Note: Your TA probably will not cover all the problems. This is totally fine, the discussion worksheets are not designed to be finished in an hour. They are deliberately made long so they can serve as a resource you can use to practice, reinforce, and build upon concepts discussed in lecture, readings, and the homework.

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: The following are definitions for $\mathcal{O}(\cdot), \Theta(\cdot)$, and $\Omega(\cdot)$:

- $f(n) = \mathcal{O}(g(n))$ 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 these definitions 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).

1 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 $\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$, 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 $\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0$, 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. (you should check on your own that these statements are correct!)

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

Hint: think about oscillating functions!

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).

2 Asymptotic Complexity Comparisons

(a) Order the following functions so that for all i, j, if f_i comes before f_j in the order then $f_i = O(f_j)$. Do not justify your answers.

- $f_1(n) = 3^n$
- $f_2(n) = n^{\frac{1}{3}}$
- $f_3(n) = 12$
- $f_4(n) = 2^{\log_2 n}$
- $f_5(n) = \sqrt{n}$
- $f_6(n) = 2^n$
- $f_7(n) = \log_2 n$
- $f_8(n) = 2^{\sqrt{n}}$
- $f_9(n) = n^3$

As an answer you may just write the functions as a list, e.g. $f_8, f_9, f_1, \ldots$

Solution: $f_3, f_7, f_2, f_5, f_4, f_9, f_8, f_6, f_1$

(b) In each of the following, indicate whether $f = O(g)$, $f = \Omega(g)$, or both (in which case $f = \Theta(g)$). ***Briefly*** justify each of your answers. Recall that in terms of asymptotic growth rate, logarithmic $<$ polynomial $<$ exponential.

	$f(n)$	$g(n)$
(i)	$\log_3 n$	$\log_4(n)$
(ii)	$n \log(n^4)$	$n^2 \log(n^3)$
(iii)	$\sqrt{n}$	$(\log n)^3$
(iv)	$n + \log n$	$n + (\log n)^2$

Solution:

(i) $f = \Theta(g)$; using the log change of base formula, $\frac{\log n}{\log 3}$ and $\frac{\log n}{\log 4}$ differ only by a constant factor.

(ii) $f = O(g)$; $f(n) = 4n \log(n)$ and $g(n) = 3n^2 \log(n)$, and the polynomial in g has the higher degree.

(iii) $f = \Omega(g)$; any polynomial dominates a product of logs. We can also obtain this result via

the limit proof below:

$$\begin{aligned}
\lim_{n\to\infty} \frac{f(n)}{g(n)} &= \lim_{n\to\infty} \frac{\sqrt{n}}{(\log n)^3} \\
&= \lim_{n\to\infty} \frac{\frac{1}{2\sqrt{n}}}{3(\log n)^2 \cdot \frac{1}{n}} && \text{[L'Hôpital's rule]} \\
&= \lim_{n\to\infty} \frac{\sqrt{n}}{6(\log n)^2} \\
&= \lim_{n\to\infty} \frac{\sqrt{n}}{24 \log n} && \text{[L'Hôpital's rule again]} \\
&= \lim_{n\to\infty} \frac{\sqrt{n}}{48} && \text{[L'Hôpital's rule one more time]} \\
&= \infty
\end{aligned}$$

(iv) $f = \Theta(g)$; Both f and g grow as $\Theta(n)$ because the linear term dominates the other. We can also obtain this result via the limit proof below:

$$\begin{aligned}
\lim_{n\to\infty} \frac{f(n)}{g(n)} &= \lim_{n\to\infty} \frac{n + \log n}{n + (\log n)^2} \\
&= \lim_{n\to\infty} \frac{1 + \frac{1}{n}}{1 + \frac{2 \log n}{n}} && \text{[L'Hôpital's rule]} \\
&= 1
\end{aligned}$$

3 Recurrence Relations

Master Theorem: If the recurrence relation is of the form $T(n) = aT\left(\frac{n}{b}\right) + O(n^d)$ for some constants $a > 0$, $b > 1$, and $d \geq 0$, then

$$T(n) = \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

Remember that if the recurrence relation is *not* in the form $T(n) = aT\left(\frac{n}{b}\right) + O(n^d)$, you can't use Master Theorem! If this is the case, then you can try the following strategies:

1. **"Unraveling" the recurrence relation.**

   Essentially, we try to recursively keep on plugging in the smaller subproblems (e.g. $T(n/2)$ or $T(n-1)$) to $T(n)$ to try to find a pattern or simply directly compute the entire expression.

2. **Draw a tree!**

   We use a tree representation to count the total number of calls on each subproblem, doing so by summing up the work per level.

3. **Squeeze**

   For certain recurrence relations where we can't directly compute the solution, we can indirectly solve it by computing upper and lower bounds for the runtime based on known recurrence relations. Specifically, if the upper and lower bounds have the same asymptotic growth, then we have our answer!

4. **Squeeze + Guess & Check**

   If we try using the previous method and can't end up with asymptotically equivalent upper/lower bounds, then we resort to guess-and-checking reasonable runtimes between the bounds to arrive at the solution (look up "The Substitution Method for Solving Recurrences – Brilliant" to see how to do this). For the purposes of this class, if you ever have to resort to using this method, the expression for $T(n)$ will always look "nice."

There are a lot more strategies that are out-of-scope for this class, and if you're curious we highly recommend you to read about them in the following link: `http://www.iiitdm.ac.in/old/Faculty_Teaching/Sadagopan/pdf/DAA/recurrence-relations-V3.pdf`.

Solve the following recurrence relations, assuming base cases $T(0) = T(1) = 1$:

(a) $T(n) = 2 \cdot T(n/2) + O(n)$

Solution: We can use the Master Theorem here! Noting that $a = 2, b = 2$, and $d = 1$, we have that $\log_b a = \log_2(2) = 1 = d$. Thus, via the Master Theorem, we have

$$T(n) = O(n^d \log n) = O(n \log n)$$

(b) $T(n) = T(n-1) + n$

Solution: Since we can't use Master Theorem here, we use the "unravelling" strategy as follows:

$$\begin{aligned}
T(n) &= T(n-1) + n \\
&= (T(n-2) + (n-1)) + n \\
&= ((T(n-3) + (n-2)) + (n-1)) + n \\
&= \cdots \text{Unravelling} \cdots \\
&= T(1) + 2 + 3 + \cdots (n-2) + (n-1) + n \\
&= 1 + 2 + 3 + \cdots + (n-2) + (n-1) + n \\
&= \sum_{i=1}^{n} i \\
&= \frac{n(n+1)}{2} \\
&= O(n^2)
\end{aligned}$$

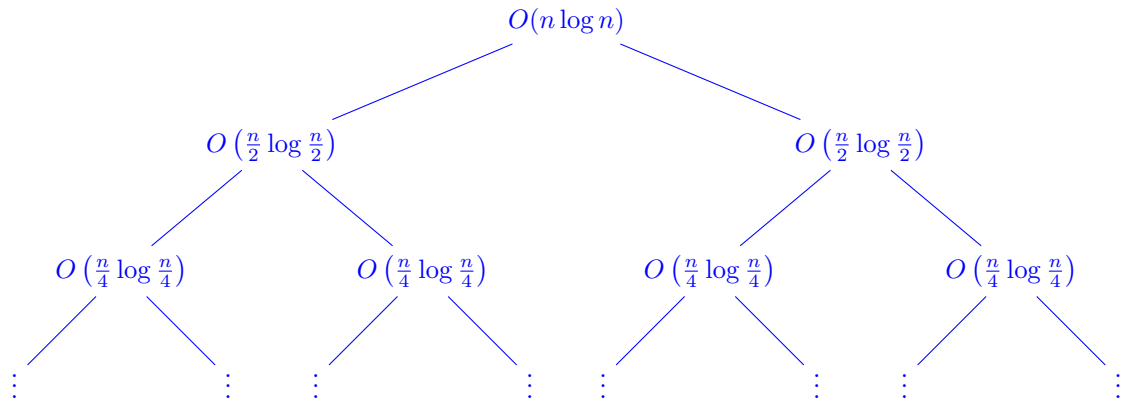
(c) $T(n) = 3 \cdot T(n-2) + 5$

Solution: More unravelling! Here we go again:

$$\begin{aligned}
T(n) &= 3T(n-2) + 5 \\
&= 3^2 T(n-4) + 5 \cdot 3 + 5 \\
&= 3^3 T(n-6) + 5 \cdot 3^2 + 5 \cdot 3 + 5 \\
&= 3^4 T(n-8) + 5 \cdot 3^3 + 5 \cdot 3^2 + 5 \cdot 3 + 5 \\
&= \cdots \\
&= 3^{\lfloor n/2 \rfloor} T(n \bmod 2) + 5 \cdot 3^{\lfloor n/2 \rfloor - 1} + 5 \cdot 3^{\lfloor n/2 \rfloor - 2} + \cdots + 5 \cdot 3^2 + 5 \cdot 3 + 5 \\
&= 1 \cdot 3^{\lfloor n/2 \rfloor} + 5 \cdot 3^{\lfloor n/2 \rfloor - 1} + 5 \cdot 3^{\lfloor n/2 \rfloor - 2} + \cdots + 5 \cdot 3^2 + 5 \cdot 3 + 5 \\
&= 3^{\lfloor n/2 \rfloor} + \frac{5 \cdot (3^{\lfloor n/2 \rfloor} - 1)}{3 - 1} \\
&= \frac{7}{2} \cdot 3^{\lfloor n/2 \rfloor} - \frac{5}{2} \\
&= O(3^{n/2})
\end{aligned}$$

(d) $T(n) = 2 \cdot T(n/2) + O(n \log n)$

Solution: We use the tree drawing technique here. We draw the following recursion tree, where the nodes represent the work done by a recursive call:



Summing up all the levels, we get

$$\begin{aligned}
T(n) &= O(n \log n) + O\left(n \log\left(\frac{n}{2}\right)\right) + O\left(n \log\left(\frac{n}{4}\right)\right) + \cdots + O(1) \\
&= O\left(\sum_{i=0}^{\lfloor \log n \rfloor} n \log\left(\frac{n}{2^i}\right)\right) \\
&= O\left(\sum_{i=0}^{\lfloor \log n \rfloor} n \left(\log n - i\right)\right) \\
&= O\left(n\left(\log^2 n - \sum_{i=0}^{\lfloor \log n \rfloor} i\right)\right) \\
&= O\left(n\left(\log^2 n - \frac{1}{2}\log^2 n\right)\right) \\
&= O(n \log^2 n)
\end{aligned}$$

4 Find the valley

You are given an array A of integers of length N. A has the following property: it is decreasing until element j, at which point it is increasing. In other words, there is some j such that if $i < j$ we have $A[i] > A[i+1]$ and if $i \geq j$ we have $A[i] < A[i+1]$. Assuming you do not already know j, give an efficient algorithm to find j.

For simplicity, you may assume that N is a power of 2.

Solution:

(i) **Main idea** If A has 1 element, we just output 1. Otherwise, we check if $A[N/2] < A[N/2+1]$. If yes, then we recursely call our algorithm on the subarray $A[1 : N/2]$. Otherwise, we recursively call our algorithm on the subarray $A[N/2+1 : N]$ (and add $N/2$ to the answer).

(ii) **Proof of correctness** The algorithm is correct if A has size 1, since the only possible value for j is 1. If $A[N/2] < A[N/2+1]$, then we know $j \leq N/2$, so j is contained within the subarray we recurse on, and we can inductively assume the recursive call finds the correct index. Otherwise, we know $j > N/2$, and we again have that j is contained within the subarray we recurse on, and again can inductively assume the recursive call finds the correct index (prior to adding $N/2$).

(iii) **Running time analysis** The algorithm takes $O(1)$ time to compare $A[N/2], A[N/2+1]$. After each comparison we halve the problem size, so we go from a length N array to length 1 in $\log N$ recursive calls. So the overall runtime is $O(\log N)$.

5 (OPTIONAL) Hadamard matrices

The Hadamard matrices $H_0, H_1, H_2, \ldots$ are defined as follows:

- H_0 is the 1×1 matrix $[1]$
- For $k > 0$, H_k is the $2^k \times 2^k$ matrix

$$H_k = \left[\begin{array}{c|c} H_{k-1} & H_{k-1} \\ \hline H_{k-1} & -H_{k-1} \end{array}\right]$$

(a) Write down the Hadamard matrices H_0, H_1, and H_2.

Solution: $H_0 = 1$

$$H_1 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

$$H_2 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}$$

(b) Compute the matrix-vector product $H_2 \cdot v$ where H_2 is the Hadamard matrix you found above, and

$$v = \begin{bmatrix} 1 \\ -1 \\ -1 \\ 1 \end{bmatrix}$$

Note that since H_2 is a 4×4 matrix, and the vector has length 4, the result will be a vector of length 4.

Solution:

$$H_2 v = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \\ -1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 4 \end{bmatrix}$$

(c) Now, we will compute another quantity. Take v_1 and v_2 to be the top and bottom halves of v respectively. Therefore, we have that

$$v_1 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}, v_2 = \begin{bmatrix} -1 \\ 1 \end{bmatrix}, v = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}$$

Compute $u_1 = H_1(v_1 + v_2)$ and $u_2 = H_1(v_1 - v_2)$ to get two vectors of length 2. Stack u_1 above u_2 to get a vector u of length 4. What do you notice about u?

Solution:

$$H_1(v_1 + v_2) = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$H_1(v_1 - v_2) = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 2 \\ -2 \end{bmatrix} = \begin{bmatrix} 0 \\ 4 \end{bmatrix}$$

We notice that $u = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 4 \end{bmatrix} = H_2 v$

(d) Suppose that

$$v = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}$$

is a column vector of length $n = 2^k$. v_1 and v_2 are the top and bottom half of the vector, respectively. Therefore, they are each vectors of length $\frac{n}{2} = 2^{k-1}$. Write the matrix-vector product $H_k v$ in terms of H_{k-1}, v_1, and v_2 (note that H_{k-1} is a matrix of dimension $\frac{n}{2} \times \frac{n}{2}$, or $2^{k-1} \times 2^{k-1}$). Since H_k is a $n \times n$ matrix, and v is a vector of length n, the result will be a vector of length n.

Solution: $H_k v = \begin{bmatrix} H_{k-1}v_1 + H_{k-1}v_2 \\ H_{k-1}v_1 - H_{k-1}v_2 \end{bmatrix} = \begin{bmatrix} H_{k-1}(v_1 + v_2) \\ H_{k-1}(v_1 - v_2) \end{bmatrix}$

(e) Use your results from (c) to come up with a divide-and-conquer algorithm to calculate the matrix-vector product $H_k v$, and show that it can be calculated using $O(n \log n)$ operations. Assume that all the numbers involved are small enough that basic arithmetic operations like addition and multiplication take unit time. You do not need to prove correctness.

Solution: Compute the 2 subproblems described in part (d), and combine them as described in part (c). Let $T(n)$ represent the number of operations taken to find $H_k v$. We need to find the vectors $v_1 + v_2$ and $v_1 - v_2$, which takes $O(n)$ operations. And we need to find the matrix-vector products $H_{k-1}(v_1 + v_2)$ and $H_{k-1}(v_1 - v_2)$, which take $T(\frac{n}{2})$ number of operations. So, the recurrence relation for the runtime is:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

Using the master theorem, this give us $T(n) = O(n \log n)$.