DSA & System Design Training

02/24/2026 15:53:02

Gemini said Let's dive straight into the foundation of all algorithmic design and system…

Gemini said

Let’s dive straight into the foundation of all algorithmic design and system architecture: Time and Space Complexity. I will guide you through this with a focus on production reality and interview rigor. I'll ask guiding questions as we progress through our phases.

================================

PHASE 1 — CONCEPT FOUNDATION

================================

What is it? Time and Space Complexity, evaluated using Big O notation, is the mathematical framework we use to describe how an algorithm's runtime or memory consumption grows as the input size (N) approaches infinity. We care about the worst-case scenario (Big O) to guarantee system stability under maximum load. ⏱️💾

Internal Working (Step-by-Step)

1. **Drop constants:** $O\left(2N\right)$ becomes $O\left(N\right)$. At scale, hardware differences mask constant factors; the growth curve is what crashes a system.
2. **Drop non-dominant terms:** $O\left(N^2+N\right)$ becomes $O\left(N^2\right)$. For an input of $1,000,000$, the N^2 term dictates the limits of the universe; the N is negligible.
3. **Space complexity isn't just variables:** It includes auxiliary space (data structures created) AND the call stack (often forgotten in recursion).

Memory Model Differences (Python vs. Java)

- **Python:** Everything is a heap-allocated object. A simple `list` is an array of pointers to objects spread across memory. This causes poor CPU cache locality and significant memory overhead (an integer can be 28+ bytes).
- **Java:** Primitives (`int` , `char`) are stored directly in the array on the heap (or on the stack if local). They are contiguous in memory, leading to excellent CPU cache locality. Objects behave more like Python's references.

Common Implementation Pitfalls

- **Python:** Using `list.pop(0)`. In Python, a list is an array, not a linked list. Removing the first element requires shifting all subsequent elements, making a seemingly simple operation $O\left(N\right)$.
- **Java:** String concatenation in a loop using `+`. Strings are immutable. `s = s + "a"` creates a completely new string object and copies the old characters, resulting in $O\left(N^2\right)$ time. You must use `StringBuilder` for $O\left(N\right)$.

Dry Run Example

```python
# title Crisis Response Event-Driven Architecture
def find_pairs(arr):
    n = len(arr)
    # Outer loop runs N times
    for i in range(n):
        # Inner loop runs N times for EVERY outer loop iteration
        for j in range(n):
            print(arr[i], arr[j])
```

Time Complexity: $O\left(N \times N\right) = O\left(N^2\right)$. *Space Complexity:* $O\left(1\right)$ auxiliary space, as we only allocate pointers `i` and `j`.

================================

PHASE 2 — REAL-WORLD SYSTEM MAPPING

================================

Where is it used? Complexity analysis is the exact math used for Capacity Planning, SLA (Service Level Agreement) guarantees, and Database Indexing.

Example: Database Lookups & Caching

```
Client → API Gateway → Redis Cache (O(1)) → Database B-Tree Index (O(log N)) → Disk
```

Why this matters at scale:

- **The Problem:** If you have 1 billion users, an $O\left(N\right)$ table scan takes billions of operations. If each takes 1 microsecond, that's ~16 minutes per query. A database will queue requests and eventually crash under timeout limits.
- **The Solution:** By maintaining a B-Tree index, complexity drops to $O\left(\log N\right)$. $\log_2\left(1,000,000,000\right) \approx 30$ operations. The query drops from 16 minutes to microseconds.
- **Tradeoffs:** The cache gives $O\left(1\right)$ time but requires expensive RAM ($O\left(N\right)$ space scaling). The DB index gives $O\left(\log N\right)$ time and costs Disk Space ($O\left(N\right)$ space) plus $O\left(\log N\right)$ write-time overhead to maintain the tree on inserts.

================================

PHASE 3 — INTERVIEW PATTERN RECOGNITION

================================

Signals & Traps In a FAANG interview, the size of the input constraints explicitly tells you the expected Time Complexity. This is your cheat code:

- $N \leq 10^9$: Expected $O\left(1\right)$ or $O\left(\log N\right)$ (Math or Binary Search)
- $N \leq 10^5$: Expected $O\left(N\right)$ or $O\left(N \log N\right)$ (Linear scan, Hash Map, Sorting, Sliding Window)
- $N \leq 10^3$: Expected $O\left(N^2\right)$ (Nested loops, 2D Dynamic Programming)
- $N \leq 20$: Expected $O\left(2^N\right)$ or O\left(N\right!\right) (Backtracking, Recursion)

Brute Force First Always state the naive approach out loud. "The brute force is $O\left(N^2\right)$ by comparing every pair. But since our input constraint is 10^5 , we need to optimize to $O\left(N \log N\right)$ or $O\left(N\right)$, which points me toward sorting or using a Hash Map."

================================

PHASE 4 — PRACTICE PROBLEMS

================================

Here are your initial evaluation problems. We will not write algorithms yet; your task is purely complexity analysis and justification.

1. **Easy:** What are the time and space complexities of a function that takes an array of size N, creates a new array, and copies only the even numbers into it?
2. **Medium:** Consider a loop: `for i from 1 to N`, and an inner loop: `for j from 1 to i`. What is the exact Big O time complexity and why?
3. **Medium+:** You write a recursive function to calculate the N-th Fibonacci number using standard recursion: `F(n) = F(n-1) + F(n-2)`. What is the Time AND Space complexity?
4. **Hard:** A dynamic array (like Python's `list` or Java's `ArrayList`) doubles its capacity when it gets full. What is the *amortized* time complexity of appending an element, and why?
5. **FAANG Style:** You are designing a system that processes a stream of 10^6 stock market ticks per second. How much time (in milliseconds) is your absolute theoretical maximum time budget per operation to maintain real-time throughput on a single thread?
6. **System Design Flavor:** You are asked to cache $100,000,000$ short URLs. A short URL is an 8-character string, mapping to a 100-character long URL string. Roughly estimate the RAM in Gigabytes required to store this in an $O(1)$ lookup hash map in memory, ignoring language-specific object overhead.

Review these problems. Which one would you like to tackle first? Provide your reasoning and complexity analysis for that specific problem, and we will review it together.