**Data Structures and Algorithms (DSA)**

Table of Contents

[Introduction 3](#_Toc203679215)

[Data Structures: 3](#_Toc203679216)

[Algorithms: 3](#_Toc203679217)

[Relationship: 4](#_Toc203679218)

[Importance: 4](#_Toc203679219)

[Database 4](#_Toc203679220)

[Data warehouse 5](#_Toc203679221)

[Big Data 5](#_Toc203679222)

[The memory layout of a C program 5](#_Toc203679223)

[Text Segment (Code Segment): 5](#_Toc203679224)

[Data Segment: 5](#_Toc203679225)

[BSS Segment (Block Started by Symbol): 6](#_Toc203679226)

[Heap: 6](#_Toc203679227)

[Stack: 6](#_Toc203679228)

[Key Differences Between Stack and Heap Allocations 7](#_Toc203679229)

[Comparison Chart 7](#_Toc203679230)

[Asymptotic Analysis 8](#_Toc203679231)

[Given two algorithms for a task, how do we find out which one is better? 8](#_Toc203679232)

[For example, 8](#_Toc203679233)

[Running times for this example: 9](#_Toc203679234)

[Does Asymptotic Analysis always work? 9](#_Toc203679235)

[Asymptotic Notations: 10](#_Toc203679236)

[Big O Notation Tutorial - A Guide to Big O Analysis 10](#_Toc203679237)

[Big O Definition 10](#_Toc203679238)

[Importance of Big O Notation 11](#_Toc203679239)

[A Quick Way to find Big O of an Expression 11](#_Toc203679240)

[Analysis of Algorithms | Θ (Theta) Notation 12](#_Toc203679241)

[Definition 12](#_Toc203679242)

[Mathematical Representation: 12](#_Toc203679243)

[Graphical Representation: 13](#_Toc203679244)

[Analysis of Algorithms | Big-Omega Ω Notation 13](#_Toc203679245)

[What is Big-Omega Ω Notation? 14](#_Toc203679246)

[Definition of Big-Omega Ω Notation? 14](#_Toc203679247)

[Big O vs Theta Θ vs Big Omega Ω Notations 15](#_Toc203679248)

[1. Big O notation (O): 15](#_Toc203679249)

[2. Big Omega notation (Ω) : 16](#_Toc203679250)

[3. Theta notation (Θ) : 16](#_Toc203679251)

[Difference Between Big oh, Big Omega and Big Theta : 17](#_Toc203679252)

[Worst, Average and Best Case Analysis of Algorithms 18](#_Toc203679253)

[1. Worst Case Analysis (Mostly used) 18](#_Toc203679254)

[2. Best Case Analysis (Very Rarely used) 19](#_Toc203679255)

[3. Average Case Analysis (Rarely used) 19](#_Toc203679256)

[Why is Worst Case Analysis Mostly Used? 19](#_Toc203679257)

[Average Case : 19](#_Toc203679258)

[Best Case : 19](#_Toc203679259)

[Worst Case: 19](#_Toc203679260)

[Time Complexity & Space Complexity 20](#_Toc203679261)

[Time Complexity 20](#_Toc203679262)

[Definition: 20](#_Toc203679263)

[**Purpose:** 20](#_Toc203679264)

[Representation: 20](#_Toc203679265)

[Examples: 20](#_Toc203679266)

[Importance: 21](#_Toc203679267)

[Space Complexity: 21](#_Toc203679268)

[Definition: 21](#_Toc203679269)

[Purpose: 21](#_Toc203679270)

[Representation: 21](#_Toc203679271)

[Examples: 21](#_Toc203679272)

[Importance: 21](#_Toc203679273)

[Key Differences: 21](#_Toc203679274)

[To determine the time complexity of a program 21](#_Toc203679275)

[1. Identify the input size: 22](#_Toc203679276)

[2. Analyze individual statements: 22](#_Toc203679277)

[3. Sum up the complexities: 23](#_Toc203679278)

[4. Focus on the dominant term: 23](#_Toc203679279)

[5. Use Big O notation: 23](#_Toc203679280)

[Example 1: Linear Search 23](#_Toc203679281)

[Example 2: Nested Loops 23](#_Toc203679282)

[Example 3: Simple Operations 24](#_Toc203679283)

[Abstract Data Types 24](#_Toc203679284)

[Features of ADT 24](#_Toc203679285)

[Why Use ADTs? 25](#_Toc203679286)

[Example of Abstraction 26](#_Toc203679287)

[Difference Between ADTs and UDTs 26](#_Toc203679288)

[Aspect 26](#_Toc203679289)

[Abstract Data Types (ADTs) 26](#_Toc203679290)

[User-Defined Data Types (UDTs) 26](#_Toc203679291)

[Examples of ADTs 27](#_Toc203679292)

[1.List ADT 27](#_Toc203679293)

[2.Stack ADT 28](#_Toc203679294)

[3. Queue ADT 29](#_Toc203679295)

[Advantages and Disadvantages of ADT 30](#_Toc203679296)

[C Structures 31](#_Toc203679297)

[Example:1 31](#_Toc203679298)

[Explanation: 31](#_Toc203679299)

[Example:2 31](#_Toc203679300)

[Output 32](#_Toc203679301)

[Structure Pointer in C 32](#_Toc203679302)

[Let's take a look at an example: 32](#_Toc203679303)

[Output 33](#_Toc203679304)

[Accessing Member using Structure Pointers 33](#_Toc203679305)

[Differencing and Using (.) Dot Operator 33](#_Toc203679306)

[Program 33](#_Toc203679307)

[Output 34](#_Toc203679308)

[Using ( -> ) Arrow Operator 34](#_Toc203679309)

[Program 34](#_Toc203679310)

[Output 35](#_Toc203679311)

[Explanation 35](#_Toc203679312)

[Getting Started with Array Data Structure 35](#_Toc203679313)

[Array 35](#_Toc203679314)

[Basic terminologies of Array 35](#_Toc203679315)

[Memory representation of Array 35](#_Toc203679316)

[Operations on Array 36](#_Toc203679317)

[Array Traversal 36](#_Toc203679318)

[Types of Array Traversal 36](#_Toc203679319)

[C Program to Insert an Element in an Array 36](#_Toc203679320)

[Insert Element at Specific Position 37](#_Toc203679321)

[Insert an Element at the End of an Array 38](#_Toc203679322)

# 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):

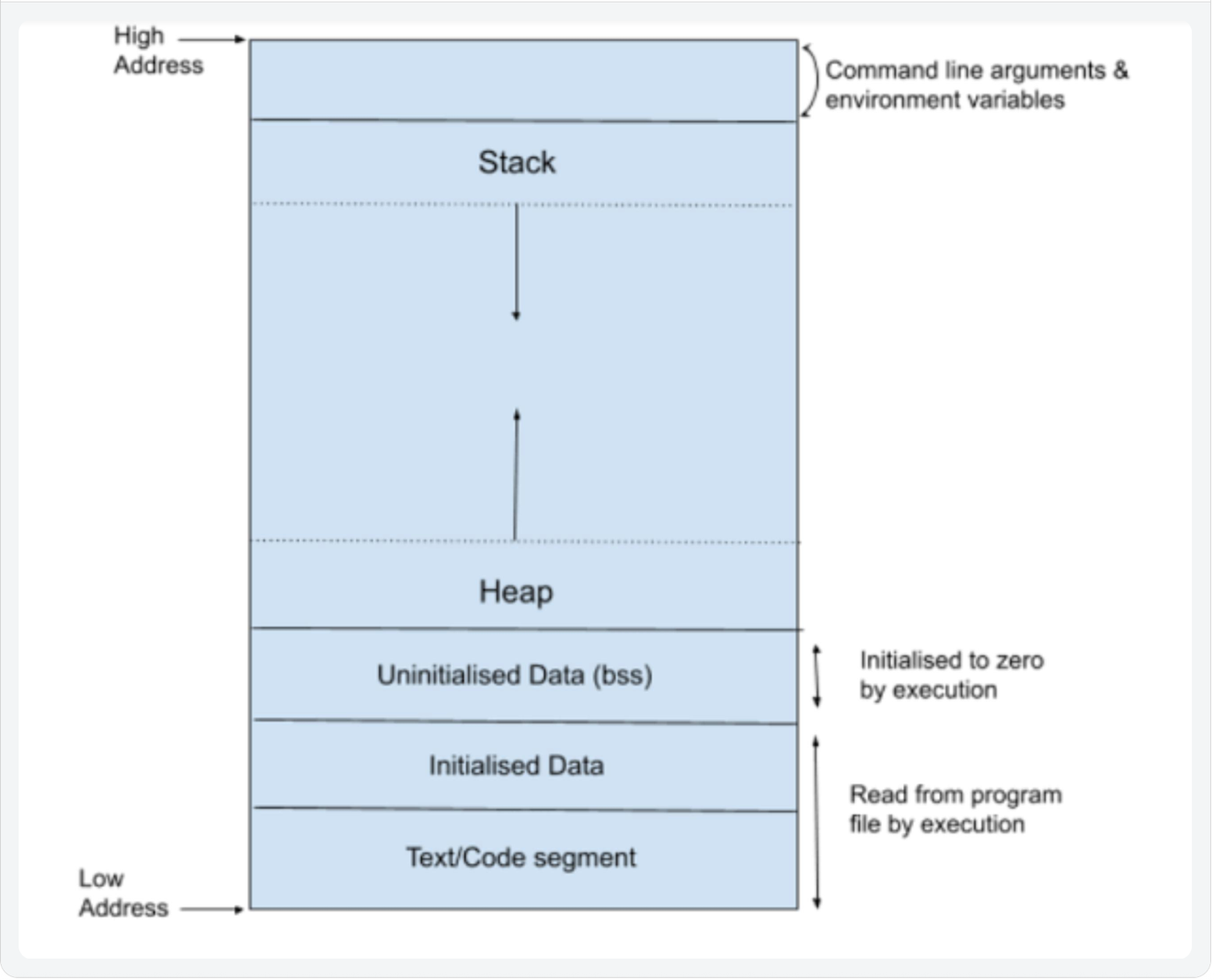
This segment stores uninitialized global and static variables. The operating system initializes the memory in this segment to zero before the program starts execution. Like the data segment, it is a read-write segment.

## 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:

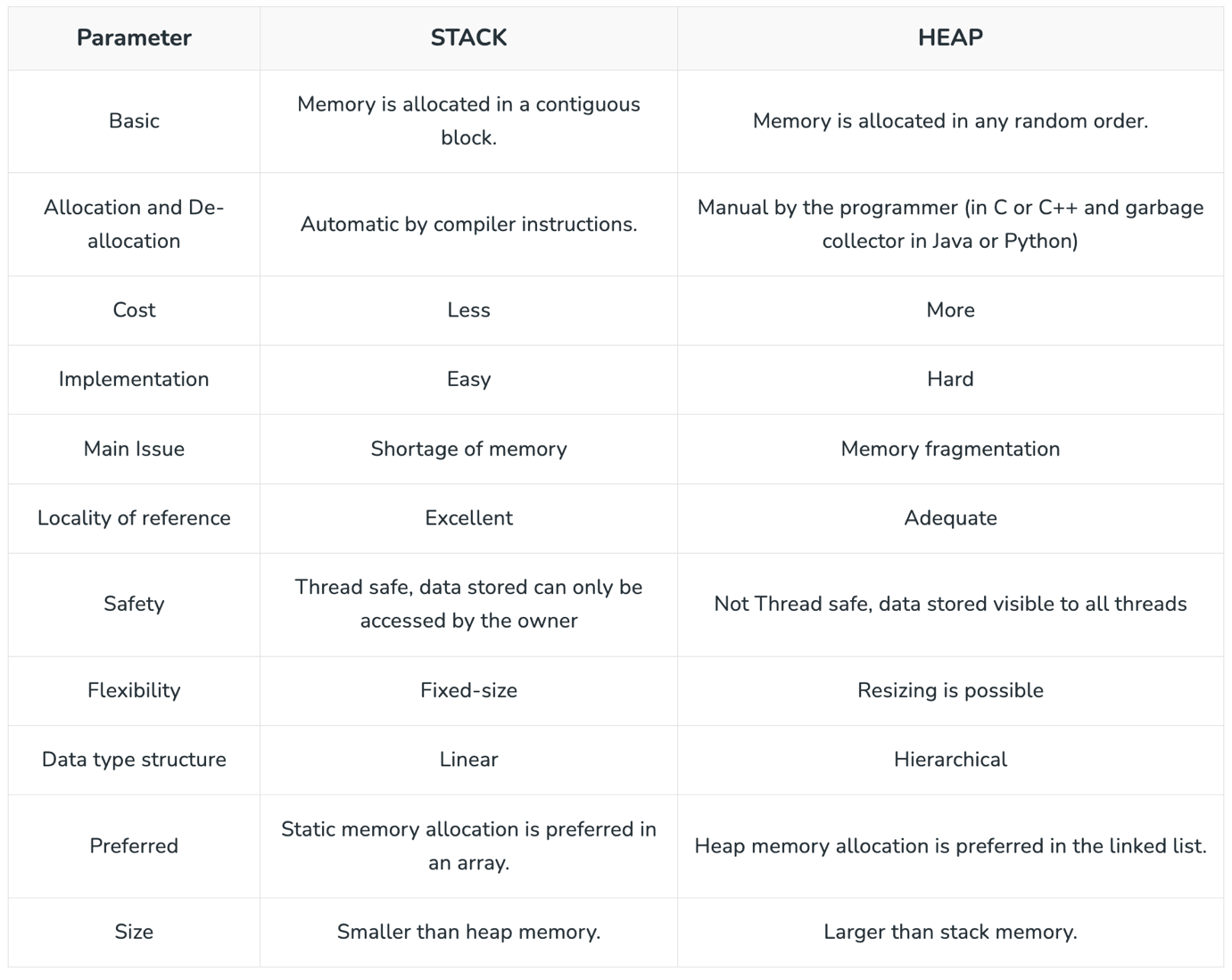
This segment is used for local variables, function arguments, and return addresses during function calls. It operates on a Last-In, First-Out (LIFO) principle, where new data is pushed onto the top of the stack and removed when no longer needed (e.g., when a function returns). The stack typically grows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

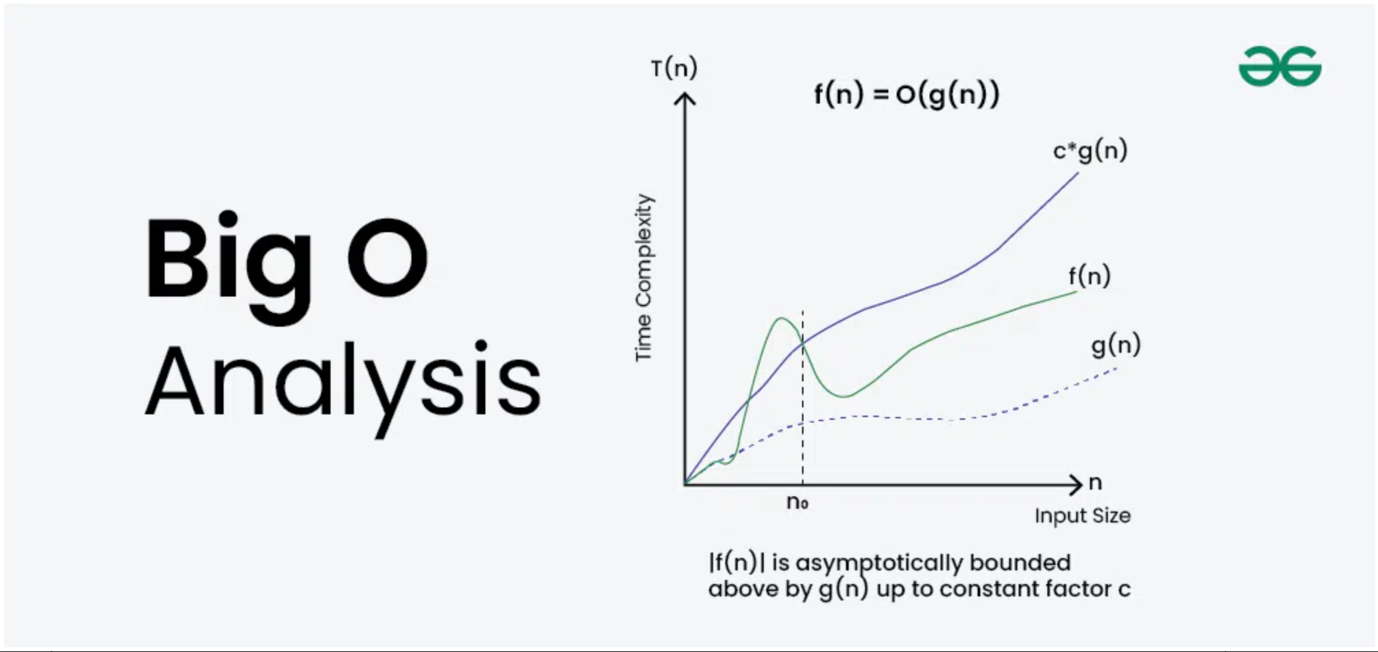
Also, in Asymptotic analysis, we always talk about input sizes larger than a constant value. It might be possible that those large inputs are never given to your software and an asymptotically slower algorithm always performs better for your particular situation. So, you may end up choosing an algorithm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

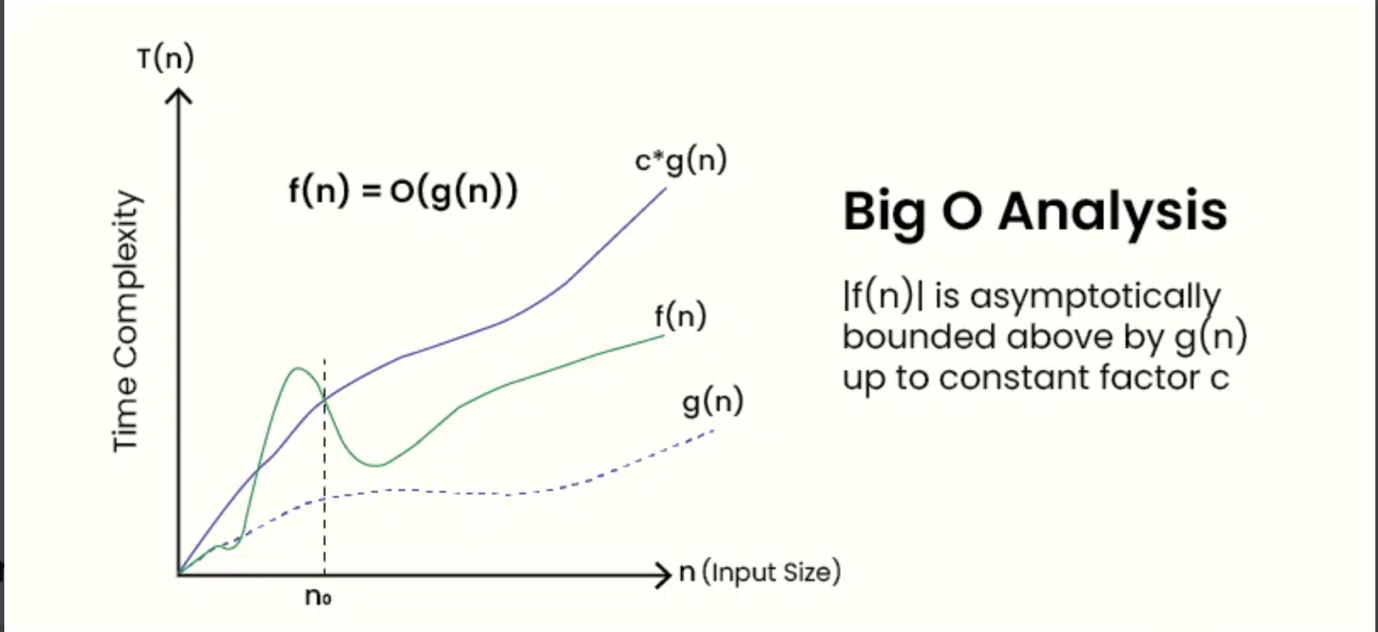
* Describes the asymptotic behavior (order of growth of time or space in terms of input size) of a function, not its exact value.
* Can be used to compare the efficiency of different algorithms or data structures.
* It provides an**upper limit** on the time taken by an algorithm in terms of the size 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).

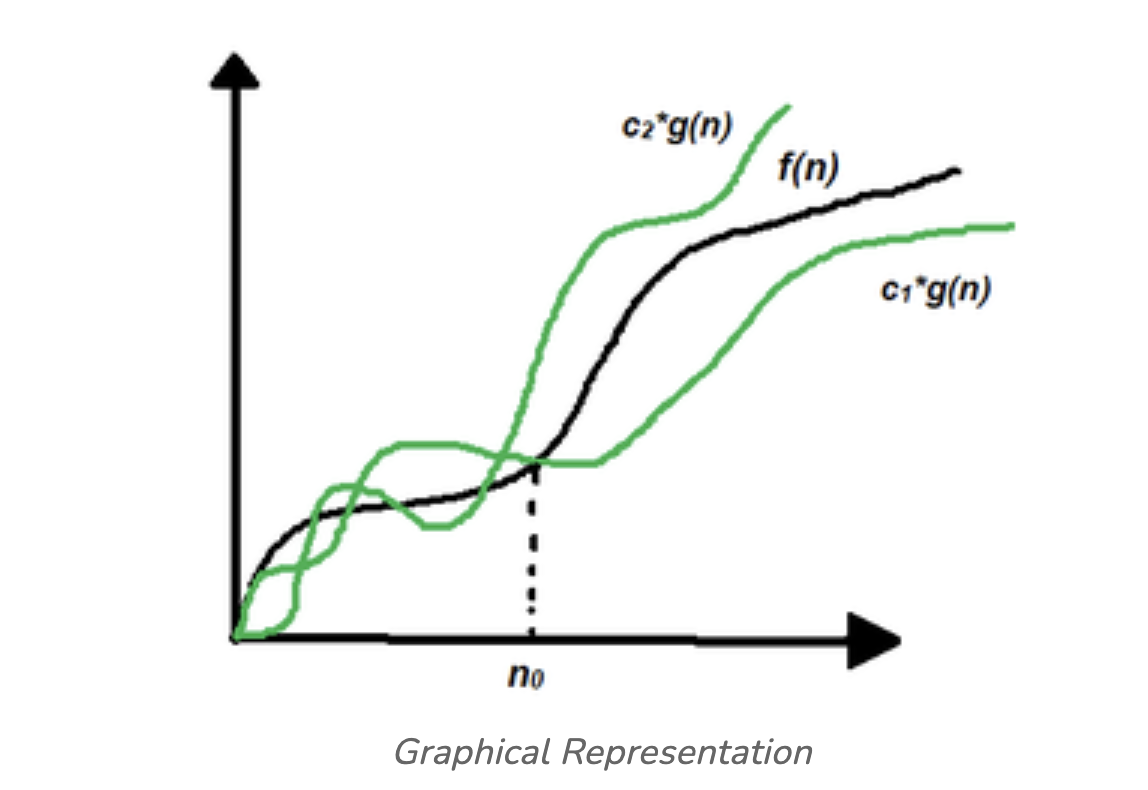
Definition**:** Let g and f be the function from the set of natural numbers to itself. The function f is said to be Θ(g), if there are constants c1, c2 > 0 and a natural number n0 such that c1\* g(n) ≤ f(n) ≤ c2 \* g(n) for all n ≥ n0

### **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:**



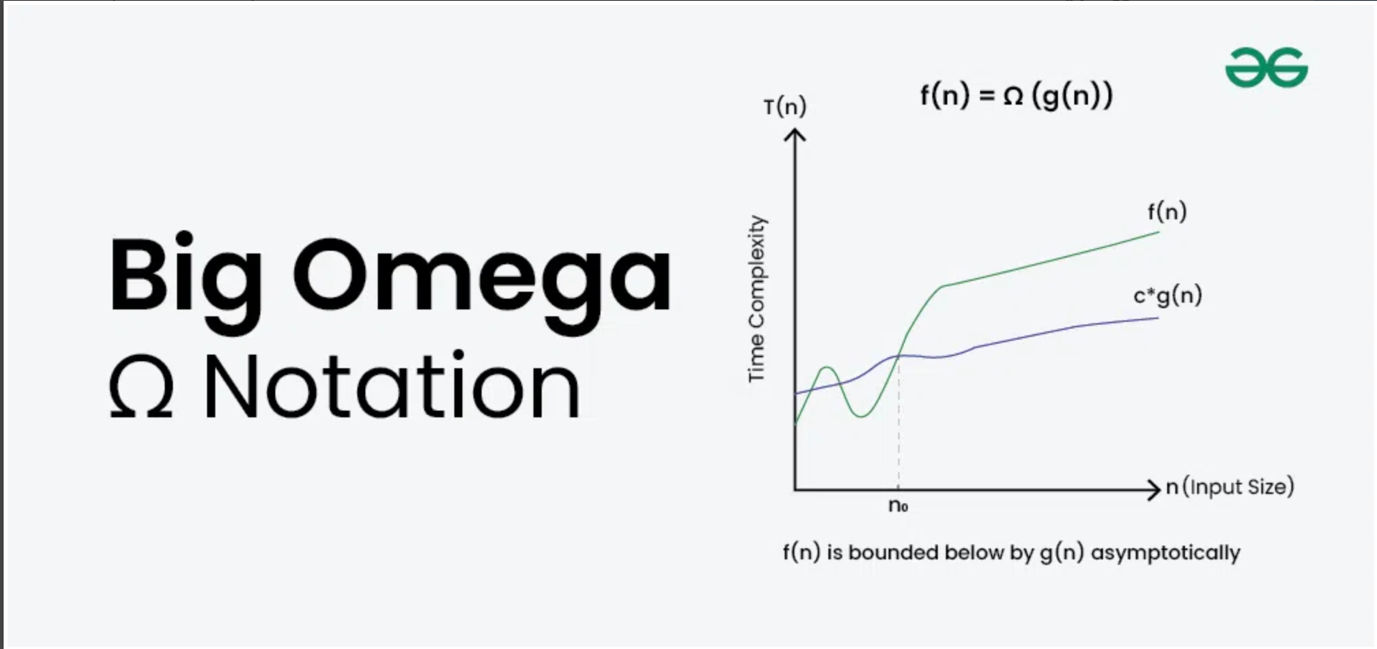
In simple language, Big - Theta(Θ) notation specifies asymptotic bounds (both upper and lower) for a function f(n) and provides the average time complexity of an algorithm.

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?

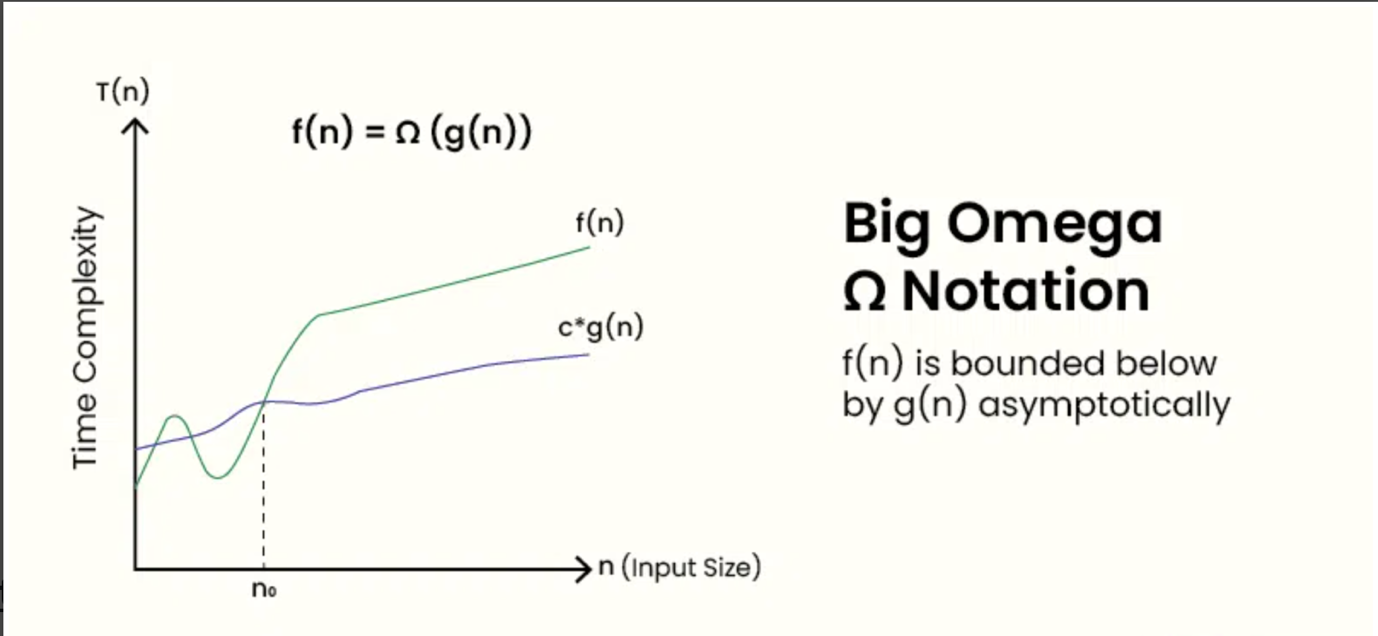
**Big-Omega Ω Notation**, is a way to express the **asymptotic lower bound**of an algorithm’s time complexity, since it analyses the**best-case** situation of algorithm. It provides a **lower limit** on the time taken by an algorithm in terms of the size of the input. It’s denoted as**Ω(f(n))**, where**f(n)**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



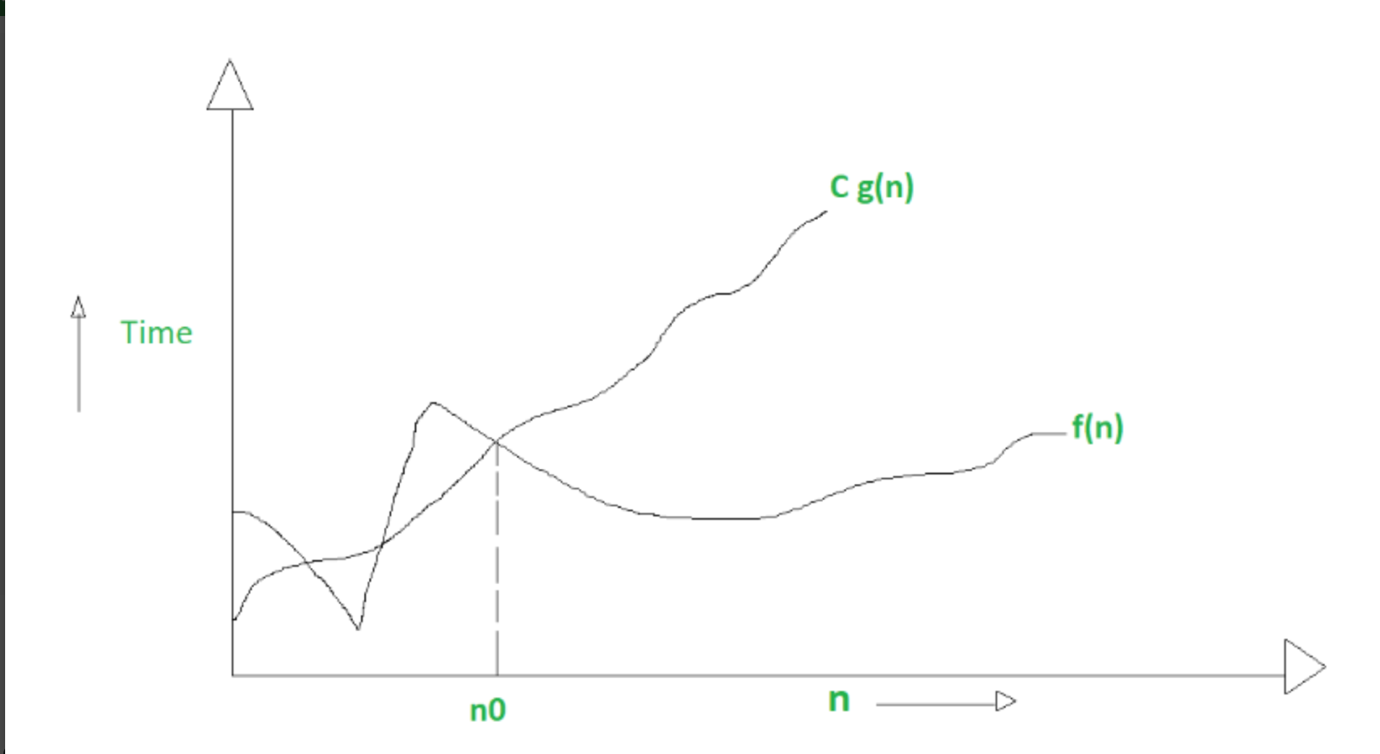
## Big O vs Theta Θ vs Big Omega Ω Notations

### 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*

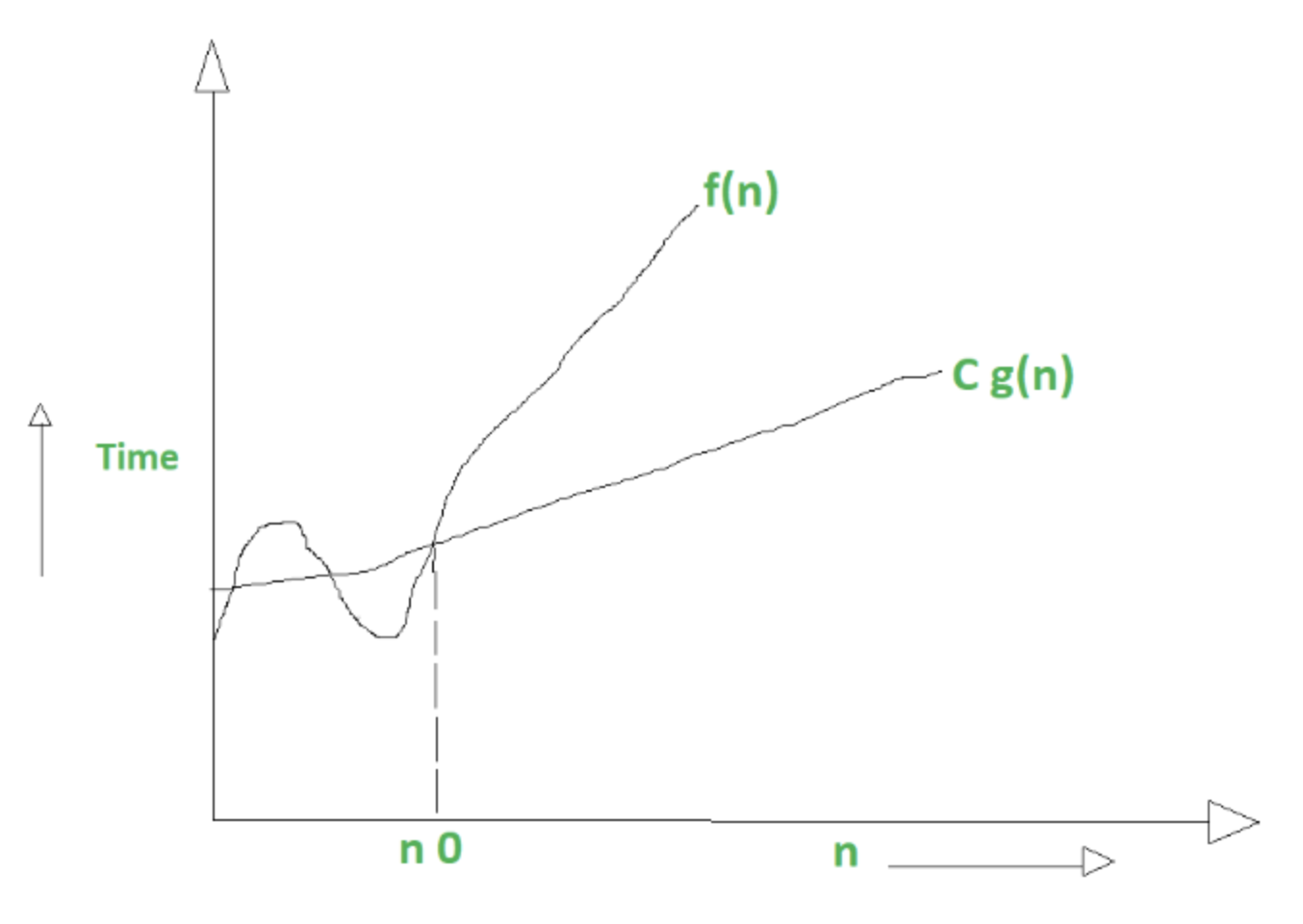
**n** = used to give upper bound a function.   
If a function is **O(n)**, it is automatically **O(n-square)** as well.



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)).**

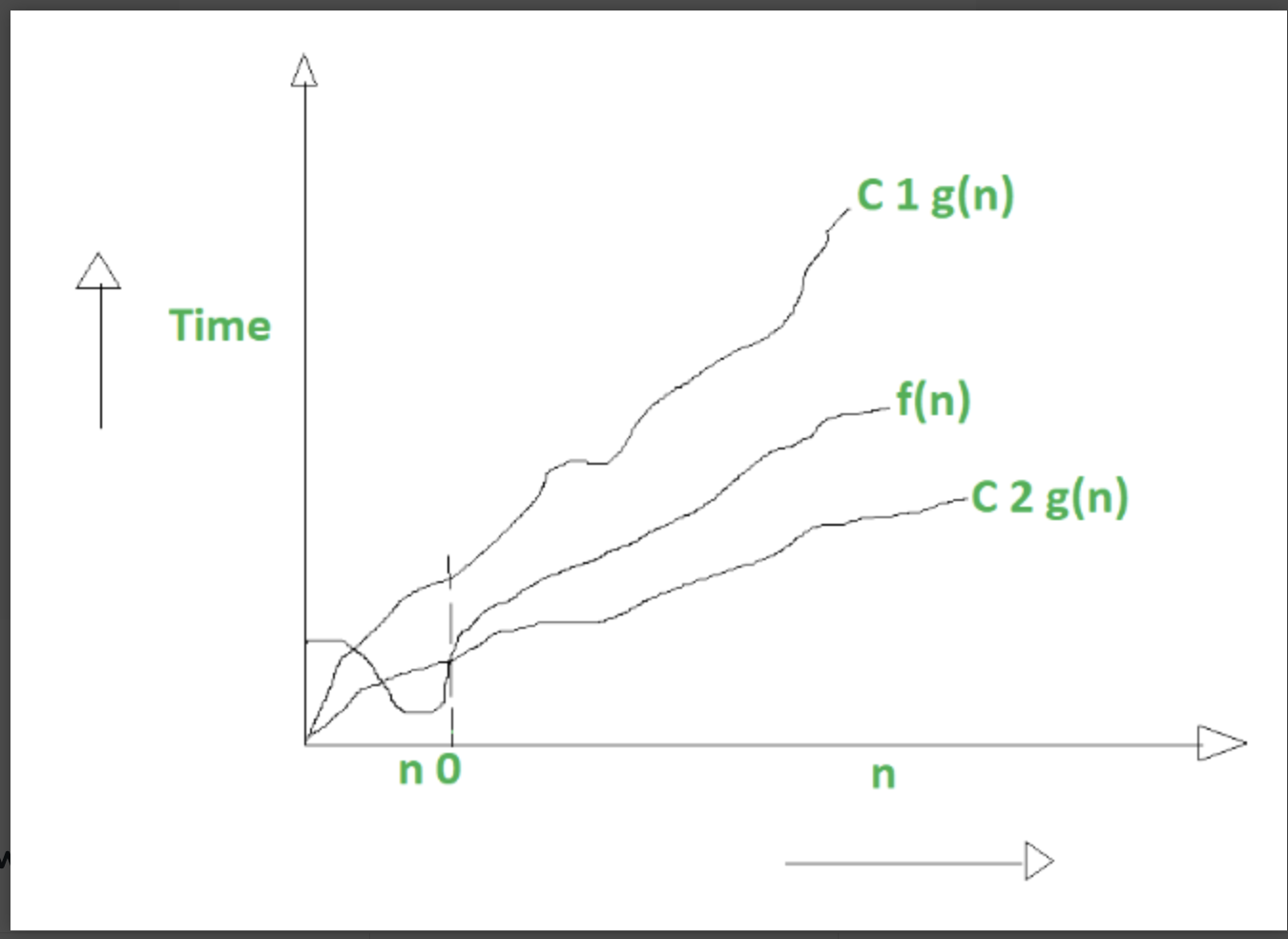
***Mathematically,***

*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 |

# Worst, Average and Best Case Analysis of Algorithms

In the previous post, we discussed how Asymptotic analysis overcomes the problems of the naive way of analysing algorithms. Now let us learn about What is Worst, Average, and Best cases of an algorithm:

### **1.** Worst Case Analysis (Mostly used)

* In the worst-case analysis, we calculate the upper bound on the running time of an algorithm. We must know the case that causes a maximum number of operations to be executed.
* For [Linear Search](https://www.geeksforgeeks.org/linear-search/), the worst case happens when the element to be searched (x) is not present in the array. When x is not present, the search()function compares it with all the elements of arr[] one by one.
* This is the most commonly used analysis of algorithms (We will be discussing below why). Most of the time we consider the case that causes maximum operations.

### **2.** Best Case Analysis (Very Rarely used)

* In the best-case analysis, we calculate the lower bound on the running time of an algorithm. We must know the case that causes a minimum number of operations to be executed.
* For [linear search](https://www.geeksforgeeks.org/linear-search/), the best case occurs when x is present at the first location. The number of operations in the best case is constant (not dependent on n). So the order of growth of time taken in terms of input size is constant.

### **3.** Average Case Analysis (Rarely used)

* In average case analysis, we take all possible inputs and calculate the computing time for all of the inputs. Sum all the calculated values and divide the sum by the total number of inputs.
* We must know (or predict) the distribution of cases. For the linear search problem, let us assume that all cases are [uniformly distributed](http://en.wikipedia.org/wiki/Uniform_distribution_%28discrete%29) (including the case of x not being present in the array). So we sum all the cases and divide the sum by (n+1). We take (n+1) to consider the case when the element is not present.

## Why is Worst Case Analysis Mostly Used?

Average Case :The average case analysis is not easy to do in most practical cases and it is rarely done. In the average case analysis, we need to consider every input, its frequency and time taken by it which may not be possible in many scenarios

Best Case :The Best Case analysis is considered bogus. Guaranteeing a lower bound on an algorithm doesn't provide any information as in the worst case, an algorithm may take years to run.

Worst Case:This is easier than average case and gives an upper bound which is useful information to analyse software products.

# Time Complexity & Space Complexity

Time complexity and space complexity are two fundamental concepts used to analyse the efficiency of algorithms. Time complexity describes how the execution time of an algorithm grows as the input size increases, while space complexity describes how the memory usage of an algorithm grows with the input size.

## Time Complexity

### Definition:

Time complexity measures the amount of time an algorithm takes to run as a function of the input size.

### **Purpose:**

It helps predict how an algorithm's performance will scale with larger inputs.

### Representation:

Time complexity is often expressed using Big O notation, which describes the upper bound of the growth rate.

### Examples:

* **Constant time (O(1)):** An algorithm that takes the same amount of time regardless of the input size (e.g., accessing an element in an array by its index).
* **Linear time (O(n)):** An algorithm where the execution time grows proportionally to the input size (e.g., iterating through an array once).
* **Quadratic time (O(n²)):** An algorithm where the execution time grows with the square of the input size (e.g., nested loops iterating through an array).
* **Logarithmic time (O(log n)):** An algorithm where the execution time increases logarithmically with the input size (e.g., binary search).

### Importance:

* Time complexity analysis helps in choosing efficient algorithms for large datasets, avoiding performance bottlenecks.

## Space Complexity:

### Definition:

Space complexity measures the amount of memory space an algorithm requires to run as a function of the input size.

### Purpose:

It helps understand the memory footprint of an algorithm and its potential impact on system resources.

### Representation:

Space complexity is also often expressed using Big O notation.

### Examples:

* **Constant space (O(1)):** An algorithm that uses a fixed amount of memory regardless of the input size (e.g., a few variables).
* **Linear space (O(n)):** An algorithm that uses memory proportional to the input size (e.g., storing an array of n elements).

### Importance:

* Space complexity analysis helps optimize memory usage, especially in resource-constrained environments.

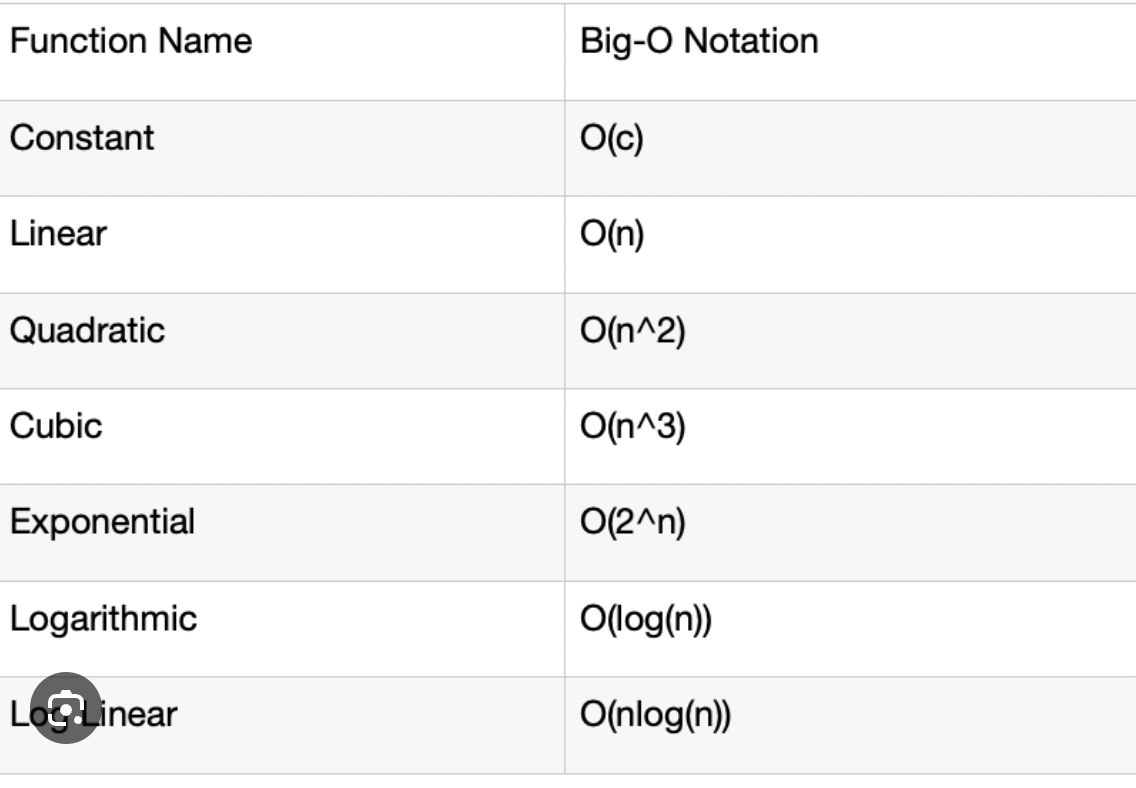
### Key Differences:

* **Time complexity focuses on execution time, while space complexity focuses on memory usage.**
* **Both are crucial for evaluating algorithm efficiency.**
* **Algorithms can have different time and space complexities.**

In essence, time complexity helps you understand how fast an algorithm is, while space complexity helps you understand how much memory it needs. Both are essential for writing efficient and scalable code.

# To determine the time complexity of a program

Analyse the code to count the number of elementary operations (like assignments, comparisons, arithmetic operations) executed in relation to the input size. Use Big O notation to express the growth rate of the execution time as the input size increases. Focus on the dominant operations within loops or recursive calls.



Here's a step-by-step breakdown:

## 1. Identify the input size:

Determine the variable that represents the input size (usually 'n'). For example, in a list traversal, 'n' would be the number of elements in the list.

## 2. Analyze individual statements:

* + Simple statements (assignments, arithmetic operations, variable access) usually take constant time, denoted as O(1).
  + Loops: If a loop iterates 'n' times, and the code inside the loop takes constant time, the loop's time complexity is O(n). If the loop iterates 'n/2' times, it's still O(n).
  + Nested loops: If you have nested loops, where the inner loop also iterates 'n' times for each iteration of the outer loop, the time complexity is O(n\*n) or O(n²).
  + Recursive calls: Analyze the number of recursive calls in relation to the input size. For example, binary search has a time complexity of O(log n) due to halving the search space in each call.

## 3. Sum up the complexities:

Add the time complexities of all the statements or blocks of code. If you have a sequence of operations, their complexities are added. If you have nested loops, you multiply their complexities.

## 4. Focus on the dominant term:

In Big O notation, you usually ignore constant factors and lower-order terms. For example, O(2n + 10) simplifies to O(n), and O(n² + n + 1) simplifies to O(n²).

## 5. Use Big O notation:

Big O notation (O(1), O(log n), O(n), O(n log n), O(n²), etc.) provides a way to express the upper bound of the algorithm's growth rate.

### Example 1: Linear Search

def linear\_search(list, target):

for element in list:

if element == target:

return True

return False

* Input size: n (length of the list).
* The for loop iterates n times in the worst case (when the element is not found or is at the end).
* Each iteration performs a comparison (constant time).
* Therefore, the time complexity is O(n).

### Example 2: Nested Loops

def print\_pairs(list):

for i in list:

for j in list:

print(i, j)

* Input size: n (length of the list).
* The outer loop iterates n times.
* The inner loop also iterates n times for each iteration of the outer loop.
* Therefore, the total number of operations is approximately n \* n = n^2.
* The time complexity is O(n2).

### Example 3: Simple Operations

def simple\_operations(n):

a = 10 # Constant time operation

b = n \* 2 # Linear time operation

c = n \* n # Quadratic time operation

return a + b + c

* a = 10 takes constant time, O(1).
* b = n \* 2 takes linear time, O(n).
* c = n \* n takes quadratic time, O(n2).
* Overall time complexity is dominated by the highest order term, O(n2).

# Abstract Data Types

An **Abstract Data Type (ADT)** is a conceptual model that defines a set of operations and behaviours for a data structure, **without specifying how these operations are implemented** or how data is organized in memory. The definition of ADT only mentions what **operations are to be performed** but not **how** these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. **It is called "abstract" because it provides an implementation-independent view.**

The process of providing only the essentials and hiding the details is known as abstraction.

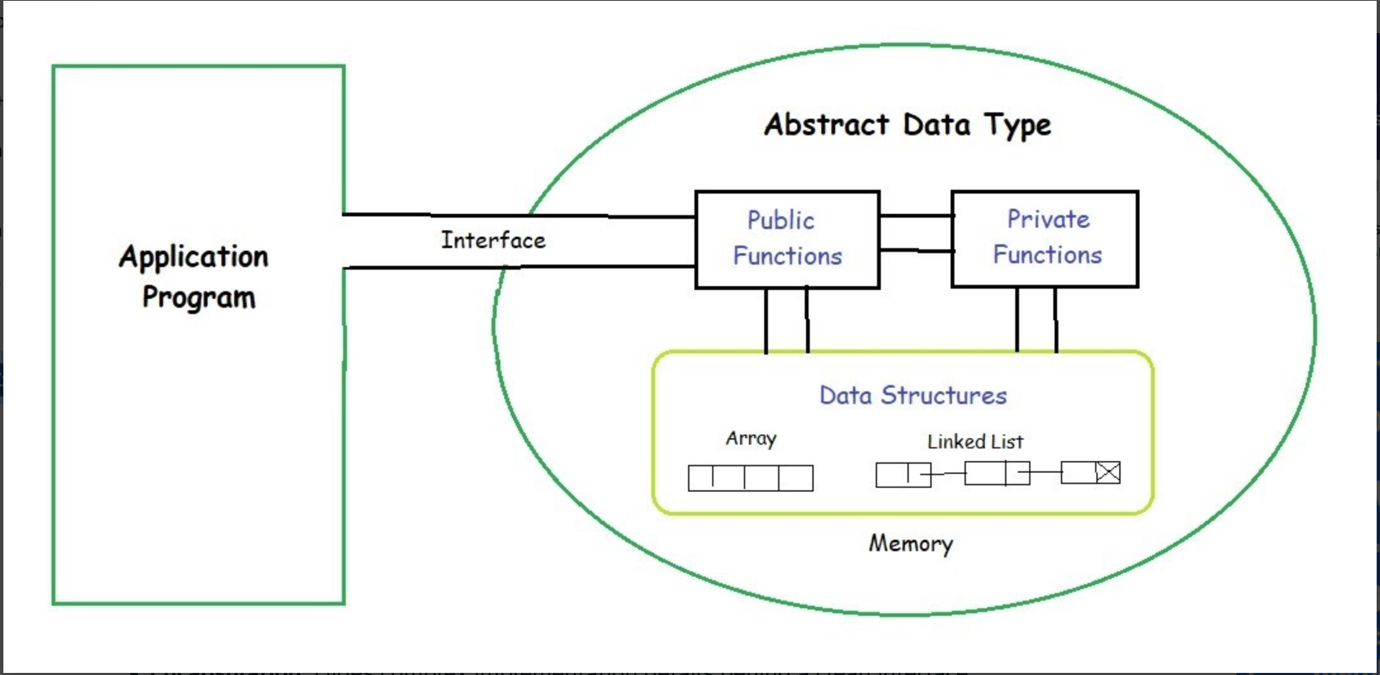
## Features of ADT

Abstract data types (ADTs) are a way of encapsulating data and operations on that data into a single unit. Some of the key features of ADTs include:

* **Abstraction:** The user does not need to know the implementation of the data structure only essentials are provided.
* **Better Conceptualization:** ADT gives us a better conceptualization of the real world.
* **Robust:** The program is robust and has the ability to catch errors.
* **Encapsulation**: ADTs hide the internal details of the data and provide a public interface for users to interact with the data. This allows for easier maintenance and modification of the data structure.
* **Data Abstraction**: ADTs provide a level of abstraction from the implementation details of the data. Users only need to know the operations that can be performed on the data, not how those operations are implemented.
* **Data Structure Independence**: ADTs can be implemented using different data structures, such as arrays or linked lists, without affecting the functionality of the ADT.
* **Information Hiding:** ADTs can protect the integrity of the data by allowing access only to authorized users and operations. This helps prevent errors and misuse of the data.
* **Modularity**: ADTs can be combined with other ADTs to form larger, more complex data structures. This allows for greater flexibility and modularity in programming.

Overall, ADTs provide a powerful tool for organizing and manipulating data in a structured and efficient manner.

**This image demonstrates how an Abstract Data Type (ADT) hides internal data structures (like arrays, linked lists) using public and private functions, exposing only a defined interface to the application program.**

****

## Why Use ADTs?

The key reasons to use ADTs in Java are listed below:

* **Encapsulation:** Hides complex implementation details behind a clean interface.
* **Reusability**: Allows different internal implementations (e.g., array or linked list) without changing external usage.
* **Modularity:** Simplifies maintenance and updates by separating logic.
* **Security:** Protects data by preventing direct access, minimizing bugs and unintended changes.

## **Example of Abstraction**

* For example, we use primitive values like int, float, and char with the understanding that these data types can operate and be performed on without any knowledge of their implementation details. ADTs operate similarly by defining **what operations are possible without detailing their implementation.**

## Difference Between ADTs and UDTs

* The table below demonstrates the difference between ADTs and UDTs.

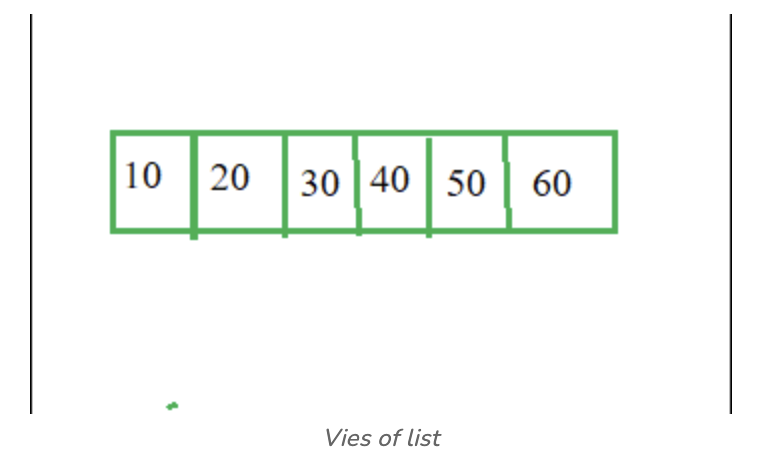
| Aspect | Abstract Data Types (ADTs) | User-Defined Data Types (UDTs) |
| --- | --- | --- |
| **Definition** | Defines a class of objects and the operations that can be performed on them, along with their expected behavior (semantics), but without specifying implementation details. | A custom data type created by combining or extending existing primitive types, specifying both structure and operations. |
| **Focus** | What operations are allowed and how they behave, without dictating how they are implemented. | How data is organized in memory and how operations are executed. |
| **Purpose** | Provides an abstract model to define data structures in a conceptual way. | Allows programmers to create concrete implementations of data structures using primitive types. |
| **Implementation Details** | Does not specify how operations are implemented or how data is structured. | Specifies how to create and organize data types to implement the structure. |
| **Usage** | Used to design and conceptualize data structures. | Used to implement data structures that realize the abstract concepts defined by ADTs. |
| **Example** | List ADT, Stack ADT, Queue ADT. | Structures, classes, enumerations, records. |

## Examples of ADTs

Now, let's understand three common ADT's: List ADT, Stack ADT, and Queue ADT.

### 1.List ADT

The List ADT (Abstract Data Type) is a sequential collection of elements that supports a set of operations **without specifying the internal implementation**. It provides an ordered way to store, access, and modify data.



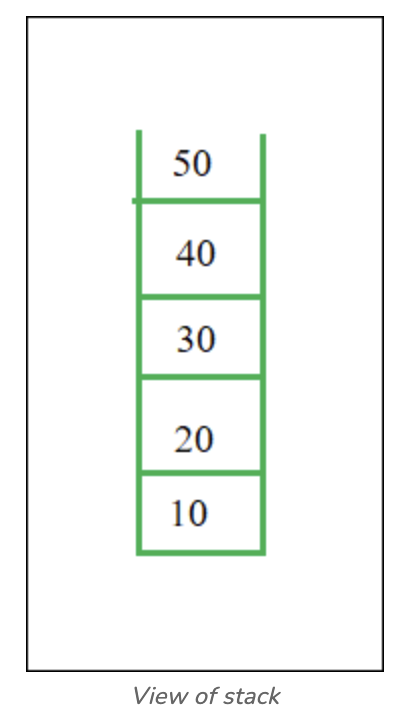
**Operations:**

The List ADT need to store the required data in the sequence and should have the following operations**:**

* **get():** Return an element from the list at any given position.
* **insert():** Insert an element at any position in the list.
* **remove():** Remove the first occurrence of any element from a non-empty list.
* **removeAt():** Remove the element at a specified location from a non-empty list.
* **replace():** Replace an element at any position with another element.
* **size():** Return the number of elements in the list.
* **isEmpty():** Return true if the list is empty; otherwise, return false.
* **isFull():** Return true if the list is full, otherwise, return false. Only applicable in fixed-size implementations (e.g., array-based lists).

### 2.Stack ADT

The Stack ADT is a linear data structure that follows the LIFO (Last In, First Out) principle. It allows elements to be added and removed only from one end, called the top of the stack.



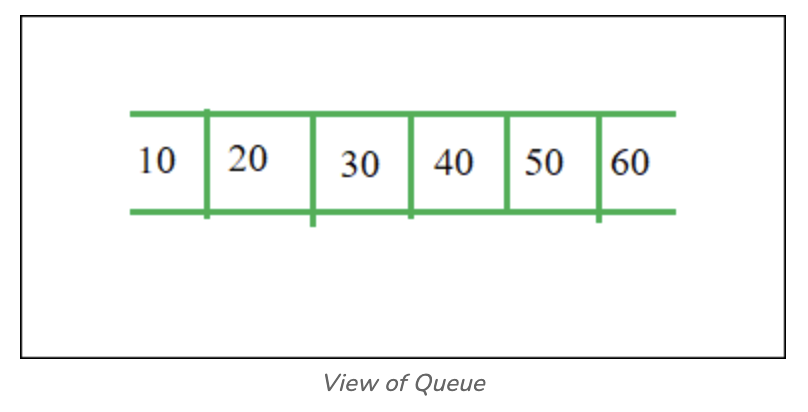
**Operations:**

In Stack ADT, the order of insertion and deletion should be according to the FILO or LIFO Principle. Elements are inserted and removed from the same end, called the top of the stack. It should also support the following operations:

* **push():** Insert an element at one end of the stack called the top.
* **pop():** Remove and return the element at the top of the stack, if it is not empty.
* **peek():** Return the element at the top of the stack without removing it, if the stack is not empty.
* **size():** Return the number of elements in the stack.
* **isEmpty():** Return true if the stack is empty; otherwise, return false.
* **isFull():** Return true if the stack is full; otherwise, return false. Only relevant for fixed-capacity stacks (e.g., array-based).

### 3. Queue ADT

The Queue ADT is a linear data structure that follows the FIFO (First In, First Out) principle. It allows elements to be inserted at one end (rear) and removed from the other end (front).



**Operations:**

The Queue ADT follows a design similar to the Stack ADT, but the order of insertion and deletion changes to FIFO. Elements are inserted at one end (called the rear) and removed from the other end (called the front). It should support the following operations:

* **enqueue():** Insert an element at the end of the queue.
* **dequeue():** Remove and return the first element of the queue, if the queue is not empty.
* **peek():** Return the element of the queue without removing it, if the queue is not empty.
* **size():** Return the number of elements in the queue.
* **isEmpty():** Return true if the queue is empty; otherwise, return false.

### Advantages and Disadvantages of ADT

Abstract data types (ADTs) have several advantages and disadvantages that should be considered when deciding to use them in software development. Here are some of the main advantages and disadvantages of using ADTs:

#### Advantage:

The advantages are listed below:

* **Encapsulation**: ADTs provide a way to encapsulate data and operations into a single unit, making it easier to manage and modify the data structure.
* **Abstraction**: ADTs allow users to work with data structures without having to know the implementation details, which can simplify programming and reduce errors.
* **Data Structure Independence**: ADTs can be implemented using different data structures, which can make it easier to adapt to changing needs and requirements.
* **Information Hiding**: ADTs can protect the integrity of data by controlling access and preventing unauthorized modifications.
* **Modularity**: ADTs can be combined with other ADTs to form more complex data structures, which can increase flexibility and modularity in programming.

#### Disadvantages:

The disadvantages are listed below:

* **Overhead**: Implementing ADTs can add overhead in terms of memory and processing, which can affect performance.
* **Complexity**: ADTs can be complex to implement, especially for large and complex data structures.
* **Learning** Curve: Using ADTs requires knowledge of their implementation and usage, which can take time and effort to learn.
* **Limited Flexibility:** Some ADTs may be limited in their functionality or may not be suitable for all types of data structures.
* **Cost**: Implementing ADTs may require additional resources and investment, which can increase the cost of development.

# C Structures

In C, a **structure**is a user-defined data type that can be used to group items of possibly different types into a single type. The **struct**keyword is used to define a structure. The items in the structure are called its **members** and they can be of any valid data type. Applications of structures involve creating data structures Linked List and Tree. Structures in C are also used represent real world objects in a software like Students and Faculty in a college management software.

## Example:1

#include <stdio.h>

// Defining a structure

struct A {

int x;

};

int main() {

// Creating a structure variable

struct A a;

// Initializing member

a.x = 11;

printf("%d", a.x);

return 0;

}

Explanation: In this example, a structure **A** is defined to hold an integer member **x**. A variable **a**of type **struct A** is created and its member **x** is initialized to **11** by accessing it using dot operator. The value of **a.x**is then printed to the console.

## Example:2

#include <stdio.h>

// Defining a structure to represent a student

struct Student {

char name[50];

int age;

float grade;

};

int main() {

// Declaring and initializing a structure

// variable

struct Student s1 = {"Rahul",20, 18.5};

// Designated Initializing another stucture

struct Student s2 = {.age = 18, .name =

"Vikas", .grade = 22};

// Accessing structure members

printf("%s\t%d\t%.2f\n", s1.name, s1.age,

s1.grade);

printf("%s\t%d\t%.2f\n", s2.name, s2.age,

s2.grade);

return 0;

}

## **Output**

Rahul 20 18.50

Vikas 18 22.00

# Structure Pointer in C

A structure pointer is a pointer variable that stores the address of a structure. It allows the programmer to manipulate the structure and its members directly by referencing their memory location rather than passing the structure itself. In this article let's take a look at structure pointer in C.

## **Let's take a look at an example:**

#include <stdio.h>

struct A {

int var;

};

int main() {

struct A a = {30};

// Creating a pointer to the structure

struct A \*ptr;

// Assigning the address of person1 to the pointer

ptr = &a;

// Accessing structure members using the pointer

printf("%d", ptr->var);

return 0;

}

## **Output**

30

## Accessing Member using Structure Pointers

There are two ways to access the members of the structure with the help of a structure pointer:

1. Differencing and Using (.) Dot Operator.
2. Using ( -> ) Arrow operator.

### Differencing and Using (.) Dot Operator

First method is to first dereference the structure pointer to get to the structure and then use the dot operator to access the member. Below is the program to access the structure members using the structure pointer with the **help of the dot operator.**

## **Program**

#include <stdio.h>

#include <string.h>

struct Student {

int roll\_no;

char name[30];

char branch[40];

int batch;

};

int main() {

struct Student s1 = {27, "Geek", "CSE", 2019};

// Pointer to s1

struct Student\* ptr = &s1;

// Accessing using dot operator

printf("%d\n", (\*ptr).roll\_no);

printf("%s\n", (\*ptr).name);

printf("%s\n", (\*ptr).branch);

printf("%d", (\*ptr).batch);

return 0;

}

## **Output**

27

Geek

CSE

2019

## Using ( -> ) Arrow Operator

C language provides an array operator (->) that can be used to directly access the structure member without using two separate operators. Below is the program to access the structure members using the structure pointer with the **help of the Arrow operator.**

## Program

#include <stdio.h>

#include <string.h>

struct Student {

int roll\_no;

char name[30];

char branch[40];

int batch;

};

int main() {

struct Student s1 = {27, "Geek", "CSE", 2019};

// Pointer to s1

struct Student\* ptr = &s1;

// Accessing using dot operator

printf("%d\n", ptr->roll\_no);

printf("%s\n", ptr->name);

printf("%s\n", ptr->branch);

printf("%d", ptr->batch);

return 0;

}

## **Output**

27

Geek

CSE

2019

Explanation**:**In this code, a **struct Person** is defined with name and age as members. A pointer ptr is used to store the address of person1. The arrow operator (**->**) is used to access and modify the members of the structure via the pointer, updating the **name** and **age** of person1, and printing the updated values.

# Getting Started with Array Data Structure

Arrayis a collection of items of the same variable type that are stored at contiguous memory locations. It is one of the most popular and simple data structures used in programming.

### Basic terminologies of Array

* **Array Index:** In an array, elements are identified by their indexes. Array index starts from 0.
* **Array element:**Elements are items stored in an array and can be accessed by their index.
* **Array Length:** The length of an array is determined by the number of elements it can contain.

### Memory representation of Array

In an array, all the elements are stored in contiguous memory locations. So, if we initialize an array, the elements will be allocated sequentially in memory. This allows for efficient access and manipulation of elements.

## Operations on Array

### **Array Traversal**

Array traversal refers to the process of accessing and processing each element of an array sequentially. This is one of the most fundamental operations in programming, as arrays are widely used data structures for storing multiple elements in a single variable.

**How Array Traversal Works?**

When an array is created, it occupies a contiguous block of memory where elements are stored in an indexed manner. Each element can be accessed using its index, which starts from 0 in most programming languages.

For example, consider an array containing five integers:

*arr = [10, 20, 30, 40, 50]*

Here:

* The first element (10) is at **index 0**.
* The second element (20) is at **index 1**.
* The last element (50) is at **index 4**.

Array traversal means accessing each element from start to end (or sometimes in reverse order), usually by using a loop.

### **Types of Array Traversal**

Array traversal can be done in multiple ways based on the requirement:

#### **Sequential (Linear) Traversal**

* + This is the most common way of traversing an array.
  + It involves iterating through the array one element at a time from the first index to the last.
  + Used for printing elements, searching, or performing calculations (such as sum or average).

#### **Reverse Traversal**

* + Instead of starting from index 0, the traversal begins from the last element and moves towards the first.
  + This is useful in cases where we need to process elements from the end.

# C Program to Insert an Element in an Array

C arrays have a fixed size, so we cannot dynamically increase their memory. However, we can insert an element if the array already have enough memory space to accommodate the new elements.

## Insert Element at Specific Position

To insert an element at a specific position, shift all elements to the right of the specified position one step to the right. This will create an empty space at the specified position. Then, insert the new element at the desired position.

#include <stdio.h>

void insert(int arr[], int \*n, int pos, int val) {

// Shift elements to the right

for (int i = \*n; i > pos; i--)

arr[i] = arr[i - 1];

// Insert val at the specified position

arr[pos] = val;

// Increase the current size

(\*n)++;

}

int main() {

int arr[7] = {10, 20, 30, 40, 50};

int n = 5;

int pos = 3;

int val = 25;

// Insert the value at the specified position

insert(arr, &n, pos, val);

for (int i = 0; i < n; i++)

printf("%d ", arr[i]);

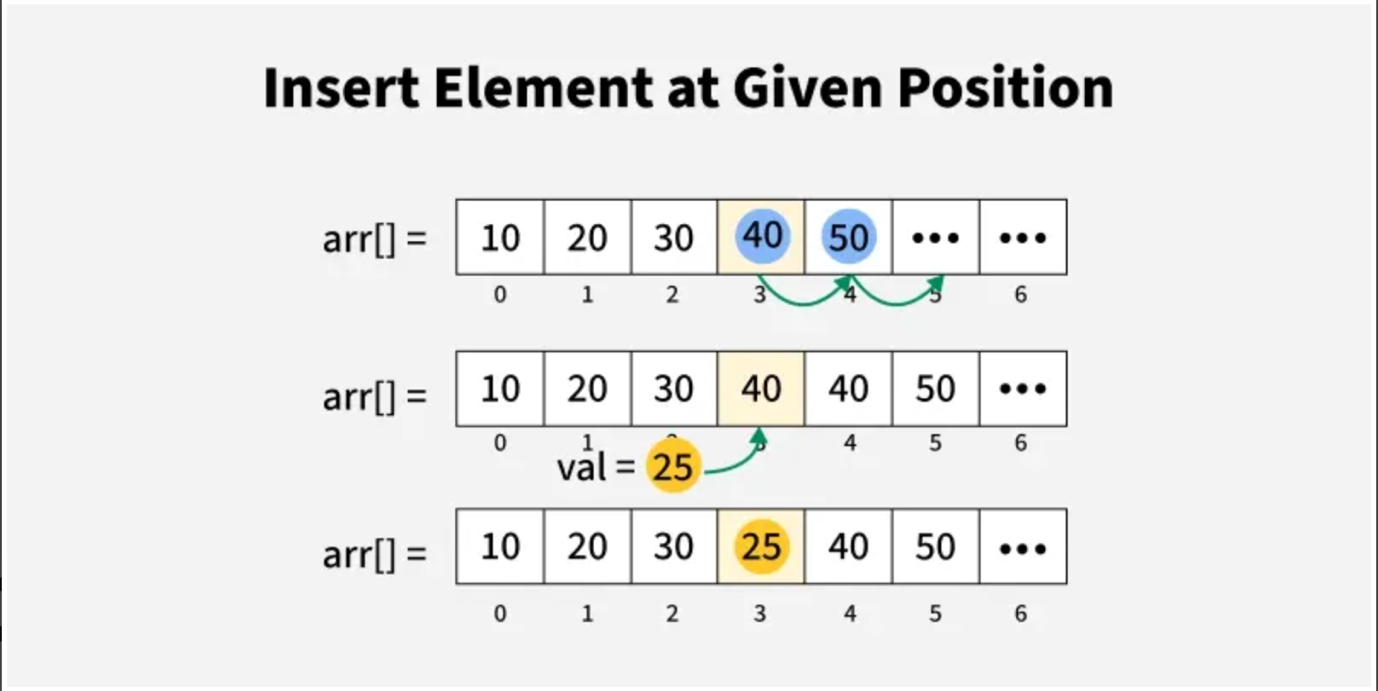
return 0;

}

Output

10 20 30 25 40 50

**Explanation:** In the given program, the function**insert()**shifts all elements starting from the insertion index 3 one step to the right. The new value 25 is then inserted at the desired position, and the size of the array is incremented.



**Time Complexity:** O(n) (for Shifting) + O(1) (for incrementing size) = O(n)  
**Auxiliary Space:** O(1)

## Insert an Element at the End of an Array

If you are inserting an element at the end, there's no need for shifting elements. You can directly add the element at the next available position and increase the size.

#include <stdio.h>

void insertLast(int arr[], int \*n, int val) {

// Insert val at last

arr[\*n] = val;

// Increase the current size

(\*n)++;

}

int main() {

int arr[7] = {10, 20, 30, 40, 50};

int n = 5;

int val = 25;

// Insert the value at the end

insertLast(arr, &n, val);

for (int i = 0; i < n; i++)

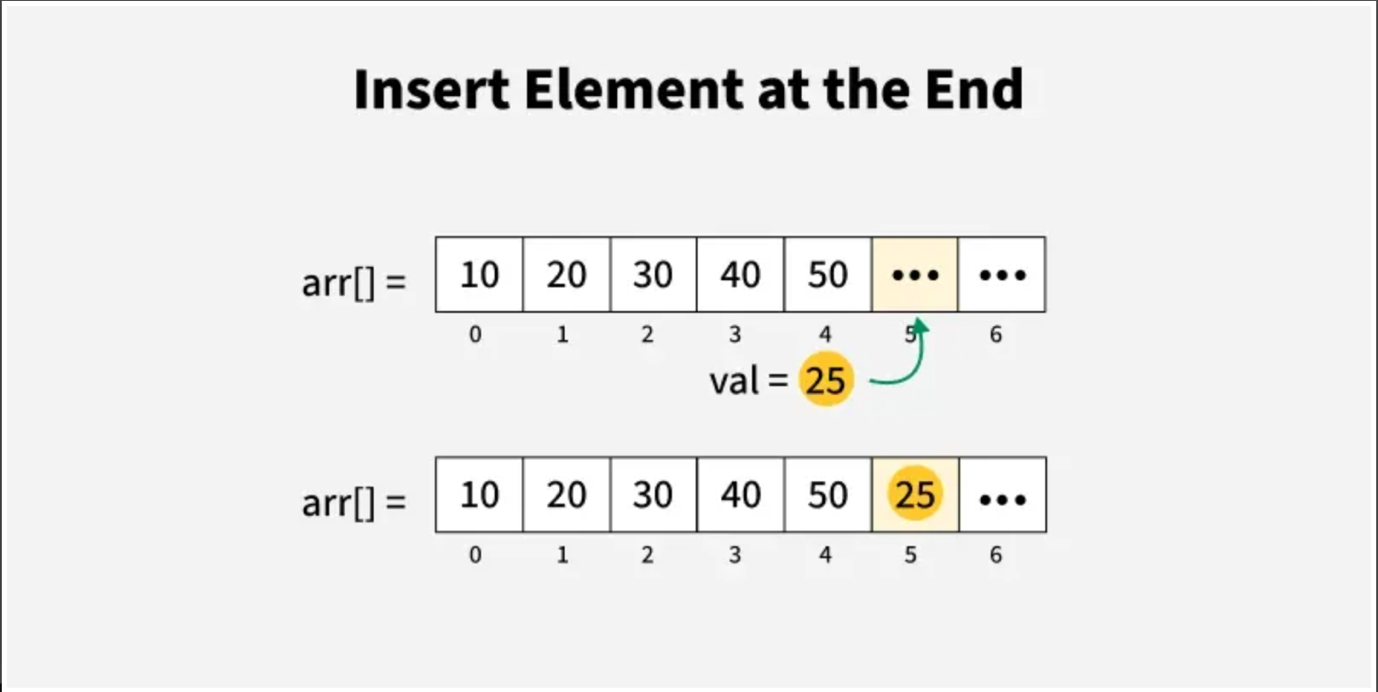
printf("%d ", arr[i]);

return 0;

}

Output

10 20 30 40 50 25



**Time Complexity:** O(1) (for decrementing size)  
**Auxiliary Space:** O(1)