Note: Your TA may not get to all the problems. This is totally fine, the discussion worksheets are not designed to be finished in an hour. The discussion worksheet is also a resource you can use to practice, reinforce, and build upon concepts discussed in lecture, readings, and the homework.

1 Asymptotic Notation Intro

In this class, we care a lot about the runtime of algorithms. However, we don't care too much about concrete performance on small input sizes (most algorithms do well on small inputs). Instead we want to compare the *long-term (asymptotic)* growth of the runtimes.

Asymptotic Notation: We define the following notation for two functions $f(n), g(n) \geq 0$:

- $f(n) = \mathcal{O}(g(n))$ if there exists a $c > 0$ where after large enough n, $f(n) \leq c \cdot g(n)$. *(Asymptotically, f grows at most as much as g)*
- $f(n) = \Omega(g(n))$ if $g(n) = \mathcal{O}(f(n))$. *(Asymptotically, f grows at least as much as g)*
- $f(n) = \Theta(g(n))$ if $f(n) = \mathcal{O}(g(n))$ and $g(n) = \mathcal{O}(f(n))$. *(Asymptotically, f and g grow the same)*

If we compare this to the order on the numbers, $\mathcal{O}$ is a lot like $\leq$, Ω is a lot like $\geq$, and Θ is a lot like $=$ (except all are with regard to asymptotic behavior).

(a) For each pair of functions $f(n)$ and $g(n)$, state whether $f(n) = \mathcal{O}(g(n))$, $f(n) = \Omega(g(n))$, or $f(n) = \Theta(g(n))$. For example, for $f(n) = n^2$ and $g(n) = 2n^2 - n + 3$, write $f(n) = \Theta(g(n))$.

(i) $f(n) = n$ and $g(n) = n^2 - n$
Solution: n grows slower than n^2 so $f(n) = \mathcal{O}(g(n))$

(ii) $f(n) = n^2$ and $g(n) = n^2 + n$
Solution: We compare the largest terms in asymptotics so we get that these two functions grow at roughly the same rate. $f(n) = \Theta(g(n))$

(iii) $f(n) = 8n$ and $g(n) = n \log n$
Solution: As a rule, for any $c > 0$, $n^c = \mathcal{O}(n^c \log n)$. If we apply this here with $c = 1$, we get $f(n) = \mathcal{O}(g(n))$.
Formally,

$$\lim_{n\to\infty} \frac{8n}{n \log n} = \lim_{n\to\infty} \frac{8}{\log n} = 0$$

(iv) $f(n) = 2^n$ and $g(n) = n^2$
Solution: Polynomial functions grow slower than exponential functions. So, $f(n) = \Omega(g(n))$

(v) $f(n) = 3^n$ and $g(n) = 2^{2n}$
Solution: $f(n) = \mathcal{O}(g(n))$. $2^{2n} = 4^n$, and if $a < b$, $a^n = \mathcal{O}(b^n)$. So $3^n = \mathcal{O}(4^n)$.
Formally,

$$\lim_{n\to\infty} \frac{3^n}{2^{2n}} = \lim_{n\to\infty} \left(\frac{3}{4}\right)^n = 0$$

(b) For each of the following, state the order of growth using Θ notation, e.g. $f(n) = \Theta(n)$.

(i) $f(n) = 50$
Solution: $f(n) = \Theta(1)$

(ii) $f(n) = n^2 - 2n + 3$
Solution: $f(n) = \Theta(n^2)$

(iii) $f(n) = n + \cdots + 3 + 2 + 1$
Solution: $f(n) = \frac{n(n+1)}{2} = \Theta(n^2)$

(iv) $f(n) = n^{100} + 1.01^n$
Solution: $f(n) = \Theta(1.01^n)$

(v) $f(n) = n^{1.1} + n \log n$
Solution: $n^{1.1}$ grows more than $n \log n$, so $f(n) = \Theta(n^{1.1})$.
Formally, we can notice

$$\lim_{n\to\infty} \frac{n^{1.1}}{n \log n} = \lim_{n\to\infty} \frac{n^{0.1}}{\log n} = \lim_{n\to\infty} \frac{0.1n^{-0.9}}{\frac{1}{n}} = \lim_{n\to\infty} 0.1n^{0.1} = \infty$$

(vi) $f(n) = (g(n))^2$ where $g(n) = \sqrt{n} + 5$
Solution:

$$f(n) = (\sqrt{n} + 5)^2 = n + 10\sqrt{n} + 25$$
$$f(n) = \Theta(n)$$

Solution: In general, we can observe the following properties of $\mathcal{O}/\Theta/\Omega$ from this:

- If $d > c$, $n^c = \mathcal{O}(n^d)$, but $n^c \neq \Omega(n^d)$ (this is sort of saying that n^d grows strictly more than n^c).
- Asymptotic notation only cares about "highest-growing" terms. For example, $n^2 + n = \Omega(n^2)$.
- Asymptotic notation does not care about leading constants. For example $50n = \Theta(n)$.
- Any exponential with base > 1 grows more than any polynomial
- The base of the exponential matters. For example, $3^n = \mathcal{O}(4^n)$, but $3^n \neq \Omega(4^n)$.
- If $d > c$, $n^c \log n = \mathcal{O}(n^d)$.

2 Asymptotics and Limits

If we would like to prove asymptotic relations instead of just using them, we can use limits.

Asymptotic Limit Rules: If $f(n), g(n) \geq 0$:

- If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, then $f(n) = \mathcal{O}(g(n))$.
- If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$, for some $c > 0$, then $f(n) = \Theta(g(n))$.
- If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$, then $f(n) = \Omega(g(n))$.

Note that these are all sufficient conditions involving limits, and are not true definitions of $\mathcal{O}$, Θ, and Ω.

(a) Prove that $n^3 = \mathcal{O}(n^4)$.

Solution:

$$\lim_{n\to\infty} \frac{n^3}{n^4} = \lim_{n\to\infty} \frac{1}{n} = 0$$

So $f(n) = \mathcal{O}(g(n))$

(b) Find an $f(n), g(n) \geq 0$ such that $f(n) = \mathcal{O}(g(n))$, yet $\lim_{n\to\infty} \frac{f(n)}{g(n)} \neq 0$.

Solution: Let $f(n) = 3n$ and $g(n) = 5n$. Then $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \frac{3}{5}$, meaning that $f(n) = \Theta(g(n))$. However, it's still true in this case that $f(n) = \mathcal{O}(g(n))$ (just by the definition of Θ).

(c) Prove that for any $c > 0$, we have $\log n = \mathcal{O}(n^c)$.

Hint: Use L'Hôpital's rule: If $\lim_{n\to\infty} f(n) = \lim_{n\to\infty} g(n) = \infty$, then $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)}$ (if the RHS exists)

Solution: By L'Hôpital's rule,

$$\lim_{n\to\infty} \frac{\log n}{n^c} = \lim_{n\to\infty} \frac{n^{-1}}{cn^{c-1}} = \lim_{n\to\infty} \frac{1}{cn^c} = 0$$

Therefore, $\log n = \mathcal{O}(n^c)$.

(d) Find an $f(n), g(n) \geq 0$ such that $f(n) = \mathcal{O}(g(n))$, yet $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ does not exist. In this case, you would be unable to use limits to prove $f(n) = \mathcal{O}(g(n))$.

Solution: Let $f(x) = x(\sin x + 1)$ and $g(x) = x$. As $\sin x + 1 \leq 2$, we have that $f(x) \leq 2 \cdot g(x)$ for $x \geq 0$, so $f(x) = \mathcal{O}(g(x))$.

However, if we attempt to evaluate the limit, $\lim_{x\to\infty} \frac{x(\sin x+1)}{x} = \lim_{x\to\infty} \sin x + 1$, which does not exist (sin oscillates forever).

3 Runtime and Correctness of Mergesort

In general, this class is about design, correctness, and performance of algorithms. Consider the following algorithm called *Mergesort*, which you should hopefully recognize from 61B:

```
function MERGE(A[1,...,n], B[1,...,m])
    i, j ← 1
    C ← empty array of length n + m
    while i ≤ n or j ≤ m do
        if i ≤ n and (j > m or A[i] < B[j]) then
            C[i + j − 1] ← A[i]
            i ← i + 1
        else
            C[i + j − 1] ← B[j]
            j ← j + 1
    return C
function MERGESORT(A[1,...,n])
    if n ≤ 1 then return A
    mid ← ⌊n/2⌋
    L ← MERGESORT(A[1,...,mid])
    R ← MERGESORT(A[mid + 1,...,n])
    return MERGE(L, R)
```

Recall that MERGESORT takes an arbitrary array and returns a sorted copy of that array. It turns out that MERGESORT is asymptotically optimal at performing this task (however, other sorts like Quicksort are often used in practice).

(a) Let $T(n)$ represent the number of operations MERGESORT performs given a array of length n. Find a base case and recurrence for $T(n)$, use asymptotic notation.

Solution:

For the base case, we simply have $T(1) = 1$ (almost nothing is done). For the recursive case $n \geq 2$, we get $T(n) = 2T(n/2) + \mathcal{O}(n)$.

On a high level, MERGE takes two pointers through A and B respectively, and keeps adding the next-smallest element from A or B. We notice that MERGE looks at each element from A or B only once. So given arrays of length m and n, MERGE takes $\mathcal{O}(m+n)$. Note that the $\mathcal{O}(n+m)$ is important here, the time is not just $n+m$ (there's more than one array access for each element in A or B, for example).

A call to MERGESORT involves two recursive calls to pieces of size $n/2$ and one call to MERGE given both halves, so the runtime here is

$$T(n) = \underbrace{2T(n/2)}_{\text{two recursive calls to MERGESORT}} + \underbrace{\mathcal{O}(n)}_{\text{call to MERGE}}$$

(b) Solve this recurrence. What asymptotic runtime do you get?

Solution:

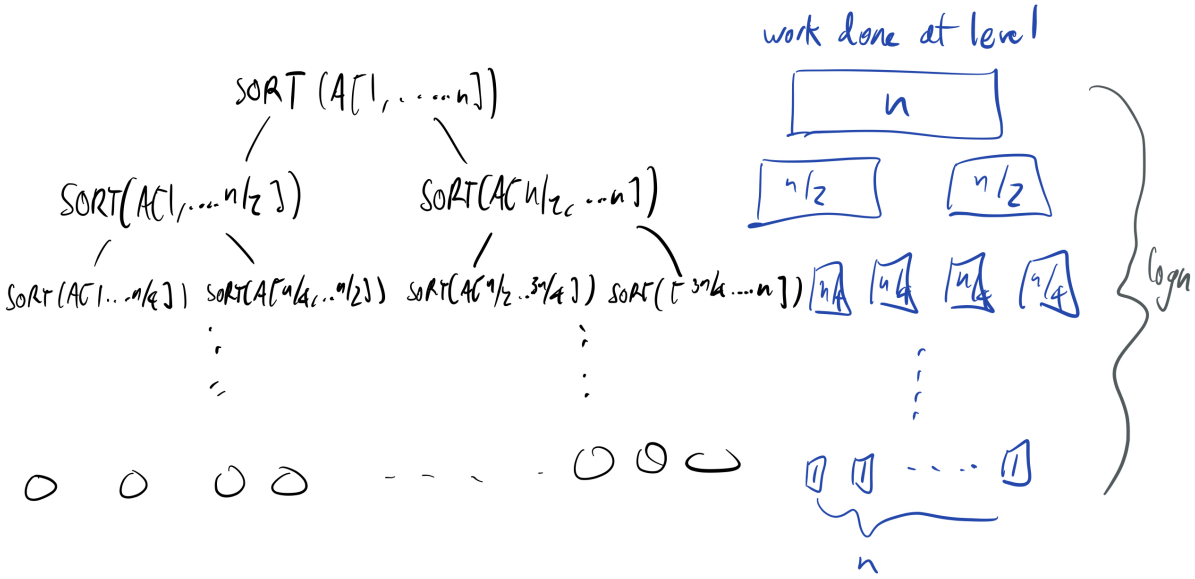
$T(n) = \mathcal{O}(n \log n)$

A general strategy to solve these is to consider what happens during repeated expansion:

$$
\begin{aligned}
&T(n)\\
&T(n) = 2T(n/2) + n\\
&T(n) = 4T(n/4) + n + n\\
&T(n) = 8T(n/8) + n + n + n\\
&\vdots\\
&T(n) = 2^{\log n} T\left(\frac{n}{2^{\log n}}\right) + n(\log n) = n \log n
\end{aligned}
$$

Each time we expand we get an extra n added to the back, and each time we expand the input size to $T()$ halves. But we can expand only $\log n$ times until we get $T(1)$ and the expansion ends.

You can also visualize this as a full binary tree with $\log n$ levels, with n work done at each level, and each node representing the work done to sort that part of the array.



Question: This doesn't consider the fact that $T(n) = 2T(n/2) + \mathcal{O}(n)$ only means that there is some $c > 0$ where for large enough n, $T(n) \leq 2T(n/2) + c \cdot n$. We assumed $T(n) = 2T(n/2) + n$ to make the analysis simpler. How can you modify the analysis to account for this formally?

Note: This is not the only way to solve recurrences like these, but it is a good way to solve recurrences in general. We will soon talk about an important tool called *The Master Theorem*, which does the geometric series expansion for you and lets you solve recurrences with a simple rule.

(c) Consider the correctness of MERGE. What is the desired property of C once MERGE completes? What is required of the arguments to MERGE for this to happen?

Solution:

We desire C to be sorted, and to contain all elements from A and B. MERGE requires that A and B are individually sorted.

Question: This falls short of a full proof of correctness for MERGE. How would you fully proof correctness for merge? (*Hint:* a good place to start would be to think about 'invariants', or properties that hold for C as i and j increase)

(d) Using the property you found for MERGE, show that MERGESORT is correct.

Solution:

We establish by induction. Let

$P(n)$: MERGESORT$(A[1, \ldots, n])$ is correct for any array A of length n

For the base case, $P(1)$ is true because a length-1 array is already sorted, and that is what we return.

Now assume for a particular n that $P(k)$ holds for all $k < n$ (this is strong induction). L and R are then guaranteed to be sorted, then by part (c) their merge will be sorted. As L and R contain all the elements of the array, we return a sorted copy of the array.