CS161 Summer 2022 Problem Session 1

It's Friday! And we (probably) have no power! Might as well distract ourselves with some powerful practice problems.

You can work at your own pace (there are more problems here than can be finished in an hour!), and either solo or with a group. We will circulate so that we can listen to your thoughts and offer help. The solutions are also given throughout, but we recommend that you check a solution only once you have made a complete and earnest attempt to solve the problem... you will learn (and retain) much more that way!

The Problems

Problem 1: Asymptotic Families

This is a very hard problem and a big leap past what we covered in lecture. Consider starting it, then working on some of the others, then coming back.

Arrange the following functions into "families" such that within a family, each function is $\Theta()$ of every other function in the family. Then arrange the families from bottom to top so that any member of a family is $O()$ of every member of every family below it.

For example, for the set $\log_2 n, \frac{n}{2}, n^2, 2n, \log_{22} n$, the list would look like:

- $\log_2 n, \log_{22} n$
- $\frac{n}{2}, 2n$
- n^2

Here's the list. (You do not need to try to attempt a formal Big-O proof for each relationship.

Rather, find informal ways of convincing yourself.)

$$4^n, 4^{\log_2 n}, n!, n \log 4n, n \log n \log \log n, 2^n, (n+4)!, \log_2 4, 4n \log n,$$
$$n^n, n^2, n^2 2^n, 2^{2n}, n \log \log n, (4n)!, \log_4 n, n \log n^4, \frac{4}{n}, \frac{n}{\log n}, (\log_4 n)^2$$

Some advice on how to proceed:

- Sometimes, two expressions will be the same except for one part, and then you can just compare those parts directly.
- Consider comparing expressions term-by-term. Is each term in one always greater than each term in the other, in a way that ends up being a non-constant?

This is quite challenging! You wouldn't need to do anything this grueling on an exam, but it's nice to work through something like this once.

- $\frac{4}{n}$ (admittedly it is hard to imagine an algorithm with running time that would behave like this, but terms like $\frac{1}{\epsilon}$, where ϵ is some bound on error, show up all the time in big-O expressions)
- $\log_2 4$ (a constant, 2)
- $\log_4 n$
- $(\log_4 n)^2$ – this beats $\log_4 n$ just as n^2 beats n
- $\frac{n}{\log n}$ – any polynomial beats any log term, so the n alone is enough to beat the log-only functions. What about the division by a log? Well, $\frac{n}{\log n}$ is to $\log n$ as n is to $\log^2 n$ (the square of $\log n$).
- $n \log \log n$
- $n \log 4n$ (this is the same as $n \log 4 + n \log n$), $4n \log n$, $n \log n^4$ (this is the same as $4n \log n$) – $\log n$ beats $\log \log n$ just as n beats $\log n$
- $n \log n \log \log n$ – as $n \to \infty$, both $\log n$ and $\log \log n$ are positive regardless of the base. Compared to either $n \log n$ or $n \log \log n$, $n \log n \log \log n$ has an extra positive, non-constant term. (This may seem like a silly function, but we did briefly see it show up as the running time of a multiplication algorithm.)
- $4^{\log_2 n}$ (this is equal to $(2^2)^{\log_2 n} = (2^{(\log_2 n)})^2 = n^2$), n^2
- 2^n – any exponential beats any polynomial
- $n^2 2^n$ – this shows up in some solutions to the Traveling Salesman Problem, which we will mention later in the course. It clearly beats 2^n, but loses to 4^n because the extra 2^n beats the extra n^2.
- $4^n, 2^{2n}$ – these are identical since $4^n = (2^2)^n = 2^{2 \cdot n}$, and 4^n beats 2^n since every term of the former dominates every term of the latter, and there can be arbitrarily many terms as $n \to \infty$
- $n!$ (any factorial beats any exponential, since as $n \to \infty$, the terms in the factorial product grow but the terms in the exponential product stay the same)
- $(n+4)!$ – this is $(n+4)(n+3)(n+2)(n+1)n! = \Theta(n^4 n!)$, so it dominates $\Theta(n!)$
- n^n – This is tricky. It beats $n!$ because each of the n terms in n^n is n, but the terms in $n!$ are $n, n-1, ..., 1$. It beats $(n+4)!$ because the extra $\Theta(n^4)$ does not make a difference – we can just increase n by 4. But...
- $(4n)!$ – this has 4 times as many terms as n^n, and $\frac{3}{4}$ of those are greater than n.

Problem 2: Divide and Struggle

Recall from Pre-Homework 1 that Indy likes to ask this interview question: Given a list L of n (not necessarily distinct) integers, determine whether there are two elements in the list that sum to exactly k, where k is some integer.

Indy barely remembers the Divide and Conquer paradigm from his algorithms class because the only things he needs to divide up these days are his competitors and his fat stacks of cash. But he thinks there might be a divide and conquer solution to his interview question. He has a 15 minute break between pitches to VCs, so he thinks about it.

(We will assume for convenience, as we often do, that n is an integer power of 2.)

(a) Here is Indy's first attempt at solving his own problem:

DetectPair(L, k):

- Base case: If there are only 2 elements in the list, return `TRUE` if the two elements sum to k, and `FALSE` otherwise.
- Otherwise: Split the list into two halves. Recursively call **DetectPair** on each half (keeping the same value of k). If at least one of those returns `TRUE`, return `TRUE`. Otherwise, return `FALSE`.

What do you think the big-O running time of **DetectPair** is? And is it correct?

(b) Here is Indy's second attempt at solving his problem, after he has had a 2 minute micronap and $\Omega(n)$ cans of Red Bull:

DetectPair2.0(L, k):

- Base case: If there are only 2 elements in the list, return `TRUE` if the two elements sum to k, and `FALSE` otherwise.
- Otherwise:
  - Split the list into two halves.
  - Recursively call **DetectPair2.0** on each half (keeping the same value of k).
  - If at least one of those calls returns `TRUE`, return `TRUE`.
  - Otherwise, for each element e_i in the left half, calculate $k - e_i$. Iterate through the right half to see if one of those elements is $k - e_i$. If so, return `TRUE`.
  - Finally, if we get this far, return `FALSE`.

What do you think the big-O running time of **DetectPair2.0** is? (Assume that each numerical calculation takes constant time.) And is it correct?

(a) The bad news about **DetectPair** is that it is wrong. To quote the relevant part of the above: the only pairs of elements that are ever actually checked are the first and second, and the third and fourth, and so on. So if, e.g., the only duplicated elements are the first and third, this algorithm will incorrectly return `FALSE`.

The good news about **DetectPair** is that it is fast: it runs in $O(n)$ time. At the level just above the leaves, there are $\frac{n}{2}$ subproblems (comparing two leaves), each of which takes constant time. Then the next level up processes $\frac{n}{4}$ pairs of two Boolean results (with each such comparison also taking constant time), and so on; the total work is $\frac{n}{2} \cdot 1 + \frac{n}{4} \cdot 1 + ...$, which is bounded above by n.[a]

(b) The good news about **DetectPair2.0** is that it is correct, because it ends up checking every possible pair of indices. That is, for any index, it is compared with one index at the level above the leaves, and with two other indices at the level above that, and so on... at the root level, it is compared with the remaining $\frac{n}{2}$ indices!

The bad news is that since there are $\binom{n}{2} = \frac{n(n-1)}{2} = \Theta(n^2)$ pairs of indices, this algorithm is $\Omega(n^2)$. (Using Problem 3 on Homework 1, you can see that it is $O(n^2)$.)

[a] Strictly speaking, we haven't talked about the work to create the subproblems, but this is also constant – the places to split the list can be calculated in constant time (so this work is just like the work of the comparisons), and there is no need to make a bunch of copies of parts of the list.

Problem 3: ThergeSort?

Waverly is wondering why MergeSort breaks the list into exactly two parts of size $\frac{n}{2}$ each. She considers breaking it into k parts of size $\frac{n}{k}$ each, for some integer $k > 2$. (For simplicity, you don't need to worry about parts being of uneven size – that is, you can assume that n is always some power of k.) This all looks good until Waverly considers the Merge step... it's not immediately clear how to make the same idea work with more than two lists.

(a) Suppose that $k = 3$. Can the Merge step be easily adapted to fit Waverly's idea while keeping the linear time guarantee? If so, show how it can be done. If not, explain where the difficulty arises.

(b) Now suppose that k is an arbitrary variable, not a constant, so that $O(nk)$, for example, is not the same thing as $O(n)$. Can the Merge step be easily adapted to fit Waverly's idea while keeping the linear time guarantee? If so, show how it can be done. If not, explain where the difficulty arises.

(a) We can easily adapt Merge to work with three lists. We just have three pointers instead of two. The major difference is that now whenever we find the smallest element to add next, we have to check three lists. However, only entails doing two comparisons (check the first candidate against the second, then check the third candidate against the smallest of the first two). Then the same argument from lecture holds – we advance each of the three pointers $\frac{n}{3}$ steps, and the comparisons at each step take constant time, so the procedure is still linear in n.

(b) Now we have a problem, because when we have to look at the pointers of k lists to choose the next smallest element, this takes $O(k)$ time. That is, when the number of lists can get arbitrarily big, we can no longer handwave away that number as a constant. Later in the quarter, we will see a way to deal with this somewhat by using a priority queue, but even then, we cannot make that check constant-time. So Waverly's idea does not hold water.

Problem 4: Inversion-Counting

Consider a list L of n integers. An *inversion* is a pair of indices i, j of the list, with $i < j$, such that $L[i] > L[j]$. That is, the elements at these indices are in a sense "out of order" relative to each other.

For example, in the list $[3, 1, 4, 1, 5, 9, 2]$, there are seven inversions: Indices 0 and 1 (3 vs. 1), Indices 0 and 3 (3 vs. 1), Indices 0 and 6 (3 vs. 2), Indices 2 and 3 (4 vs. 1), Indices 2 and 6 (4 vs. 2), Indices 4 and 6 (5 vs. 2), Indices 5 and 6 (9 vs. 2).

(a) Give an exact (not big-O) expression for the maximum number of inversions that can be present in a list of length n (with $n \geq 2$). Justify your answer mathematically.

(b) Describe an $O(n^2)$ algorithm to count the number of inversions in such a list. (Think like Brutus!)

(c) Now let's try to do better! Notice that the number of inversions is directly related to how close to sorted the list is. This suggests that maybe we can adapt our MergeSort algorithm to count inversions as we're sorting – specifically, by modifying the Merge step.

Without changing the backbone of how MergeSort works, find a way to add some additional variables/calculations to the Merge step such that the procedure also returns the number of inversions in the list.

(d) Informally justify that the runtime of this modified procedure is still $O(n \log n)$, just as for MergeSort.

(e) Suppose we wanted to instead print all the pairs of inversions, instead of just counting them. Explain why no $O(n \log n)$ algorithm for that problem exists.

(a) The worst case occurs when *every* pair of indices is an inversion, which happens when the entire list is in reverse sorted order. There are $\binom{n}{2} = \frac{n(n-1)}{2}$ distinct pairs of indices, so this is the maximum number of inversions.

(b) We can just check every pair of indices (e.g., with a double for loop) for an inversion, and keep track of the total number of inversions.

(c) When we perform a Merge step, we merge two already-sorted lists, so there are no inversions *within* either of the lists. So we only need to count the number of pairs of inversions that *span* the two lists – i.e., we can compare all indices in the left list with all indices in the right list. Wait a minute – isn't this what Indy tried to do in Problem 2(b)? Well, here we have a trick up our sleeve.

Suppose that our left list is $[1, 4, 6, 7]$, and our right list is $[2, 3, 5, 8]$. We start our pointers off at 1 and 2, respectively, and then we choose 1 as our first new list element and advance the first pointer to 4. Then we choose 2 as our next new list element.

At the time we choose 2, we note that it is actually smaller than **every** remaining element in the left list! Why? Because if we are choosing 2 as the next smallest element for our new list, we must have already chosen everything smaller from the left list!

Therefore we can add three inversions to our total – just the remaining length of the left list. (We don't need to go through this list to find out how big it is – we know how big it was to start, and we know where the pointer is now, so we can calculate the remaining length directly.)

Continuing in this fashion, we find that 3 also gives us three inversions, 5 gives us two, and 8 gives us none (after all, it is the biggest element overall!) We total all these up and pass the total up the tree, and the final answer is the total of all the inversions counted by all the Merge steps.

A couple lingering questions:

- *How do we know this doesn't double-count?* Convince yourself that every pair of distinct indices is on opposite sides of **exactly one** Merge step.
- *What if there are ties?* Then we have to be careful to choose the left list's value first whenever the two lists have a tie. (We only count inversions when processing an element from the right list, so by the time we do this, all duplicates will have been removed from the left list. And duplicates in the right list will be handled separately, which is what we want.)

(d) The new procedure involves keeping track of a new variable and passing it up the tree, but all of these operations take constant time – no worse than the constant work we already do at each step of a Merge operation.

(e) In part (a), we saw that there can be $\Theta(n^2)$ inversions. Since printing each one takes some time, any such algorithm is $\Omega(n^2)$.

(Answers to all three of these remaining problems are on the next pages.)

Problem 5: Polynomial Big-O

We have informally seen that only the highest-degree term of a polynomial function matters for Big-O. That is, $0.01n^3+9999n^2-7$ is $O(n^3)$. In this problem, we'll make that rigorous.

Suppose we have a function $f(n) = \sum_{i=0}^{d} a_i n^i$, with $a_0, ..., a_d$ being constants with a_d nonnegative. (It wouldn't really be useful in the context of algorithmic analysis for this leading coefficient to be negative!) Note that each of the other coefficients could be positive, negative, or zero. (In the example in the first paragraph, we have $a_0 = -7, a_1 = 0, a_2 = 9999, a_3 = 0.01$.)

Prove that $f(n)$ is $O(n^d)$. The tricky part here is figuring out which values of c and n_0 to choose...

Problem 6: Cramped Aisle

There's an aisle of theater seats numbered 1 through n. Seat 1 is against a wall, so the only entrance to the aisle is by seat n. Therefore, when someone enters the aisle and wants to get to some seat k, they must walk by seats n, $n-1$, ..., $k+1$, and anyone already sitting in those seats must stand up to let the new person go by. (Then they sit down again.)

You have a list of the order in which the n ticketholders (who are also numbered 1 through n, matching their seat numbers) will arrive. Describe an algorithm to calculate the total number of instances of standing up. (The same person might stand up multiple times, and all of those instances count separately.) Your algorithm must run in $O(n \log n)$ time.

Problem 7: A Big-O Surprise

It is natural to assume that *all* functions fall into some kind of hierarchy of families as in Problem 1, but this turns out not to be true! That is, it is possible to find functions $f(n), g(n)$ such that $f(n)$ is not $O(g(n))$, and $g(n)$ is not $O(f(n))$. Can you give an example of such a pair?

Answer to Problem 5

To show that $f(n) = O(n^d)$, we need to exhibit a c and n_0 such that for all $n \geq n_0$, $f(n) = \sum_{i=0}^{d} a_i n^i \leq c \cdot n^d$.

What value of n_0 should we choose? We know that as n increases, things will only get more favorable for our claim, since n^d will grow faster than lower-order terms. So let's choose $n_0 = 1$, where we will be at our weakest, and think about what value of c we'll need.

If we plug in 1 for every power of n in the polynomial, what do we end up with? The sum of all of the coefficients! So if we take c to be at least as large as the sum of all the coefficients? Then the bound certainly holds for $n = 1$.

What about for larger n? Well, because $n^i \leq n^d$ for all $i \leq d$, we have

$$f(n) = \sum_{i=0}^{d} a_i n^i \leq \sum_{i=0}^{d} c_i n^d \quad \text{s.t. } c_i \geq a_i \ \forall i \in [0, d]$$

$$\leq n^d \sum_{i=0}^{d} c_i = n^d \cdot c$$

(where we were able to move the n^d out of the summation because it has no dependency on i.) Therefore we have chosen a c and n_0 and demonstrated that they establish an upper bound, so the proof is complete.

Answer to Problem 6

Notice that if the theatergoers arrived in the order $1, 2, ..., n$, nobody would ever have to stand up. In fact, every instance in which person a has to stand up for person b corresponds to an inversion, i.e., a is numbered higher than b but arrives before b. Therefore the problem can be solved using our inversion-counting algorithm from Problem 4.

Answer to Problem 7

We saw an example of this on Pre-Homework 1: we can choose $f(n) = 1$ for odd n and n for even n, and $g(n) = 1$ for even n and n for odd n. Then no constant multiplier can make $g(n)$ upper-bound $f(n)$ on even n (since $g(n)$ is always 1 there), and no constant multiplier can make $f(n)$ upper-bound $g(n)$ on odd n (since $f(n)$ is always 1 there).

Another example pair (which I believe is the "classic" answer to this question) is $f(x) = \sin x, g(x) = \cos x$. The argument is similar to the above.