**Big O Notation — The Core of Algorithm Efficiency**

## 🧠 ****1️⃣ What is Big O?****

**Big O** tells us **how the runtime or memory usage of an algorithm grows** as the size of input (n) increases.

We don’t care about exact milliseconds;  
we care about **how fast it grows** relative to input size.

It’s a mathematical way to measure scalability.

| **Example** | **Input** | **Time Taken** | **Pattern** |
| --- | --- | --- | --- |
| Print one number | 1 → 10 → 1000 | almost same | O(1) |
| Loop through array | 10 → 1000 → 1e6 | grows linearly | O(n) |
| Nested loops | 10 → 1000 → 1e6 | grows quadratic-ally | O(n²) |
| Binary Search | 10 → 1000 → 1e6 | grows slowly (halves each time) | O(log n) |

## ****3 Constant Time — O(1)****

### Definition:

Execution time **does not depend** on the size of input n,It runs the same number of steps no matter how big the input is.

**Example:** Accessing an array element, adding two numbers, simple arithmetic.

// O(1) Example: Accessing an element

function getFirstElement(arr) {

return arr[0]; // Always 1 operation, no matter array size

}

const numbers = [10, 20, 30, 40, 50];

console.log(getFirstElement(numbers)); // Output: 10

✅ **Key point:** Adding more elements doesn’t change the number of steps.

## 2️⃣ ****Linear Time — O(n)****

**Definition:** Execution time grows **linearly** with input size n.  
Each element is touched once.

**Example:** Sum of all elements in an array.

// O(n) Example: Sum all elements

function sumArray(arr) {

let sum = 0; // 1 step

for (let i = 0; i < arr.length; i++) {

sum += arr[i]; // n steps for n elements

}

return sum;

}

const numbers = [1, 2, 3, 4, 5];

console.log(sumArray(numbers)); // Output: 15

✅ **Key point:** If array doubles in size, number of operations roughly doubles.

## 3️⃣ ****Quadratic Time — O(n²)****

**Definition:** Execution time grows **proportional to n²**.  
Usually occurs when we have **nested loops** over the input.

**Example:** Printing all pairs in an array.

// O(n^2) Example: Print all pairs

function printPairs(arr) {

for (let i = 0; i < arr.length; i++) { // Outer loop → n times

for (let j = 0; j < arr.length; j++){ // Inner loop → n times

console.log(arr[i], arr[j]); // Total steps ~ n \* n = n^2

}

}

}

const numbers = [1, 2, 3];

printPairs(numbers);

✅ **Key point:** If input size doubles, operations **quadruple**. Avoid nested loops when possible in interviews.

## ****4 Logarithmic Time — O(log n)****

**Definition:** Execution time grows **logarithmically** with input size n.  
Usually occurs when **we divide the problem in half each step** (binary search, divide and conquer).

**Example:** Binary Search in a sorted array.

// O(log n) Example: Binary Search

function binarySearch(arr, target) {

let left = 0;

let right = arr.length - 1;

while (left <= right) {

let mid = Math.floor((left + right) / 2); // 1 step

if (arr[mid] === target) {

return mid; // Found → return

} else if (arr[mid] < target) {

left = mid + 1; // Search right half

} else {

right = mid - 1; // Search left half

}

}

return -1; // Not found

}

const sortedNumbers = [1, 3, 5, 7, 9, 11, 13];

console.log(binarySearch(sortedNumbers, 7)); // Output: 3

✅ **Key point:** Each step **cuts the array in half**, so even if array size grows a lot, the number of steps increases very slowly.

| **Complexity** | **How it grows** | **Example** |
| --- | --- | --- |
| O(1) | Constant | Access array element |
| O(n) | Linear | Sum array elements |
| O(n²) | Quadratic | Nested loops / all pairs |
| O(log n) | Logarithmic | Binary search |

## 💡 ****Tips for JavaScript****

Built-in array methods like push, pop are O(1).

indexOf or includes → O(n) (linear search).

Nested loops → watch out for O(n²) accidentally.

Using Map/Set for lookups → O(1) average.

## ****JavaScript Big O Visual Demo****

// Big O Visual Demo in JavaScript

const { performance } = require('perf\_hooks'); // Node.js timing

// Helper to measure time

function measureTime(fn, arr, label) {

const start = performance.now();

fn(arr);

const end = performance.now();

console.log(`${label}: ${(end - start).toFixed(4)} ms`);

}

// 1️⃣ O(1) — Constant Time

function constantTime(arr) {

let first = arr[0]; // Access first element

// Simulate tiny work

let x = first + 10;

return x;

}

// 2️⃣ O(n) — Linear Time

function linearTime(arr) {

let sum = 0;

for (let i = 0; i < arr.length; i++) {

sum += arr[i];

}

return sum;

}

// 3️⃣ O(n^2) — Quadratic Time

function quadraticTime(arr) {

let count = 0;

for (let i = 0; i < arr.length; i++) {

for (let j = 0; j < arr.length; j++) {

count += arr[i] \* arr[j];

}

}

return count;

}

// 4️⃣ O(log n) — Logarithmic Time (Binary Search)

function logarithmicTime(arr, target) {

let left = 0, right = arr.length - 1;

while (left <= right) {

let mid = Math.floor((left + right) / 2);

if (arr[mid] === target) return mid;

else if (arr[mid] < target) left = mid + 1;

else right = mid - 1;

}

return -1;

}

// Test with increasing array sizes

const sizes = [10, 100, 1000, 5000, 10000]; // You can increase gradually

sizes.forEach(n => {

const arr = Array.from({ length: n }, (\_, i) => i);

console.log(`\nArray size: ${n}`);

measureTime(constantTime, arr, "O(1) Constant Time");

measureTime(linearTime, arr, "O(n) Linear Time");

measureTime(quadraticTime, arr, "O(n^2) Quadratic Time (small n only!)");

measureTime(() => logarithmicTime(arr, n - 1), arr, "O(log n) Binary Search");

});

### ****How It Works****

**O(1):** Accessing first element — runtime stays nearly the same regardless of array size.

**O(n):** Summing all elements — runtime grows linearly as array size increases.

**O(n²):** Nested loops — runtime grows quadratically (use smaller n to avoid freezes).

**O(log n):** Binary search — runtime grows very slowly even if n is huge.