STACK LEARNER

Mastering DSA for Developers: Start Your DSA Journey

Powered By: Stack Learner Limited

18 April 2025

Meet Your **Instructors**

HM Nayem
Senior Software Engineer

- Your Mentor
- Technical Screener & Senior Software Engineer At Toptal
- CTO At Stack Learner
- Author of Two Programming Books
- Taken 1000+ Technical Lectures
- Created 1500+ Technical Videos

What We Are Going To Do In This Workshop?

1. Explore Common DSA
2. Observe Real World Relationships with DSA
3. Enjoy A Magic Show

DAY 01

Introduction to DSA, Asymptotic Analysis & Problem Solving

by HM NAYEM

TODAY'S **AGENDA**

DAY 1

Introduction
9:00 - 9:15

Difference Between Data Structure & Algorithm
9:15 - 9:30

Common Data Structures
9:30 - 9:55

Common Operations
10:10 - 10:40

How to Choose The Correct DSA
10:40 - 11:00

Asymptotic Analysis
11:00 - 11:30

Let's Start . . .

You don’t know DSA. Admit it.
We’re not here to "learn" in the traditional sense — we’re here to enjoy the journey, ask questions, break things, and build true understanding from the ground up.

No pressure. No ego. Just curiosity and growth.
Let’s have some fun!

DSA

Every Programmer Already Know and Using DSA

DSA stands for **Data structures & Algorithms.** These are two different terms.

- **Data Structures:** The storage where you can store data.

  In computer science, a **data structure** is like those storage items — it's a way of organizing and storing data so you can use it effectively.

  Imagine you're organizing your house. You have different storage items like:
  - A drawer for socks
  - A bookshelf for books
  - A shoe rack for shoes
  - A fridge for food

- **Algorithms:** The procedures, the operations we need to work on the data.

  In programming, an algorithm is a set of instructions to manipulate data and solve a problem.

  Now, imagine you're baking a cake. You follow **a step-by-step recipe:**

  - Get ingredients
  - Mix them
  - Bake for 30 minutes
  - Serve

Why Algorithm always comes with Data Structures

Common Data Structures

List of Common Structures

Array	Linked List	Stack	Queue	Hash Table
Set	Tree	Heap	Trie	Graph

Common Operations

Operation	Meaning
Insert	Add new data into the structure or memory
Delete	Remove data from a specific location
Update	Modify existing data
Search/Find	Locate specific data based on a value or key
Read/Access	Retrieve data from a location without modifying it
Traverse	Visit all elements in a structured way (e.g., one by one, top to bottom)
Sort	Rearrange data based on a rule (ascending, alphabetical, etc.)

Common Operations

Operation	Meaning
Transform	Convert data into another format/structure
Merge	Combine two or more data collections into one
Split	Divide a data collection into parts
Map	Apply a function to every element (common in functional programming)
Filter	Select elements based on a condition
Aggregate	Compute a summary (sum, average, count, etc.)

Why do we need different data structures and algorithms?

Untidy Expensive Room

VS

Tidy Cheap Room

How To Choose The Correct Data Structure

Between these which one will perform better?

```javascript
function quickSort(arr) {
  function swap(array, i, j) {
    let temp = array[i];
    array[i] = array[j];
    array[j] = temp;
  }

  function partition(array, low, high) {
    let pivot = array[high];
    let i = low - 1;

    for (let j = low; j < high; j++) {
      if (array[j] <= pivot) {
        i++;
        swap(array, i, j);
      }
    }

    swap(array, i + 1, high);
    return i + 1;
  }

  function quickSortRecursive(array, low, high) {
    if (low < high) {
      let pi = partition(array, low, high);
      quickSortRecursive(array, low, pi - 1);  // Left sub-array
      quickSortRecursive(array, pi + 1, high); // Right sub-array
    }
  }

  let array = arr.slice();
  quickSortRecursive(array, 0, array.length - 1);
  return array;
}
```

VS

```javascript
function bubbleSort(arr) {
  let n = arr.length;
  for (let i = 0; i < n; i++) {
    for (let j = 0; j < n - i - 1; j++) {
      if (arr[j] > arr[j + 1]) {
        [arr[j], arr[j + 1]] = [arr[j + 1], arr[j]];
      }
    }
  }
  return arr;
}
```

STACK LEARNER

- But how can we know which data structures will perform better on which data structures?
- How do we measure?
- What is the metric?

Asymptotic Analysis

What is Asymptotic Analysis?

Asymptotic analysis is **a method used to describe the limiting behavior of a function or algorithm as its input size increases towards infinity.** It's particularly useful in computer science for understanding the efficiency and scalability of algorithms by focusing on how their performance changes with large inputs, ignoring constant factors and low-level details

Key Concepts:

Focus on Growth Rate

Asymptotic analysis focuses on the rate at which an algorithm's resource usage (e.g., time or space) grows as the input size increases.

Asymptotic Notations

It employs mathematical notations like Big O, Big Omega, and Big Theta to express these growth rates.

Ignoring Constants

Constant factors and specific input sizes are often ignored because they become less relevant as the input grows large.

Key Concepts:

Scalability

Asymptotic analysis helps determine how well an algorithm will perform with larger datasets or inputs, providing insights into its scalability.

Performance Comparison

It allows for the comparison of different algorithms' efficiencies by looking at their asymptotic growth rates.

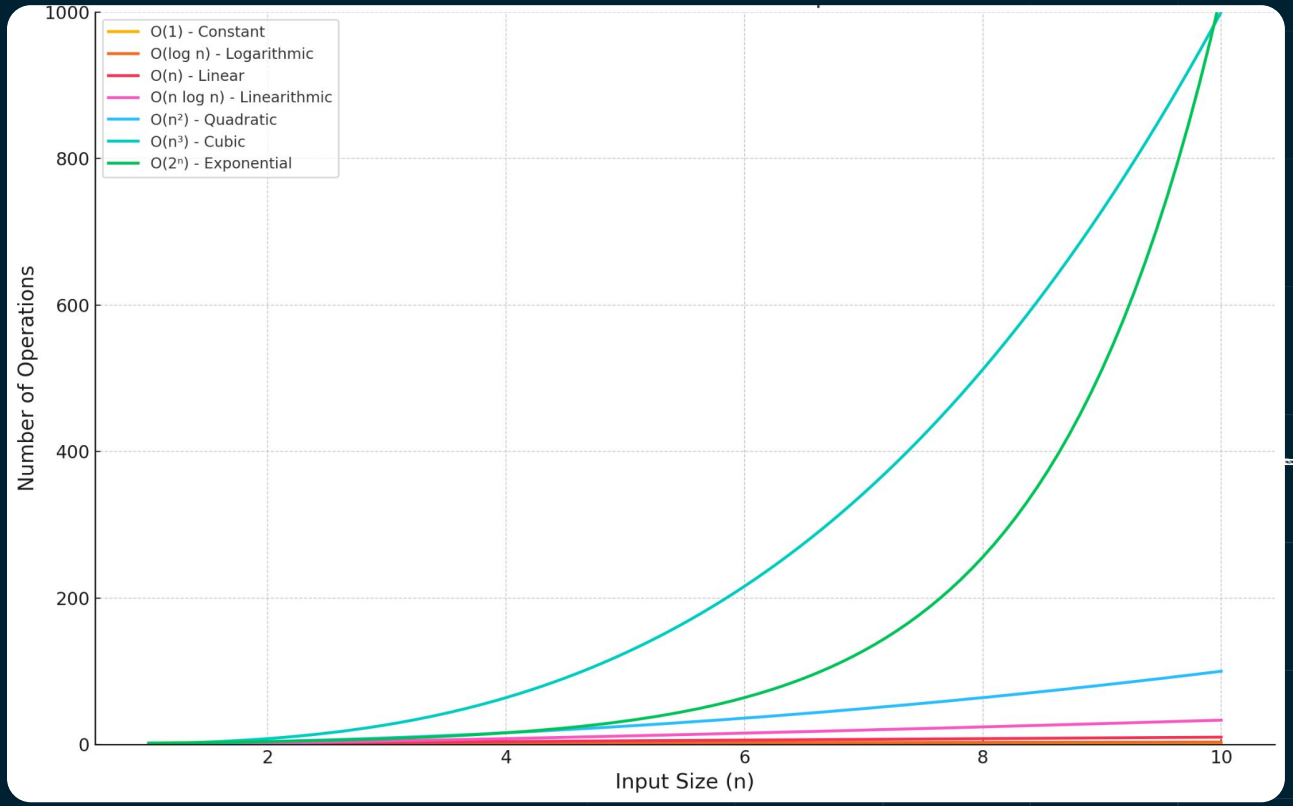
Notations

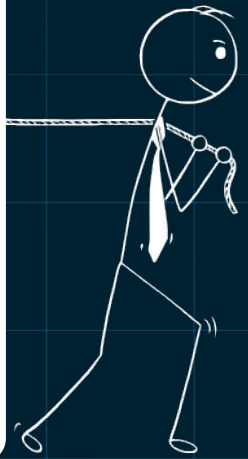
Big (O) **Upper Bound**	In the worst case scenario the algorithm will take at most this much time. Used the most in algorithm analysis. Ex. You were looking for a book and found it at the last.
Big (Ω) **Lower Bound**	In the best case scenario the algorithm will take this much time. Ex. You found the desired book at the first attempt.
Big (Θ) **Tight Bound**	Describe the exact growth rate. The algorithm takes this much time in all cases. Used when the upper and lower bounds are the same.

Growth Functions

Notation	Name	Example	Meaning (as n grows)
O(1)	Constant Time	Accessing array by index arr[5]	Always takes same time
O(log n)	Logarithmic Time	Binary Search	Very efficient; grows slowly
O(n)	Linear Time	Traversing an array	Time grows directly with n
O(n log n)	Linearithmic Time	Merge Sort, Quick Sort (avg)	Between linear and quadratic
$O(n^2)$	Quadratic Time	Nested loops (e.g., Bubble Sort)	Grows fast as n increases
$O(n^3)$	Cubic Time	3-nested loops (e.g., matrix mult.)	Slower than quadratic
$O(2^n)$	Exponential Time	Recursive Fibonacci	Explodes very quickly
O(n!)	Factorial Time	Solving traveling salesman brute-force	Extremely slow, impractical

Growth of Common Time Complexities





Best, Worst, and Average Case

Term	Meaning	Example (Linear Search)
Best Case	Fastest scenario (minimum time)	Element is at the first position
Worst Case	Slowest scenario (maximum time)	Element is at the end or not present
Average Case	Expected time over all inputs	Element is somewhere randomly

The Domination Rules

When combining multiple time complexities:

- Take the term with the highest growth rate.
- It dominates the rest as input size n grows large.

This is called the “dominant term” rule in asymptotic analysis.

Because Big-O is about **growth trends**, not exact counts. For large n:

- n^3 grows much faster than n^2, which grows faster than n log n, etc.
- So $n^3 + n^2 + \log n$ is **$O(n^3)$**

This is like adding pennies to millions - the pennies become **negligible**.

Space Complexity

Space complexity includes:

- Input storage
- Temporary variables
- Function call stack
- Data structures used

Thank You!