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 $O(\cdot), \Theta(\cdot)$, and $\Omega(\cdot)$:

- $f(n) = O(g(n))$ 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 $g(n) = O(f(n))$. *(Asymptotically, f grows at least as much as g)*
- $f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ and $g(n) = O(f(n))$. *(Asymptotically, f and g grow the same)*

If we compare these definitions to the order on the numbers, 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). Generally in this class, to determine the type of asymptotic relationship between functions, we use limits:

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

- If $\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$, then $f(n) = O(g(n))$.
- 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: these are all sufficient conditions involving limits, and are not true definitions of O, Θ, and Ω.

1 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. $f_8, f_9, f_1, \ldots$

(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)	$n + \log n$	$n + (\log n)^2$

2 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 are the same asymptotically, 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) $T(n) = 2 \cdot T(n/2) + O(n)$

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

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

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

3 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.

4 Maximum Subarray Sum

Given an array A of n integers, the maximum subarray sum is the largest sum of any contiguous subarray of A (including the empty subarray). In other words, the maximum subarray sum is:

$$\max_{i \leq j} \sum_{k=i}^{j} A[k]$$

For example, the maximum subarray sum of $[-2, 1, -3, 4, -1, 2, 1, -5, 4]$ is 6, the sum of the contiguous subarray $[4, -1, 2, 1]$. Design an $O(n \log n)$-time algorithm that finds the maximum subarray sum.

Hint: Split the array into two equally-sized pieces. What are the possibilities for the subarray, and how does this apply if we want to use divide and conquer?

5 Pareto Optimality

Given a set of points $P = \{(x_1, y_1), (x_2, y_2) \dots (x_n, y_n)\}$, a point $(x_i, y_i) \in P$ is Pareto-optimal if there does not exist any $j \neq i$ such that such that $x_j > x_i$ and $y_j > y_i$. In other words, there is no point in P above and to the right of (x_i, y_i). Design a $O(n \log n)$-time divide-and-conquer algorithm that given P, outputs all Pareto-optimal points in P. Provide an algorithm description and runtime analysis.

(Hint: Split the array by x-coordinate. Show that all points returned by one of the two recursive calls is Pareto-optimal, and that you can get rid of all non-Pareto-optimal points in the other recursive call in linear time).

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

For all your answers, simplify if possible.

(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!

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

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