CS 161 (Stanford, Winter 2025) Section 1

Asymptotic analysis

Review of definitions

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

Definition 1: (Big-Oh notation)

$f(n) = O(g(n))$ 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)

$f(n) = \Omega(g(n))$ 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)

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

Some additional definitions

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)

$f(n) = o(g(n))$ 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)

$f(n) = \omega(g(n))$ 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 questions

1.1 Limit definitions of asymptotic relationships

If you've taken a calculus course, the first thing you'll learn is the definition of a limit. For our purposes, we'll only use limits as n approaches ∞:

Definition 6: (Limit at infinity)

$\lim_{n\to\infty} f(n) = c$ if for any $\epsilon > 0$, there exists n_0 such that $|f(n) - c| < \epsilon$ for all $n \geq n_0$.

$\lim_{n\to\infty} f(n) = \infty$ if for any $m > 0$, there exists n_0 such that $f(n) > m$ for all $n \geq n_0$.

(a) Show that if $\lim_{n\to\infty} f(n)/g(n) = c > 0$, then $f(n) = \Theta(g(n))$.

(b) Show that if $\lim_{n\to\infty} f(n)/g(n) = 0$, then $f(n) = o(g(n))$.

(c) Assuming $\lim_{n\to\infty} f(n)/g(n)$ exists, give conditions in terms of this limit that correspond to $f(n) = O(g(n))$, $f(n) = \Omega(g(n))$, and $f(n) = \omega(g(n))$. (No need to prove that your conditions are correct.)

Solution

(a) Assume $\lim_{n\to\infty} f(n)/g(n) = c$ for some $c > 0$. By selecting $\epsilon = c/2$, there is some n_0 such that $c/2 < f(n)/g(n) < 3c/2$ for all $n \geq n_0$. Rearranging, we have that $f(n) > c/2 \cdot g(n)$, showing that $f(n) = \Omega(g(n))$, and $f(n) < 3c/2 \cdot g(n)$, showing that $f(n) = O(g(n))$. Therefore $f(n) = \Theta(g(n))$.

(b) Assume $\lim_{n\to\infty} f(n)/g(n) = 0$. Then for every constant $c > 0$, there is some n_0 such that $f(n)/g(n) < c$ for all $n \geq n_0$. Rearranging, $f(n) < c \cdot g(n)$ for all $n \geq n_0$, which is the definition of $f(n) = o(g(n))$.

(c)
- If $\lim_{n\to\infty} f(n)/g(n) = c \geq 0$, then $f(n) = O(g(n))$.
- If $\lim_{n\to\infty} f(n)/g(n) = c > 0$ or $\lim_{n\to\infty} f(n)/g(n) = \infty$, then $f(n) = \Omega(g(n))$.
- If $\lim_{n\to\infty} f(n)/g(n) = \infty$, then $f(n) = \omega(g(n))$.

1.2 Properties of asymptotic relationships

Are the following statements true or false? $f(n)$, $g(n)$, and $h(n)$ are all non-negative functions of the integers $0, 1, 2, \ldots$

(a) If $f(n) = O(g(n))$, then $g(n) = \Theta(g(n) + f(n))$.

(b) If $f(n) = O(g(n))$, then $f(n) \cdot h(n) = O(g(n) \cdot h(n))$.

(c) If $f(n) = O(g(n))$, then either $f(n) = o(g(n))$ or $f(n) = \Theta(g(n))$.

(d) If $f(n) = O(g(n))$ and $h(n)$ is an integer-valued function of n that increases to ∞, then $h(f(n)) = O(h(g(n)))$.

(e) If $f(n) = O(g(n))$ and $h(n)$ is an integer-valued function of n that increases to ∞, then $f(h(n)) = O(g(h(n)))$.

(f) If Algorithm 1 runs in time $f(n)$ on inputs of size n, and Algorithm 2 runs in time $g(n)$ on inputs of size n, where $f(n) = o(g(n))$, then we should always prefer Algorithm 1 over Algorithm 2.

Solution

(a) True. Let $c > 0$ be a constant such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$. Then $g(n) + f(n) \leq g(n) + c \cdot g(n) = (1 + c)g(n)$ for all $n \geq n_0$, so $g(n) + f(n) = O(g(n))$. Since f is nonnegative, $g(n) + f(n) \geq g(n)$, so $g(n) + f(n) = \Omega(g(n))$. Therefore $g(n) = \Theta(g(n) + f(n))$.

(b) True. Let $c > 0$ be a constant such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$. Multiplying both sides by $h(n) \geq 0$ shows that $f(n) \cdot h(n) \leq c \cdot g(n) \cdot h(n)$, so $f(n) \cdot h(n) = O(g(n) \cdot h(n))$.

(c) False. For example, consider $f(n) = 1$ and

$$g(n) = \begin{cases} 1 & \text{if } n \text{ is odd} \\ n & \text{if } n \text{ is even} \end{cases}.$$

Then $f(n) \leq g(n)$ for all n, so $f(n) = O(g(n))$. But $f(n) = g(n)$ for all odd n, which means that $f(n)$ cannot be less than $c \cdot g(n)$ for all $c > 0$, hence $f(n)$ is not $o(g(n))$. And $f(n) = \frac{1}{n}g(n)$ for all even n, meaning that $f(n)$ cannot be greater than $c\dot{g}(n)$ for any $c > 0$, so $f(n)$ is not $\Omega(g(n))$.

(d) False. Consider $f(n) = 2 \log n$, $g(n) = \log n$, and $h(n) = 2^n$. Then $f(n) = O(g(n))$ and $h(n)$ is increasing, but $h(f(n)) = 2^{2 \log n} = n^2$, while $h(g(n)) = 2^{\log n} = n$. Therefore $h(f(n))$ is not $O(h(g(n))$.

(e) True. Let $c > 0$ be a constant such that for all $n \geq n_0$, $f(n) \leq c \cdot g(n)$. Let n' be the smallest integer such that $h(n') \geq n_0$. Then $h(n) \geq n_0$ for all $n \geq n'$ because h is increasing. Therefore $f(h(n)) \leq c \cdot g(h(n))$ for all $n \geq n'$, which shows that $f(h(n)) = O(g(h(n))$.

(f) False. There are many reasons why we might prefer Algorithm 2. For example, if we know we will only use the algorithm on small inputs, Algorithm 2 may be faster in real life despite being worse for large n. Algorithm 2 might also have other desirable properties separate from runtime, such as protecting privacy, treating inputs fairly, or providing an explanation for its output.

2 Induction

You're a chef at an internationally-renowned pancake restaurant, and have just made a big stack of n pancakes. However, you're serving a picky customer who wants the pancakes to be sorted by size, with the largest on the bottom of the stack and the smallest at the top. You

can sort the stack by inserting your spatula somewhere in the stack and flipping the pancakes above that point, reversing their order. Give an algorithm for sorting the stack of pancakes that requires $O(n)$ flip operations.

Hint: use induction to show that a stack of n pancakes can be sorted using $2n - 3$ flips.

Bonus question: Can we use this approach to beat all known sorting algorithms and sort a list in $O(n)$ time? Why or why not?

Solution

Inductive hypothesis: Any stack of n pancakes can be sorted using $2n - 3$ flips.
Base case: A stack of two pancakes requires at most one flip to reverse the order.
Inductive step: Assume a stack of k pancakes can be sorted using $2k - 3$ flips. Here is a procedure for sorting $k + 1$ pancakes. Identify the largest pancake, then flip just below it so that it is on top of the stack, then flip the entire stack so that the largest pancake is on the bottom. Then sort the remaining k pancakes using our inductive hypothesis. This requires a total of $2k - 3 + 2 = 2(k + 1) - 3$ flips, so the inductive hypothesis holds for $k + 1$ pancakes.
Conclusion: By induction, the hypothesis holds for all $n \geq 2$.
Bonus: This does not give us an $O(n)$ sorting algorithm because the flip operation cannot be implemented in $O(1)$ time on a computer.

3 Bad 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 are examples of a false proof where an untrue claim has been 'proven' using induction (with some errors or missing details, of course). Your task is to investigate the 'proofs' and identify the mistakes made.

3.1

Fake Claim 1: All numbers are equal.

Inductive hypothesis: Any set of k numbers are all equal to each other.

Base Case: For $k = 1$, any one number is equal to itself.

Inductive Step: Suppose the inductive hypothesis holds for k; we will show that it is also true for $k + 1$. Given a set of $k + 1$ numbers in an arbitrary order, the first k of them are all equal by our inductive hypothesis. Similarly, the last k are all equal. Therefore all $k + 1$ must be equal to each other (see diagram below).

$$\underbrace{x_1,\ \overbrace{x_2,\ \ldots, x_k,}^{k \text{ numbers}}}_{k \text{ numbers}}\ x_{k+1}$$

Conclusion: By induction, the claim holds for all $k \geq 1$.

Solution

The inductive step implicitly assumes that there is some overlap between the first k and the last k numbers in the set of $k+1$ numbers, but this is not true in a set of 2 numbers. Therefore the inductive chain breaks at the very first step.

3.2

Fake Claim 2: For every non-negative integer n, $2^n = 1$.

Inductive Hypothesis: For all integers n such that $0 \leq n \leq k$, $2^n = 1$.

Base Case: For $n = 0$, $2^0 = 1$.

Inductive Step: Suppose the inductive hypothesis holds for k; we will show that it is also true for $k+1$, i.e. $2^{k+1} = 1$. We have

$$\begin{aligned} 2^{k+1} &= \frac{2^{2k}}{2^{k-1}} \\ &= \frac{2^k \cdot 2^k}{2^{k-1}} \\ &= \frac{1 \cdot 1}{1} && \text{(by strong induction hypothesis)} \\ &= 1 \end{aligned}$$

Conclusion: By strong induction, the claim follows.

Solution

The error in this proof occurs in the inductive step. Given that we induct on k with a base case of $k = 0$, in order for the inductive hypothesis to apply to the denominator, the exponent, $k-1$, must be a non-negative integer. This requires the implicit assumption that $k \geq 1$. The inductive step must hold for $k = 0$, so the assumption that $k \geq 1$ is invalid and the inductive step fails.

3.3

Fake Claim 3:

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

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

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

Conclusion: By weak induction, the claim follows.

Solution

The first part of the long derivation of the inductive step is wrong — the summation in the parentheses only contains $k - 1$ summands! It should contain k terms, so it is missing the last term.

4 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 X and Y 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 X and Y can be calculated in $O(1)$ time.

4.1

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

Solution

The naive solution is that we will multiply row by column to get each element of the new matrix. Each new element of the new matrix is a sum of a row multiplied by a column, which takes n time, and there are n^2 new element to compute, resulting in a runtime of $O(n^3)$.

4.2

Now if we divide up X and Y into quarters 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. (You may assume that n is a power of 2.)

Solution

The recurrence relation of this approach is $T(n) = 8T(\frac{n}{2}) + O(n^2)$ because you have 8 subproblems, and cutting subproblem size by 2, while doing n^2 additions to combine the subproblems. Using the recurrence, we know that at the last level of recursion we will have $8^{log(n)}$ subproblems of size $O(1)$.

$$8^{log(n)} = n^{log(8)} = n^3$$

Thus, this approach is at least $O(n^3)$. Looks like we did not improve the running time at all!

4.3

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?

Solution

The recurrence relation of this algorithm is $T(n) = 7T(\frac{n}{2}) + O(n^2)$ because you have 7 subproblems, and cutting subproblem size by 2, while doing n^2 additions to combine the subproblems. Using similar calculation to above, we calculate that there are $\approx n^{2.81}$ subproblems of size $O(1)$. We'll show later in class that just as with Karatsuba multiplication, this determines the runtime to be $O(n^{2.81})$ (i.e. the work done at higher levels of the subproblem decomposition is asymptotically negligible).