CS 161 (Stanford, Fall 2025) Section 1

Asymptotic Analysis

Asymptotic Analysis Definitions

Let f, g be functions from the positive integers to the non-negative reals.

Definition 1: (Big-Oh notation)

$f = O(g)$ if there exist constants $c > 0$ and n_0 such that for all $n \geq n_0$,

$$f(n) \leq c \cdot g(n).$$

Definition 2: (Big-Omega notation)

$f = \Omega(g)$ if there exist constants $c > 0$ and n_0 such that for all $n \geq n_0$,

$$f(n) \geq c \cdot g(n).$$

Definition 3: (Big-Theta notation)

$f = \Theta(g)$ if $f = O(g)$ and $f = \Omega(g)$.

Note: You will use "Big-Oh notation", "Big-Omega notation", and "Big-Theta notation" A LOT in class. Additionally, you may occasionally run into "little-oh notation" and "little-omega notation":

Definition 4:(Little-oh notation)

$f = o(g)$ if **for every constant** $c > 0$ there exist a constant n_0 such that for all $n \geq n_0$,

$$f(n) < c \cdot g(n).$$

Definition 5:(Little-omega notation)

$f = \omega(g)$ if **for every constant** $c > 0$ there exist a constant n_0 such that for all $n \geq n_0$,

$$f(n) > c \cdot g(n).$$

1 Asymptotic Analysis Problems

1.1

For each of the following functions, prove whether $f = O(g)$, $f = \Omega(g)$, or $f = \Theta(g)$. For example, by specifying some explicit constants n_0 and $c > 0$ such that the definition of Big-Oh, Big-Omega, or Big-Theta is satisfied. *Bonus: prove little-Oh and little-Omega.*

(a)	$f(n) = n\log(n^3)$	$g(n) = n\log n$
(b)	$f(n) = 2^{2n}$	$g(n) = 3^n$
(c)	$f(n) = \sum_{i=1}^{n} \log i$	$g(n) = n\log n$

1.2

Give an example of f, g such that f is not $O(g)$ and g is not $O(f)$.

1.3

Prove that if $f = \Omega(g)$ then f is not in $o(g)$.

2 Matrix Multiplication

In lecture, you have seen how digit multiplication can be improved upon with divide and conquer. Let us see a more generalized example of Matrix multiplication. Assume that we have matrices A and B and we'd like to multiply them. Both matrices have n rows and n columns.

For this question, you can make the simplifying assumption that the product of any two entries from A and B can be calculated in O(1) time.

2.1

What is the naive solution and what is its runtime? Think about how you multiply matrices.

2.2

Now if we divide up the problem like this:

$$XY = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}$$

We now have a divide and conquer strategy! Find the recurrence relation of this strategy and the runtime of this algorithm.

2.3

Can we do better? It turns out we can by calculating only 7 of the sub problems:

$$P_1 = A(F - H) \qquad P_5 = (A + D)(E + H)$$
$$P_2 = (A + B)H \qquad P_6 = (B - D)(G + H)$$
$$P_3 = (C + D)E \qquad P_7 = (A - C)(E + F)$$
$$P_4 = D(G - E)$$

And we can solve XY by

$$XY = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix}$$

We now have a more efficient divide and conquer strategy! What is the recurrence relation of this strategy and what is the runtime of this algorithm?

3 How NOT to prove claims by induction

In this class, you will prove a lot of claims, many of them by induction. You might also prove some wrong claims, and catching those mistakes will be an important skill!

The following is an example of a false proof where an obviously untrue claim has been 'proven' using induction (with some errors or missing details, of course). Your task is to investigate the 'proof' and identify the mistakes made.

3.1

Fake Claim 1:

$$\underbrace{\frac{1}{1 \cdot 2} + \frac{1}{2 \cdot 3} + \ldots}_{n \text{ terms}} = \frac{3}{2} - \frac{1}{n}. \tag{1}$$

Inductive Hypothesis: (1) holds for $n = k$

Base Case: For $n = 1$,

$$\frac{1}{1 \cdot 2} = 1/2 = \frac{3}{2} - \frac{1}{1}.$$

Inductive Step: Suppose the inductive hypothesis holds for $n = k$; we will show that it is also true for $n = k + 1$. We have

$$\begin{aligned} \left(\frac{1}{1 \cdot 2} + \frac{1}{2 \cdot 3} + \ldots \frac{1}{(k-1) \cdot k}\right) + \frac{1}{k \cdot (k+1)} &= \frac{3}{2} - \frac{1}{k} + \frac{(k+1) - k}{k \cdot (k+1)} \quad \text{(by weak induction hypothesis)} \\ &= \frac{3}{2} - \frac{1}{k} + \frac{1}{k} - \frac{1}{k+1} \\ &= \frac{3}{2} - \frac{1}{k+1}. \end{aligned}$$

Conclusion: By weak induction, the claim follows.

4 Induction: Snowball Fight

On a flat ice sheet, an *odd* number of penguins are standing such that their pairwise distances to each other are all different. At the strike of dawn, each penguin throws a snowball at another penguin that is closest to them. Show that there is always some penguin that doesn't get hit by a snowball.

5 Skyline (Pseudocode and Big-O)

You are handed a scenic black-and-white photo of the skyline of a city. The photo is n-pixels tall and m-pixels wide, and in the photo, buildings appear as black (pixel value 0) and sky background appears as white (pixel value 1). In any column, all the black pixels are below all the white pixels. In this problem, you will design and analyze efficient algorithms that find the location of a tallest building in the photo. (It could be that there are multiple tallest buildings that all have the same height; in this case, your algorithm should return any one of them.)

The input is an $n \times m$ matrix (for $n, m \geq 1$), where the buildings are represented with 0s, and the sky is represented by 1s. The matrix is indexed from top to bottom, from left to right. Each column of the matrix has a single building that is represented by 0's. The output is an integer representing the location of a tallest building. For example, for the input 6×5 matrix below, $A[0,0] = 1$, $A[3,0] = 0$, $A[3,2] = 1$, a tallest building has height 5 and is in location 1 (assuming we are 0-indexing). Thus the output is 1.

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & \mathbf{0} & 1 & 1 & 1 \\ 1 & \mathbf{0} & 1 & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & 1 & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & 1 & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix}$$

5.1

Find an algorithm that finds a tallest building in time $O(m \log n)$. You may assume that your input is a $n \times m$ matrix A, and you may access an element in the i-th row and j-th column in constant time.

5.2

Find an algorithm that finds a tallest building in time $O(m+n)$ and write it out in pseudocode. You may assume that your input is a $n \times m$ matrix A, and you may access an element in the i-th row and j-th column in constant time.

5.3

For some values of (n, m) the algorithm from part (a) is more efficient, while for others, the algorithm from part (b) is more efficient. For each of the values of n in terms of m below, determine which of the above algorithm runtimes is more efficient (or that they are equally efficient) in terms of big-O notation. The case $n = m$ is filled in as an example (in blue).

$n = ?$	100	$\sqrt{m}$	$\frac{m}{\log m}$	m	$m \log m$	2^m
Runtime for (a)				$O(m \log m)$		
Runtime for (b)				$O(m)$		
Which is better?				(b)		