# DSA

## Module 1 - Introduction to Algorithms & Programming

By Pallavi Pannu

# Agenda

1. Introduction to Algorithms.
2. How to construct an algorithm?
3. Introduction to Time and Space Complexity.
4. Time complexities notation ( Big-O, Omega, theta)
5. Interview questions for Time complexities.
6. Arrays
7. 1d arrays, how to access an element?

# What is Algorithm?

It is a combination of sequence of finite steps to solve a particular problem.

Example :

MTN()  // **Multiply 2 numbers**

{

1. Take two numbers (a,b)
2. Multiply a and b and store in c
3. print(c)

}

# Properties of Algorithm

1. It should terminate after finite time.
2. It should produce at least 1 output.
3. Every statement in the algorithm should be unambiguous.
4. Algorithm is independent of programming language.
5. Every step in the algorithm should perform some operation.

# Steps required to construct algorithm

1. Problem Definition **(What is input and what is output).**
2. Design Algorithm **(divide and conquer, graphs, dynamic programming,etc).**
3. Draw a flowchart.
4. Testing or verify
5. Implementation or coding using some language.
6. Analysis **(Time and Space Complexity).**

# Time and Space Complexity

While running a program, 2 main resources are needed

1. CPU
2. Memory

How much CPU time —> Time complexity

How much memory —> Space complexity

# Time Complexities Notation

## Big-O Notation (Upper Bound)

### Definition

- **Big-O** describes the **worst-case** or **upper bound** of an algorithm's runtime.
- It answers: *"How slow can this algorithm get as input grows?"*

### Analogy

- **"Your phone's battery lasts *at most* 24 hours."**
  - It could last 10h or 24h, but never *more* than 24h.

## 2. Big-Ω Notation (Lower Bound)

### Definition

- **Big-Ω** describes the **best-case** or **lower bound** of an algorithm's runtime.
- It answers: *"How fast can this algorithm be in the best scenario?*

### Analogy

- **"Your phone's battery lasts *at least* 5 hours."**
  - It could last 5h or longer, but never *less* than 5h.

# 3. Big-Θ Notation (Tight Bound)

## Definition

- **Big-Θ** describes the **exact growth rate** when the best-case and worst-case are the same.
- It answers: *"What is the precise runtime growth of this algorithm?"*

## Comparison Summary

| Notation | Meaning | Example (Linear Search) | Analogy |
| --- | --- | --- | --- |
| **Big-O** | Upper bound | $\mathrm{O(n)}$<br>$O(n)$ | "Takes *at most* 24h." |
| **Big-Ω** | Lower bound | $\Omega(1)$<br>$\Omega(1)$ | "Takes *at least* 5h." |
| **Big-Θ** | Tight bound | $\Theta(\mathrm{n})$<br>$\Theta(n)$ (if best/worst cases match) | "Takes *exactly* 8–10h." |

## Common Misconceptions

1. **"Big-O is the average case."**
    - ❌ No! Big-O is *worst-case*. Average case is different (often harder to compute).
2. **"Big-Θ is always possible to define."**
    - ❌ No! Only if best and worst cases have the same growth rate.

# Interview Questions to Test Understanding

1. **Q**: Is $O(n)$ the same as $\Theta(n)$?

   **A**: No! $O(n)$ is an upper bound (could be better), while $\Theta(n)$ is exact.

2. **Q**: If an algorithm is $\Omega(n^2)$, can it also be $O(n^3)$?

   **A**: Yes! $\Omega(n^2)$ means it's *at least* quadratic, but could be worse (e.g., cubic).

3. **Q**: What's the $\Theta$-complexity of binary search?

   **A**: $\Theta(\log n)$ (best/worst cases are both logarithmic).

## Key Takeaways

- **Big-O** = Worst-case (upper bound).
- **Big-Ω** = Best-case (lower bound).
- **Big-Θ** = Exact growth rate (if best/worst cases match).

# Types of Time Complexities

1. Constant Complexity → $O(1)$
2. Logarithmic Complexity → $O(\log n)$
3. Linear Complexity → $O(n)$
4. Quadratic Complexity → $O(n^2)$
5. Cubic Complexity → $O(n^3)$
6. Polynomial Complexity → $O(n^c)$
7. Exponential Complexity → $O(c^n)$

# Arrays

An array is a **contiguous block of memory** storing elements of the **same type**, accessible via indices.

**Key Properties**:

1. **Fixed Size**: Static size (can be resized dynamically)
2. **Zero-Based Indexing**: First element at index `0`.
3. **Constant-Time Access**: O(1) access to any element (via index).

# 1d arrays

int a[10] = {10,20,30,....,100}

1000    1002    1004    —> memory location

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 10 | 20 | 30 | 40 | … | …. | … | … | …. | 100 |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | | | | | | | 9 |

Index

**How to access?**

- loc(a[5]) = 1000 + (5-0)*2 = 1010
- loc(a[9]) = 1000+(9-0)*2=1018

Given : Arr[lower_bound …… upper bound], Base address, bytes

**loc(a[i]) = Base address+ (i - lower_bound) * c**