Algorithm complexity (Big O notation)

🧠 Algorithm Complexity (Big O Notation) — Explained in Detail

🔷 What Is Algorithm Complexity?

Algorithm complexity refers to how the **performance** of an algorithm changes relative to the size of the input data. It's measured in terms of:

- **Time Complexity**: How long it takes to run
- **Space Complexity**: How much memory it uses

We use **Big O Notation** to describe **worst-case** performance (typically), helping you reason about **scalability**.

🔷 Why Big O?

Big O notation helps answer:

> "If my input grows from 10 items to 10,000, how will performance scale?"

It **abstracts away hardware** and **implementation details** to focus on the **growth rate**.

🔷 Common Time Complexities

Notation	Name	Example	Growth Rate
`O(1)`	Constant	Accessing array element	🟢 Best performance
`O(log n)`	Logarithmic	Binary search	⚡ Fast on large input
`O(n)`	Linear	Loop through array	📈 Scales with size
`O(n log n)`	Linearithmic	Merge Sort, Quick Sort avg	🔁 Efficient sorting
`O(n²)`	Quadratic	Nested loops (e.g. bubble sort)	🐌 Slows fast on large n

$O(2^n)$	Exponential	Recursive Fibonacci	🧨 Explodes quickly
$O(n!)$	Factorial	Permutations	🔥 Very expensive

📌 Examples

1. O(1) — Constant Time

```
function getFirst(arr) {
  return arr[0];
}
```

No matter how big the array is, it always does **one operation**.

2. O(n) — Linear Time

```
function sum(arr) {
  let total = 0;
  for (let num of arr) {
    total += num;
  }
  return total;
}
```

Performs one operation for each element.

3. O(n²) — Quadratic Time

```
function printPairs(arr) {
  for (let i = 0; i < arr.length; i++) {
    for (let j = 0; j < arr.length; j++) {
      console.log(arr[i], arr[j]);
    }
  }
}
```

Nested loop causes exponential growth.

---

4. O(log n) — Logarithmic Time

```
function binarySearch(arr, target) {
  let low = 0, high = arr.length - 1;
  while (low <= high) {
    let mid = Math.floor((low + high) / 2);
    if (arr[mid] === target) return mid;
    else if (arr[mid] < target) low = mid + 1;
    else high = mid - 1;
  }
  return -1;
}
```

Reduces the search space by half each time.

🔷 Space Complexity

Describes how much **additional memory** is required:

```
function makeDouble(arr) {
  let result = [];
  for (let num of arr) {
    result.push(num * 2);
  }
  return result;
}
// Space: O(n) (creates a new array)
```

If the function modifies in place (e.g., sorts in-place), space complexity can be `O(1)`.

🔷 Visual Summary

```
Best → Worst:

O(1) < O(log n) < O(n) < O(n log n) < O(n²) < O(2ⁿ) < O(n!)
```

🔷 Tips to Optimize Big O

- **Avoid nested loops** when possible
- Use **hash maps/objects** for faster lookup (`O(1)` access)
- Choose the right **sorting/search algorithms**
- Use **memoization** or **dynamic programming** to avoid redundant work

---

🧠 Use Cases in Real Projects

Scenario	Desired Complexity
Search in sorted array	`O(log n)` (binary search)
Filter/transform data	`O(n)`
Sort items for display	`O(n log n)`
Compare all user pairs	`O(n²)` (avoid if large)
Generate combinations	`O(2ⁿ)` or worse

🧾 Summary

- **Big O** helps you reason about algorithm **scalability**
- It describes how performance **grows with input size**
- Choose algorithms with **lower complexity** for large data
- Understand **trade-offs** (e.g., speed vs. memory)

Would you like a **cheat sheet image** or **interactive examples** to better visualize Big O complexities?