Theorem is most applicable here. This is because the* $f(n)$ *term in this statement is indeed bound by* $n^{log_2(3)}$. *Although this relationship may initially be difficult to see, with some analysis, we can come to the conclusion we need.*

   The simplest way to go about this is to use the limit definition seen in Equation 1.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \tag{1}$$

Where $f(n) = n(\log n)$ *and* $g(n) = (n^{log_2(3)})$. *The conditionals to prove big O is that the limit defined in Equation 1 is less than infinity.*

$$\lim_{n\to\infty} \frac{n \log n}{n^{1.5}} \tag{2}$$
$$= \lim_{n\to\infty} \frac{\log n}{((n^{0.58496250072})} \tag{3}$$

All we need to do now is show that $n^{0.58496250072}$ is larger than $logn$. This is a little difficult, so instead all I will show is that $n^{1/2}$ is larger than $logn$. But, we already demonstrated this earlier in part 4 of problem 1.

Now we can go back to Equation 3. Because we know that $\sqrt{n}$ (or $n^{0.58496250072}$) will eventually exceed and grow larger than the $logn$ term, we can conclude that the limit tends to 0. This satisfies the originally stated condition that the limit has to be less than infinity. Therefore, we can conclude that $n(\log n) \in O\left(n^{1.58496250072}\right)$.

This leads us to conclude that $T(n) \in \Theta(n^{log_2(3)})$

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: *I think it is pretty obvious to see that this recurrence fulfills at least the first component of Case 3. $nlog(n)$ is significantly larger than $n^{log_4(3)}$, even with only a cursory glance, we can find that just the n term of the former is larger than the entire latter term. But now, we have to assess the second component of Case 3:*

$$af(\frac{n}{b}) \leq cf(n) \tag{1}$$

For some constant $c < 1$ and for all sufficiently large n. Now lets plug some values in and take a look:

$$3 * ((\frac{n}{4})log(\frac{n}{4})) \leq cnlog(n) \tag{2}$$

Now if we apply a certain log rule:

$$\frac{3n}{4}(log(n) - log(4)) \leq c * n * log(n) \tag{3}$$

We can make a modification such that this becomes simpler to solve (specifically by removing a negative term that is guaranteed to reduce the value of our left side.

$$\frac{3n}{4}(log(n)) \leq \frac{3n}{4}(log(n) - log(4)) \leq c * n * log(n) \tag{4}$$

$$\frac{3n}{4}(log(n)) \leq c * n * log(n) \tag{5}$$

$$\frac{3n}{4} \leq c * n \tag{6}$$

Because the above statement is clearly true for values of c that are greater than 3/4 (and we can still satisfy the statement with values of c less than 1) we can conclude that Case 3 of the Master theorem applies and conclude that $T(n) \in \Theta(nlog(n))$

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: *Off a cursory glance, I want to guess that this recurrence relation is* $\Theta(nlog(n))$. *But, my answer is made simpler by the fact that the problem says that this recurrence is* $\in O(nlog_2(n))$.

Goal:

$$T(n) \leq n * log_2(n) = O(nlog_2(n)) \tag{1}$$

Base cases:

$$n = 4 \tag{2}$$

$$T(4) = 4T(\frac{4}{4}) + 4 \leq 4 * log_2(4) \tag{3}$$

$$4 * T(1) + 4 \leq 4 * 2 \tag{4}$$

$$4 * 1 + 4 \leq 4 * 2 \tag{5}$$

$$8 \leq 8 \tag{6}$$

This does not need to be verified for larger values of n because the multiplicative growth of $n * log_2(n)$ outstrips the individual growths of the 4*T(n/4) term as well as the n term. But $n = 4$ is value n_0 because that is the point where the right side of the inequality outstrips the left.

Hypothesis:

$$\forall n \leq x_0, T(n) \leq 4 * log_2(4) \tag{7}$$

Inductive Step:

$$T(x_0 + 1) = 4T(\frac{x_0 + 1}{4}) + (x_0 + 1) \tag{8}$$

$$\leq 4 * (\frac{x_0 + 1}{4} * log_2(\frac{x_0 + 1}{4})) + (x_0 + 1) \tag{9}$$

$$= (x_0 + 1) * 4 * (log_2(x_0 + 1) - log_2(4)) + x_0 + 1 \tag{10}$$

$$= (x_0 + 1) * 4 * (log_2(x_0 + 1) - 2) + x_0 + 1 \tag{11}$$

$$= (x_0 + 1) * (4 * log_2(x_0 + 1) - 8) + x_0 + 1 \tag{12}$$

$$= 4 * x_0 * log_2(x_0 + 1) - 8 * x_0 + 4 * log_2(x_0 + 1) - 8 + x_0 + 1 \tag{13}$$

$$= 4 * x_0 * log_2(x_0 + 1) + 4 * log_2(x_0 + 1) - 7 * x_0 - 7 \tag{14}$$

If we cast out the non-dominant terms:

$$\approx 4 * x_0 * log_2(x_0 + 1) \tag{15}$$

From Equation 15, we can conclude that $T(n) \in O(nlog(n))$

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: *There's two ways to approach this problem that immediately jump out to me. The first way is to sort this list using mergesort (a divide and conquer approach) and simply return the pair of elements at either the beginning or end of the list (depending on how we sort). The other method is to recursively divide and conquer the problem. To do so, I divide each input until I am left with a pair of inputs. From there we can do a series of comparisons (max 2 per return) to determine which a, b we are returning. The reason I don't have all 12 possible permutations of selections (4P2) is because we never return only a pair of b's nor an ordering where a "b" is above an "a". This eliminates the 2 combinations attributed to returning only b's and another 4 attributed to returning bx, ax. Further, we eliminate another two combinations where we return a1, b2 and a2, b1 which are impossible to reach because we have to return a2 and a1, respectively because the respective a-terms are larger than the b-terms.*

```
funky(lowIndex, highIndex, list):
    if(highIndex - lowIndex == 1):
            if(list[highIndex] > list[lowIndex]):
                return list[highIndex], list[lowIndex]
            else:
                return list[lowIndex], list[highIndex]
        elsif:
            midIndex = ((highIndex - lowIndex) / 2) + lowIndex
            a1, b1 = funky(lowIndex, midIndex, list)
            a2, b2 = funky(midIndex + 1, highIndex, list)

            if (a1 > a2):
                if(a2 > b1):
                    return a1, a2
                else:
                    return a1, b1
            else:
                if (a1 > b2):
                    return a2, a1
                else:
                    return a2, b2
```

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: *This problem is most similar to a in-degree ranking problem. If we consider this problem as a graph problem where we are trying to return the node with the largest in-degree (greatest number of vertices pointed towards it), then this problem becomes simpler as we are able to abstract away some of the nuances of the problem. Although there are a couple of ways of approaching this problem, including counting in-degree/out-degree of each node and the like, the method that is simplest is to traverse the graph until we end up in a node with a out-degree of 0. The superstar knows no one at the conference. Therefore, we can simply traverse the graph from a random node, and mark previously visited nodes so that we do not enter them again. Per iteration, per jump from node to node, we eliminate another node from the set of possible superstar candidates. This enables us to have a worst case Ø(n) algorithm for determining the superstar. My implementation will assume that the nodes are stored in an array with an associated linkedlist that stores the corresponding exit edges.*

Instead, I will proceed along the route prescribed by the course. There are two "worst" cases that I have to consider. The first case is that everyone knows everyone (except for the superstar who knows no one). The other worst case is that everyone knows only the superstar. Depending on the approach, either worst case can arise. The key is to select an approach that can handle/avoid both. I believe the approach I outlined above, albeit in a linked-list form will be the most efficient. This approach will feature an algorithm that will select any element as a superstar candidate and then check the subsequent element for "knowing of" eliminating at least one element per comparison.

```
superstar(linkedlist nodes){
    node* i = nodes->node;
    node* j = i->next;
    while(i->next != null){
        if ( knows(i, j) ){ \\eliminate i from the superstar candidacy
            i = j;
            j = j->next;
            nodes->node = nodes->node->next;
        } else { \\ eliminate j from superstar candidacy
            i->next = i->next->next;
            j = j->next;
        }
    }

    return i;

}
```

When talking to Professor Hott, I realized that there is a solution that employs a recursive divide and conquer method using principles similar to the principles applied above. This method would initially divide the list into pairs of attendees. At the bottom-most level, we would perform the knows(a_j, a_i) on a pair of elements, returning the only attendee that could still be a super-star candidate, and then performing that operation on each pair-of-pairs until only one superstar candidate remains at the top level, giving you the superstar. But the above solution is a lot prettier and pretty much guaranteed to be the most efficient so I'm going to leave it up.

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.