1.2 Computational Complexity

Computational complexity is a core area of computer science focused on analyzing and classifying algorithms based on their efficiency. It studies how the **time** and **space** (memory) requirements of algorithms scale with the size of the input.

- Big-O -big O notation
- Big-Ω -big omega notation
- Big-Θ Big-Theta notation

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

Types of Computational Complexity

a.Time Complexity

- Measures **how execution time grows** with input size.
- Expressed using **asymptotic notation** (Big-O, Big-Ω, Big-Θ).

Example:
Linear search on an array of size *n* → time complexity **O(n)**

b.Space Complexity

- Measures **memory used** by an algorithm.
- Includes:
  - Input space
  - Auxiliary (extra) space

Example:
Using an extra array of size *n* → space complexity **O(n)**

Big-O Notation (O)

Definition

Big-O represents the **upper bound** of an algorithm's time (or space) complexity.
It tells us the **worst-case performance**.

Mathematical Definition

An algorithm has time complexity **O(f(n))** if:

$$T(n) \leq c \cdot f(n), \quad \text{for all } n \geq n_0$$

where

- c and n_0 are positive constants.

Meaning

"The algorithm will **not take more than** this amount of time."

Big-Ω Notation (Ω)

Definition

Big-Ω represents the **lower bound** of an algorithm's complexity.
It describes the **best-case performance**.

Mathematical Definition

An algorithm has time complexity **Ω(f(n))** if:

$$T(n) \geq c \cdot f(n), \quad \text{for all } n \geq n_0$$

Meaning

"The algorithm will take **at least** this much time."

Big-Θ Notation (Θ)

Definition

Big-Θ gives a **tight bound**.

It represents both **upper and lower bounds**.

Mathematical Definition

An algorithm has time complexity **Θ(f(n))** if:

$$c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n), \quad \text{for all } n \geq n_0$$

Meaning

"The algorithm's growth rate is **exactly** this."

1. **Example for** Big-O Notation (O)

Example: **Linear Search**

```
for i in range(n):
    if arr[i] == key:
        return i
```

Explanation:

- If the element is **not present** or present at the **last position**,
- The loop runs **n times**.

Complexity:

- **Big-O:** `O(n)`

Meaning:

The algorithm will **not take more than n steps**.

2. Example for Big-Ω Notation (Ω)

Example: **Linear Search**

```
if arr[0] == key:
    return 0
```

Explanation:

- The element is found at the **first position**.
- Only **one comparison** is needed.

Complexity:

- **Big-Ω:** `Ω(1)`

Meaning:

The algorithm will take **at least constant time**.

Example for Big-Θ Notation (Θ)

Example: **Printing all elements**

```
for i in range(n):
    print(arr[i])
```

Explanation:

- Loop always runs **n times**.
- Best, average, and worst cases are the same.

Complexity:

- **Big-Θ:** `Θ(n)`

Meaning:

The algorithm grows **exactly linearly** with input size.