Big O Notation: Time Complexity & Examples Explained

Table of Contents

1. What Is Big O Notation?

Big O notation is a mathematical notation used in computer science to describe the **upper bound (worst-case scenario)** of an algorithm's runtime or space complexity in terms of the input size (n). It helps compare how efficiently algorithms scale as input grows.

Example:

- **O(1)** → Constant time (e.g., accessing an array element).
- **O(n)** → Linear time (e.g., traversing a list).

- **O(n^2)** → Quadratic time (e.g., nested loops).

2. Big O Notation: Formal Definition

A function `f(n)` is **O(g(n))** if there exist constants C and n_0 such that:

[0 ≤ f(n) ≤ C \cdot g(n) \quad \text{for all} \quad n ≥ n_0]

This means `f(n)` grows **no faster** than `g(n)` multiplied by some constant C for large `n`.

3. Importance of Big O Notation

- **Compares algorithm efficiency** (e.g., O(n) vs. O(n^2)).
- **Predicts scalability** (how performance degrades with larger inputs).
- **Optimizes code** by identifying bottlenecks.
- **Guides algorithm selection** (e.g., choosing Merge Sort (O(n log n)) over Bubble Sort (O(n^2))).

4. Understanding Big O Notation

Big O simplifies functions by focusing on **dominant terms** (highest growth rate) and ignoring constants.

Example:

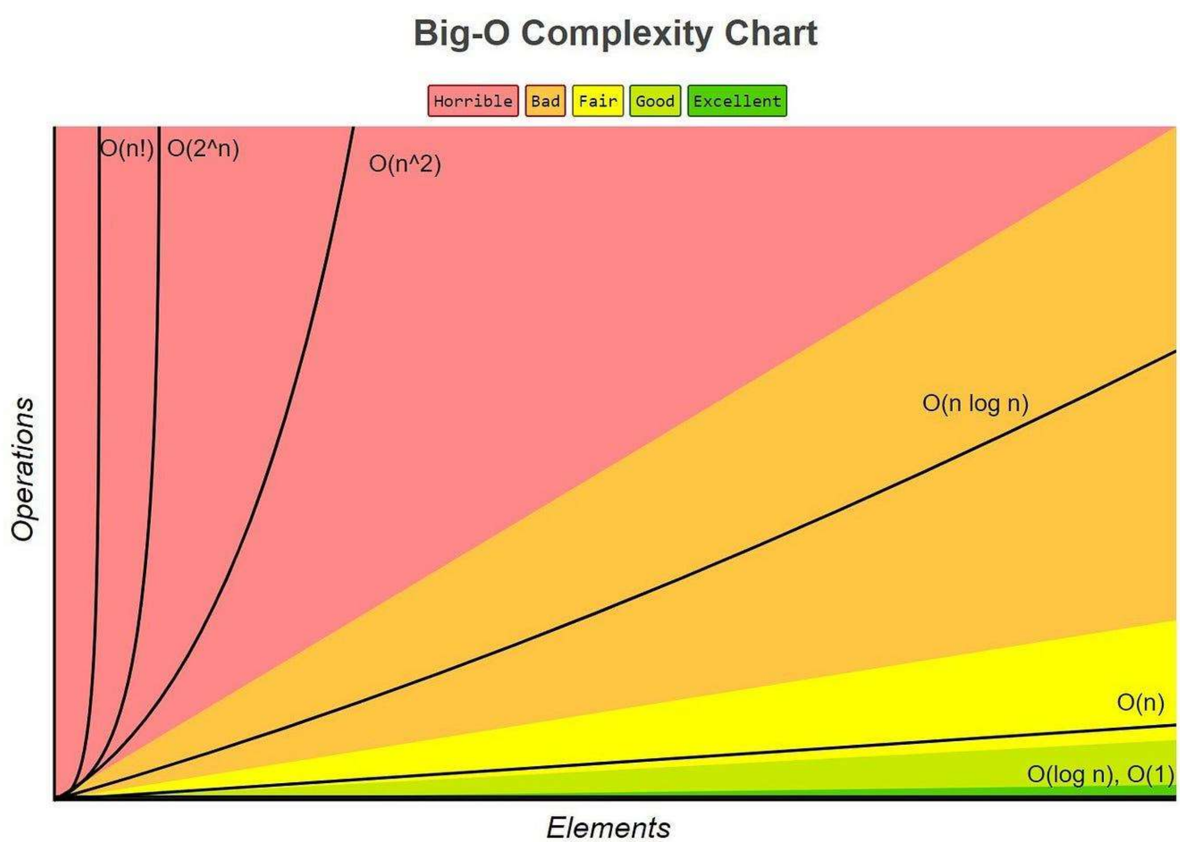
- $f(n) = 6n^4 - 2n^3 + 5$ → **O(n^4)** (only the fastest-growing term matters).
- $f(n) = 3n^2(2n + 1)$ → Expands to $6n^3 + 3n^2$ → **O(n^3)**.

5. Big O Notation Examples

Notation	Name	Example

O(1)	Constant	Array access
O(log n)	Logarithmic	Binary search
O(n)	Linear	Linear search
O(n log n)	Linearithmic	Merge Sort
O(n^2)	Quadratic	Bubble Sort
O(2^n)	Exponential	Recursive Fibonacci
O(n!)	Factorial	Permutations

6. Complexity Comparison Between Typical Big Os



- **O(1) < O(log n) < O(n) < O(n log n) < O(n^2) < O(2^n) < O(n!)**

7. Usage Of Big O Notation

- **Mathematics:** Approximating series (e.g., Taylor series).
- **Computer Science:** Analyzing algorithms (time/space complexity).
- **Infinite vs. Infinitesimal Asymptotics:**
    - **Infinite:** Behavior as $n \to \infty$ (e.g., algorithm runtime).
    - **Infinitesimal:** Behavior as $n \to 0$ (e.g., error analysis).

8. Properties of Big O Notation

1. **Reflexivity:** $f(n) = O(f(n))$.
2. **Transitivity:** If $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$.
3. **Constant Factor:** $k \cdot O(f(n)) = O(f(n))$.
4. **Sum Rule:** $O(f(n)) + O(g(n)) = O(\max(f(n), g(n)))$.
5. **Product Rule:** $O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n))$.

9. Big O With Multiple Variables

Used when an algorithm depends on **multiple inputs** (e.g., rows m and columns n in a matrix).

Example:

- $f(m, n) = O(m + n)$ → Linear in both dimensions.
- $f(m, n) = O(m \cdot n)$ → Quadratic if nested loops.

10. Matters of Notation

- **Correct:** $f(n) \in O(g(n))$ (set notation).
- **Common but misleading:** $f(n) = O(g(n))$ (asymmetric equality).

11. List of Orders of Common Functions

Complexity	Description	Example
O(1)	Constant time	Hash table lookup
*O(log n)**	Iterated logarithm	Disjoint-set (Union-Find)
O(log n)	Logarithmic	Binary search
O(n)	Linear	Iterating an array
O(n log n)	Linearithmic	Merge Sort
$O(n^2)$	Quadratic	Bubble Sort
$O(2^n)$	Exponential	Subset generation
O(n!)	Factorial	Traveling Salesman (brute-force)

12. Related Asymptotic Notations

Notation	Meaning	Definition
Big O (O)	Upper bound	$f(n) \le C \cdot g(n)$
Big Omega (Ω)	Lower bound	$f(n) \ge C \cdot g(n)$
Big Theta (Θ)	Tight bound	$C_1 \cdot g(n) \le f(n) \le C_2 \cdot g(n)$
Little o (o)	Strictly smaller	$f(n) < C \cdot g(n)$
Little omega (ω)	Strictly larger	$f(n) > C \cdot g(n)$

13. Time and Space Complexity

- **Time Complexity:** How runtime grows with input size (e.g., O(n)).
- **Space Complexity:** How memory usage grows (e.g., O(1) for in-place sorting).

14. Best, Average, Worst, Expected Complexity

Complexity	Best Case	Average Case	Worst Case	Expected Case
O(1)	O(1)	O(1)	O(1)	O(1)
O(log n)	O(1)	O(log n)	O(log n)	O(log n)
O(n)	O(n)	O(n)	O(n)	O(n)
O(n^2)	-	O(n^2)	O(n^2)	O(n^2)

15. How Does Big O Notation Make a Runtime Analysis of an Algorithm?

1. **Identify operations** (loops, recursion).
2. **Count steps** relative to input size (n).
3. **Drop constants & lower-order terms**.
4. **Express in Big O** (e.g., O(n^2)).

Example:

```
for i in range(n):          # O(n)
    for j in range(n):      # O(n) → Total: O(n²)
        print(i, j)
```

16. Real-World Applications of Big O Notation

- **Database indexing** (B-trees for O(log n) searches).
- **Sorting algorithms** (QuickSort vs. Bubble Sort).
- **Network routing** (Dijkstra's algorithm).
- **Machine learning** (optimizing training time).

17. Conclusion

Big O notation is **essential** for:

✓ **Comparing algorithm efficiency**

✓ **Optimizing performance**

✓ **Designing scalable systems**

Mastering it helps in **software engineering, data science, AI, and competitive programming**.

18. FAQs

1. What is Big O notation? Give examples.

- **O(1):** Array access.
- **O(n):** Linear search.
- **$O(n^2)$:** Bubble Sort.

2. Why is Big O notation used?

To analyze **algorithm scalability** and efficiency.

3. What are time complexity and Big O notation?

- **Time complexity:** Measures runtime growth.

- **Big O:** Describes worst-case upper bound.

4. What is another name for Big O notation?

Asymptotic notation.

5. What are the rules of Big O notation?

- Keep **dominant term**.
- Ignore **constants & lower-order terms**.
- Use **worst-case analysis**.