**Data Structures and Algorithms (DSA)**

Table of Contents

[Introduction 2](#_Toc202799168)

[Data Structures: 2](#_Toc202799169)

[Algorithms: 2](#_Toc202799170)

[Relationship: 2](#_Toc202799171)

[Importance: 2](#_Toc202799172)

[Database 3](#_Toc202799173)

[Data warehouse 3](#_Toc202799174)

[Big Data 3](#_Toc202799175)

[The memory layout of a C program 4](#_Toc202799176)

[Text Segment (Code Segment): 4](#_Toc202799177)

[Data Segment: 4](#_Toc202799178)

[BSS Segment (Block Started by Symbol): 4](#_Toc202799179)

[Heap: 4](#_Toc202799180)

[Stack: 4](#_Toc202799181)

[Key Differences Between Stack and Heap Allocations 5](#_Toc202799182)

[Comparison Chart 6](#_Toc202799183)

[Big O Notation Tutorial - A Guide to Big O Analysis 6](#_Toc202799184)

[Big O notation 6](#_Toc202799185)

[Big O Definition 7](#_Toc202799186)

[Importance of Big O Notation 7](#_Toc202799187)

[A Quick Way to find Big O of an Expression 8](#_Toc202799188)

[Example 1 8](#_Toc202799189)

[Example 2 8](#_Toc202799190)

[Properties of Big O Notation 8](#_Toc202799191)

[1. Reflexivity 8](#_Toc202799192)

[2. Transitivity 9](#_Toc202799193)

[3. Constant Factor 9](#_Toc202799194)

[4. Sum Rule 9](#_Toc202799195)

[5. Product Rule 9](#_Toc202799196)

[6. Composition Rule 9](#_Toc202799197)

[Common Big-O Notations 9](#_Toc202799198)

[1. Linear Time Complexity: Big O(n) Complexity 10](#_Toc202799199)

[2. Logarithmic Time Complexity: Big O(log n) Complexity 10](#_Toc202799200)

[3. Quadratic Time Complexity: Big O(n2) Complexity 10](#_Toc202799201)

[4. Cubic Time Complexity: Big O(n3) Complexity 11](#_Toc202799202)

[5. Polynomial Time Complexity: Big O(nk) Complexity 11](#_Toc202799203)

[6. Exponential Time Complexity: Big O(2n) Complexity 11](#_Toc202799204)

[7. Factorial Time Complexity: Big O(n!) Complexity 12](#_Toc202799205)

[Mathematical Examples of Runtime Analysis 13](#_Toc202799206)

[Algorithmic Examples of Runtime Analysis 13](#_Toc202799207)

[Algorithm Classes with Number of Operations 14](#_Toc202799208)

[Comparison of Big O Notation, Big Ω (Omega) Notation, and Big θ (Theta) Notation 15](#_Toc202799209)

# 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):

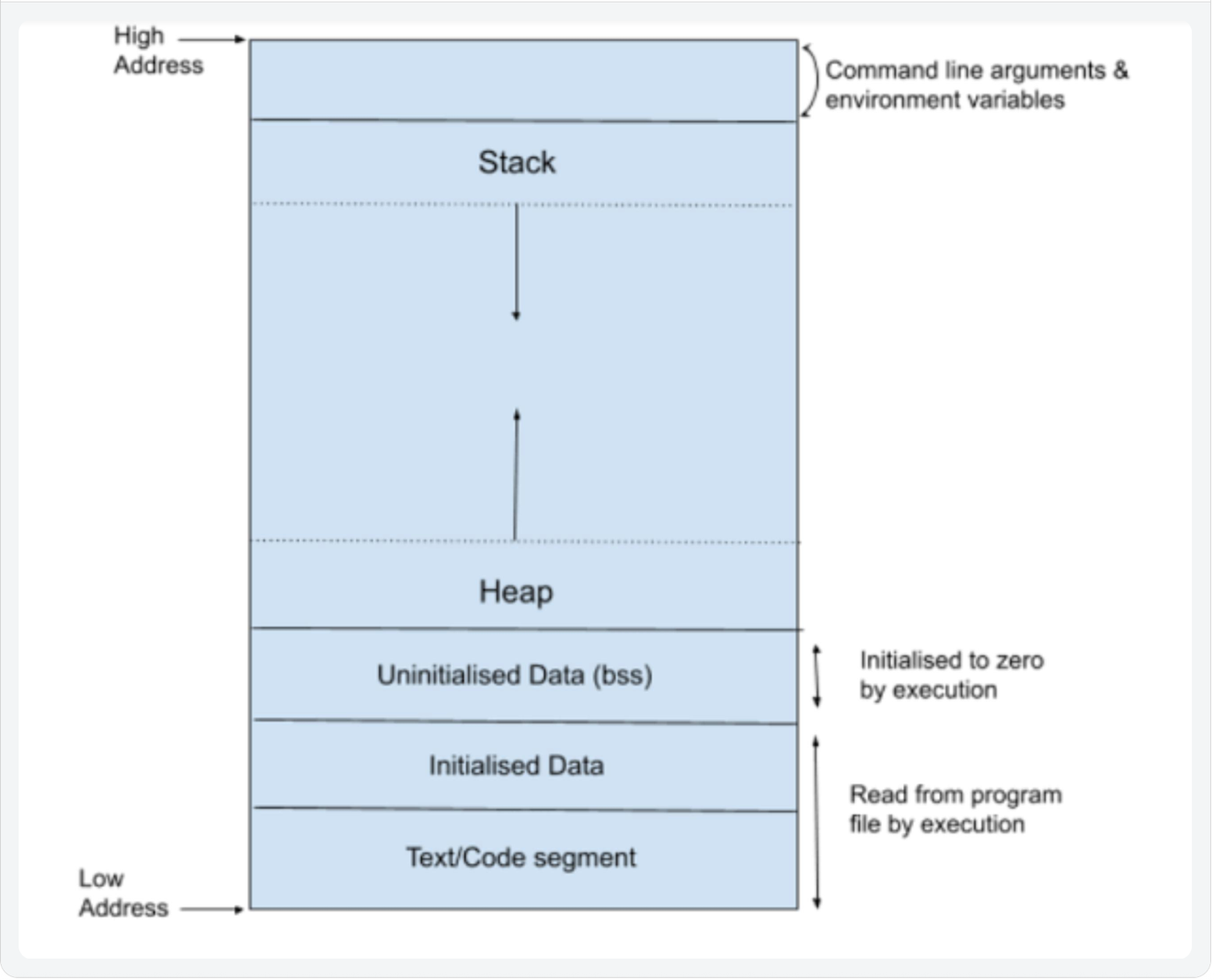
This segment stores uninitialized global and static variables. The operating system initializes the memory in this segment to zero before the program starts execution. Like the data segment, it is a read-write segment.

## 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:

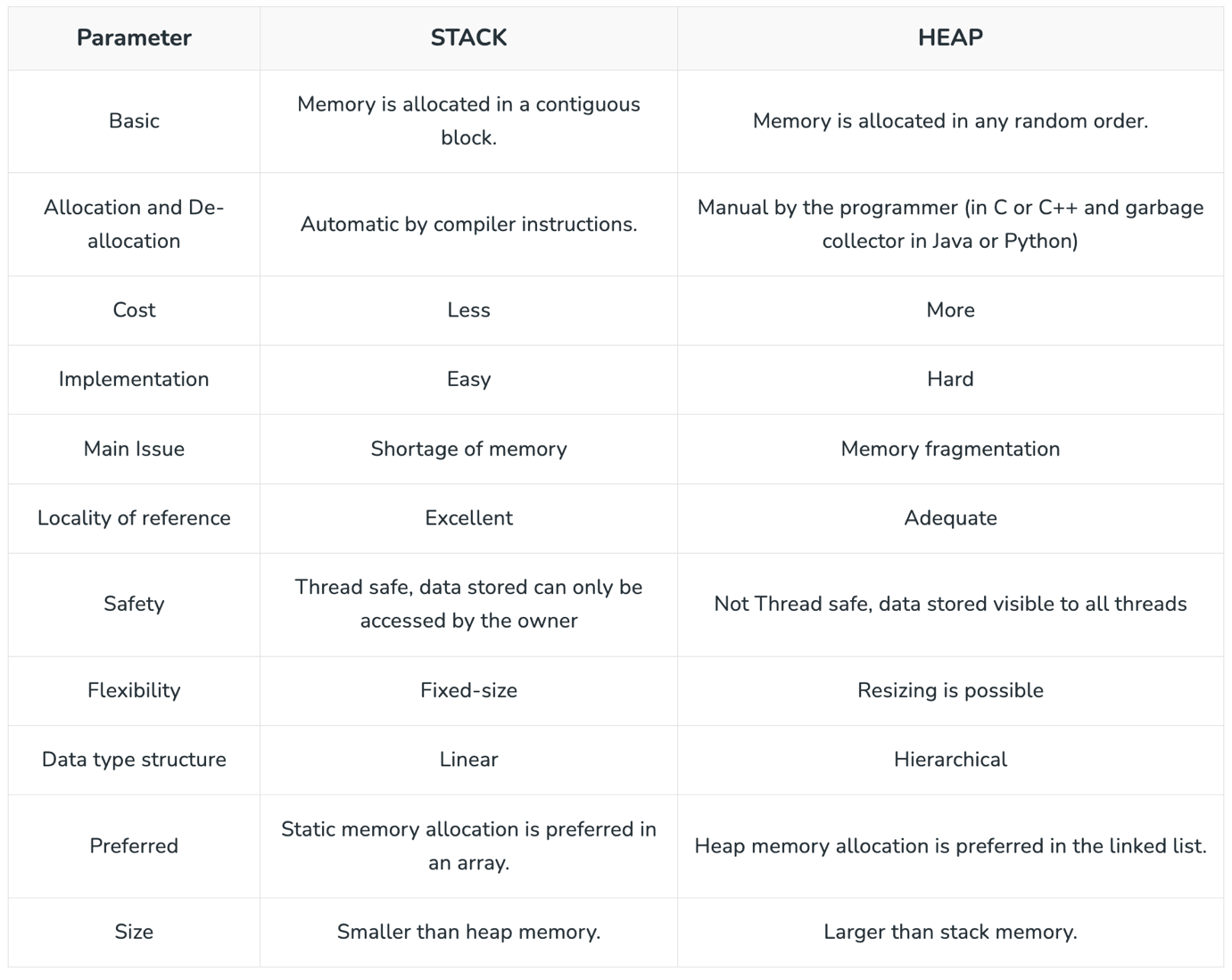
This segment is used for local variables, function arguments, and return addresses during function calls. It operates on a Last-In, First-Out (LIFO) principle, where new data is pushed onto the top of the stack and removed when no longer needed (e.g., when a function returns). The stack typically grows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

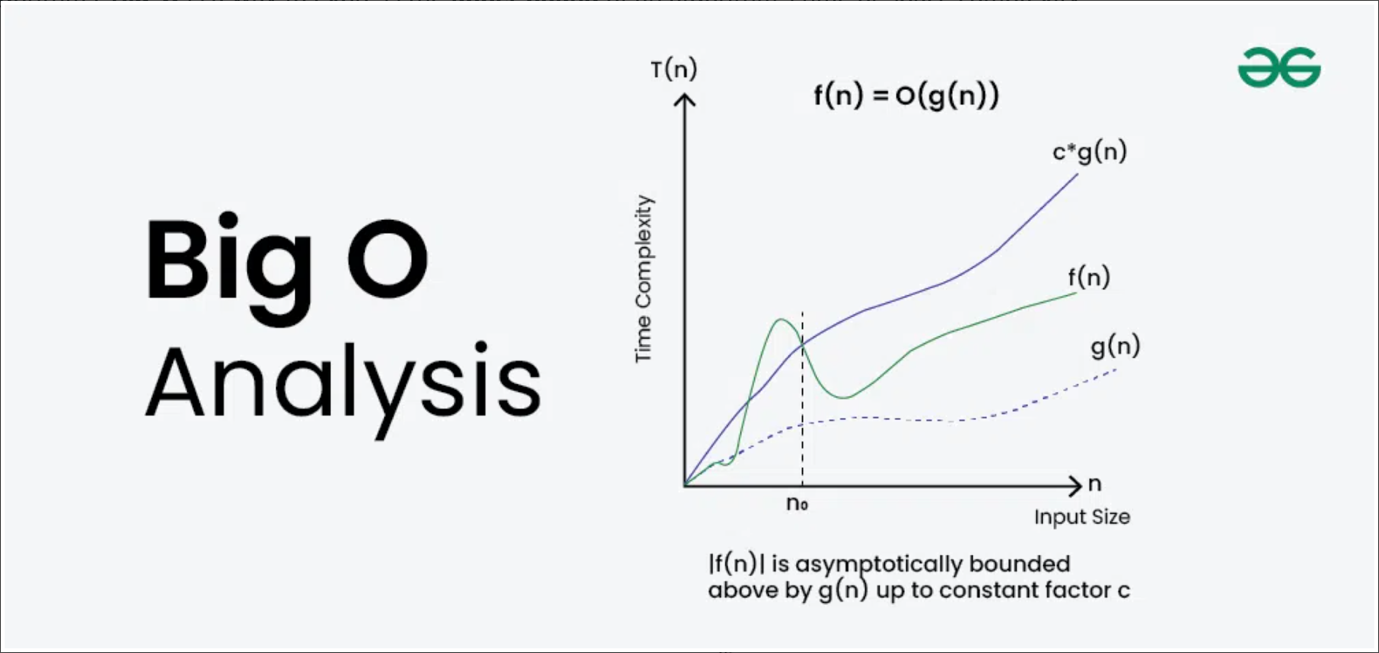
## **Comparison Chart**



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

Big O notationis 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.

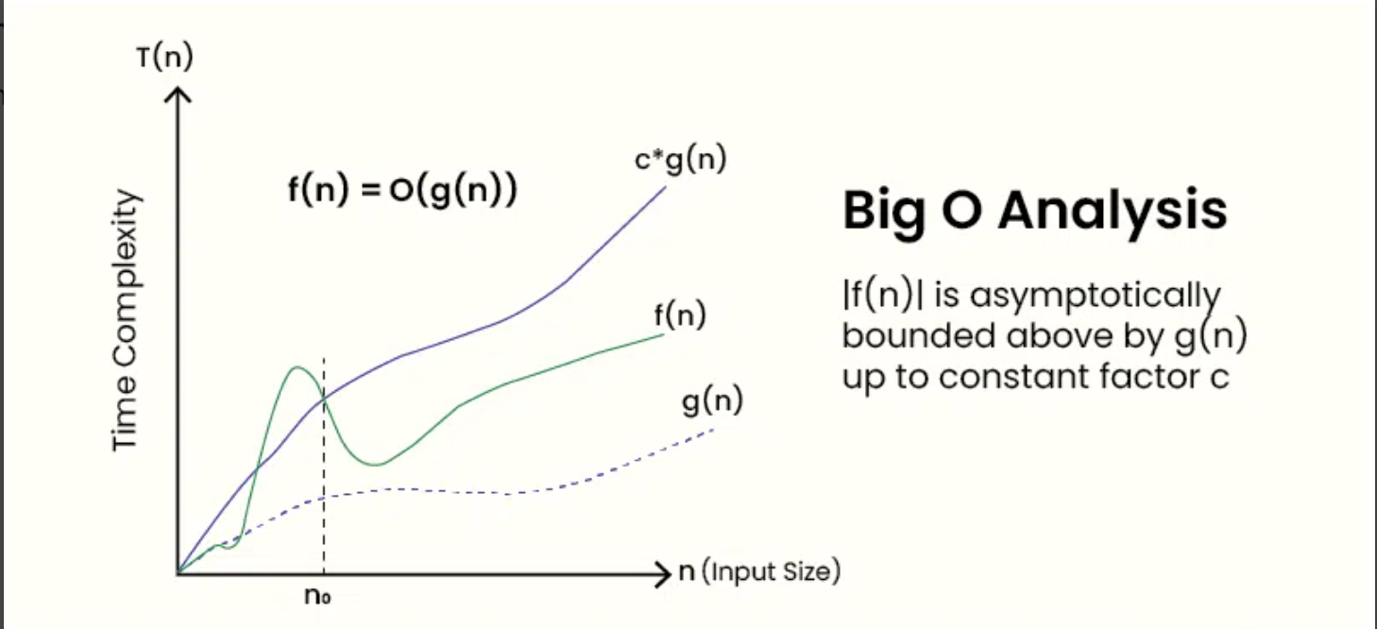
* Describes the asymptotic behaviour (order of growth of time or space in terms of input size) of a function, not its exact value.
* Can be used to compare the efficiency of different algorithms or data structures.
* It provides an**upper limit** on the time taken by an algorithm in terms of the size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

## Properties of Big O Notation

Below are some important Properties of Big O Notation:

### 1. Reflexivity

For any function f(n), f(n) = O(f(n)).

**Example:**

*f(n) = n2, then f(n) = O(n2).*

### 2. Transitivity

If f(n) = O(g(n)) and g(n) = O(h(n)), then f(n) = O(h(n)).

**Example:**

*If f(n) = n^2, g(n) = n^3, and h(n) = n^4, then f(n) = O(g(n)) and g(n) = O(h(n)).   
Therefore, by transitivity, f(n) = O(h(n)).*

### 3. 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:**

*f(n) = n, g(n) = n2. Then f(n) = O(g(n)). Therefore, 2f(n) = O(g(n)).*

### 4. Sum Rule

If f(n) = O(g(n)) and h(n) = O(k(n)), then f(n) + h(n) = O(max( g(n), k(n) ) When combining complexities, only the largest term dominates.

**Example:**

*f(n) = n2, h(n) = n3. Then , f(n) + h(n) = O(max(n2 + n3) = O ( n3)*

### 5. Product Rule

If f(n) = O(g(n)) and h(n) = O(k(n)), then f(n) \* h(n) = O(g(n) \* k(n)).

**Example:**

*f(n) = n, g(n) = n2, h(n) = n3, k(n) = n4. Then f(n) = O(g(n)) and h(n) = O(k(n)). Therefore, f(n) \* h(n) = O(g(n) \* k(n)) = O(n6).*

### 6. Composition Rule

If f(n) = O(g(n)) and g(n) = O(h(n)), then f(g(n)) = O(h(n)).

**Example:**

*f(n) = n2, g(n) = n, h(n) = n3. Then f(n) = O(g(n)) and g(n) = O(h(n)). Therefore, f(g(n)) = O(h(n)) = O(n3).*

## Common Big-O Notations

Big-O notation is a way to measure the time and space complexity of an algorithm. It describes the upper bound of the complexity in the worst-case scenario. Let’s look into the different types of time complexities:

### 1. Linear Time Complexity: Big O(n) Complexity

Linear time complexity means that the running time of an algorithm grows linearly with the size of the input.

For example, consider an algorithm that[traverses through an array to find a specific element](https://www.geeksforgeeks.org/linear-search/):

bool findElement(int arr[], int n, int key)

{

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

if (arr[i] == key) {

return true;

}

}

return false;

}

### 2. Logarithmic Time Complexity: Big O(log n) Complexity

Logarithmic time complexity means that the running time of an algorithm is proportional to the logarithm of the input size.

For example, a[binary search algorithm](https://www.geeksforgeeks.org/binary-search/)has a logarithmic time complexity:

int binarySearch(int arr[], int l, int r, int x)

{

if (r >= l) {

int mid = l + (r - l) / 2;

if (arr[mid] == x)

return mid;

if (arr[mid] > x)

return binarySearch(arr, l, mid - 1, x);

return binarySearch(arr, mid + 1, r, x);

}

return -1;

}

### 3. Quadratic Time Complexity: Big O(n2) Complexity

Quadratic time complexity means that the running time of an algorithm is proportional to the square of the input size.

For example, a simple [bubble sort algorithm](https://www.geeksforgeeks.org/bubble-sort/) has a quadratic time complexity:

void bubbleSort(int arr[], int n)

{

for (int i = 0; i < n - 1; i++) {

for (int j = 0; j < n - i - 1; j++) {

if (arr[j] > arr[j + 1]) {

swap(&arr[j], &arr[j + 1]);

}

}

}

}

### 4. Cubic Time Complexity: Big O(n3) Complexity

Cubic time complexity means that the running time of an algorithm is proportional to the cube of the input size.

For example, a naive[matrix multiplication algorithm](https://www.geeksforgeeks.org/matrix-multiplication/) has a cubic time complexity:

void multiply(int mat1[][N], int mat2[][N], int res[][N])

{

for (int i = 0; i < N; i++) {

for (int j = 0; j < N; j++) {

res[i][j] = 0;

for (int k = 0; k < N; k++)

res[i][j] += mat1[i][k] \* mat2[k][j];

}

}

}

### 5. Polynomial Time Complexity: Big O(nk) Complexity

Polynomial time complexity refers to the time complexity of an algorithm that can be expressed as a polynomial function of the input size**n**. In Big **O** notation, an algorithm is said to have polynomial time complexity if its time complexity is**O(nk)**, where **k** is a constant and represents the degree of the polynomial.

Algorithms with polynomial time complexity are generally considered efficient, as the running time grows at a reasonable rate as the input size increases. Common examples of algorithms with polynomial time complexity include**linear time complexity O(n)**, **quadratic time complexity O(n2)**, and **cubic time complexity O(n3)**.

### 6. Exponential Time Complexity: Big O(2n) Complexity

Exponential time complexity means that the running time of an algorithm doubles with each addition to the input data set.

For example, the problem of[generating all subsets of a set](https://www.geeksforgeeks.org/backtracking-to-find-all-subsets/) is of exponential time complexity:

void generateSubsets(int arr[], int n)

{

for (int i = 0; i < (1 << n); i++) {

for (int j = 0; j < n; j++) {

if (i & (1 << j)) {

cout << arr[j] << " ";

}

}

cout << endl;

}

}

### 7. Factorial Time Complexity: Big O(n!) Complexity

Factorial time complexity means that the running time of an algorithm grows factorially with the size of the input. This is often seen in algorithms that generate all permutations of a set of data.

Here’s an example of a factorial time complexity algorithm, which generates all permutations of an array:

void permute(int\* a, int l, int r)

{

if (l == r) {

for (int i = 0; i <= r; i++) {

cout << a[i] << " ";

}

cout << endl;

}

else {

for (int i = l; i <= r; i++) {

swap(a[l], a[i]);

permute(a, l + 1, r);

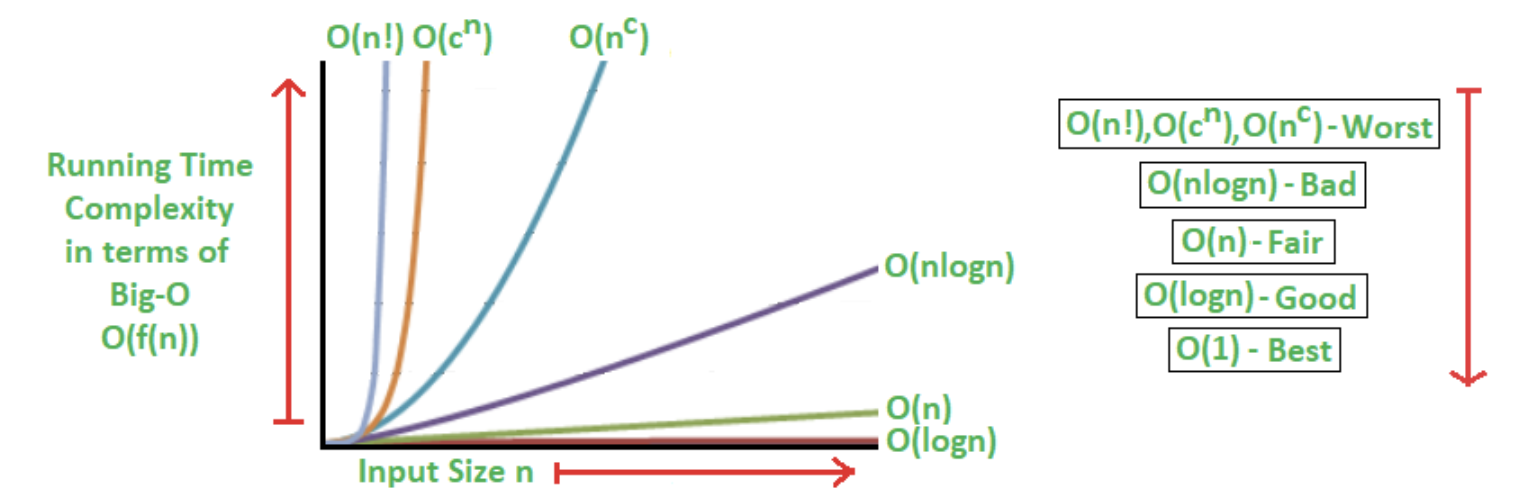
swap(a[l], a[i]); // backtrack

}

}

}

If we plot the most common Big O notation examples, we would have graph like this:



## **Mathematical Examples of Runtime Analysis**

Below table illustrates the runtime analysis of different orders of algorithms as the input size (n) increases.

| **n** | **log(n)** | **n** | **n \* log(n)** | **n^2** | **2^n** | **n!** |
| --- | --- | --- | --- | --- | --- | --- |
| 10 | 1 | 10 | 10 | 100 | 1024 | 3628800 |
| 20 | 2.996 | 20 | 59.9 | 400 | 1048576 | 2.432902e+1818 |

## **Algorithmic Examples of Runtime Analysis**

Below table categorizes algorithms based on their runtime complexity and provides examples for each type.

| **Type** | **Notation** | **Example Algorithms** |
| --- | --- | --- |
| Logarithmic | O(log n) | Binary Search |
| Linear | O(n) | Linear Search |
| Superlinear | O(n log n) | Heap Sort, Merge Sort |
| Polynomial | O(n^c) | Strassen’s Matrix Multiplication, Bubble Sort, Selection Sort, Insertion Sort, Bucket Sort |
| Exponential | O(c^n) | Tower of Hanoi |
| Factorial | O(n!) | Determinant Expansion by Minors, Brute force Search algorithm for Traveling Salesman Problem |

## Algorithm Classes with Number of Operations

Below are the classes of algorithms and their number of operations assuming that there are no constants.

| **Big O Notation Classes** | **f(n)** | **Big O Analysis (number of operations) for n = 10** |
| --- | --- | --- |
| **constant** | O(1) | 1 |
| **logarithmic** | O(logn) | 3.32 |
| **linear** | O(n) | 10 |
| **O(nlogn)** | O(nlogn) | 33.2 |
| **quadratic** | O(n2) | 102 |
| **cubic** | O(n3) | 103 |
| **exponential** | O(2n) | 1024 |
| **factorial** | O(n!) | 10! |

## Comparison of Big O Notation, Big Ω (Omega) Notation, and Big θ (Theta) Notation

Below is a table comparing Big O notation, Ω (Omega) notation, and θ (Theta) notation:

| **Notation** | **Definition** | **Explanation** |
| --- | --- | --- |
| Big O (O) | f(n) ≤ C \* g(n) for all n ≥ n0 | Describes the upper bound of the algorithm's running time. Used most of the time. |
| Ω (Omega) | f(n) ≥ C \* g(n) for all n ≥ n0 | Describes the lower bound of the algorithm's running time . Used less |
| θ (Theta) | C1 \* g(n) ≤ f(n) ≤ C2 \* g(n) for n ≥ n0 | Describes both the upper and lower bounds of the algorithm's **running time**. Also used a lot more and preferred over Big O if we can find an exact bound. |

In each notation:

* **f(n)** represents the function being analyzed, typically the algorithm's time complexity.
* **g(n)** represents a specific function that bounds**f(n)**.
* **C, C1​,**and **C2**​ are constants.
* **n0**​ is the minimum input size beyond which the inequality holds.

These notations are used to analyse algorithms based on their**worst-case (Big O)**,**best-case (Ω)**, and **average-case (θ)** scenarios.