Big O Notation Tutorial – A Guide to Big O Analysis

Last Updated : 29 Mar, 2024

Big O notation is a powerful tool used in computer science to describe the time complexity or space complexity of algorithms. It provides a standardized way to compare the efficiency of different algorithms in terms of their worst-case performance. Understanding **Big O notation** is essential for analyzing and designing efficient algorithms.

In this tutorial, we will cover the basics of **Big O notation**, its significance, and how to analyze the complexity of algorithms using **Big O**.

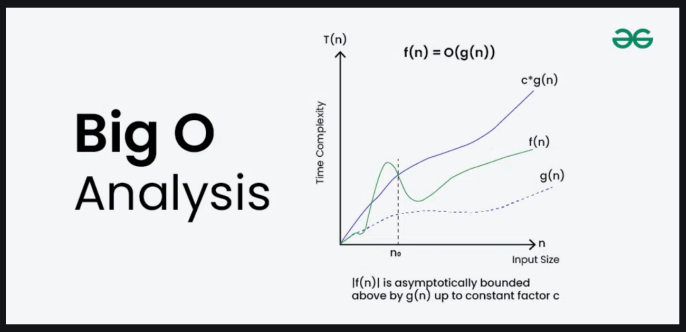


Table of Content

What is Big-O Notation?

Big-O, commonly referred to as "**Order of**", is a way to express the **upper bound** of an algorithm's time complexity, since it analyses the **worst-case** situation of algorithm. It provides an **upper limit** on the time taken by an algorithm in terms of the size of the input. 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 notation*** *is used to describe the performance or complexity of an algorithm. Specifically, it describes the* ***worst-case scenario*** *in terms of* ***time*** *or* ***space complexity.***

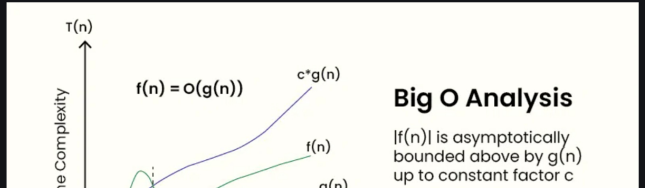
Important Point:

- **Big O notation** only describes the asymptotic behavior of a function, not its exact value.
- The **Big O notation** can be used to compare the efficiency of different algorithms or data structures.

Definition of Big-O Notation:

Given two functions **f(n)** and **g(n)**, we say that **f(n)** is **O(g(n))** if there exist constants **c > 0** and **n_0 >= 0** such that **f(n) <= c*g(n)** for all **n >= n_0**.

In simpler terms, **f(n)** is **O(g(n))** if **f(n)** grows no faster than **c*g(n)** for all n >= n_0 where c and n_0 are constants.

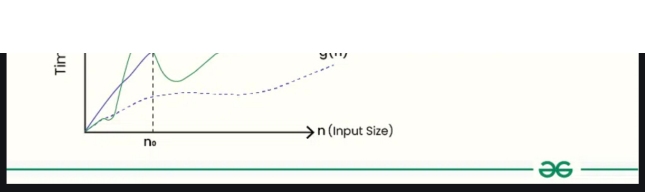


Avoid these Ads for Just $17 Per Month
Limited Time Offer
Get 1 Year Free Extension!
Go Premium

AD
EVEN TERRORISTS
Learn more

AD
fiverr pro.
Give your in-house team the talent they deserve.
Find Top Freelancers →
Jonathan C.
Voiceover Artist

AD
With hand-picked talent, you know you're in good hands.
Count on an expert →
fiverr.

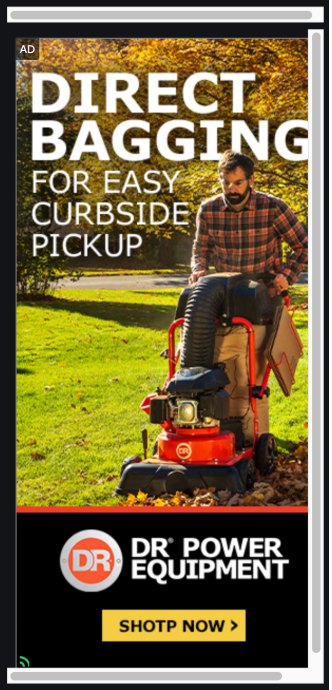


Why is Big O Notation Important?

Big O notation is a mathematical notation used to describe the worst-case time complexity or efficiency of an algorithm or the worst-case space complexity of a 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.



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:

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

2. Transitivity:

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

Example:

$f(n) = n^3$, $g(n) = n^2$, $h(n) = n^4$. Then f(n) = O(g(n)) and g(n) = O(h(n)). Therefore, 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:

$f(n) = n$, $g(n) = n^2$. 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(g(n)), then f(n) + h(n) = O(g(n)).

Example:

$f(n) = n^2$, $g(n) = n^3$, $h(n) = n^4$. Then f(n) = O(g(n)) and h(n) = O(g(n)). Therefore, f(n) + h(n) = O(g(n)).

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:

*$f(n) = n$, $g(n) = n^2$, $h(n) = n^3$, $k(n) = n^4$. Then f(n) = O(g(n)) and h(n) = O(k(n)). Therefore, f(n) * h(n) = O(g(n) * k(n)) = $O(n^5)$.*

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) = n^2, g(n) = n, h(n) = n^3. Then f(n) = O(g(n)) and g(n) = O(h(n)). Therefore, f(g(n)) = O(h(n)) = O(n^3).

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:

Code Snippet

```
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 has a logarithmic time complexity:

Code Snippet

```
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(n^2) 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 has a quadratic time complexity:

Code Snippet

```
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(n^3) 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 has a cubic time complexity:

Code Snippet

```
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(n^k) 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(n^k)**, 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(n^2)**, and **cubic time complexity O(n^3)**.

6. Exponential Time Complexity: Big O(2^n) 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 is of exponential time complexity:

Code Snippet

```
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;
    }
}
```

Factorial Time Complexity: Big O(n!) Complexity

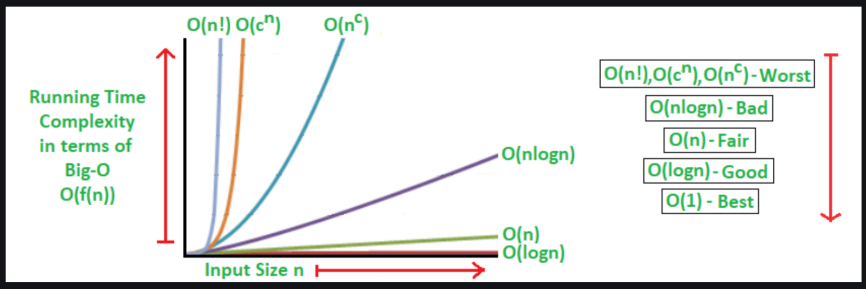
Factorial time complexity means that the running time of an algorithm grows factorially with the size of the input. This is often seen in algorithms that generate all permutations of a set of data.

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

Code Snippet

```
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:



How to Determine Big O Notation?

Big O notation is a mathematical notation used to describe the **asymptotic behavior** of a function as its input grows infinitely large. It provides a way to characterize the efficiency of algorithms and data structures.

Steps to Determine Big O Notation:

1. Identify the Dominant Term:

- Examine the function and identify the term with the highest order of growth as the input size increases.
- Ignore any constant factors or lower-order terms.

2. Determine the Order of Growth:

- The order of growth of the dominant term determines the Big O notation.

3. Write the Big O Notation:

- The Big O notation is written as O(f(n)), where f(n) represents the dominant term.
- For example, if the dominant term is n^2, the Big O notation would be O(n^2).

4. Simplify the Notation (Optional):

- In some cases, the **Big O notatio**n can be simplified by removing constant factors or by using a more concise notation.
- For instance, **O(2n)** can be simplified to **O(n).**

Example:

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

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

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 and Execution Time:

Below are the classes of algorithms and their execution times on a computer executing **1 million operation per second (1 sec = 10^6 µsec = 10^3 msec)**:

Big O Notation Classes	f(n)	Big O Analysis (number of operations) for n = 10	Execution Time (1 instruction/µsec)
constant	O(1)	1	1 µsec
logarithmic	O(logn)	3.32	3 µsec
linear	O(n)	10	10 µsec
O(nlogn)	O(nlogn)	33.2	33 µsec
quadratic	$O(n^2)$	10^2	100 µsec
cubic	$O(n^3)$	10^3	1msec
exponential	$O(2^n)$	1024	10 msec
factorial	O(n!)	10!	3.6288 sec

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)	$f(n) \leq C * g(n)$ for all $n \geq n_0$	Describes the upper bound of the algorithm's running time in the **worst case**.
Ω (Omega)	$f(n) \geq C * g(n)$ for all $n \geq n_0$	Describes the lower bound of the algorithm's running time in the **best case**.
θ (Theta)	$C_1 * g(n) \leq f(n) \leq C_2 * g(n)$ for $n \geq n_0$	Describes both the upper and lower bounds of the algorithm's **running time**.

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.
- n_0 is the minimum input size beyond which the inequality holds.

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

Frequently Asked Questions about Big O Notation:

Question 1. What is Big O Notation?

Answer: *Big O Notation is a mathematical notation used to describe the upper bound of an algorithm's time complexity in terms of how it grows relative to the size of the input.*

Question 2. Why is Big O Notation important?

Answer: *It helps us analyze and compare the efficiency of algorithms by focusing on the worst-case scenario and understanding how their performance scales with input size.*

Question 3. How is Big O Notation calculated?

Answer: *Big O Notation is determined by identifying the dominant operation in an algorithm and expressing its time complexity in terms of n, where n represents the input size.*

Question 4. What does O(1) mean in Big O Notation?

Answer: *O(1) signifies constant time complexity, indicating that an algorithm's execution time does not change regardless of the input size.*

Question 5. What is the significance of different Big O complexities like O(log n) or O(n^2)?

Answer: *Different complexities like O(log n) or O(n^2) represent how an algorithm's performance scales as the input size increases, providing insights into its efficiency and scalability.*

Question 6. Can Big O Notation be applied to space complexity as well?

Answer: *Yes, Big O Notation can also be used to analyze and describe an algorithm's space complexity, indicating how much memory it requires relative to the input size.*

Related Article:

- Examples of Big-O analysis
- Design and Analysis of Algorithms
- Types of Asymptotic Notations in Complexity Analysis of Algorithms
- Analysis of Algorithms | Big – Ω (Big- Omega) Notation
- Analysis of Algorithms | little o and little omega notations

"The DSA course helped me a lot in clearing the interview rounds. It was really very helpful in setting a strong foundation for my problem-solving skills. Really a great investment, the passion Sandeep sir has towards DSA/teaching is what made the huge difference." - **Gaurav | Placed at Amazon**

Before you move on to the world of development, **master the fundamentals of DSA** on which every advanced algorithm is built upon. Choose your preferred language and start learning today:

DSA In JAVA/C++
DSA In Python
DSA In JavaScript
Trusted by Millions, Taught by One- Join the best DSA Course Today!

Sou... + Follow 97

Previous Article
Introduction to Amortized Analysis

Next Article
Difference between Big O vs Big Theta Θ vs Big Omega Ω Notations

Similar Reads

Similar Reads

Analysis of Algorithms | Big - Θ (Big Theta) Notation

In the analysis of algorithms, asymptotic notations are used to evaluate the performance of an algorithm, in its best cases and worst cases. This article will discuss Big - Theta notations...

6 min read

Analysis of Algorithms | Big-Omega Ω Notation

In the analysis of algorithms, asymptotic notations are used to evaluate the performance of an algorithm, in its best cases and worst cases. This article will discuss Big-Omega Notation...

9 min read

Asymptotic Notation and Analysis (Based on input size) in Complexity Analysis of...

Asymptotic Analysis is defined as the big idea that handles the above issues in analyzing algorithms. In Asymptotic Analysis, we evaluate the performance of an algorithm in terms of...

8 min read

Difference between Big O vs Big Theta Θ vs Big Omega Ω Notations

Prerequisite - Asymptotic Notations, Properties of Asymptotic Notations, Analysis of Algorithms1. Big O notation (O): It is defined as upper bound and upper bound on an algorith...

4 min read

Top 30 Big-O Notation Interview Questions & Answers 2023

Big O notation plays a role, in computer science and software engineering as it helps us analyze the efficiency and performance of algorithms. Whether you're someone preparing for an...

12 min read

Maximize big when both big and small can be exchanged

Given N Big Candies and M Small Candies. One Big Candy can be bought by paying X small candies. Alternatively, one big candy can be sold for Y small candies. The task is to find the...

5 min read

Exponential notation of a decimal number

Given a positive decimal number, find the simple exponential notation (x = a·10^b) of the given number. Examples: Input : 100.0 Output : 1E2 Explanation: The exponential notation of 100.0...

5 min read

Convert Infix To Prefix Notation

Given an infix expression, the task is to convert it to a prefix expression. Infix Expression: The expression of type a 'operator' b (a+b, where + is an operator) i.e., when the operator is...

12 min read

Program to convert Infix notation to Expression Tree

Given a string representing infix notation. The task is to convert it to an expression tree.Expression Tree is a binary tree where the operands are represented by leaf nodes and...

12 min read

Knuth's Up-Arrow Notation For Exponentiation

Knuth's up-arrow notation, also known as Knuth's arrow notation, is a mathematical notation for exponentiation that was introduced by Donald Knuth in his book "Concrete Mathematics". I...

4 min read

Article Tags : Algorithms | Analysis of Algorithms | DSA | Algorithms-Analysis of Algorithms

Practice Tags : Algorithms

GeeksforGeeks
Sanchhaya Education Private Limited

Corporate & Communications Address:- A-143, 9th Floor, Sovereign Corporate Tower, Sector- 136, Noida, Uttar Pradesh (201305) | Registered Address:- K 061, Tower K, Gulshan Vivante Apartment, Sector 137, Noida, Gautam Buddh Nagar, Uttar Pradesh, 201305

Company
About Us
Legal
In Media
Contact Us
Advertise with us
GFG Corporate Solution
Placement Training Program

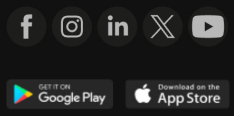
Languages
Python
Java
C++
PHP
GoLang
SQL
R Language
Android Tutorial

DSA
Data Structures
Algorithms
DSA for Beginners
Basic DSA Problems
DSA Roadmap
Top 100 DSA Interview Problems
DSA Roadmap by Sandeep

Data Science & ML
Data Science With Python
Data Science For Beginner
Machine Learning
ML Maths
Data Visualisation
Pandas
NumPy
NLP

Web Technologies
HTML
CSS
JavaScript
TypeScript
ReactJS
NextJS
Bootstrap
Web Design

Python Tutorial
Python Programming Examples
Python Projects
Python Tkinter
Web Scraping
OpenCV Tutorial
Python Interview Question
Django



GeeksforGeeks Community
Tutorials Archive
Jain
All Cheat Sheets
Deep Learning

Computer Science
Operating Systems
Computer Network
Database Management System
Software Engineering
Digital Logic Design
Engineering Maths
Software Development
Software Testing

DevOps
Git
Linux
AWS
Docker
Kubernetes
Azure
GCP
DevOps Roadmap

System Design
High Level Design
Low Level Design
UML Diagrams
Interview Guide
Design Patterns
OOAD
System Design Bootcamp
Interview Questions

Inteview Preparation
Competitive Programming
Top DS or Algo for CP
Company-Wise
Recruitment Process
Company-Wise Preparation
Aptitude Preparation
Puzzles

School Subjects
Mathematics
Physics
Chemistry
Biology
Social Science
English Grammar
Commerce
World GK

GeeksforGeeks Videos
DSA
Python
Java
C++
Web Development
Data Science
CS Subjects

@GeeksforGeeks, Sanchhaya Education Private Limited, All rights reserved