# CS 2500: Algorithms

## Lecture 1: Introduction and Logistics

Shubham Chatterjee

Missouri University of Science and Technology, Department of Computer Science

August 19, 2024

# About Me

**Dr. Shubham Chatterjee**
Assistant Professor

**Research Interests:** Neural IR, LLMs, Conversational AI.
**Previously:** University of Edinburgh, UK.
**PhD:** University of New Hampshire, Durham, USA.

# Teaching Assistant

**Priyesh Ranjan**
Ph.D. Student

**Research Interests.** Federated learning and attacker detection in federated learning setups, time series data analysis, and healthcare data analytics using neural networks.

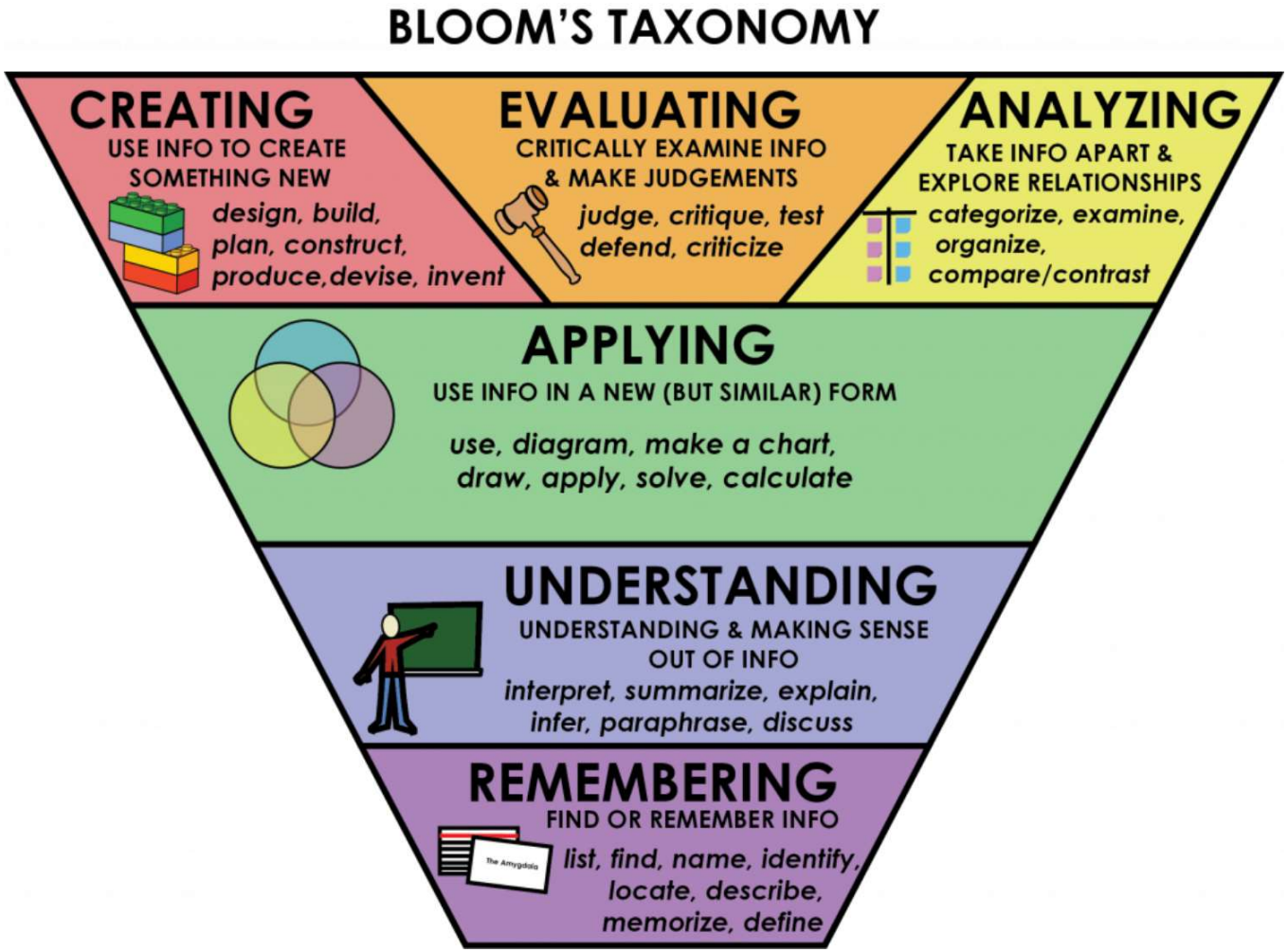
# Learning Objectives

- Analyze the asymptotic performance of algorithms in terms of runtime and storage memory.
- Write rigorous correctness proofs for algorithms.
- Demonstrate a familiarity with major algorithms.
- Apply important algorithmic design paradigms and methods of analysis.
- Synthesize efficient algorithms in common engineering design situations.

# Student Outcomes

- Argue the correctness of algorithms using inductive proofs and invariants.
- Analyze worst-case runtime of algorithms in asymptotic notation.
- Employ the divide-and-conquer paradigm to design algorithms when a problem calls for it and evaluate their performance by solving recurrences.
- Use graphs to model engineering problems, when appropriate, and perform graph search.
- Design greedy algorithms to solve sequential decision problems efficiently.
- Design dynamic programming algorithms to solve sequential decision problems optimally using value iteration.
- Identify computationally hard problems and design tractable, approximate algorithms.

# Taxonomy of Learning Objectives in Cognitive Domain



# Grading Information

**Components:**

- Weekly homework assignments. [40%]
  - Problems + coding.
  - Released every Tuesday.
  - Due next Tuesday.
  - **No homework this week!**
- 1 midterm examination. [20%]
- 1 final comprehensive examination. [20%]
- Recitations. [20%]

**Final Grade:**

- $[90 - 100]$: A
- $[80 - 90)$: B
- $[70 - 80)$: C
- $[60 - 70)$: D
- $< 60$: F

# Mutual Expectations: Students

Students are expected to:

- Be present in class (physically and mentally).
- Ask at least one question every session.
- Present homework solutions.
- Do their own work and contribute significantly in team activities.
- Study and repeat necessary class materials independently.

# Mutual Expectations: Instructor

The instructor is expected to:

- Make lecture notes available before the class.
- Return graded homework in a timely manner.
- Be available for questions regarding class material during class, online, and if necessary by appointment.
- Notify students that are in danger of not meeting the class goals early on.
- Provide ungraded test exams (quizzes) for students' self-assessment.

# A Note on Expectations

- It is not sufficient to just be present in class and submit homework.
- Obtaining an A requires that you study and review materials from lecture notes, assignments, and discussions with the help of the book.
- If stuck, please see the instructor.

# Prerequisites for this Course (Required)

Before taking this algorithms class, students should have completed the following courses or have equivalent knowledge:

- **Discrete Mathematics**
  - Logic, sets, relations, functions, combinatorics, graph theory, proof techniques.
  - Importance: Provides the mathematical foundation for analyzing algorithms.
- **Data Structures**
  - Arrays, linked lists, stacks, queues, trees, graphs, hash tables, heaps.
  - Importance: Understanding of data structures is crucial for algorithm design.
- **Introduction to Programming**
  - Basic programming concepts, control structures, functions, recursion.
  - Importance: Comfort with writing and debugging code is necessary for implementing algorithms.

# Prerequisites for this Course (Optional but Recommended)

Before taking this algorithms class, students are recommended to complete the following courses or have equivalent knowledge:

- **Basic Calculus**
  - Limits, derivatives, integrals, sequences.
  - Importance: Useful for understanding growth rates in algorithmic analysis.
- **Basic Probability and Statistics**
  - Probability theory, random variables, distributions, expectation, variance.
  - Importance: Helpful for analyzing algorithms with randomness or probabilistic analysis.

# Definition of an Algorithm

**Definition:** An algorithm is a finite set of instructions that, if followed, accomplishes a particular task. In addition, it must meet the following criteria:

- **Input:** Takes zero or more inputs.
- **Output:** Produces at least one output.
- **Definiteness:** Each instruction must be clear and unambiguous.
- **Finiteness:** The algorithm should terminate after a finite number of steps.
- **Effectiveness:** Any instruction should be basic enough to be performed, ideally by a person using pencil and paper.

# Example: Making a Cup of Tea

Let's consider the steps to make a cup of tea as an example of an algorithm:

- **Input:**
  - Water
  - Tea leaves or a tea bag
  - A cup
  - Sugar (optional)
  - Milk (optional)
- **Output:** A prepared cup of tea.
- **Definiteness:** Each step is clear and unambiguous:
  - Boil water.
  - Add tea leaves or a tea bag to the cup.
  - Pour boiling water into the cup.
  - Steep for 3-5 minutes.
  - Add sugar and milk if desired.
  - Stir and serve.

# Example: Making a Cup of Tea

- **Finiteness:** The algorithm has a finite number of steps and will terminate once the tea is made.
- **Effectiveness:** Each step is simple enough to be performed by a person using basic tools (e.g., a kettle, a spoon).

# Focus Areas

1. **How to devise algorithms?**
   - Divide and Conquer
   - Greedy Algorithms
   - Dynamic Programming
   - Backtracking
   - Branch and Bound
2. **How to validate algorithms?**
   - Check if the algorithm is correct.
3. **Performance Analysis**
   - Time complexity
   - Space complexity
4. **How to test algorithms?**
   - Debugging
   - Profiling

# Performance Analysis

**Space Complexity:** The amount of memory an algorithm takes to run to completion.

- The space $S(P)$ required consists of two parts:
  1. **Fixed part** $(C)$**:** Independent of input size, e.g., instruction space, space for constants.
  2. **Variable part** $(S_p)$**:** Space needed for variables.

$$S(P) = C + S_p$$

**Time Complexity:** The amount of computational time an algorithm takes to run to completion.

# Space Complexity Example

**Example 1:**

**Algorithm abc(a, b, c)**

1: **return** $(a + b + c)$

**Space Complexity:** $S_p = 0$ (Fixed part only)

# Space Complexity Example

**Example 2:**

**Algorithm Sum(a, n)**

```
1: s ← 0
2: for i ← 1 to n do
3:       s ← s + a[i]
4: end for
5: return s
```

**Space Complexity:**

- Variables: $s$, $i$, $n$, $a[i]$
- Fixed part: $s$, $i$, $n$ which occupy 3 bytes (1 byte each).
- $S(P) = 3 + n$ (Fixed part + Variable part)

# Time Complexity: Finding a Book in a Library

Imagine you are in a library with 1,000 books, and you need to find a specific book.

# Method 1: Checking Each Book Individually

- Go book by book.
- Start with the first book, check if it's the one, then move on to the next, and so on.
- This approach is more straightforward: you check each book once.

**Summary**

You perform a linear search, going through the books one by one.

# Method 2: Dividing and Conquering the Search

- The books are arranged alphabetically by title.
- Go to the middle of the library and check if the book is to the left or right.
- Continue dividing the number of books you're searching through until you find the book.

**Summary**

You efficiently narrow down the search by repeatedly halving the number of books to check.

# From Intuition to Formalism

- We've seen different ways to search for a book, with varying levels of efficiency.
- How can we formalize these observations to compare them more easily?
- Let's introduce a tool to describe and compare these methods: a formal notation.

# The Role of Input Size

- In general, we are not so much interested in the time and space complexity for small inputs.
- For example, while the difference in time complexity between linear (method 1) and binary (method 2) search is meaningless for a sequence with $n = 10$, it is gigantic for $n = 2^{30}$.

# The Role of Input Size

## Question

Let us assume two algorithms A and B that solve the same class of problems.

- The time complexity of A is $5000n$, the one for B is $1.1^n$ for an input with $n$ elements.
- For $n = 10$, A requires $50,000$ steps, but B only 3, so B is superior to A.

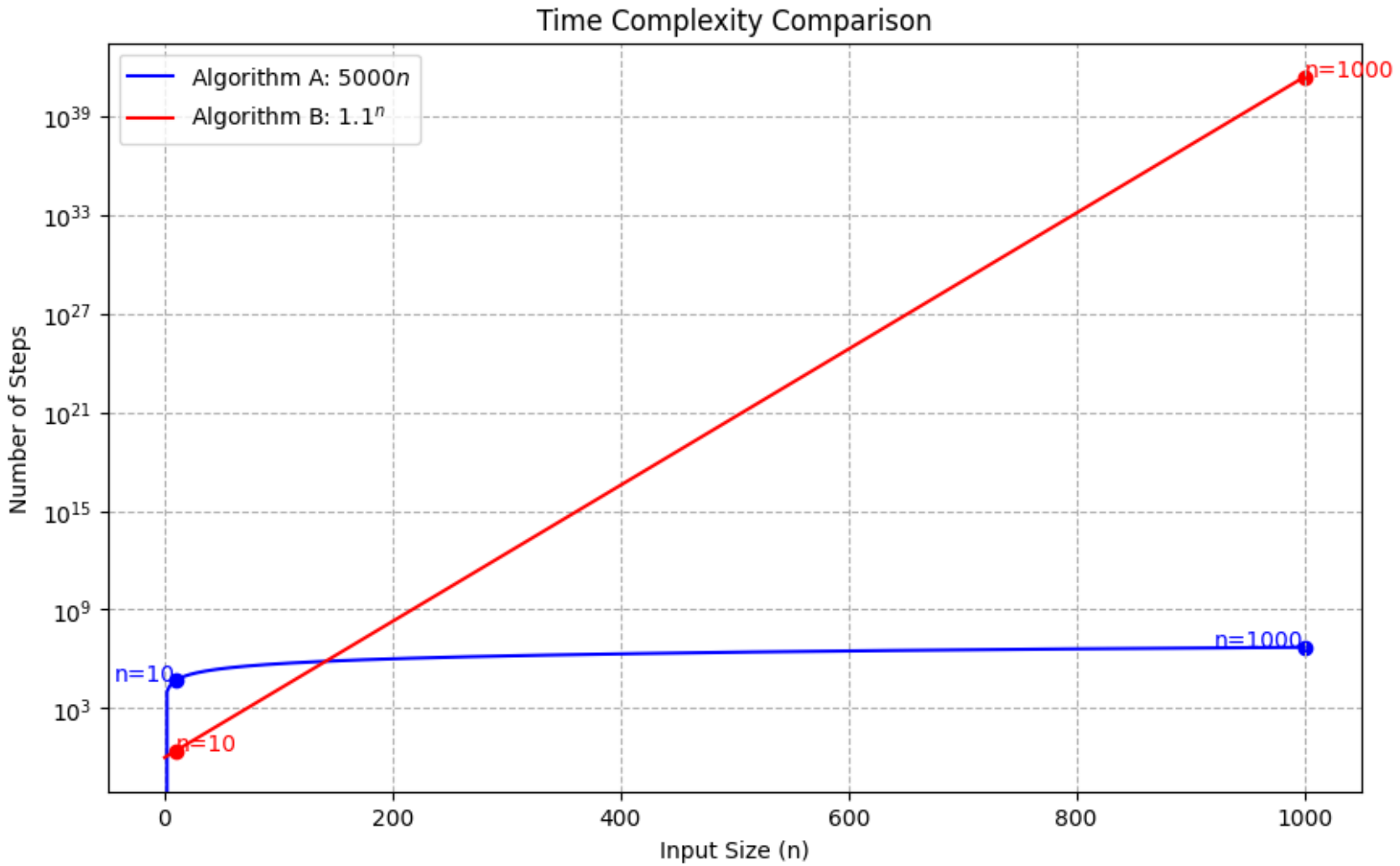
**Is this true or false? Justify your answer.**

# The Role of Input Size

**Answer: False**

- For $n = 1000$, A requires $5,000,000$ steps, while B requires $2.5 \times 10^{41}$ steps.
- While B appears better for small $n$, as $n$ grows, A becomes far more efficient.
- This demonstrates the importance of understanding how algorithms scale with input size.

# The Role of Input Size



# The Role of Input Size

- This means that algorithm B cannot be used for large inputs, while algorithm A is still feasible.
- So what is important is the **growth** of the complexity functions.
- The growth of time and space complexity with the increasing input size $n$ is a suitable measure for the comparison of algorithms.

# The Growth of Functions: Big-O Notation

- The growth of functions is usually described using the Big-O notation.
- Big-O Notation is a way to describe the efficiency of an algorithm in terms of how it scales with input size.
- It helps us categorize algorithms and understand their behavior as the problem size grows.

# Big-O Notation

**Big-O:** Let $f$ and $g$ be functions from the integers or the real numbers to the real numbers. The function $f(n)$ is $O(g(n))$ if there exist positive constants $C$ and $k$ such that:

$$|f(n)| \leq C \cdot |g(n)| \text{ for all } n > k$$

**Example:**

$$f(n) = 3n^2 + 2 \quad \text{is } O(n^2)$$

**Common Time Complexities:**

- $O(1)$ – Constant computing time
- $O(n)$ – Linear computing time
- $O(n^2)$ – Quadratic computing time
- $O(2^n)$ – Exponential computing time

# Big-O Notation

- When we analyze the growth of complexity functions, $f(n)$ and $g(n)$ are always positive. (**Homework: Why?**)
- We can simplify the Big-O requirement to:

$$f(n) \leq C \cdot g(n) \quad \text{whenever } n > k$$

- To show that $f(n)$ is $O(g(n))$, we only need to find one pair $(C, k)$.

# Example: Showing $f(n) = 3n + 2$ is $O(n)$

- Let $f(n) = 3n + 2$ and $g(n) = n$.
- Choose $C = 4$ and $k = 1$.
- For $n > 1$, $3n + 2 \leq 4n$, so $f(n)$ is $O(n)$.

**See the Visualization in Action:**

Click Here to View the Animation

# Big-O Notation

- The idea behind the Big-O notation is to establish an **upper boundary** for the growth of a function $f(n)$ for large $n$.
- This boundary is specified by a function $g(n)$ that is usually much **simpler** than $f(n)$.
- We accept the constant $C$ in the requirement $f(n) \leq C \cdot g(n)$ whenever $n > k$, because **C does not grow with n**.
- We are only interested in large $n$, so it is OK if $f(n) > C \cdot g(n)$ for $n \leq k$. (**Homework: Why?**)

# Big-O Notation

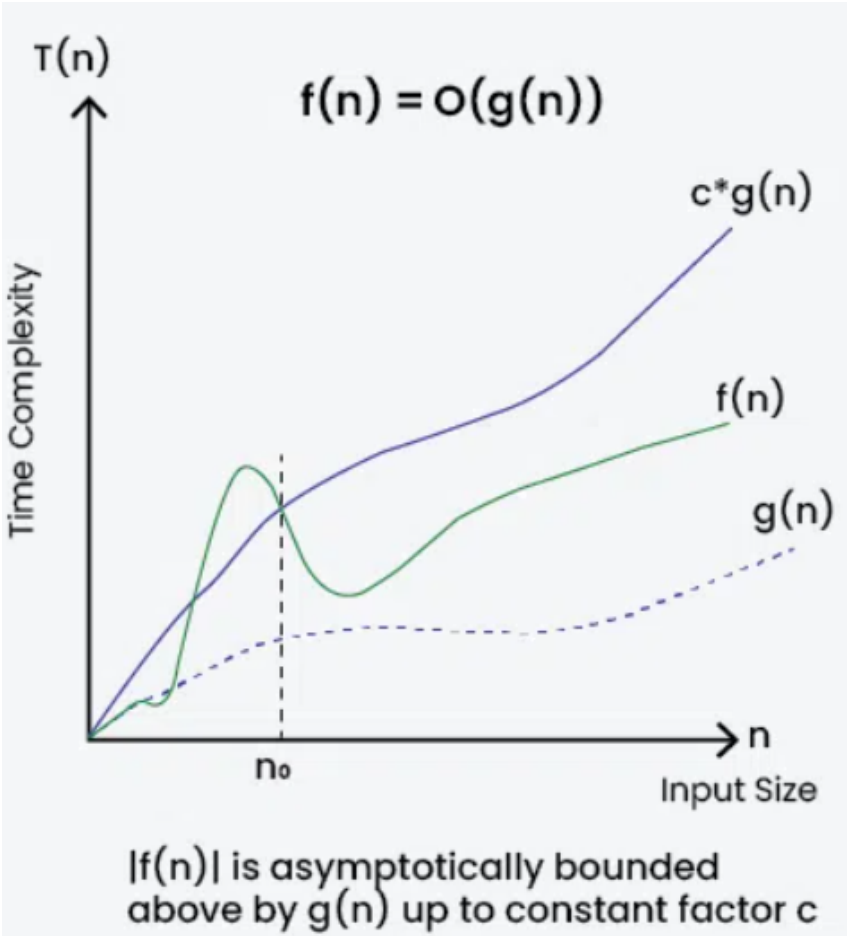



Figure: Plot for Big-O

# Big-O: Example

**Question**

Show that $f(n) = n^2 + 2n + 1$ is $O(n^2)$.

# Big-O: Example

## Solution

For $n > 1$, we have:

$$n^2 + 2n + 1 \leq n^2 + 2n^2 + n^2$$
$$\Rightarrow n^2 + 2n + 1 \leq 4n^2$$

Therefore, for $C = 4$ and $k = 1$:

$$f(n) \leq 4n^2 \quad \text{whenever } n > 1$$

$$\Rightarrow f(n) \text{ is } O(n^2).$$

**See the Visualization in Action:**

Click Here to View the Animation

# Properties of Big-O Notation

**Reflexivity:** For any function $f(n)$, $f(n) = O(f(n))$.

**Example**

If $f(n) = n^2$, then $f(n) = O(n^2)$.

# Properties of Big-O Notation

**Transitivity:** If $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$.

## Example

If $f(n) = n^3$, $g(n) = n^2$, $h(n) = n^4$, then:

- $f(n) = O(g(n))$
- $g(n) = O(h(n))$
- Therefore, $f(n) = O(h(n))$

# Properties of Big-O Notation

**Constant Factor:** For any constant $c > 0$ and functions $f(n)$ and $g(n)$, if $f(n) = O(g(n))$, then $cf(n) = O(g(n))$.

## Example

If $f(n) = n$, $g(n) = n^2$, then:

- $f(n) = O(g(n))$
- Therefore, $2f(n) = O(g(n))$

# Properties of Big-O Notation

**Sum Rule:** If $f(n) = O(g(n))$ and $h(n) = O(g(n))$, then $f(n) + h(n) = O(g(n))$.

## Example

If $f(n) = n^2$, $g(n) = n^3$, $h(n) = n^4$, then:

- $f(n) = O(g(n))$
- $h(n) = O(g(n))$
- Therefore, $f(n) + h(n) = O(g(n))$

# Properties of Big-O Notation

**Product Rule:** If $f(n) = O(g(n))$ and $h(n) = O(k(n))$, then $f(n) \times h(n) = O(g(n) \times k(n))$.

## Example

If $f(n) = n$, $g(n) = n^2$, $h(n) = n^3$, $k(n) = n^4$, then:

- $f(n) = O(g(n))$
- $h(n) = O(k(n))$
- Therefore, $f(n) \times h(n) = O(g(n) \times k(n)) = O(n^5)$

# Properties of Big-O Notation

**Composition Rule:** If $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(g(n)) = O(h(n))$.

## Example

If $f(n) = n^2$, $g(n) = n$, $h(n) = n^3$, then:

- $f(n) = O(g(n))$
- $g(n) = O(h(n))$
- Therefore, $f(g(n)) = O(h(n)) = O(n^3)$

# Common Functions in Big-O Notation

- Popular functions $g(n)$ are $n \log n, 1, 2^n, n^2, n!, n, n^3, \log n$.
- Listed from slowest to fastest growth:
  - $1$
  - $\log n$
  - $n$
  - $n \log n$
  - $n^2$
  - $n^3$
  - $2^n$
  - $n!$

# Steps to Determine Big-O Notation

1. **Identify the Dominant Term:**
   - Examine the function and identify the term with the highest order of growth as the input size increases.
   - Ignore any constant factors or lower-order terms.
2. **Determine the Order of Growth:**
   - The order of growth of the dominant term determines the Big-O notation.
3. **Write the Big-O Notation:**
   - The Big-O notation is written as $O(f(n))$, where $f(n)$ represents the dominant term.
   - For example, if the dominant term is $n^2$, the Big-O notation would be $O(n^2)$.
4. **Simplify the Notation (Optional):**
   - In some cases, the Big-O notation can be simplified by removing constant factors or by using a more concise notation.
   - For instance, $O(2n)$ can be simplified to $O(n)$.

# Example: Determining Big-O Notation

## Example

**Function:** $f(n) = 3n^3 + 2n^2 + 5n + 1$

1. **Dominant Term:** $3n^3$
2. **Order of Growth:** Cubic $(n^3)$
3. **Big-O Notation:** $O(n^3)$
4. **Simplified Notation:** $O(n^3)$