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)	$f(n) = \sum_{i=1}^{n} \log i$	$g(n) = n\log n$

Solution

(a) $f(n) = \Theta(g(n))$. Since $f(n) = n\log(n^3) = 3n\log n$. To prove Big-Oh, chose any c above 3 (for example $c = 4$), then $f(n) = 3n\log n \leq 4n\log n = cg(n) \quad \forall n \geq n_0$ of any n_0 of your choice. To prove big Omega, chose any c below 3 (for example $c = 2$, then $f(n) = 3n\log n \geq 2n\log n = cg(n) \quad \forall n \geq n_0$ of any n_0 of your choice.

(b) $f(n) = \Omega(g(n))$. Since $f(n) = 2^{2n} = 4^n$. Chose $c = 1, n_0 = 1$, then $f(n) = 4^n \geq 1 \times 3^n = cg(n) \quad \forall n \geq n_0$. To disprove Big-Oh, use contradiction.

$$4^n \leq c3^n$$
$$n\log 4 \leq \log c + n\log 3$$
$$n \leq \frac{\log c}{\log 4 - \log 3}$$

So pick any $c \text{ and } n > \frac{\log c}{\log 4 - \log 3}$. We have a contradiction.

(c) Inspect summation

$$\sum_{i=1}^{n} \log i = \log 1 + \log 2 + \log 3 + ... + \log n$$
$$\sum_{i=1}^{n} \log i \leq \log n + \log n + \log n + ... + \log n$$
$$\sum_{i=1}^{n} \log i \leq n\log n$$

So we have proven Big-Oh.

In order to prove Big-Omega, inspect summation again

$$\sum_{i=1}^{n} \log i = \log 1 + \log 2 + \log 3 + ... + \log(\frac{n}{2}) + ... + \log n$$
$$\sum_{i=1}^{n} \log i \geq \sum_{i=n/2}^{n} \log i \geq \log(\frac{n}{2}) + \log(\frac{n}{2}) + ... + \log(\frac{n}{2})$$
$$\sum_{i=1}^{n} \log i \geq \frac{n}{2} \log(\frac{n}{2}) = \frac{n}{2}(\log(n) - \log(2))$$

So

$$\frac{n}{2}(\log(n) - \log(2)) \geq cn \log n$$
$$\frac{1}{2} \log(n) - c \log n \geq \frac{\log 2}{2}$$
$$(1 - 2c) \log n \geq 1$$

Now we can pick $c_0 = \frac{1}{4}, n_0 = 4$

1.2

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

Solution

There are many such examples. Here is one:

$$f(n) = n.$$
$$g(n) = \begin{cases} 1 & \text{if } n \text{ is odd} \\ n^2 & \text{if } n \text{ is even} \end{cases}.$$

1.3

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

Solution

Assume **by contradiction** that $f = \Omega(g)$ and also $f = o(g)$. By definition of $\Omega()$, there exist constants c and n_Ω such that for all $n \geq n_\Omega$,

$$f(n) \geq c \cdot g(n). \quad (1)$$

On the other hand, by definition of $o()$, there exists n_o such that for all $n \geq n_o$,

$$f(n) < c \cdot g(n). \tag{2}$$

Consider n which is greater than both n_Ω and n_o (e.g. their max +1). Then (1) and (2) should both hold, but this is a contradiction!

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.

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 elements to compute, resulting in a runtime of $O(n^3)$.

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.

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 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!

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?

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 the runtime of this method to be $\approx n^{2.81}$.

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 are examples 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 'proofs' and identify the mistakes made.

3.1

Fake Claim 1: 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

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

Conclusion: By strong induction, the claim follows.

Solution

The error in this proof occurs in the inductive step. 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.

3.2

Fake Claim 2:

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

Inductive Hypothesis: (3) 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

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

Conclusion: By weak induction, the claim follows.

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 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.

Solution

Claim: For every non-negative number n, if there are $2n+1$ penguins, then there exists a penguin that doesn't get hit by a snowball.

Inductive hypothesis: The above claim holds for all $n \leq k$.

Base Case: When $n = 1$, there are 3 penguins. Since the pairwise distances between these penguins are different, 2 of these penguins must form the closest pair, and will throw their snowballs at each other. Thus, the third penguin will not get hit with any snowball.

Inductive Step: Suppose the inductive hypothesis holds for $n \leq k$; we will show that it also holds for $n = k+1$. There are $2(k+1)+1$ penguins. Choose any 2 penguins that are closest to each other and call them A and B. These two penguins will throw snowballs at each other. Then, there are $2k+1$ penguins remaining and by the inductive hypothesis, at least one of these penguins will not get hit by a snowball.

Conclusion: By induction, the claim follows.

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.

Solution

At a high level, we perform a row-wise binary search. More precisely, we split the current matrix in half; if there is at least one zero in the middle row, we recurse on the top half (that contains the middle row), and if not, we recurse on the bottom half. We stop when we see there is only one row left in the matrix, and at this point simply return the index of any column that has a 0, since this will correspond to the tallest building.

Pseudocode:

```
def TallestTower( an n x m matrix A ):
    if n == 1:
        for j = 0, ..., m-1:
            if A[0, j] == 0:
                return j
        return "There are no buildings in this city!"
    if A[n//2, :] has at least one zero:
        return TallestTower( A[:n//2, :] ) //assuming the
        current array notation is right-inclusive
    else:
        return TallestTower( A[n//2 + 1:, :] )
```

Note that there are multiple ways to do this part, including returning early if a certain row contains exactly one zero (only one building even reaches that height). Another

one is to do a binary search on each building to find the height, and then return the largest.

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.

Solution

Starting with the bottom left corner, move upwards until you see a 1, and then move right until we see a 0, and repeat until we reach the rightmost column. The tallest building is height $n - i - 1$ where i is the row we finish on, with the exception that if we hit the top of the photo, then the height of the tallest building is height $n - i$ with $i = 0$. Each time we move either upwards or rightwards (note that the i variable only decreases), so at most $m + n$ steps are made.
Pseudocode:

```
def TallestTower( an n x m matrix A ):
    best = None
    i = n - 1
    if n > 0:
        for j = 0, ..., m-1:
            if A[i, j] == 0:
                best = j
            // move up until we hit the sky or top of photo
            while A[i, j] == 0 and i > 0:
                i -= 1
    return best
```

Again, we note that there are multiple ways to write pseudocode for this part, including returning early if we hit the top of the photo (that building must be the tallest) or having slightly different loops.

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)		

Solution

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