

# Time vs Space Complexity



if a function has a sum of 2nd and 3rd degree, then would the time complexity be only the 3rd degree and the 2nd degree would be ignored? Why?

Yes, if a function has a sum of **2nd-degree (quadratic) and 3rd-degree (cubic) terms**, the time complexity is determined **only by the highest-degree term** (i.e., the cubic term). The lower-degree term is ignored in Big-O notation.

## Why is the lower-degree term ignored?

Big-O notation focuses on **asymptotic behavior**—how the function grows as  $n \rightarrow \infty$ . The highest-degree term grows the fastest and dominates the function for large  $n$ .

### Example:

Suppose an algorithm has a time complexity:

$$T(n) = 5n^3 + 2n^2 + 7n + 10$$

For very large  $n$ , the  $n^3$  **term grows much faster** than  $n^2$ ,  $n$ , or the constant. Let's compare their growth:

$n$	$n^3$	$n^2$
10	1,000	100
100	1,000,000	10,000
1,000	1,000,000,000	1,000,000

Clearly,  $n^3$  **dominates** as  $n$  increases.

## Final Time Complexity:

$$T(n) = O(n^3)$$

We ignore **coefficients (like 5, 2, etc.)** and **lower-degree terms (like  $n^2$ ,  $n$ , and constants)** because they have negligible impact compared to  $n^3$  for large  $n$ .

Would you like a graphical comparison of how these terms grow? 😊



what is "asymptotic behavior" and what does it mean by "dominates the function for large  $n$ "?

## What is "Asymptotic Behavior"?

**Asymptotic behavior** refers to how a function behaves as its input  $n$  **becomes very large** ( $n \rightarrow \infty$ ). In **Big-O notation**, we care about the **long-term growth rate** of an algorithm rather than its exact values for small  $n$ .

For example:

- $f(n) = 3n^2 + 5n + 10$
- As  $n$  grows, the  $3n^2$  **term becomes much larger** than  $5n$  or  $10$ .
- So, the function behaves **asymptotically like**  $O(n^2)$ .

## What Does "Dominates the Function for Large $n$ " Mean?

A term **dominates** if it grows much faster than all other terms as  $n$  increases.

### Example: Comparing Growth Rates

Let's analyze three different terms:

- $n$  (linear growth)
- $n^2$  (quadratic growth)
- $n^3$  (cubic growth)

$n$	$n$	$n^2$	$n^3$
10	10	100	1,000
100	100	10,000	1,000,000
1,000	1,000	1,000,000	1,000,000,000

- **For small  $n$** , all terms contribute somewhat.
- **For large  $n$** ,  $n^3$  grows **way faster** than  $n^2$  or  $n$ , meaning it **dominates the function**.

Thus, if an algorithm has a complexity like  $T(n) = 2n^3 + 4n^2 + 10n + 50$ ,

- $2n^3$  **dominates**, so we write  $O(n^3)$ .
- The lower-order terms are **ignored in Big-O notation**.

## Intuition with a Real-Life Example

Think of it like comparing different speeds:

- **Walking speed** (~5 km/h)
- **Car speed** (~100 km/h)
- **Airplane speed** (~900 km/h)

If you travel a long distance, **airplane speed dominates**—the walking speed becomes irrelevant compared to the airplane.

Would you like a graph to visualize this concept? 😊



"The lower-order terms are ignored in Big-O notation." does it refer to the **code blocks of lower time complexity that runs parallel to the high time complexity coded blocks**? for example, if two loops are to run separately and one is a nested loop, then the time complexity of the non-nested loop would be ignored and only the nested loop's time complexity will be counted?

Yes! You've got the right intuition. **In Big-O notation, the lower-order terms (or less dominant code blocks) are ignored, especially when they run separately.**

## Example 1: Parallel Code Blocks with Different Complexities

Consider this **pseudo-code**:

```
python

# First loop (O(n))
for i in range(n):
    print(i)

# Nested loop (O(n^2))
for i in range(n):
    for j in range(n):
        print(i, j)
```

### Step-by-step Time Complexity Analysis:

1. First loop runs  $O(n)$  times.
2. Second loop runs  $O(n^2)$  times (since it's nested).
3. Total complexity:

$$O(n) + O(n^2) = O(n^2)$$

4. Since  $n^2$  **dominates**  $n$  for large  $n$ , we ignore  $O(n)$  and simply write:

$$O(n^2)$$

## Example 2: Two Nested Loops vs. Single Loop

```
python

# O(n^2) nested loop
for i in range(n):
    for j in range(n):
        print(i, j)

# O(n) loop
for k in range(n):
    print(k)
```

- The **nested loop runs  $O(n^2)$  times.**

- The **separate loop runs  $O(n)$  times.**
- The final complexity is  $O(n^2) + O(n) = O(n^2) \rightarrow O(n)$  is ignored.

## General Rule for Complexity Calculation

1. **If loops run sequentially**, add their complexities.
  - Example:  $O(n) + O(n^2) = O(n^2)$  (ignore the smaller term)
2. **If loops are nested**, multiply their complexities.
  - Example:  $O(n) \times O(n) = O(n^2)$
3. **Ignore constants and lower-order terms in Big-O notation.**

So yes, when two loops run **separately**, and one has a much higher complexity, we ignore the lower one because **it doesn't significantly impact performance for large  $n$ .**

Would you like me to show a visualization for this? 😊



can you share some references for this "ignoring" procedure?

Yes! The idea of **ignoring lower-order terms** and **constant factors** in **Big-O notation** is a fundamental concept in **asymptotic analysis**. Here are some standard references that explain why we do this:

## 1. Introduction to the Design and Analysis of Algorithms – Anany Levitin

### Chapter: Growth of Functions & Asymptotic Notation

- Discusses how Big-O focuses on dominant terms because they **determine runtime for large inputs.**
- **Quote:**

"In asymptotic analysis, the highest-order term dominates the growth rate of a function as  $n$  becomes large. **Lower-order terms become negligible.**"

 [Book Link \(Google Books\)](#)

## 2. Introduction to Algorithms (CLRS) – Cormen, Leiserson, Rivest, Stein

### Chapter 3: Growth of Functions

- Explains how **lower-order terms and constant coefficients** are ignored in Big-O notation.
- **Example from the book:**
  - If  $f(n) = 6n^2 + 100n + 3000$ , then we write  $O(n^2)$  because  $n^2$  dominates for large  $n$ .
- **Quote:**

"Big-O notation describes an upper bound on the growth of a function, making constant factors and lower-degree terms irrelevant."

 [Book Link \(MIT Press\)](#)

---

### 3. Algorithm Design Manual – Steven Skiena

#### Chapter: Complexity Analysis

- Explains **why asymptotic notation ignores small terms**
- **Example:**
  - If an algorithm has complexity  $O(n^3 + 5n^2 + 20n + 100)$ , the  $O(n^3)$  **term dominates**.
- **Quote:**

"Asymptotic analysis ignores details that are insignificant for large inputs. The highest-degree term dominates the function's growth."

 [Book Link \(Springer\)](#)

---

### 4. Online Resources

#### MIT OpenCourseWare – Introduction to Algorithms

- Video Lecture: [Asymptotic Analysis \(MIT 6.006\)](#)
  -  **CS50 by Harvard University** – Growth of Algorithms
  - Course Link: [CS50 Lectures](#)
- 

### Final Summary

#### Why do we ignore lower-order terms?

- Because for large  $n$ , they become **insignificant** compared to the highest-order term.
- Big-O focuses on **worst-case growth**, and **smaller terms don't affect the overall trend**.

Would you like me to elaborate on any specific reference? 😊