**Data Structures and Algorithms (DSA)**

Table of Contents

[Introduction 2](#_Toc202903013)

[Data Structures: 2](#_Toc202903014)

[Algorithms: 2](#_Toc202903015)

[Relationship: 2](#_Toc202903016)

[Importance: 2](#_Toc202903017)

[Database 3](#_Toc202903018)

[Data warehouse 3](#_Toc202903019)

[Big Data 3](#_Toc202903020)

[The memory layout of a C program 4](#_Toc202903021)

[Text Segment (Code Segment): 4](#_Toc202903022)

[Data Segment: 4](#_Toc202903023)

[BSS Segment (Block Started by Symbol): 4](#_Toc202903024)

[Heap: 4](#_Toc202903025)

[Stack: 4](#_Toc202903026)

[Key Differences Between Stack and Heap Allocations 5](#_Toc202903027)

[Comparison Chart 6](#_Toc202903028)

[Asymptotic Analysis 6](#_Toc202903029)

[Given two algorithms for a task, how do we find out which one is better? 6](#_Toc202903030)

[For example, 7](#_Toc202903031)

[Running times for this example: 8](#_Toc202903032)

[Does Asymptotic Analysis always work? 8](#_Toc202903033)

[Asymptotic Notations: 8](#_Toc202903034)

[Big O Notation Tutorial - A Guide to Big O Analysis 8](#_Toc202903035)

[BIg O Definition 9](#_Toc202903036)

[Importance of Big O Notation 10](#_Toc202903037)

[A Quick Way to find Big O of an Expression 10](#_Toc202903038)

[Analysis of Algorithms | Θ (Theta) Notation 10](#_Toc202903039)

[Definition 11](#_Toc202903040)

[Mathematical Representation: 11](#_Toc202903041)

[Graphical Representation: 11](#_Toc202903042)

[Analysis of Algorithms | Big-Omega Ω Notation 12](#_Toc202903043)

[What is Big-Omega Ω Notation? 12](#_Toc202903044)

[Definition of Big-Omega Ω Notation? 13](#_Toc202903045)

[Big O vs Theta Θ vs Big Omega Ω Notations 13](#_Toc202903046)

[1. Big O notation (O): 13](#_Toc202903047)

[2. Big Omega notation (Ω) : 14](#_Toc202903048)

[3. Theta notation (Θ) : 15](#_Toc202903049)

[Difference Between Big oh, Big Omega and Big Theta : 16](#_Toc202903050)

# Introduction

## Data Structures:

* **Definition:** Data structures are ways of organizing and storing data in a computer so that it can be used efficiently.
* **Examples:** Arrays, linked lists, stacks, queues, trees, graphs, hash tables.
* **Purpose:** To provide efficient ways to access, modify, and manage data.

## Algorithms:

* **Definition:**

Algorithms are sets of instructions or rules that specify how to solve a particular problem.

* **Examples:**

Sorting algorithms (like bubble sort, merge sort), searching algorithms (like binary search), graph algorithms (like Dijkstra's algorithm).

* **Purpose:**

To provide a systematic approach to problem-solving, often involving the manipulation of data structures.

## Relationship:

* Data structures and algorithms often work together.
* A good algorithm often relies on an efficient data structure for optimal performance.
* For example, searching for an element in a sorted array can be done efficiently using binary search (algorithm) on an array (data structure).

## Importance:

* **Efficiency:**

Using the right data structure and algorithm can significantly improve the speed and efficiency of a program, especially when dealing with large amounts of data.

* **Problem-solving:**

Data structures and algorithms provide the tools to break down complex problems into smaller, manageable steps.

* **Code optimization:**

Understanding these concepts helps in writing optimized code that is both efficient and maintainable.

* **Interviews:**

Many tech companies use coding interviews that assess a candidate's knowledge of data structures and algorithms.

# Database

A database is an electronically stored, systematic collection of data. It can contain any type of data, including words, numbers, images, videos, and files. You can use software called a database management system (DBMS) to store, retrieve, and edit data.

# Data warehouse

A data warehouse is a system used for reporting and data analysis, serving as a central repository for large amounts of historical data from various sources. It's a core component of business intelligence (BI) and helps organizations make informed decisions by providing a single, consistent view of their data.

# Big Data

Big data refers to extremely large and complex datasets that exceed the capacity of traditional data processing systems. These datasets are characterized by their volume, velocity, and variety, making them challenging to store, manage, and analyse using conventional methods.

# The memory layout of a C program

The memory layout of a C program refers to how its various components are organized within the computer's memory during execution. This organization is typically divided into several segments:

## Text Segment (Code Segment):

This segment stores the compiled machine code of the program's instructions. It is usually read-only to prevent accidental modification during runtime and can be shared among multiple instances of the same program.

## Data Segment:

This segment holds initialized global and static variables. It is a read-write segment, meaning the values of these variables can be modified during program execution. This segment also includes initialized constant variables, which are typically stored in a read-only area within the data segment.

## BSS Segment (Block Started by Symbol):

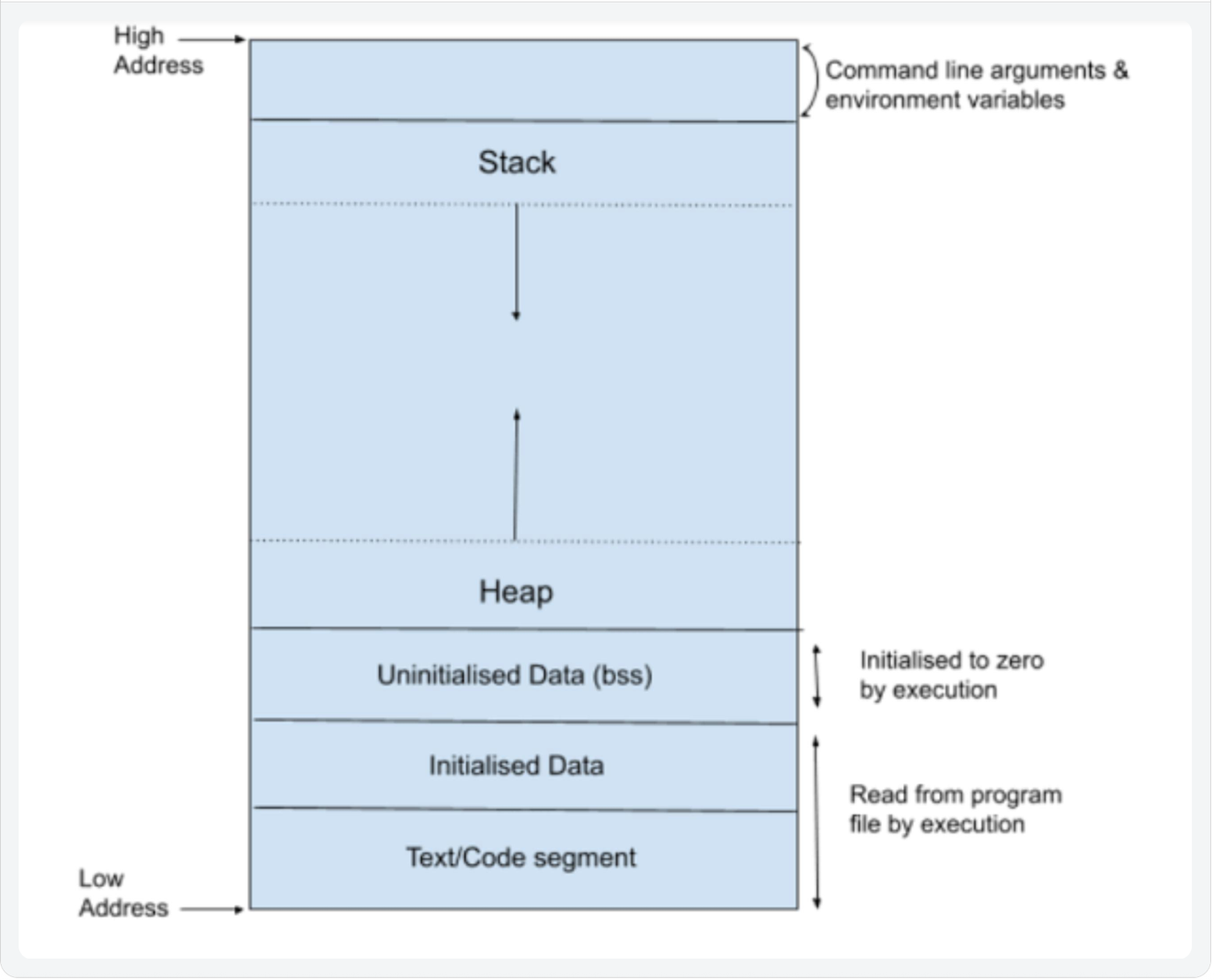
This segment stores uninitialized global and static variables. The operating system initializes the memory in this segment to zero before the program starts execution. Like the data segment, it is a read-write segment.

## Heap:

This segment is used for dynamic memory allocation, where memory is requested and released by the program during runtime using functions like malloc(), calloc(), realloc(), and free(). The heap grows upwards in memory addresses.

## Stack:

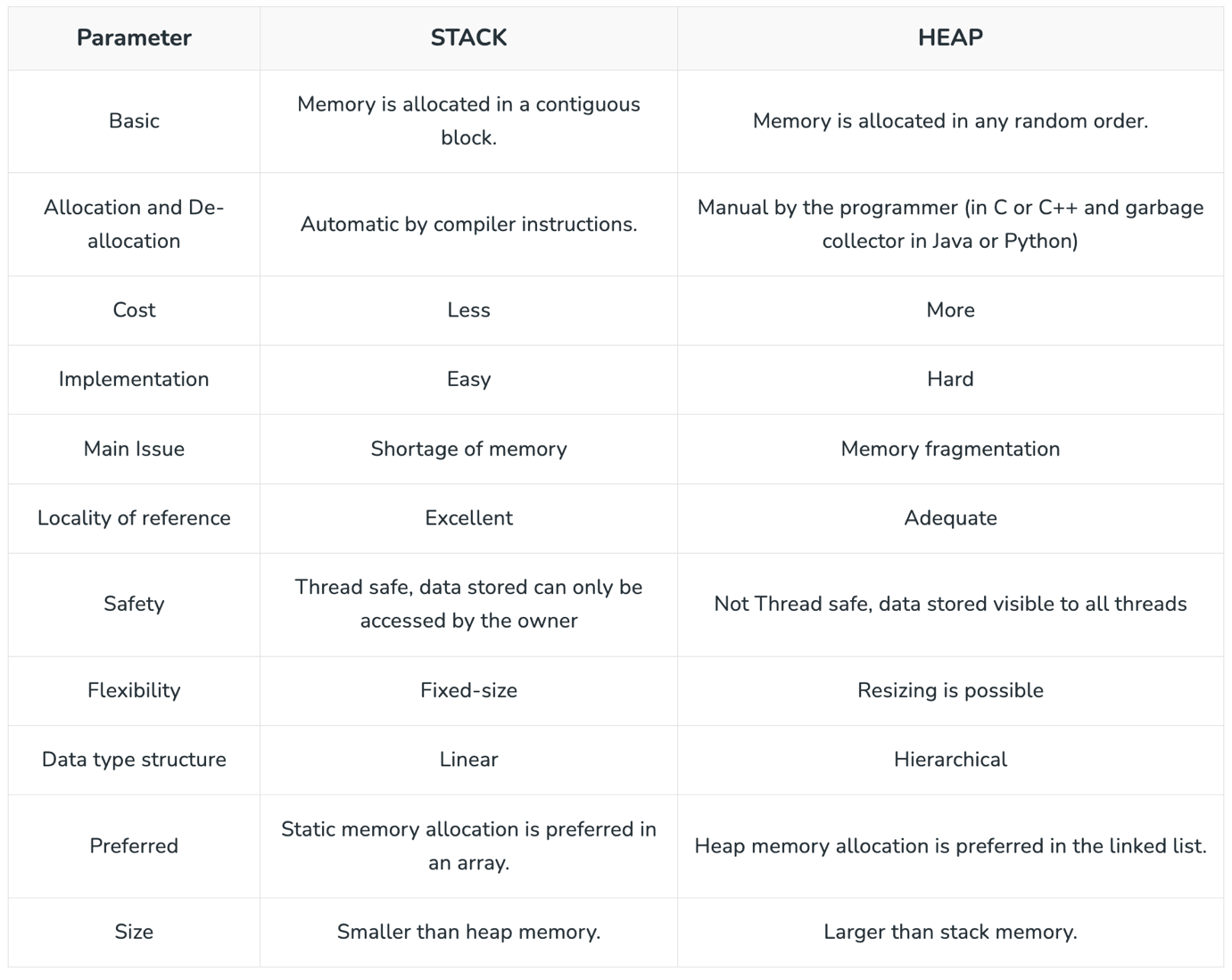
This segment is used for local variables, function arguments, and return addresses during function calls. It operates on a Last-In, First-Out (LIFO) principle, where new data is pushed onto the top of the stack and removed when no longer needed (e.g., when a function returns). The stack typically grows downwards in memory addresses.



## **Key Differences Between Stack and Heap Allocations**

1. In a stack, the allocation and de-allocation are **automatically** done by the compiler whereas, in heap, it needs to be done by the **programmer manually.**
2. Handling the Heap frame is **costlier** than handling the stack frame.
3. **Memory shortage** problem is more likely to happen in stack whereas the main issue in heap memory is **fragmentation.**
4. Stack **frame access is easier**than the heap frame as the stack has a small region of memory and is **cache-friendly** but in the case of heap frames which are dispersed throughout the memory so it causes more **cache misses.**
5. A stack is **not flexible**, the memory size allotted cannot be changed whereas a heap is flexible, and the allotted memory can be **altered.**
6. **Accessing** **time** of heap takes is more than a stack.

## **Comparison Chart**



# Asymptotic Analysis

## Given two algorithms for a task, how do we find out which one is better?

One naive way of doing this is - to implement both the algorithms and run the two programs on your computer for different inputs and see which one takes less time. There are many problems with this approach for the analysis of algorithms.

* It might be possible that for some **inputs**, the first algorithm performs better than the second. And for some **inputs** second performs better.
* It might also be possible that for some inputs, the first algorithm performs better on one **machine**, and the second works better on another **machine** for some other inputs.

For example, let us consider the search problem (searching a given item) in a sorted array.

The solution to above search problem includes:

* [**Linear Search**](https://www.geeksforgeeks.org/linear-search/) (order of growth is linear)
* [**Binary Search**](https://www.geeksforgeeks.org/binary-search/) (order of growth is logarithmic).

To understand how Asymptotic Analysis solves the problems mentioned above in analyzing algorithms,

* let us say:
  + We run the Linear Search on computer A and
  + Binary Search on computer B and
* For small values of input array size n, computer A may take less time.
* But, after a certain value of input array size, the Binary Search will definitely start taking less time compared to the Linear Search even though the Binary Search is being run on a slow machine.  Why? After certain value, the machine specific factors would not matter as the value of input would become large.
* The reason is the order of growth of Binary Search with respect to input size is logarithmic while the order of growth of Linear Search is linear.
* **So the machine-dependent constants can always be ignored after a certain value of input size.**
* Let’s say the constant for machine A is 0.2 and the constant for B is 1000 which means that A is 5000 times more powerful than B.

| **Input Size** | **Running time on A** | **Running time on B** |
| --- | --- | --- |
| **10** | 2 sec | ~ 1 h |
| **100** | 20 sec | ~ 1.8 h |
| **10^6** | ~ 55.5 h | ~ 5.5 h |
| **10^9** | ~ 6.3 years | ~ 8.3 h |

## Running times for this example:

* Linear Search running time in seconds on A: 0.2 \* n
* Binary Search running time in seconds on B: 1000\*log(n)

## Does Asymptotic Analysis always work?

Asymptotic Analysis is not perfect, but that's the best way available for analysing algorithms. For example, say there are two sorting algorithms that take 1000nLogn and 2nLogn time respectively on a machine. Both of these algorithms are asymptotically the same (order of growth is nLogn). So, With Asymptotic Analysis, we can't judge which one is better as we ignore constants in Asymptotic Analysis. For example, asymptotically [Heap Sort](https://www.geeksforgeeks.org/heap-sort/)is better than [Quick Sort](https://www.geeksforgeeks.org/quick-sort-algorithm/), but Quick Sort takes less time in practice.

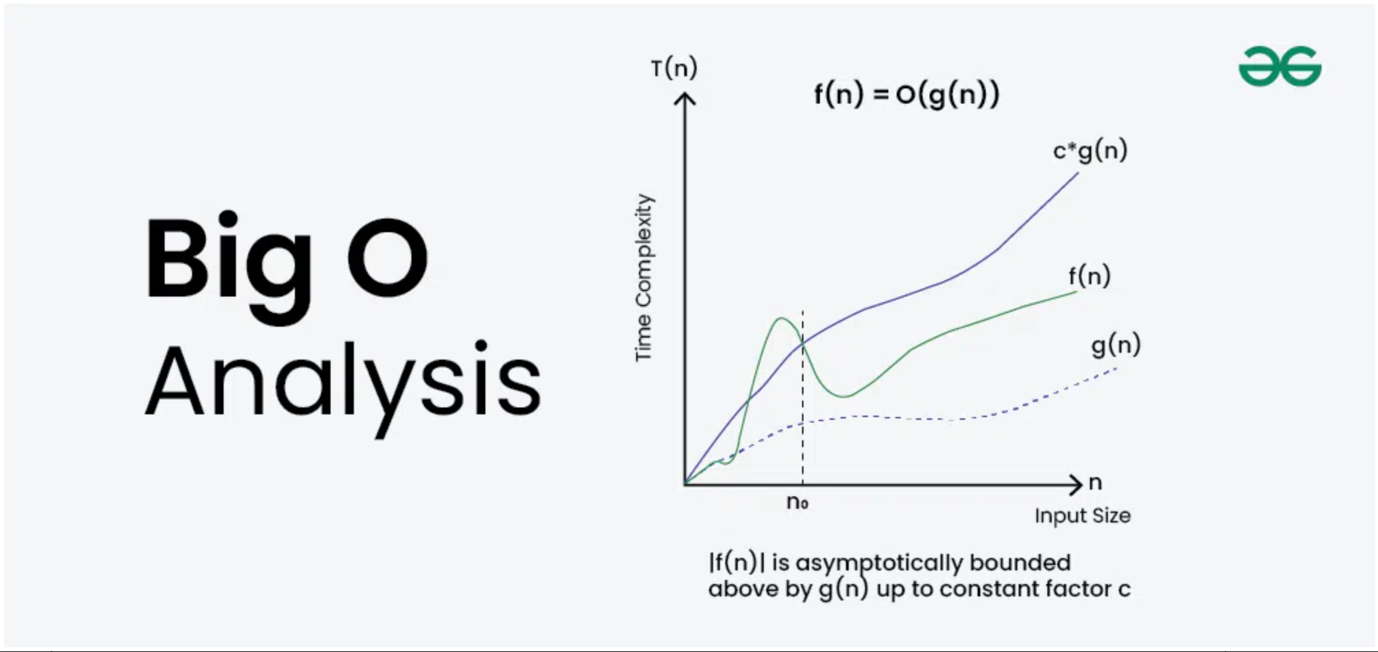
Also, in Asymptotic analysis, we always talk about input sizes larger than a constant value. It might be possible that those large inputs are never given to your software and an asymptotically slower algorithm always performs better for your particular situation. So, you may end up choosing an algorithm that is Asymptotically slower but faster for your software.

# **Asymptotic Notations:**

## Big O Notation Tutorial - A Guide to Big O Analysis

**Big O notation**is a powerful tool used in computer science to describe the time complexity or space complexity of algorithms. **Big-O** is a way to express the **upper bound**of an algorithm’s time or space complexity.

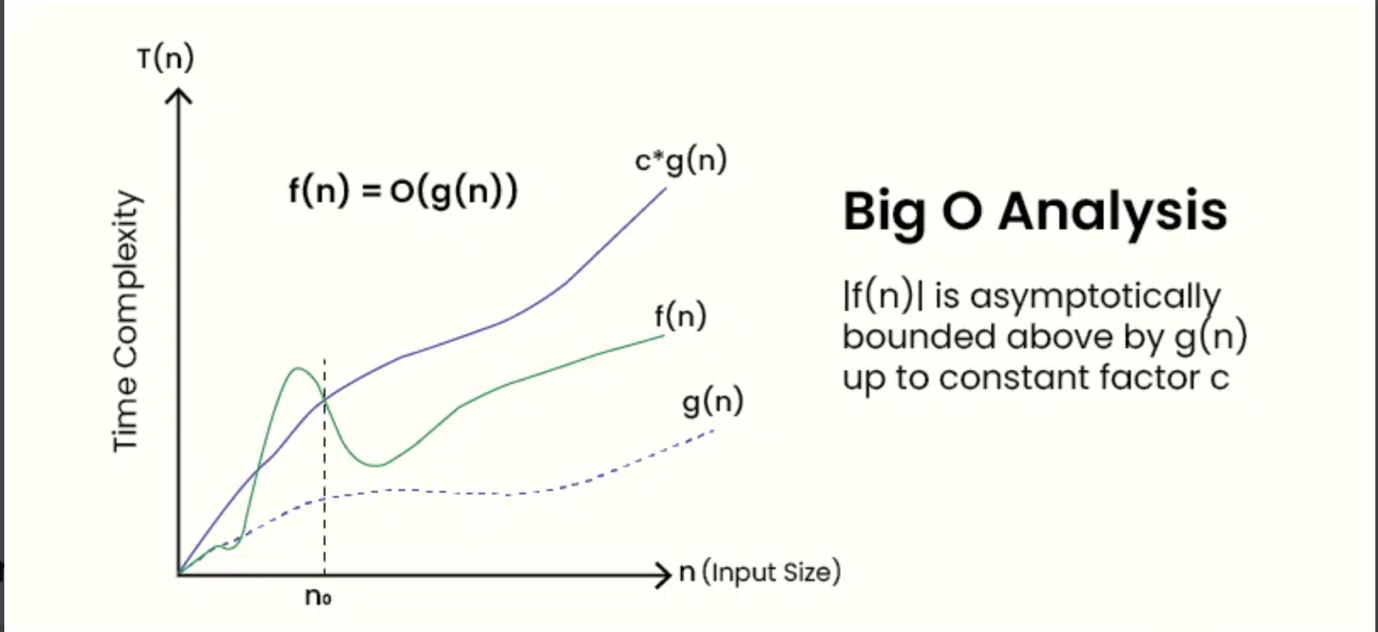
* Describes the asymptotic behavior (order of growth of time or space in terms of input size) of a function, not its exact value.
* Can be used to compare the efficiency of different algorithms or data structures.
* It provides an**upper limit** on the time taken by an algorithm in terms of the size of the input. We mainly consider the worst case scenario of the algorithm to find its time complexity in terms of Big O
* It’s denoted as**O(f(n))**, where**f(n)** is a function that represents the number of operations (steps) that an algorithm performs to solve a problem of size **n**.



### BIg O Definition

Given two functions**f(n)** and **g(n)**, we say that**f(n)** is**O(g(n))** if there exist constants**c > 0** and **n0 >=**0 such that**f(n) <= c\*g(n)** for all **n >= n0**.

In simpler terms,**f(n)** is **O(g(n))** if**f(n)** grows no faster than**c\*g(n)** for all n >= n0 where c and n0 are constants.



### Importance of Big O Notation

Big O notation is a mathematical notation used to find an upper bound on time taken by an algorithm or data structure. It provides a way to compare the performance of different algorithms and data structures, and to predict how they will behave as the input size increases.

Big O notation is important for several reasons:

* Big O Notation is important because it helps analyze the efficiency of algorithms.
* It provides a way to describe how the**runtime**or **space requirements** of an algorithm grow as the input size increases.
* Allows programmers to compare different algorithms and choose the most efficient one for a specific problem.
* Helps in understanding the scalability of algorithms and predicting how they will perform as the input size grows.
* Enables developers to optimize code and improve overall performance.

### A Quick Way to find Big O of an Expression

* Ignore the lower order terms and consider only highest order term.
* Ignore the constant associated with the highest order term**.**

Example 1*: f(n) = 3n2 + 2n + 1000Logn + 5000  
After ignoring lower order terms, we get the highest order term as 3n2  
After ignoring the constant 3, we get n2  
Therefore the Big O value of this expression is O(n2)*

Example 2 :*f(n) = 3n3 + 2n2 + 5n + 1  
Dominant Term: 3n3  
Order of Growth: Cubic (n3)  
Big O Notation: O(n3)*

## Analysis of Algorithms | Θ (Theta) Notation

In the [analysis of algorithms](https://www.geeksforgeeks.org/analysis-of-algorithms-set-1-asymptotic-analysis/), asymptotic notations are used to evaluate the performance of an algorithm by providing an exact order of growth. This article will discuss Big - Theta notations represented by a Greek letter (Θ).

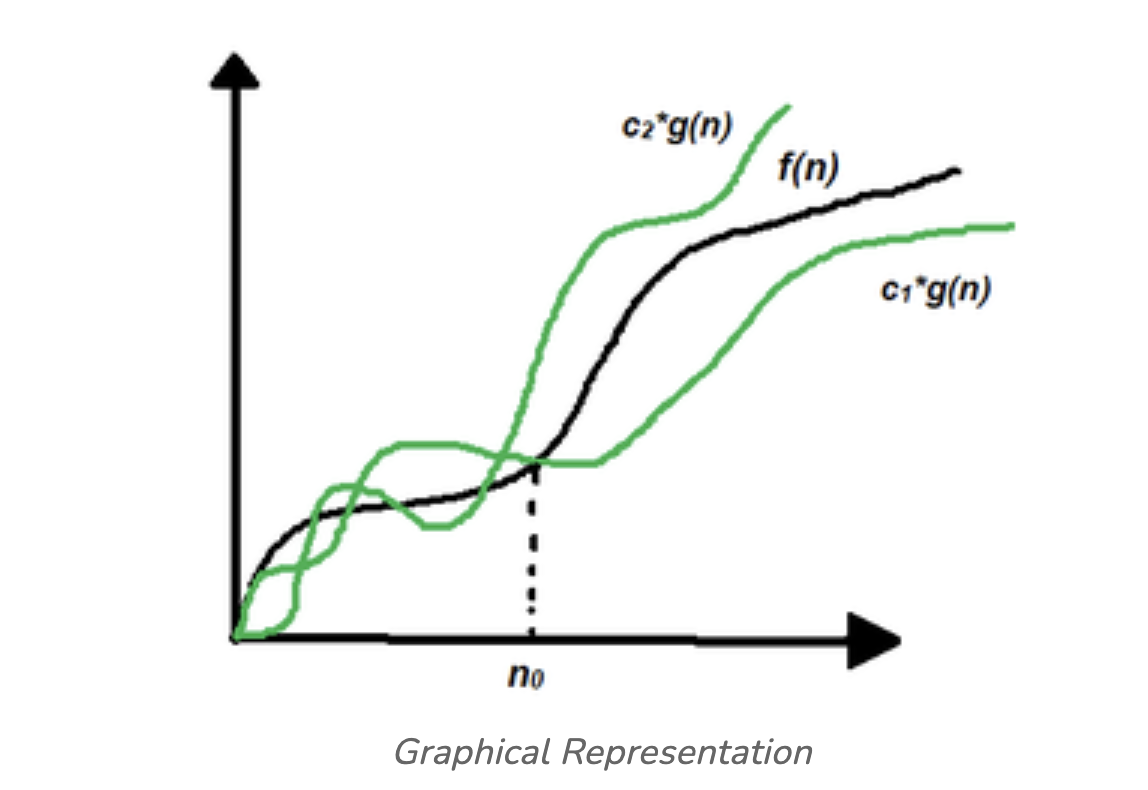
Definition**:** Let g and f be the function from the set of natural numbers to itself. The function f is said to be Θ(g), if there are constants c1, c2 > 0 and a natural number n0 such that c1\* g(n) ≤ f(n) ≤ c2 \* g(n) for all n ≥ n0

### **Mathematical Representation:**

*Θ (g(n)) = {f(n): there exist positive constants c1, c2 and n0 such that 0 ≤ c1 \* g(n) ≤ f(n) ≤ c2 \* g(n) for all n ≥ n0}  
Note: Θ(g) is a set*

The above definition means, if f(n) is theta of g(n), then the value f(n) is always between c1 \* g(n) and c2 \* g(n) for large values of n (n ≥ n0). The definition of theta also requires that f(n) must be non-negative for values of n greater than n0.

### **Graphical Representation:**



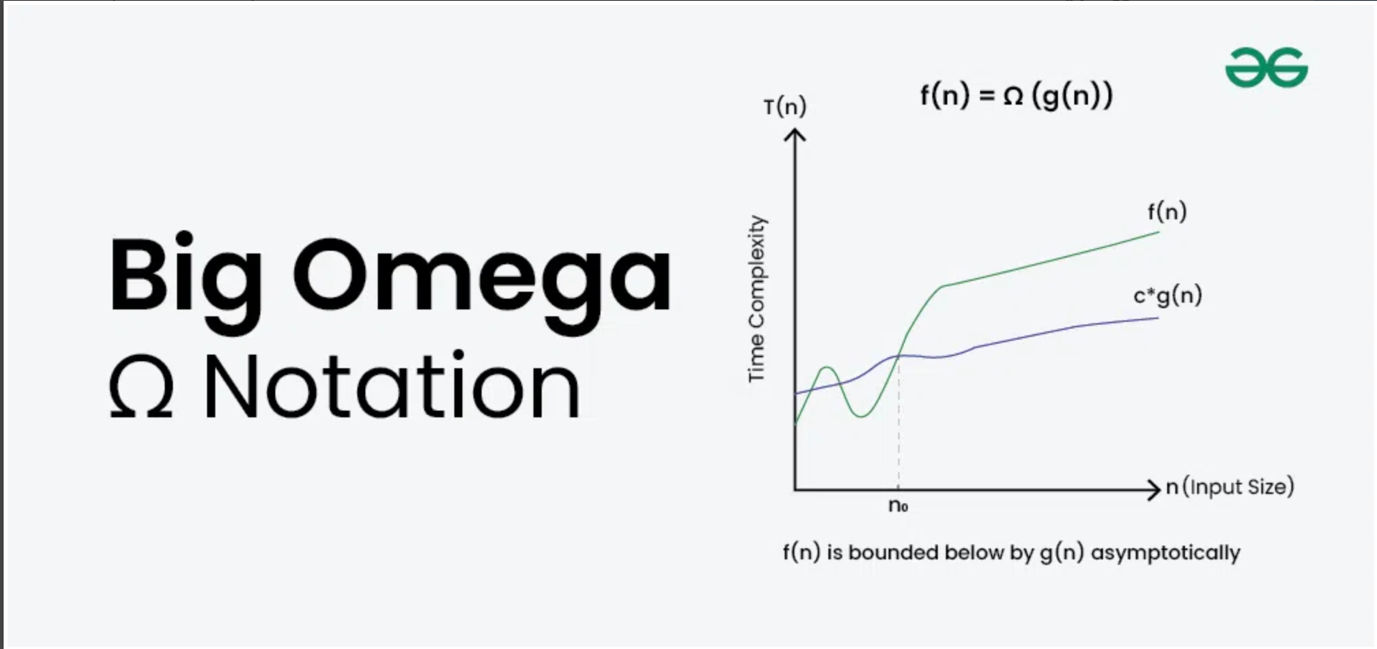
In simple language, Big - Theta(Θ) notation specifies asymptotic bounds (both upper and lower) for a function f(n) and provides the average time complexity of an algorithm.

Follow the steps below to find the average time complexity of any program:

1. Break the program into smaller segments.
2. Find all types and number of inputs and calculate the number of operations they take to be executed. Make sure that the input cases are equally distributed.
3. Find the sum of all the calculated values and divide the sum by the total number of inputs let say the function of n obtained is g(n) after removing all the constants, then in Θ notation its represented as Θ(g(n))

## Analysis of Algorithms | Big-Omega Ω Notation

In the [analysis of algorithms](https://www.geeksforgeeks.org/analysis-of-algorithms-set-1-asymptotic-analysis/), asymptotic notations are used to evaluate the performance of an algorithm, in its [best cases and worst cases](https://www.geeksforgeeks.org/analysis-of-algorithms-set-2-asymptotic-analysis/). This article will discuss Big-Omega Notation represented by a Greek letter (Ω).



### What is Big-Omega Ω Notation?

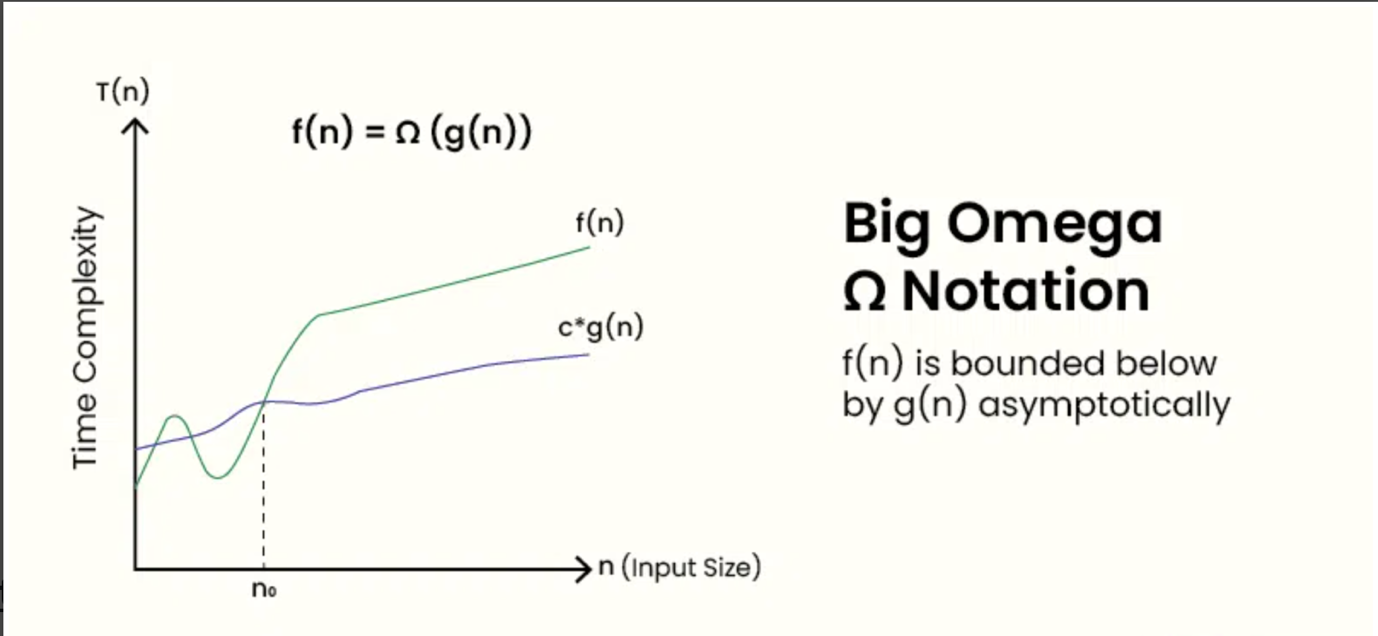
**Big-Omega Ω Notation**, is a way to express the **asymptotic lower bound**of an algorithm’s time complexity, since it analyses the**best-case** situation of algorithm. It provides a **lower limit** on the time taken by an algorithm in terms of the size of the input. It’s denoted as**Ω(f(n))**, where**f(n)** is a function that represents the number of operations (steps) that an algorithm performs to solve a problem of size **n**.

Big-Omega **Ω**Notation is used when we need to find the **asymptotic lower bound** of a function. In other words, we use Big-Omega **Ω** when we want to represent that the algorithm will take**at least** a certain amount of time or space.

### Definition of Big-Omega Ω Notation?

Given two functions **g(n)** and **f(n)**, we say that**f(n) =** **Ω(g(n))**, if there exists constants **c > 0** and **n0 >=**0 such that**f(n) >= c\*g(n)** for all **n >= n0**.

In simpler terms,**f(n)** is **Ω(g(n))** if**f(n)**will always grow faster than**c\*g(n)** for all n >= n0 where c and n0 are constants.



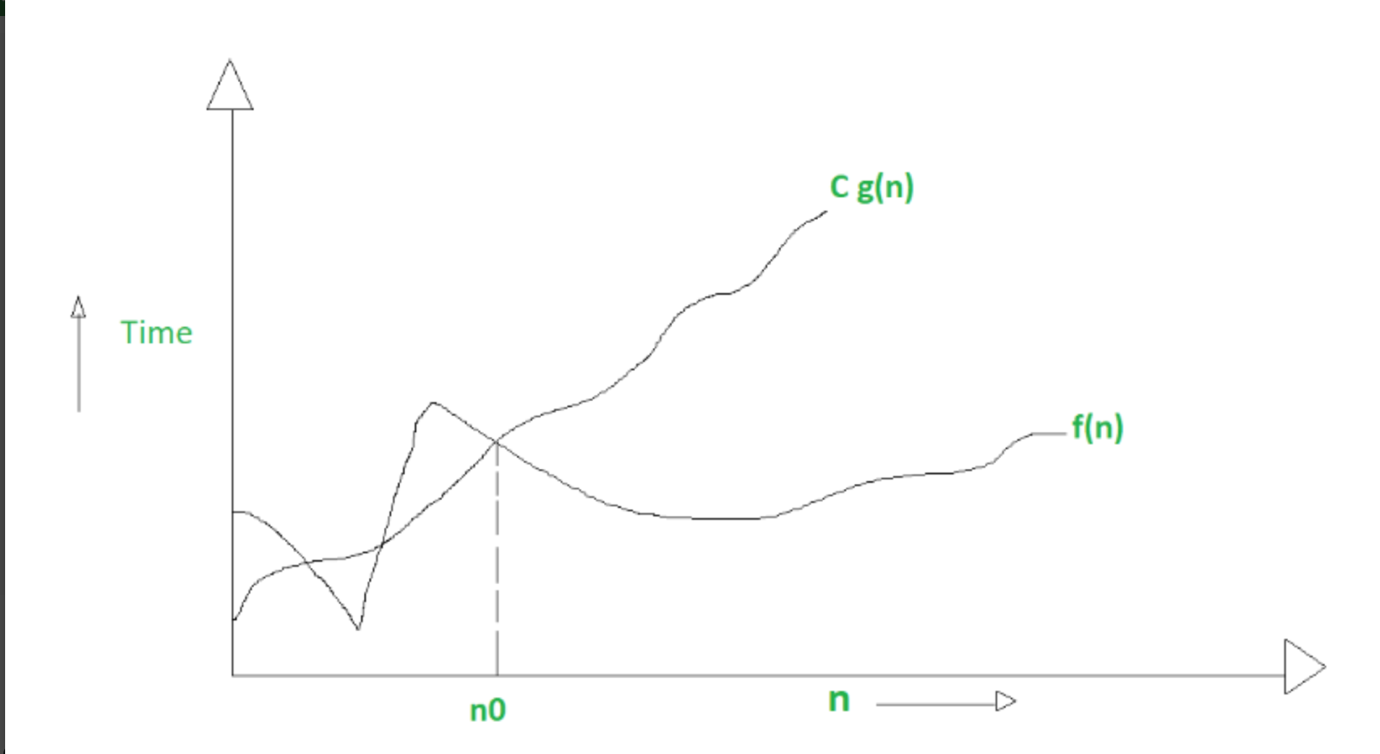
## Big O vs Theta Θ vs Big Omega Ω Notations

### 1. Big O notation (O):

It defines an upper bound on order of growth of time taken by an algorithm or code with input size. Mathematically, if**f(n)** describes the running time of an algorithm; **f(n)** is **O(g(n))** if there exist positive constant **C** and **n0** such that,

*0 <= f(n) <= Cg(n) for all n >= n0*

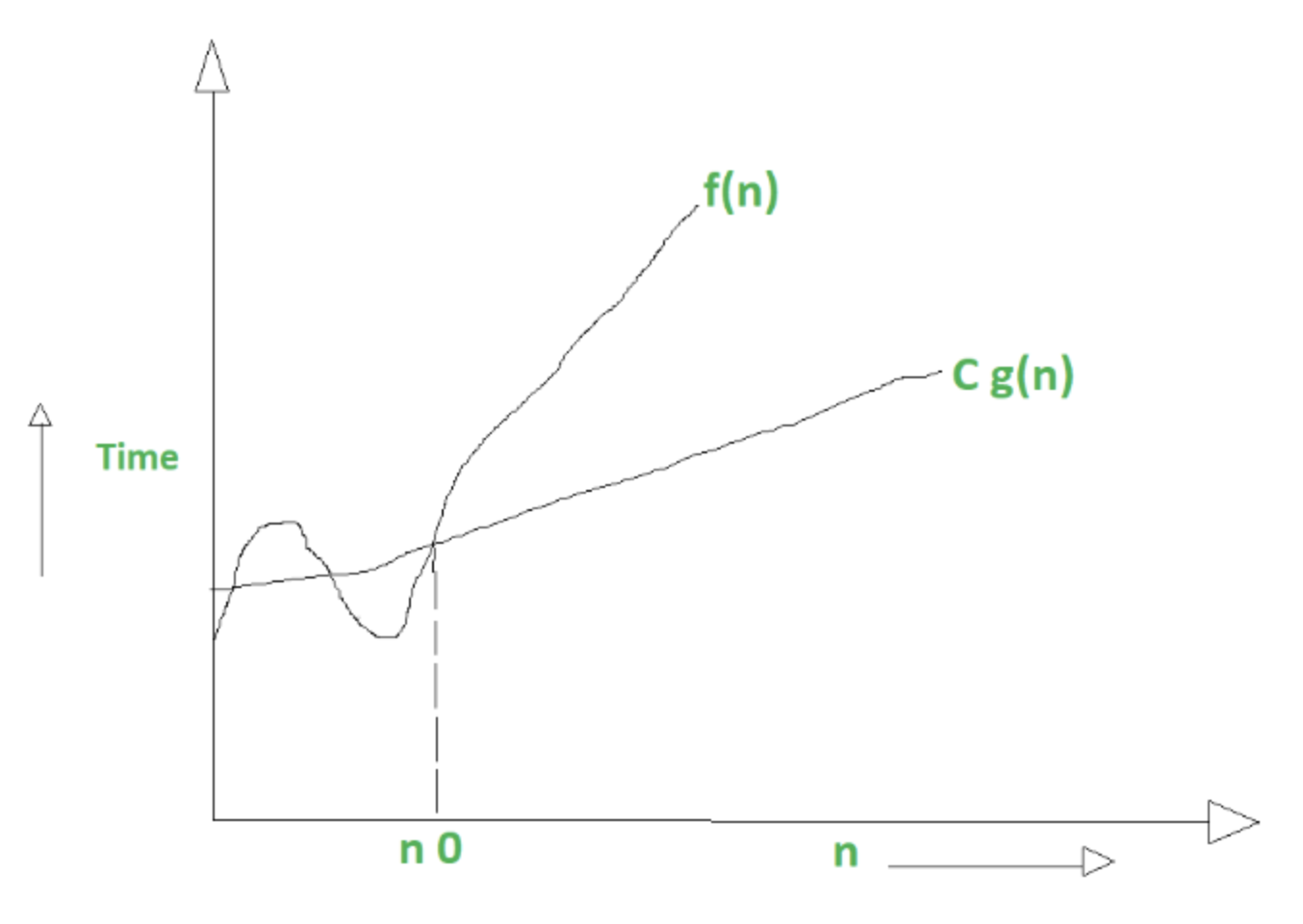
**n** = used to give upper bound a function.   
If a function is **O(n)**, it is automatically **O(n-square)** as well.



2. Big Omega notation (Ω) : It defines a lower bound on order of growth of time taken by an algorithm or code with input size. Let **f(n)** define running time of an algorithm;  
**f(n)** is said to be**Ω(g (n))** if there exists positive constant **C** and **(n0)** such that

*0 <= Cg(n) <= f(n) for all n >= n0*

**n** = used to given lower bound on a function   
If a function is **Ω(n-square)** it is automatically **Ω(n)** as well.



### 3. Theta notation (Θ) :

It defines exact order of growth of time taken by an algorithm or code with input size. Let **f(n)** define running time of an algorithm. **f(n)** is said to be **Θ(g(n))** if **f(n)** is **O(g(n))** and**f(n)** is **Ω(g(n)).**

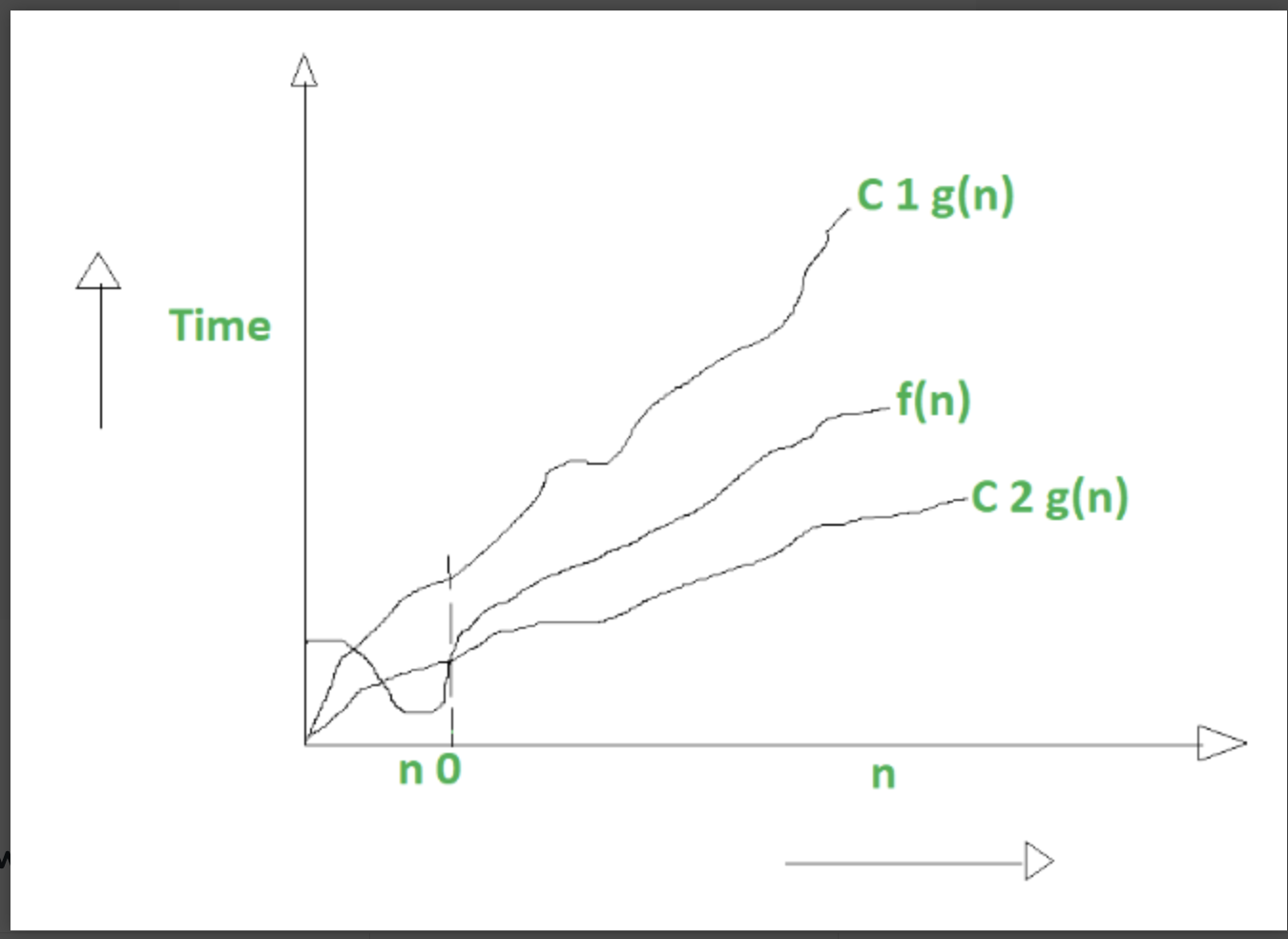
***Mathematically,***

*0 <= f(n) <= C1g(n) for n >= n0  
0 <= C2g(n) <= f(n) for n >= n0*

***Merging both the equation, we get :***

*0 <= C2g(n) <= f(n) <= C1g(n) for n >= n0*

The equation simply means there exist positive constants C1 and C2 such that f(n) is sandwich between C2 g(n) and C1g(n).



### Difference Between Big oh, Big Omega and Big Theta :

| **S.No.** | **Big O** | **Big Omega (Ω)** | **Theta (Θ)** |
| --- | --- | --- | --- |
| **1.** | It is like (<=)  rate of growth of an algorithm is less than or equal to a specific value. | It is like (>=)  rate of growth is greater than or equal to a specified value. | It is like (==)  meaning the rate of growth is equal to a specified value. |
| **2.** | The upper bound of a function is represented by Big O notation. Only the time taken function is bounded by above. B | The lower bound of a function is represented by Omega notation. | The bounding of a function from above and below is represented by theta notation. The exact asymptotic behavior is done by this theta notation. |
| **3.** | Big O - Upper Bound | Big Omega (Ω) - Lower Bound | Big Theta (Θ) - Tight Bound |
| **4.** | To find Big O notation of time/space,. we consider the case when an algorithm takes maximum time/space. | To find Big Omega notation of time/space,. we consider the case when an algorithm takes minimum time/space. | An algorithm's general time/space cannot be represented as Theta notation, if its order of growth varies with input. |
| **5.** | Mathematically: Big Oh is 0 <= f(n) <= Cg(n) for all n >= n0 | Mathematically: Big Omega is 0 <= Cg(n) <= f(n) for all n >= n0 | Mathematically - Big Theta is 0 <= C2g(n) <= f(n) <= C1g(n) for n >= n0 |