CSCI 4470 Algorithms

Part I Foundations

- 1 The Role of Algorithms in Computing
- 2 Getting Started
- **3 Characterizing Running Times**
- 4 Divide-and-Conquer
- 5 Probabilistic Analysis and Randomized Algorithms

Chapter 3 Characterizing Running Times

Using Three Properties of Loop Invariant in Insertion Sort Algorithm

Three Properties of Loop Invariant:

1. Initialization: It's true prior to the first iteration of the loop.

2. Maintenance: If it's true before an iteration of the loop, it remains true before the next iteration.

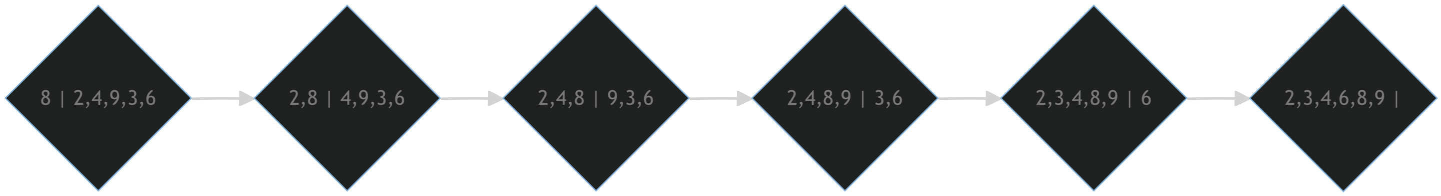
3. Termination: When the loop terminates, the invariant provides a useful property to help show that the algorithm is correct.

Insertion Sort Algorithm Analysis

Insertion Sort Algorithm for Array `a[8,2,4,9,3,6]`

1. Initially, `8` is considered sorted.
2. Consider `2`. Since `2` is less than `8`, we swap them, resulting in: `2,8,4,9,3,6`.
3. Next is `4`. It's greater than `2` but less than `8`, so it's placed between them, resulting in: `2,4,8,9,3,6`.
4. `9` is already in the correct position.

5. `3` is less than all preceding numbers except for `2`, so it's placed after `2`: `2,3,4,8,9,6`.
6. `6` is less than `9` and `8` but greater than `4`, so it's placed between `4` and `8`: `2,3,4,6,8,9`.
7. Thus, the sorted result is `a[2,3,4,6,8,9]`.



Loop Invariant Example with Insertion Sort

Consider the array a = [8,2,4,9,3,6]. Let's apply the loop invariant to prove the correctness of the insertion sort.

Initialization:
Before the first iteration, the subarray a[1] (containing only 8) is trivially sorted.

Maintenance:
If a[1] to a[i-1] are sorted, the loop places a[i] in its correct position, ensuring a[1] to a[i] are sorted at the end of the i-th iteration.

- **Examples**:
  - After the 2nd iteration, the first two elements (2,8) are sorted.
  - After the 3rd iteration, the first three elements (2,4,8) are sorted.

Termination:
After n iterations, the first n elements are sorted, implying the entire array is sorted.

- **Explanation**:
  - The loop invariant, proven by induction, confirms the algorithm's correctness.
  - At loop termination, $j = n + 1$, confirming that a[1, …, n] is sorted.

Key Points:

- Loop invariants are essential to prove algorithm correctness.
- The loop invariant concept can be applied to analyze various algorithms.

Examination Points:

- Definition of loop invariant.
- How to apply loop invariant in the Insertion Sort algorithm.
- For instance, when we move to A[2] (which is 2), it might get inserted before '8', making '2,8' the sorted subarray.

Example Loop Invariants

To find the maximum element of an array using loop invariants:

Pseudocode:

```
Find_Max(A, n)  // Added 'n' to represent the length of array A
{
    m = A[1]
    for i = 2 to n
    {
        if (A[i] > m)  // Corrected 'a[i]' to 'A[i]'
            m = A[i]  // Corrected the assignment from 'A[1]' to 'A[i]'
    }
    return m
}
```

Loop Invariant Proof:

Consider the loop invariant: At the start of each iteration of the loop, `m` is the maximum element in the sub-array `A[1, ..., i-1]`.

1. **Initialization**:
   - Before the loop starts (at `i=2`), `m` equals the maximum of the sub-array `A[1]`, which is `A[1]`.
   - The loop invariant holds true before the loop begins.
2. **Maintenance**:
   - Assume that the loop invariant holds true at the start of an arbitrary iteration `i` (meaning, `m` is the maximum of `A[1, ..., i-1]`).
   - During this iteration, if `A[i]` is greater than `m`, we update `m` to be `A[i]`.
   - At the start of the next iteration (i.e., `i+1`), `m` will be the maximum of the sub-array `A[1, ..., i]`.
   - This ensures the loop invariant holds true for the next iteration.
3. **Termination**:
   - The loop ends when `i=n+1`.
   - At this juncture, because of our loop invariant, `m` is the maximum of the sub-array `A[1, ..., n]`, which covers the whole array.
   - Thus, at termination, `m` represents the highest value in the entire array.

Analysis of Insertion Sort

Insertion sort's complexity largely depends on the number of comparisons and swaps made. Let's derive the best, average, and worst cases step by step:

1. **Best Case**: The list is already sorted.

- Shortest running time for a given input size
  Every new element we consider (starting from the second) only needs one comparison to ascertain that the list remains sorted.
  Total comparisons = $1+1+1+\ldots+1$ (for n-1 times) = $n-1$. Complexity is $O(n)$.

2. **Worst Case**: The list is sorted in the reverse order.

- Longest running time for given input size
  For every new element we consider, we might have to compare and move it past all the elements that came before it.
  Total comparisons = $1+2+3+\ldots+(n-1)$.
  Using summation, $\sum_{i=2}^{n}(i-1)=1+2+\ldots+(n-1)$.
  As per the formula for sum of the first n natural numbers: $\frac{n(n-1)}{2}$. This is $O(n^2)$.

3. **Average Case**: We make an assumption that for every element, it has an equal chance of being placed in any position.

- Average running time of all possible inputs of a given size
  On average, an element will be compared against half of the elements that came before it.
  Average comparisons = $\frac{1}{2}+\frac{2}{2}+\frac{3}{2}+\ldots+\frac{n-1}{2}$.
  This is equivalent to $\frac{1}{2}\sum_{k=1}^{n-1}k$, which equals $\frac{1}{2}\times\frac{n(n-1)}{2}$. This is also $O(n^2)$.

4. **Amortized Analysis**:

- Worst case sequence of n consecutive operations

Definition:

- Amortized analysis is used to determine the time-averaged cost of each operation in the worst case over a sequence of operations, rather than the worst-case time for a single operation.

Key Idea:

- While an individual operation might be expensive, the average cost per operation might be small when averaged over a sequence of operations.

Common Techniques:

1. **Aggregate Analysis**:
   - Analyze the sequence of operations as a whole to determine the average operation cost.
2. **Accounting Method**:
   - Assign different costs (tokens) to different operations, ensuring the total cost remains under the specified limit.
3. **Potential Method**:
   - Use a hypothetical potential energy to represent saved-up work. The difference in potential between two points in time represents the saved-up cost.

Asymptotic Notations

Asymptotic notation, often used in algorithm analysis, describes the limiting behavior of a function. The most common notations in this domain are

1. **Big O Notation (O)**: Represents an upper bound.

- O notation: asymptotic “less than” or “upper bound”
  - $f(n) = O(g(n))$ implies: $f(n) \leq g(n)$

2. **Omega Notation (Ω)**: Represents a lower bound.

- Ω notation: asymptotic “greater than” or “lower bound”
  - $f(n) = \Omega(g(n))$ implies: $f(n) \geq g(n)$

3. **Theta Notation (Θ)**: Represents asymptotic bounds that are both upper and lower, signifying that a function grows at the same rate as another, up to constant factors.

- $f(n) = \Theta(g(n))$ implies that there exist constants $c_1, c_2 > 0$ and n_0 such that $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ for all $n \geq n_0$. This indicates that $f(n)$ grows neither faster nor slower than $g(n)$ by more than a constant factor.
- Θ notation: asymptotic “bounded by” or “tight bound”

4. **Little o Notation (o)**: Describes an upper bound that is not tight.

- $f(n) = o(g(n))$ if for every positive constant c, there exists a value n_0 such that $0 \leq f(n) < c \cdot g(n)$ for all $n > n_0$.

5. **Little omega Notation (ω)**: Describes a lower bound that is not tight.

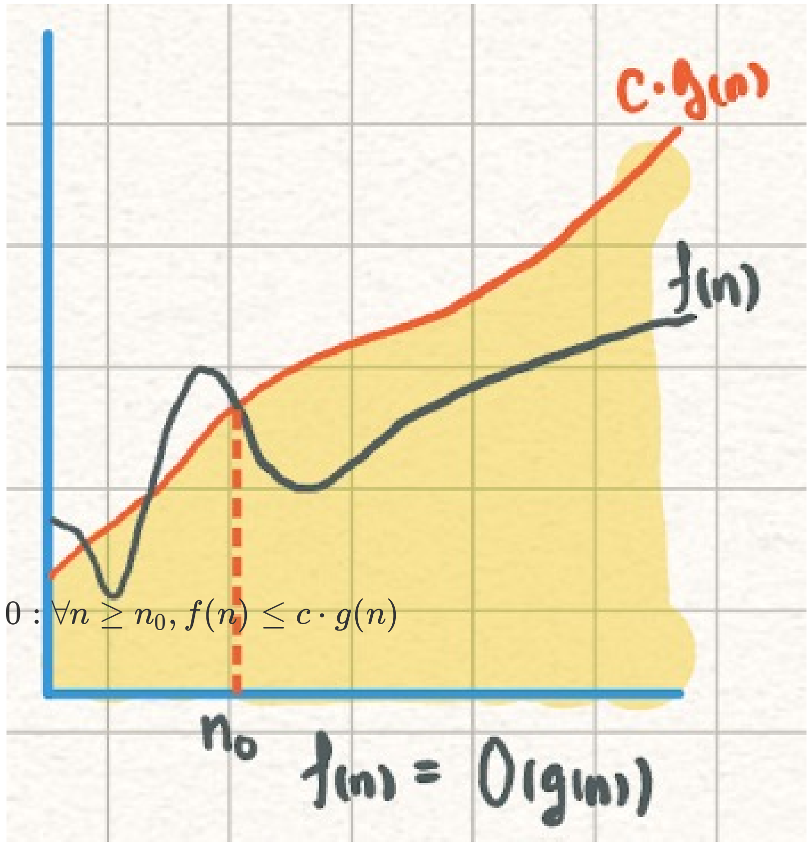
- $f(n) = \omega(g(n))$ if for every positive constant c, there exists a value n_0 such that $0 \leq c \cdot g(n) < f(n)$ for all $n > n_0$.

Big-O	Big-Ω	Big-Θ	Little-o	Little-ω
1. $O(n)$	2. $\Omega(n)$	3. $\Theta(n)$	4. $o(n)$	5. $\omega(n)$
$\leq$	$\geq$	$=$	$>$	$<$

- For Little o and Little ω, The table entries $>$ and $<$ can be replaced with “dominated by” and “dominates” respectively, or provide their formal definitions.

Big-O Notation

Big-O notation (Provides an UPPER BOUND on a function $f(n)$, typically for the Worst Case)



Definition of Big-O:

- $O(g)$ represents the set of all functions for which there exist constants $c > 0$ and $n_0 \geq 0$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

$$Big - O : f(n) \in O(g(n)) \equiv \exists\, c > 0, \exists\, n_0 \geq 0 : \forall n \geq n_0, f(n) \leq c \cdot g(n)$$

- $f(n)\ is\ O(g(n))\ implies : f(n) \leq g(n)$

  $f(n) \leq c \cdot g(n)$ is the UPPER BOUND

Big-O notation:

$f(n) \in O(g(n))$ if and only if there exist constants $c > 0$ and n_0 such that for all $n \geq n_0$, $f(n) \leq c \cdot g(n)$.

- Stating $f(n)$ is $O(g(n))$ implies that, for some constant c and beyond a certain point n_0, $f(n)$ is always bounded above by $c \cdot g(n)$.

Big-O Rules:

- If $f(n)$ is a polynomial of degree i, then $f(n)$ is $O(n^i)$.

1. Drop lower-order terms: When you have multiple terms, drop terms with smaller growth rates. For example, if you have $O(n^2 + n)$, it is $O(n^2)$.
2. Drop constant factors: Constants don't affect the rate of growth. Hence, $2n$ is $O(n)$ and not $O(2n)$.
3. Aim for the simplest expression:

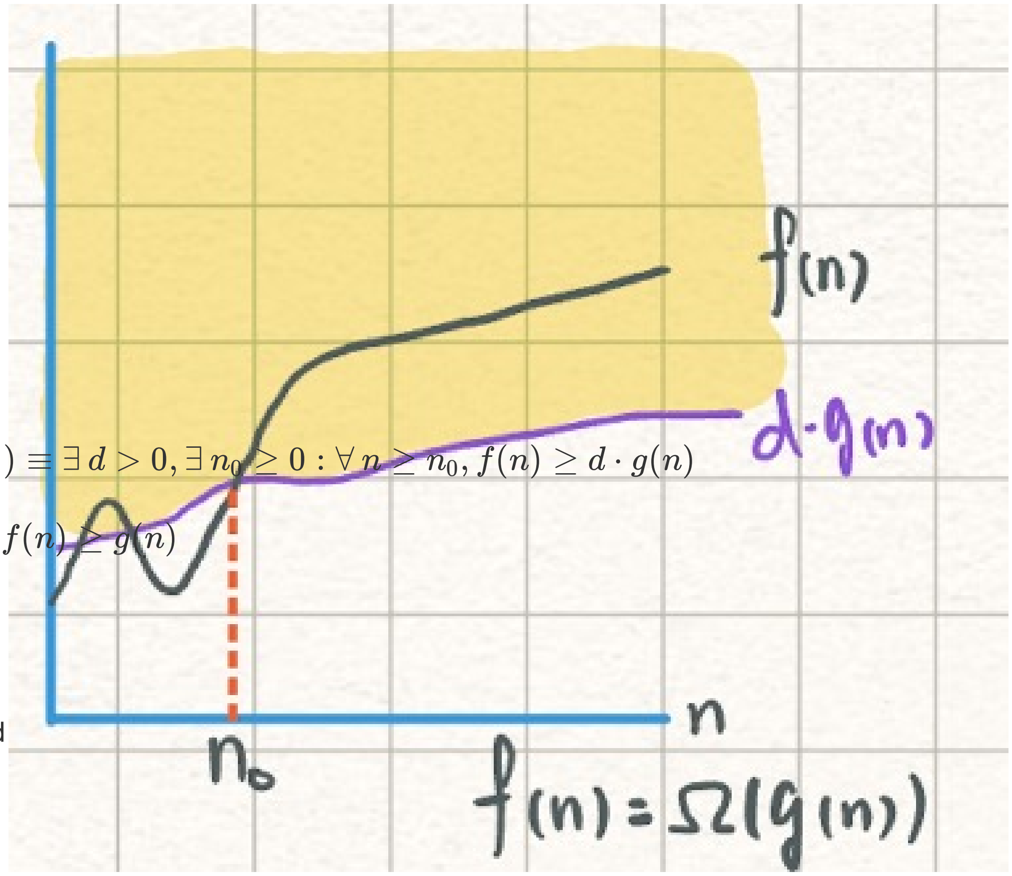
- It's preferable to say "$2n$ is $O(n)$" rather than "$2n$ is $O(n^2)$".
- Likewise, state $3n + 5$ is $O(n)$ rather than $3n + 5$ is $O(3n)$".

Big-Omega (Ω) Notation

Big-Omega notation (Provides a LOWER BOUND on a function $f(n)$, typically for the Best Case)

Definition of Big-Omega (Ω):

- $\Omega(g)$ represents the set of all functions for which there exist constants $d > 0$ and $n_0 \geq 0$ such that $f(n) \geq d \cdot g(n)$ for all $n \geq n_0$.



$$Big - Omega : f(n) \in \Omega(g(n)) \equiv \exists\, d > 0, \exists\, n_0 \geq 0 : \forall\, n \geq n_0, f(n) \geq d \cdot g(n)$$

- $f(n)\ is\ \Omega(g(n))\ implies:\ \ f(n) \geq g(n)$
- $f(n) \geq d \cdot g(n)$ is the LOWER BOUND

Ω notation: $f(n) \in \Omega(g(n))$ if and only if there exist constants $d > 0$ and n_0 such that for all $n \geq n_0$, $d \cdot g(n) \leq f(n)$.

- Stating $f(n)$ is $\Omega(g(n))$ implies that, for some constant d and beyond a certain point n_0, $f(n)$ is always bounded below by $d \cdot g(n)$.

Big-Theta (Θ) Notation

(Tight bound on $f(n)$, Average Case) Big-Theta (Θ)

Big-Theta notation (Provides a tight bound on a function $f(n)$, capturing both UPPER and LOWER BOUNDS)

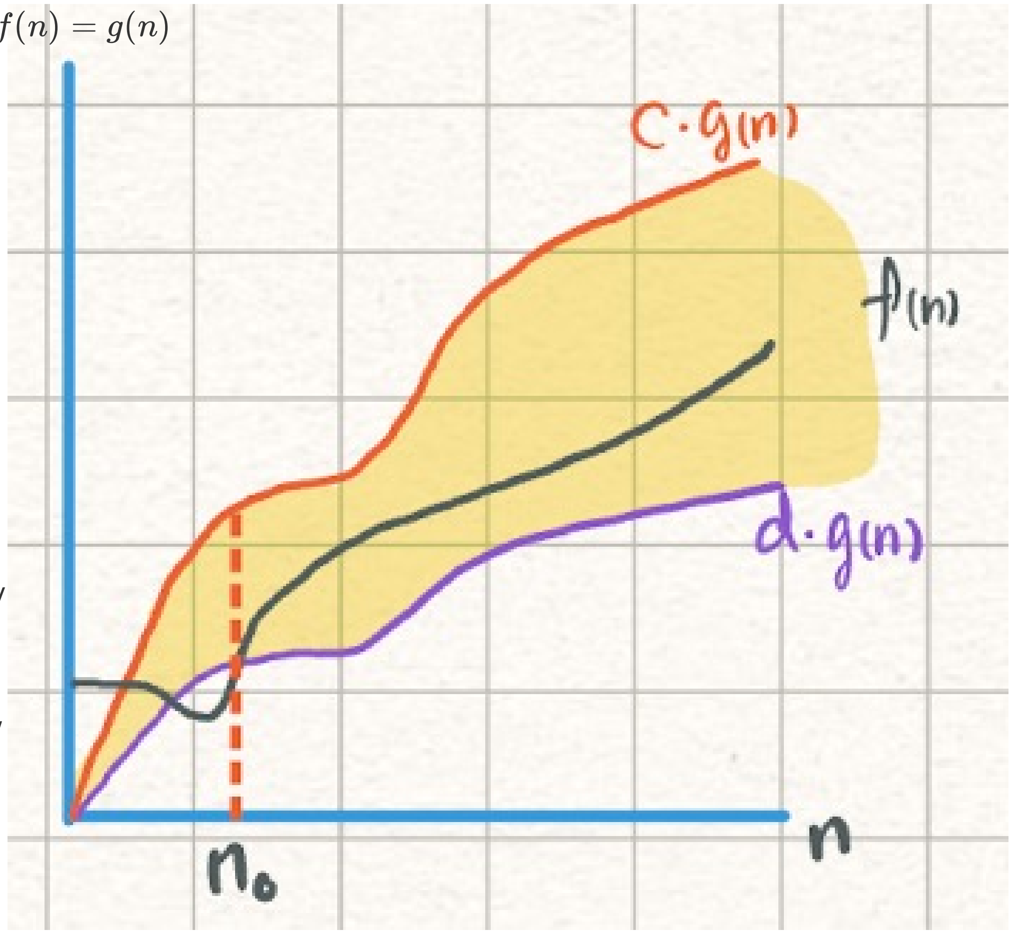
Definition of Big-Θ:

- $\Theta(g)$ represents the set of all functions for which there exist constants $c, d > 0$ and $n_0 \geq 0$ such that $d \cdot g(n) \leq f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.
- $\Theta(g)$ exists if there are positive constants c, d, and n_0 such that for all $n \geq n_0$, we have $0 \leq d \cdot g(n) \leq f(n) \leq c \cdot g(n)$.

$$Big - \Theta : f(n) \in \Theta(g(n)) \equiv f(n) \in \Omega(g(n)) \cup O(g(n))$$

$$Big - \Theta : f(n) \in \Theta(g(n)) \equiv f(n) \in \Omega(g(n)) \wedge f(n) \in O(g(n))$$

- $f(n)\ is\ \Theta(g(n))\ implies: f(n) = g(n)$



- This means that for a function $f(n)$ to be $\Theta(g(n))$, it should satisfy both the conditions of being $O(g(n))$ (upper bound) and $\Omega(g(n))$ (lower bound).
- To find the Big-Θ notation $\Theta(g(n))$, we need to find the Big-O notation $c \cdot g(n)$ and Big-Ω notation $d \cdot g(n)$ both, which are the upper bound and the lower bound
- $f(n)$ is $\Theta(g(n))$ doesn't imply that $f(n) = g(n)$ exactly, but rather it means that $f(n)$ is bounded both above and below by $g(n)$ within a constant factor.
- To determine the Big-Θ notation $\Theta(g(n))$, one needs to establish both the Big-O notation (upper bound) $c \cdot g(n)$ and the Big-Ω notation (lower bound) $d \cdot g(n)$.

- From n_0, the function $c \cdot g(n)$ is an "upper bound" that covers all of $f(n)$. This is because in the proof, it is shown that $c \cdot g(n)$ is always greater than $f(n)$.
- From n_0, the function $d \cdot g(n)$ is a "lower bound" that is covered by all of $f(n)$. This is because in the proof, it is shown that $d \cdot g(n)$ is always less than $f(n)$.

Transitivity

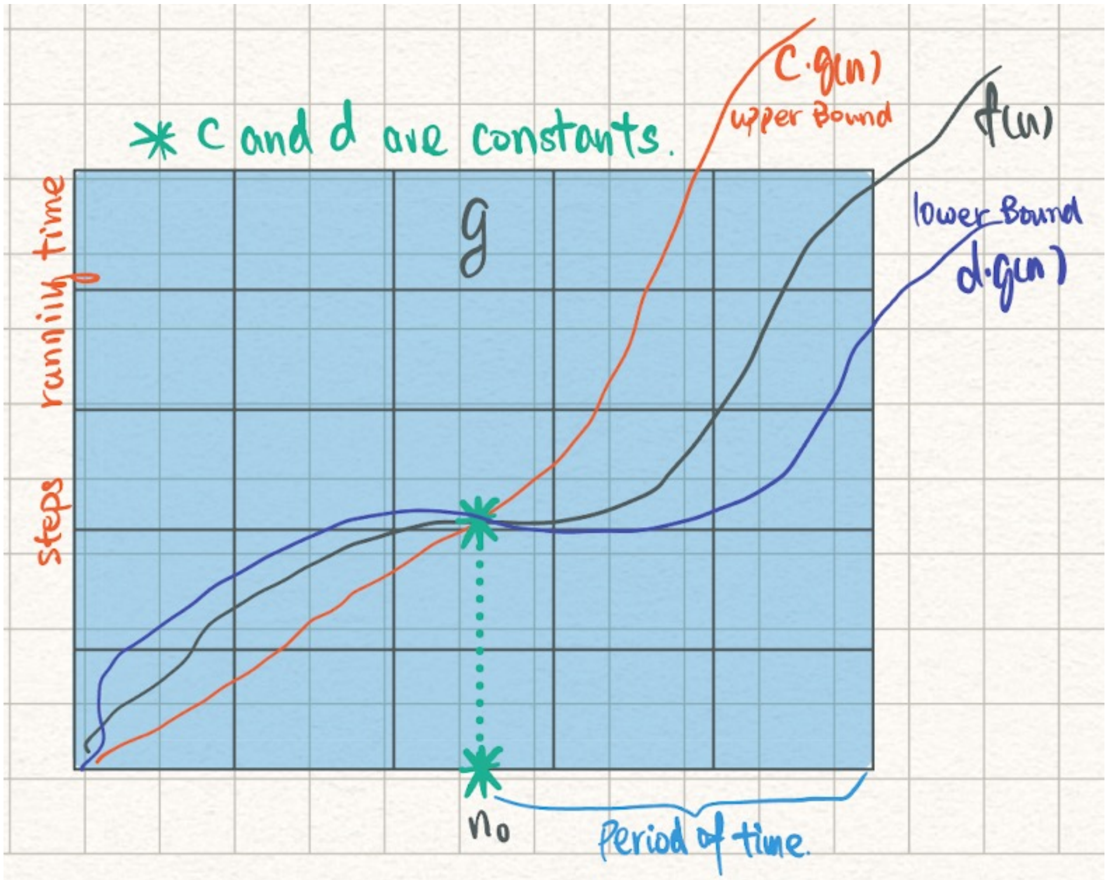
If $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$, then $f(n) = \Theta(h(n))$.
If $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$.
If $f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n))$, then $f(n) = \Omega(h(n))$.
If $f(n) = o(g(n))$ and $g(n) = o(h(n))$, then $f(n) = o(h(n))$.
If $f(n) = \omega(g(n))$ and $g(n) = \omega(h(n))$, then $f(n) = \omega(h(n))$.

Reflexivity

$f(n) = \Theta(f(n))$

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

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



Example of Big-O

Show that $3n^3 + 20n^2 + 5$ is $O(n^3)$. $f(n) \in O(n^3)$ is TRUE

Proof:

To prove that $3n^3 + 20n^2 + 5$ is $O(n^3)$, we need to find constants c and n_0 such that:

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

for all $n \geq n_0$.

Where:

- $f(n) = 3n^3 + 20n^2 + 5$
- $g(n) = n^3$

Let's examine the terms of $f(n)$:

$$3n^3 \leq 3n^3$$

$$20n^2 \leq 20n^3$$

for $n \geq 1$

$$5 \leq 5n^3$$

for $n \geq 1$

Adding these inequalities:

$$3n^3 + 20n^2 + 5 \leq 3n^3 + 20n^3 + 5n^3$$

$$3n^3 + 20n^2 + 5 \leq 28n^3$$

From the above inequality, we can see that $f(n) \leq 28g(n)$ for all $n \geq 1$. Thus, we have:

$$f(n) \in O(n^3)$$

with $c = 28$ and $n_0 = 1$.

Example 02 of Big-O

Show that $3\log(n) + 5$ is $O(\log(n))$

Example 03 of Big-Theta

Prove: $f(x) = 3x^2 + 8xlog(x)$ is $\Theta(x^2)$ Let's begin by analyzing the process

1. **Prove** $f(x) \in O(x^2)$

- To prove $f(x) \in O(x^2)$, we must show $f(x) \leq c_1x^2$ for some positive constant c_1 and for all x beyond some threshold.
- Considering $f(x)$, $3x^2 + 8xlog(x) \leq 3x^2 + 8x^2$ (because $log(x) \leq x$ for all $x > 0$).
- Thus, $f(x) \leq 11x^2$ when $x \geq 1$.
- Therefore, $f(x) \in O(x^2)$ with $c_1 = 11$ and $x_0 = 1$.

2. **Prove** $f(x) \in \Omega(x^2)$

- To prove $f(x) \in \Omega(x^2)$, we must show $f(x) \geq c_2x^2$ for some positive constant c_2 and for all x beyond some threshold.
- For $x \geq 2$, $log(x)$ is at least 1.
- Hence, $3x^2 + 8xlog(x) \geq 3x^2$.
- Thus, $f(x) \geq 3x^2$ when $x \geq 2$.
- Therefore, $f(x) \in \Omega(x^2)$ with $c_2 = 3$ and $x_0 = 2$.

3. **Conclusion**

- Given the proofs for $f(x) \in O(x^2)$ and $f(x) \in \Omega(x^2)$, we can conclude $f(x) \in \Theta(x^2)$ with $c_1 = 11$, $c_2 = 3$, and $x_0 = 2$.

Little-o Notation

Definition:

- The little-o notation represents an upper bound that is not asymptotically tight.

Mathematical Representation:

- $o(g(n)) = \{f(n)$: For any constant $c > 0$, there exists a constant $n_0 > 0$ such that $0 \leq f(n) < c \cdot g(n)$ for all $n \geq n_0\}$.

Explanation:

- In little-o notation, $f(n)$ becomes arbitrarily small compared to $g(n)$ as n approaches infinity.
- The little-o notation is used in mathematics to describe an asymptotic relationship between two functions. Specifically, $f(x) = o(g(x))$ means:

$$\lim_{x \to \infty} \frac{f(x)}{g(x)} = 0$$

- In plain language, this means that $f(x)$ grows slower than (g(x)) as x approaches infinity.

Examples Method 1:

- For instance, $2n = o(n^2)$ because $2n$ grows much slower than n^2, but $2n^2 \neq o(n^2)$ because they grow at comparable rates.

Proof Steps:

1. Define little-o notation:
   - $f(n) \in o(g(n))$ if $f(n) < c \cdot g(n)$ for all $c > 0$ and for sufficiently large $n \geq n_0$.
2. Proving $2n$ is little-o of n^2:
   - $2n < c \cdot n^2$ for all $c > 0$ and for sufficiently large $n \geq n_0$.
   - For example, choose $c = 1$, then:
     - $2n < n^2$, this is true for $n > 2$.
     - So, one valid n_0 could be 3.
3. Analyzing $2n^2$ relative to n^2:
   - $2n^2$ does not satisfy $f(n) < c \cdot g(n)$ for all $c > 0$ because when you choose $c = 1$, you get $2n^2 = n^2$, which does not meet the little-o criteria.

The limit notation

$$\lim_{x \to \infty} \frac{f(x)}{g(x)} = 0$$

is a direct way to prove that $f(n) = o(g(n))$, meaning $f(n)$ grows strictly slower than $g(n)$. Let's use this notation to validate the given example.

Example Method 2: Using Limit Notation:

Given $2n$ and $2n^2$, Evaluate if they are $\in o(n^2)$.

1. For $f(n) = 2n$ and $g(n) = n^2$ to verify $2n = o(n^2)$:
   - Compute the following limit:

$$\lim_{n\to\infty} \frac{2n}{n^2}$$

   - After simplification:

$$\lim_{n\to\infty} \frac{2}{n} = 0$$

   Since the limit is 0, this confirms that $2n = o(n^2)$.

2. For $f(n) = 2n^2$ and $g(n) = n^2$ to verify $2n^2 = o(n^2)$:
   - We compute:

$$\lim_{n\to\infty} \frac{2n^2}{n^2}$$

   - Simplifying:

$$\lim_{n\to\infty} 2 = 2$$

   Since the limit is 2 and not 0, it confirms that $2n^2$ is not $o(n^2)$.

Examples 02 Method 1:

Prove that $f(n) = 3n + 4$ is $o(n^2)$.

Proof:

We want to show $f(n) = o(g(n))$ if for any $c > 0$, there exists n_0 such that $f(n) < c \cdot g(n)$ for all $n \geq n_0$.

Steps:

1. Given $f(n) = 3n + 4$, we want $f(n) \in o(n^2)$. That is, $3n + 4 < c \cdot n^2$ for some constant $c > 0$ and for all $n \geq n_0$.
2. Multiply everything by c:
   $3nc + 4c < c^2 n^2$
3. Rearrange the terms:
   $4c < c^2 n^2 - 3nc$

4. Using $(a+b)^2 = a^2 + b^2 + 2ab$, consider $a = cn$ and $b = \frac{-3}{2}$ to give $a^2 = c^2n^2$ and $2ab = -3cn$
   $4c + \frac{9}{4} < c^2n^2 - 3nc + \frac{9}{4}$
   (which is equivalent to $(cn - \frac{3}{2})^2$)
5. Finally, we have
   $cn - \frac{3}{2} > \sqrt{4c + \frac{9}{4}}$

This shows that as n approaches infinity, $3n+4$ grows strictly slower than n^2, and therefore, $f(n) = 3n+4$ is $o(n^2)$.

Examples 02 Method 2 Limit Notation:

> Let's prove $f(n) = 3n + 4$ is $o(n^2)$ using the limit notation.

Proof: To prove that $f(n) = 3n+4$ is $o(n^2)$, we need to show

$$\lim_{n\to\infty} \frac{f(n)}{n^2} = 0$$

Steps:

1. Plug in $f(n) = 3n+4$ into the formula 將 $f(n) = 3n+4$:
   $$\lim_{n\to\infty} \frac{3n+4}{n^2}$$
2. Split the fraction into two terms:
   $$\lim_{n\to\infty} \frac{3n}{n^2} + \lim_{n\to\infty} \frac{4}{n^2}$$
3. Simplify:
   $$\lim_{n\to\infty} 3 \cdot \frac{1}{n} + \lim_{n\to\infty} \frac{4}{n^2}$$
4. As n approaches infinity, both $\frac{1}{n}$ and $\frac{4}{n^2}$ tend towards 0
   $$3 \cdot 0 + 0 = 0$$
5. Hence, we have shown
   $$\lim_{n\to\infty} \frac{f(n)}{n^2} = 0$$
6. This confirms that $f(n) = 3n+4$ is $o(n^2)$

Tricky Little-o Notation Question

Question: Prove or disprove: $f(n) = n^2 + n$ is $o(n^2)$.

Solution: To prove or disprove $f(n) = n^2 + n$ is $o(n^2)$, we need to find the limit:

$$\lim_{n\to\infty} \frac{f(n)}{n^2}$$

Steps:

1. Substitute $f(n) = n^2 + n$ into the formula:

$$\lim_{n\to\infty} \frac{n^2 + n}{n^2}$$

2. Break the fraction into two terms:

$$\lim_{n\to\infty} 1 + \lim_{n\to\infty} \frac{n}{n^2}$$

3. Simplify:

$$\lim_{n\to\infty} 1 + \lim_{n\to\infty} \frac{1}{n}$$

4. As n goes to infinity, $\frac{1}{n}$ approaches 0:

$$1 + 0 = 1$$

5. Thus, we find:

$$\lim_{n\to\infty} \frac{f(n)}{n^2} = 1$$

Conclusion:
Since the limit is not 0, $f(n) = n^2 + n$ is **not** $o(n^2)$.

Little-omega Notation

Definition:

- The little-omega notation represents a lower bound, indicating that one function grows strictly faster than another as they approach infinity.

Comparison:

- ω notation is analogous to Ω notation in the way that o-notation is analogous to O-notation.

Examples:

- $n^2/2 = \omega(n)$ but $n^2/2 \neq \omega(n^2)$.

Explanation:

- In ω-notation, the function $f(n)$ grows strictly faster than $g(n)$ as n approaches infinity.

Mathematical Representation:

- If $f(n) > c \cdot g(n)$ for all constants $c > 0$ and for all sufficiently large $n (n \geq n_0)$, then $f(n)$ is in $\omega(g(n))$.
- Another way to represent this relationship is:

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$$

Tricky Little-omega Notation Question

Question: Prove or disprove: $f(n) = n^2 \log n$ is $\omega(n^2)$.

Solution: To prove or disprove $f(n) = n^2 \log n$ is $\omega(n^2)$, we need to find the limit:

$$\lim_{n \to \infty} \frac{f(n)}{n^2}$$

Steps:

1. Substitute $f(n) = n^2 \log n$ into the formula:

$$\lim_{n \to \infty} \frac{n^2 \log n}{n^2}$$

2. Simplify the expression:

$$\lim_{n \to \infty} \log n$$

3. As n approaches infinity, $\log n$ also approaches infinity:

$$\lim_{n \to \infty} \log n = \infty$$

Conclusion:
Since the limit is infinity, we can conclude that $f(n) = n^2 \log n$ is $\omega(n^2)$.

Example: Prove and Disprove:

Question: Is 2^{n+1} in $O(2^n)$?

Proof: First, let's express 2^{n+1}:

$$2^{n+1} = 2 \cdot 2^n$$

To prove 2^{n+1} is in $O(2^n)$, we need to show there exists constants c and n_0 such that:

$$2^{n+1} \leq c \cdot 2^n$$

for all $n \geq n_0$.

Take $c = 2$ and any $n_0 \geq 0$. We have:

$$2 \cdot 2^n \leq 2 \cdot 2^n$$

Which is clearly true for all n.

Therefore, we can conclude that 2^{n+1} is in $O(2^n)$ as $2 \cdot 2^n = O(2^n)$.

This is aligned with a general principle: for any positive constant k, $k \cdot f(n) = O(f(n))$. In the example $2 \cdot 2^n \in O(2^n)$,because, $2 \cdot f(n) \in O(f(n))$

In essence, you were on the right track. This proof highlights that 2^{n+1} grows at most twice as fast as 2^n, but in terms of big O notation, they belong to the same complexity class.

Example: Prove whether 2^{2n} is in $\Theta(2^n)$:

Question: Is $2^{2n} \in \Theta(2^n)$? To do this, we need to verify if 2^{2n} is in both $O(2^n)$ and $\Omega(2^n)$.

Proof:

1. **Define the functions**:
   - $f(n) = 2^{2n} = 4^n$
   - $g(n) = 2^n$
2. **Check for $f(n) \in O(g(n))$**:
   To prove or disprove this, we need to determine if there exist constants $c > 0$ and n_0 such that:

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

   for all $n \geq n_0$.
   Checking this, we see that for $4^n \leq c \cdot 2^n$, there's no constant c for which this is true as n grows large.
3. **Check for $f(n) \in \Omega(g(n))$**:
   Observing that $f(n) = 4^n$ grows strictly faster than $g(n) = 2^n$, we can conclude $f(n)$ is in $\Omega(g(n))$.
4. **Check for $f(n) \in \omega(g(n))$**:
   For $f(n) \in \omega(g(n))$, for any given constant $c > 0$, there exists a n_0 such that:

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

for all $n \geq n_0$.
Given $f(n) = 4^n$ and $g(n) = 2^n$, it's evident that for any positive c, 4^n will eventually surpass $c \times 2^n$.

5. **Final Decision for Θ**:
$f(n)$ is not in $O(g(n))$ but is in $\Omega(g(n))$ and $\omega(g(n))$. Therefore, $f(n)$ is not in $\Theta(g(n))$.

Overall, 2^{2n} (or 4^n) is not in $\Theta(2^n)$ because it grows much faster than 2^n.

Example: To determine if $2^{f(n)} = O(2^{g(n)})$ when $f(n) = O(g(n))$.

1. **Given:** $f(n) = 2n$ and $g(n) = 4n$.
$2^{f(n)} = 2^{2n} = 4^n$ and $2^{g(n)} = 2^{4n} = 16^n$.
    - 4^n is not $O(16^n)$. Because, as n grows, 16^n grows much faster than 4^n.
2. **Given:** $f(n) = 4n$ and $g(n) = \frac{n}{2}$. **給定：** $f(n) = 4n$ 和 $g(n) = \frac{n}{2}$.
$2^{f(n)} = 2^{4n} = 16^n$ and $2^{g(n)} = 2^{\frac{n}{2}} = \sqrt{2^n}$.
    - 16^n grows significantly faster than $2^{\sqrt{n}}$. Thus, $2^{f(n)}$ is not $O(2^{g(n)})$.
3. **Table values:**

The table aims to compare values of 16^n and $2^{\sqrt{n}}$:

n	16^n	$2^{\sqrt{n}}$
2	16^2	$2^{\sqrt{2}}$
4	16^4	2^2
6	16^6	$2^{\sqrt{6}}$
8	16^8	$2^{\sqrt{8}}$
10	16^{10}	$2^{\sqrt{10}}$

1. The last lines:

$$f(n) \in \Omega(n)$$

means that $f(n)$ grows at least as fast as a linear function of n.

$$f(n) \in O(n)$$

means $f(n)$ grows at most as fast as n.

In essence, the exponential growth in $2^{f(n)}$ and $2^{g(n)}$ isn't always reflective of the relation between $f(n)$ and $g(n)$ when considering Big O notation. The core of the problem is to remember that the exponential function drastically magnifies growth differences between $f(n)$ and $g(n)$.

This example is illustrating a crucial concept in the analysis of algorithms and Big O notation. It's emphasizing that even if one function $f(n)$ is $O(g(n))$, it doesn't necessarily mean that $2^{f(n)}$ is $O(2^{g(n)})$. The exponential function can greatly amplify the growth differences between two functions.

Let's break it down:

1. **The first part** is showing that even though $f(n) = 2n$ is clearly $O(4n)$, when you take the exponential of both sides, the relationship doesn't hold. 4^n is not $O(16^n)$. This is because the exponential function greatly magnifies the growth rate difference between the two functions.
2. **The second part** is another example to emphasize the point. Even though $4n$ grows faster than $\frac{n}{2}$, when you take the exponential, 16^n grows much faster than $2^{\sqrt{n}}$.
3. **The table** is a practical demonstration of the growth rates of the two functions from the second part. As you can see, as n increases, the value of 16^n grows much more rapidly than $2^{\sqrt{n}}$.
4. **The last lines** are definitions of the Big O and Big Omega notations. They're there to provide context and remind you of the meaning of these notations.

$\lfloor Flooring \rfloor$ and $\lceil Ceiling \rceil$

- **Floor and Ceiling Functions:**
  The floor function of x, represented as $\lfloor x \rfloor$, is the biggest integer less than or equal to x.
  The ceiling function of x, represented as $\lceil x \rceil$, is the smallest integer more than or equal to x.
- **Properties of Floor and Ceiling**:
  For any real number x: $x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1$
  - Examples: $\lfloor 3.4 \rfloor = 3$, $\lceil 3.5 \rceil = 4$
    $-\lfloor x \rfloor = \lceil -x \rceil$. For instance, $-\lfloor 3.5 \rfloor = \lceil -3.5 \rceil = -3$
- **Properties of the Floor Function**:
  $\lfloor x \rfloor > x - 1$. This suggests that the floor function is at minimum $x - 1$. Hence, $\lfloor x \rfloor \in \Omega(x)$.
  $\lfloor x \rfloor \leq x$. This indicates that the floor function is at most x. Thus, $\lfloor x \rfloor \in O(x)$.
  Because $\lfloor x \rfloor$ is both an upper and lower bound for x up to a constant factor, $\lfloor x \rfloor \in \Theta(x)$.
- **Properties of the Ceiling Function**:
  $\lceil x \rceil \leq x + 1$. This denotes that the ceiling function is no more than $x + 1$. Thus, $\lceil x \rceil \in O(x)$.
  $\lceil x \rceil > x$. This means the ceiling function is strictly larger than x. Hence, $\lceil x \rceil \in \Omega(x)$.
  Given that $\lceil x \rceil$ is both an upper and lower bound for x up to a constant factor, $\lceil x \rceil \in \Theta(x)$.
- **Translation Property**:

$\lfloor n + x \rfloor = n + \lfloor x \rfloor$
$\lceil n + x \rceil = n + \lceil x \rceil$

$\lfloor Flooring \rfloor$ and $\lceil Ceiling \rceil$

- Properties of Flooring and Ceiling
  $x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1$, x is a real number
  - e.g. $\lfloor 3.4 \rfloor = 3$, $\lceil 3.5 \rceil = 4$
  - e.g. $-\lfloor x \rfloor = \lceil -x \rceil$
  - e.g. $-\lfloor 3.5 \rfloor = \lceil -3.5 \rceil$
  - e.g. $= -3 = -3$
- Properties of Flooring

$\lfloor x \rfloor > x - 1$, $\lfloor x \rfloor \in \Omega(x)$
$\lfloor x \rfloor \leq x$, $\lfloor x \rfloor \in O(x)$, Linear Growth which is always true
$\lfloor x \rfloor \in \Theta(x)$

$\lfloor n + x \rfloor = n + \lfloor x \rfloor$

$\lceil n + x \rceil = n + \lceil x \rceil$

Properties of Log Function

1. $log_a(a) = 1$
2. $log_a(a^x) = x$
3. $log(ab) = log(a) + log(b)$
4. $log_a(b^x) = xlog_a(b)$
5. $a^{log_a(x)} = x$
6. $log_a(x) = \frac{log_b(x)}{log_b(a)}$

Some Notation For Logs

- $lg(n) = log_2(n)$
- $ln(n) = log_e(n)$
- $log(n) = log_{10}(n)$
- $lg^k(n) = (lg(n))^k$
- $lg\ lg\ n = lg(lg(n))$
- $lg\ n + k = (lg(n))$
- $+k \neq lg(n + k)$

Fact's about Factorials

- Stirling's approximation ,
  $n! = \sqrt{2\pi n}(\frac{n}{e})^n(1 + \Theta(\frac{1}{n}))$, or equivalently
  $\sqrt{2\pi n}(\frac{n}{e})^n \leq n! \leq e\sqrt{n}(\frac{n}{e})^n)$
- From this, we can deduce the following
  - $n! = o(n^n)$
  - $n! = \omega(2^n)$
  - $log(n!) = \Theta(nlog(n))$

Example of Factorial:
$4! = 4 \cdot 3!$
$4! = 4 \cdot 3 \cdot 2!$
$4! = 4 \cdot 3 \cdot 2 \cdot 1!$

Properties of Factorial

$0! = 1$

Functional Iteration

- This is like function composition, but you are composing the function with itself

$$f^{(i)}(n) = \begin{cases} n & \text{if } i = 0 \\ f(f^{(i-1)}(n)) & \text{if } i > 0 \end{cases}$$

$f(n) = 2n$ then $f^{(i)}(n) = ?$