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

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.

(iv) $f = \Theta(g)$; Both f and g grow as $\Theta(n)$ because the linear term dominates the other.

3 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 $T(n) = O(n \log n)$.

4 Monotone matrices

A m-by-n matrix A is *monotone* if $n \geq m$, each row of A has no duplicate entries, and it has the following property: if the minimum of row i is located at column j_i, then $j_1 < j_2 < j_3 \ldots j_m$. For example, the following matrix is monotone (the minimum of each row is bolded):

$$\begin{bmatrix} \mathbf{1} & 3 & 4 & 6 & 5 & 2 \\ 7 & 3 & \mathbf{2} & 5 & 6 & 4 \\ 7 & 9 & 6 & 3 & 10 & \mathbf{0} \end{bmatrix}$$

Give an efficient (i.e., better than $O(mn)$-time) algorithm that finds the minimum in each row of an m-by-n monotone matrix A.

Give a 3-part solution. You do not need to write a formal recurrence relation in your runtime analysis; an informal summary of the runtime analysis such as "proof by picture" is fine.

Solution:

(i) **Main idea** If A has one row, we just scan that row and output its minimum.

Otherwise, we find the smallest entry of the $m/2$-th row of A by just scanning the row. If this entry is located at column j, then since A is a monotone matrix, the minimum for all rows above the $m/2$-th row must be located to the left of the j-th column. i.e. in the submatrix formed by

rows 1 to $m/2-1$ and columns 1 to $j-1$ of A, which we will denote by $A[1:m/2-1, 1:j-1]$. Similarly, the minimum for all rows below the $m/2$-th row must be located to the right of the j-th column. So we can recursively call the algorithm on the submatrices $A[1:m/2-1, 1:j-1]$ and $A[m/2+1:m, j+1:n]$ to find and output the minima for rows 1 to $m/2-1$ and rows $m/2+1$ to m.

(ii) **Proof of correctness** We will prove correctness by (total) induction on m.

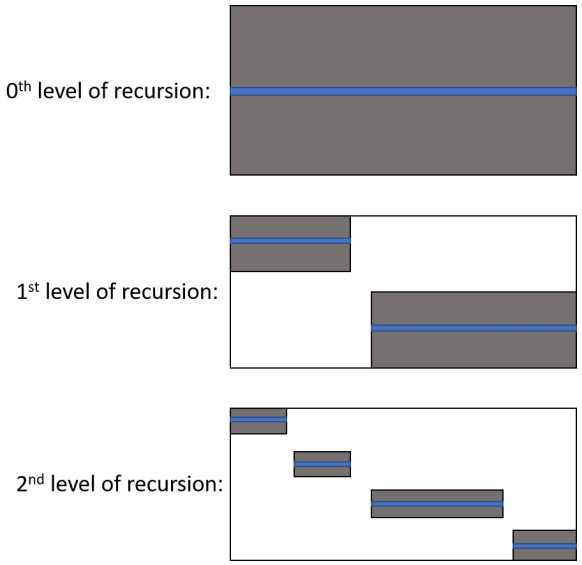
As a base case, $m=1$, and the algorithm explicitly finds and outputs the minimum of the single row.

If A has more than one row, we of course find and output the correct row minimum for row $m/2$. As argued above, the minima of rows 1 to $m/2-1$ of A are the same as the minima of the submatrix $A[1:m/2-1, 1:j-1]$, and the minima of rows $m/2+1$ to m are the same as the minima of the submatrix $A[m/2+1:m, j+1:n]$. By the induction hypothesis, the algorithm correctly outputs the minima of these matrices, and together with the $m/2$ row above, they find and output the minima of all the rows of A.

(iii) **Running time analysis** There are two ways to analyze the run time. One involves doing an explicit accounting of the total number of steps at each level of the recursion, as we did before we relied on the master theorem, or as we did in the proof of the master theorem. Since m is halved at each step of recursion, there are $\log m$ levels of recursion. At each level of recursion, the number of columns of the matrix get split into two – those associated with the left matrix and those associated with the right matrix. Moreover, the number of steps required to perform the split is just n, since it involves scanning all the entries of a single row. This means that at any level of the recursion, all the submatrices have disjoint columns, meaning if the different submatrices have n_k columns, then $\sum_k n_k \leq n$. The total number of steps required to split these matrices to go to the next level of recursion is then just $\sum_k n_k \leq n$. So there are $\log m$ levels of recursion, each taking total time n, for a grand total of $n \log m$.

Actually to be more accurate, when m=1, $\log m = 0$, so the expression should be $n(\log m + 1)$ to get the base case right.

For a "proof by picture", consider the following picture, where the grey boxes represent the submatrices we're solving the problem for at each level of recursion, and the blue lines represent the rows we're scanning at each level of the recursion. We can see the total length of the blue lines in each level of the recursion is $O(n)$.



Another way to analyze the running time is by writing and solving a recurrence relation (though again, this isn't necessary for full credit). The recurrence is easy to write out: let $T(m,n)$ be the number of steps to solve the problem for an $m \times n$ array. It takes n time to find the minimum of row $m/2$. If this row has minimum at column j, we recurse on submatrices of size at most $m/2$-by-j and $m/2$-by-$(n-j)$. So we can write the following recurrence relation: $T(m,n) \leq T(m/2, j) + T(m/2, n-j) + n$.

This does not directly look the recurrences in the master theorem — it is "2-D" since it depends upon two variables. You need some inspiration to guess the solution. We will guess that $T(m,n) \leq n(\log m + 1)$. We can prove this by strong induction on m.

Base case: $T(1,n) = n = n(\log 1 + 1)$.

Induction step:

$$T(m,n) \leq T(m/2,j) + T(m/2, n-j) + n \leq j\log(m/2) + 1 + (n-j)\log(m/2) + 1 + n$$

(by the induction hypothesis)

$$= n(\log(m/2) + 1 + 1) = n(\log m + 1).$$

5 Complex numbers review

A *complex number* is a number that can be written in the rectangular form $a + bi$ (i is the imaginary unit, with $i^2 = -1$). The following famous equation (*Euler's formula*) relates the polar form of complex numbers to the rectangular form:

$$re^{i\theta} = r(\cos\theta + i\sin\theta)$$

In polar form, $r \geq 0$ represents the distance of the complex number from 0, and θ represents its angle. Note that since $\sin(\theta) = \sin(\theta + 2\pi), \cos(\theta) = \cos(\theta + 2\pi)$, we have $re^{i\theta} = re^{i(\theta+2\pi)}$ for any r, θ.

The n-th *roots of unity* are the n complex numbers satisfying $\omega^n = 1$. They are given by

$$\omega_k = e^{2\pi ik/n}, \qquad k = 0, 1, 2, \ldots, n-1$$

(a) Let $x = e^{2\pi i 3/10}, y = e^{2\pi i 5/10}$ which are two 10-th roots of unity. Compute the product $x \cdot y$. Is this an n-th root of unity for some n? Is it a 10-th root of unity?

What happens if $x = e^{2\pi i 6/10}, y = e^{2\pi i 7/10}$?

Solution: $x \cdot y = e^{2\pi i 8/10}$. This is always an 10-th root of unity (it is in general). But because $8/10 = 4/5$, this is also a 5th root of unity.

If $x = e^{2\pi i 6/10}, y = e^{2\pi i 7/10}$, then we 'wind around' and the product becomes $e^{2\pi i 13/10} = e^{2\pi i 3/10}$.

(b) Show that for any n-th root of unity $\omega \neq 1$, $\sum_{k=0}^{n-1} \omega^k = 0$, when $n > 1$.

Hint: Use the formula for the sum of a geometric series $\sum_{k=0}^{n} \alpha^k = \frac{\alpha^{n+1}-1}{\alpha-1}$. It works for complex numbers too!

Solution: Remember that $\omega^n = 1$. So
$\sum_{k=0}^{n-1} \omega^k = \frac{\omega^n - 1}{\omega - 1} = \frac{1-1}{\omega-1} = 0$

(c) (i) Find all ω such that $\omega^2 = -1$.

Solution: $\omega = i, -i$

There are many ways to arrive at the solution, here's one: Squaring both sides, we get $\omega^4 = 1$. So we only need to consider the 4th roots of unity, $e^{2\pi i \cdot 0/4}, e^{2\pi i \cdot 1/4}, e^{2\pi i \cdot 2/4}, e^{2\pi i \cdot 3/4}$, or equivalently $1, i, -1, -i$. Geometrically, we get these by going $0, 1, 2, 3$ quarters of the way around the complex unit circle. Of these four values, the ones that square to -1 are $i, -i$.

(ii) Find all ω such that $\omega^4 = -1$.

Solution: $\omega = e^{2\pi i \cdot 1/8}, e^{2\pi i \cdot 3/8}, e^{2\pi i \cdot 5/8}, e^{2\pi i \cdot 7/8}$

Similarly to the previous part, squaring both sides we get $\omega^8 = 1$, so we only need to consider the 8th roots of unity. However, $\omega^4 \neq 1$, so ω is not a 4th root of unity. The 8th roots of unity that are not 4th roots of unity are $e^{2\pi i \cdot 1/8}, e^{2\pi i \cdot 3/8}, e^{2\pi i \cdot 5/8}, e^{2\pi i \cdot 7/8}$, and we can check that these all are solutions to $\omega^4 = -1$.

6 Extra Divide and Conquer Practice: Quantiles

Let A be an array of length n. The boundaries for the k quantiles of A are $\{a^{(n/k)}, a^{(2n/k)}, \ldots, a^{((k-1)n/k)}\}$ where $a^{(\ell)}$ is the ℓ-th smallest element in A.

Devise an algorithm to compute the boundaries of the k quantiles in time $\mathcal{O}(n \log k)$. For convenience, you may assume that k is a power of 2.

Hint: Recall that QUICKSELECT(A, ℓ) gives $a^{(\ell)}$ in $\mathcal{O}(n)$ time.

Solution: The idea is to find the median of A, partition it into two pieces around this median. Then we recursively find the medians of the two partitions, partition those further, and so on. If we do this $\log k$ times, we will have found all of the k-quantiles.

Finding the median and partitioning A takes $\mathcal{O}(n)$ time. We also do this for two arrays of size $n/2$, four times for arrays of size $n/4$, etc. So the total time taken is

$$\mathcal{O}(n + 2 \cdot n/2 + 4 \cdot n/4 + \cdots + 2^{\log k} n/2^{\log k}) = \mathcal{O}(\underbrace{n + n + \cdots + n}_{\log k \text{ times}}) = \mathcal{O}(n \log k)$$