Big- O notation

What Does "**Measuring the Work Done by a Computer**" Mean?

When you write an algorithm, you want to know:

- How fast it is
- How much memory it uses
- How it behaves when input size becomes very large

But timing a program on a computer is NOT reliable because:

1. Different computers run at different speeds
2. Different inputs give different timings
3. Different programming languages/compilers change execution speed

So, computer scientists instead analyze algorithms using **Big-O notation**.

Time Complexity: Time complexity is a way to describe how the running time of an algorithm grows as the size of the input (**n**) increases.

Time complexity tells you **how many steps** an algorithm takes **in terms of n**, not in actual seconds.
It measures **efficiency** and helps **compare** algorithms.

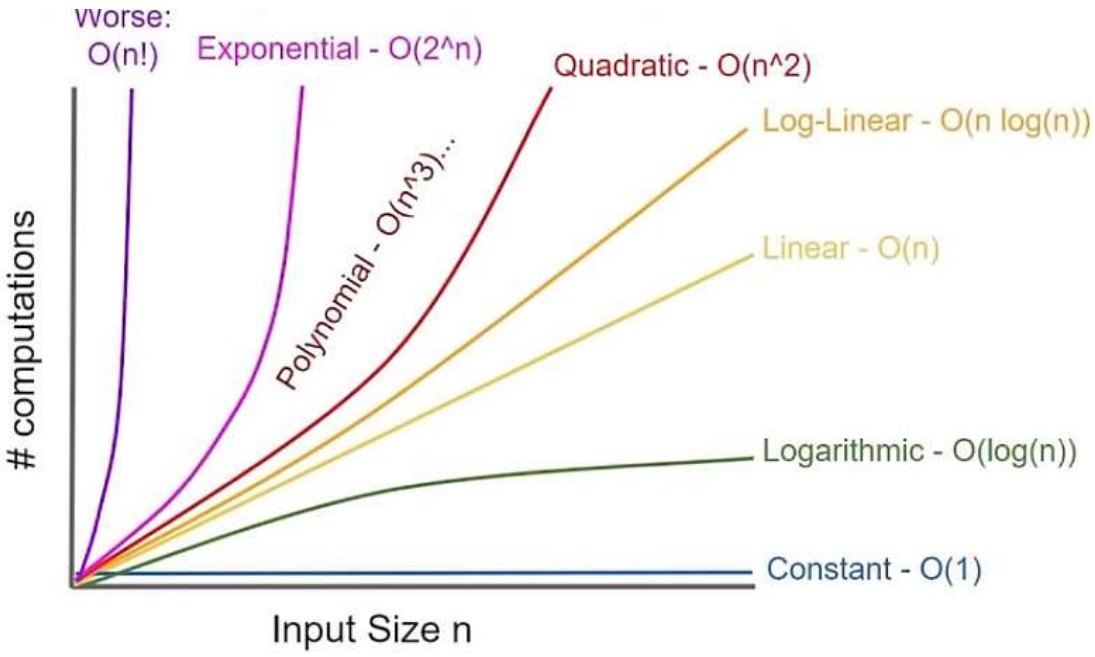
Time Complexity ≠ time taken by algorithm

Big-O Notation — Measuring Algorithm Efficiency

Big-O describes how fast the number of steps grows as input size (n) increases (becomes large).

Examples of growth:

Big-O	Meaning	Example
O(1)	Constant time	Access element at index i. The algorithm takes the same amount of time no matter how big the input is. (e.g., accessing array[5])
O(n)	Linear	Look through an entire list once. Time increases directly with input size. (e.g., scanning a list)
$O(n^2)$	Quadratic	Nested loops. Time grows as the square of the input size. (e.g. matrix addition).
$O(n^3)$	Cubic	Triple-nested loops. Time grows as the cube of the input size. (e.g. matrix multiplication, Gauss-Jordan)
O(log n)	Logarithmic	Binary search
O(n log n)	Linearithmic	Fast sorting algorithms



Ref. APNA college, Time Complexity

Why Big-O?

Because when n becomes very large, constants don't matter we **drop constants and lower-order terms**:

- $7n^2$ → behaves like n^2
- $1000n$ → behaves like n
- $n^2 + 7n + 10$ → behaves like n^2

For example: $3n^2 + 7n \rightarrow O(n^2)$
(7n is much smaller than $3n^2$ for large n.)

Example of ignoring incorrectly:

- $n \log n$ → cannot drop $\log n$,
- (Because it is multiplied, not added)

Understanding Loops in Big-O

A. Loops that run sequentially → add their steps

```
for i in n;                → n steps
for j in n;                → n steps

Total = n + n = 2n → O(n)
//Because one loop finishes before the next starts
```

B. Nested loops → multiply their steps

```
for i in n;
   for j in n;
      step()
```

Total steps = $n \times n = n^2 \rightarrow$ **$O(n^2)$**

C. Divide-and-conquer loops (cutting data in half)

binary search

Each step halves the data:
$n \rightarrow n/2 \rightarrow n/4 \rightarrow n/8 \rightarrow \ldots \rightarrow 1$

Number of halving = $\log_2(n) \rightarrow$ **$O(\log n)$**

EXAMPLE 1: Searching for a Name in a Phonebook

Sequential Search

Look at every name one by one.

Worst case: the name is last → n comparisons
Means, **O(n)**

Binary Search

Phonebook is sorted.
Each step halves the remaining names.

Steps needed ≈ $\log_2(n)$
Means, **O(log n)**

EXAMPLE 2: Adding an Item to a Sorted List

Method 1: Add to end + sort

Sorting takes:

- O(n log n) (fast algorithms)
- OR $O(n^2)$ (slow algorithms)

Either way: Sorting dominates

So final is: O(n log n) or $O(n^2)$ depending on sorting method.

Method 2: Binary search to find location + shift items

Binary search: O(log n)

Shifting items in an array:
O(n)

Total: O(log n + n) → **O(n)** (drop lower term)

EXAMPLE 3: Matrix Addition

Two nested loops:

```
for row in n
   for col in n
       sum[row][col] = A[row][col] + B[row][col]
```

Total = $n \times n = n^2$. Means, **$O(n^2)$**

This is why matrix addition is relatively cheap.

EXAMPLE 4: Matrix Multiplication

Three nested loops:

```
for i in n;
   for j in n;
      for k in n;
         product[i][j] += A[i][k] * B[k][j]
```

Total steps = $n \times n \times n = n^3$

$O(n^3)$ (classic multiplication)

EXAMPLE 5: Gauss-Jordan Elimination

Several parts:

- systemSolvable → $O(n^2)$
- dividing pivot row → $O(n^2)$
- eliminating other rows → $O(n^3)$

Total is dominated by: **$O(n^3)$**

(by ignoring lower order term)

This is why solving linear systems is expensive.

Summary:

Problem	**Steps**	**Big-O**
Sequential search	n	O(n)
Binary search	log n	O(log n)
Matrix addition	n^2	$O(n^2)$
Matrix multiplication	n^3	$O(n^3)$
Gauss–Jordan elimination	n^3	$O(n^3)$
Sorting (good algorithms)	n log n	O(n log n)

Big-O measures how fast an algorithm grows with bigger input.

- Ignore constants
- Ignore lower-order terms
- Focus only on the part that grows fastest
- Count loops to determine the order
- Nested loops multiply
- Sequential loops add

Worst-case analysis helps compare algorithms to find the one that scales better.