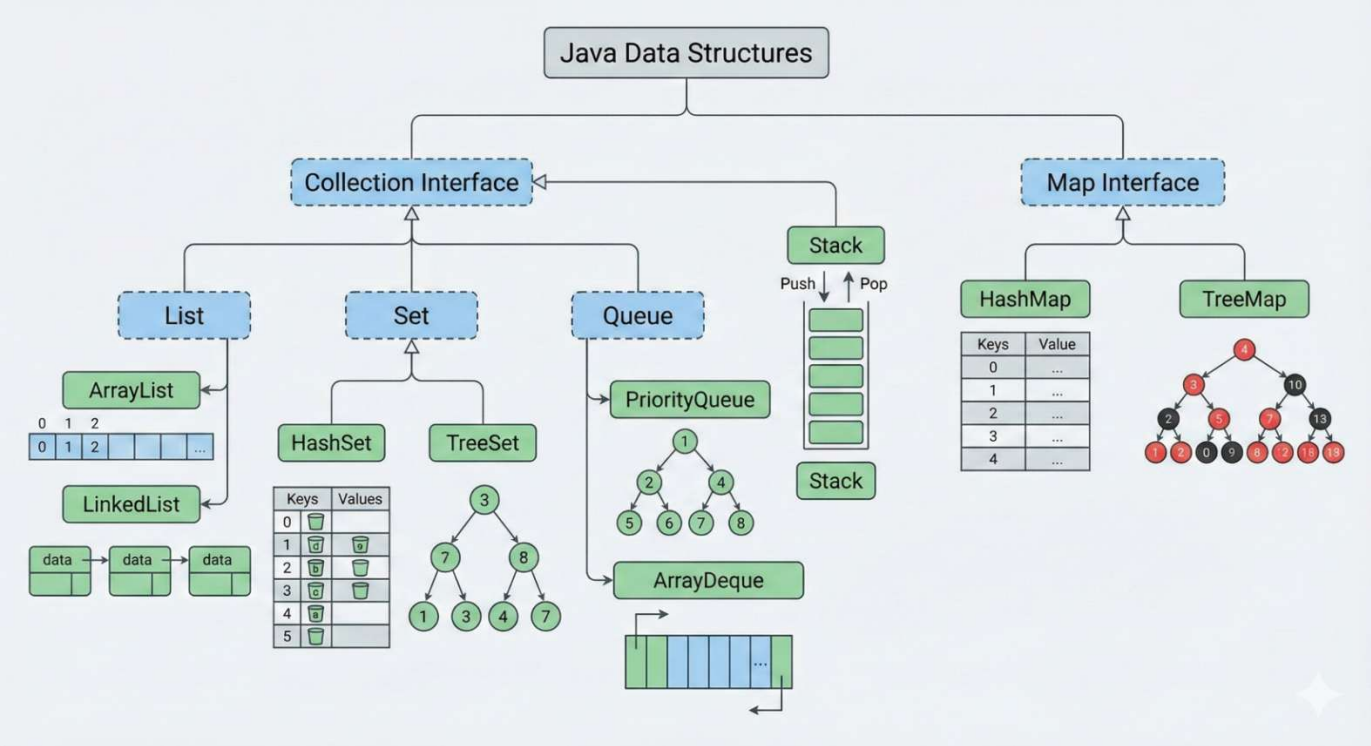
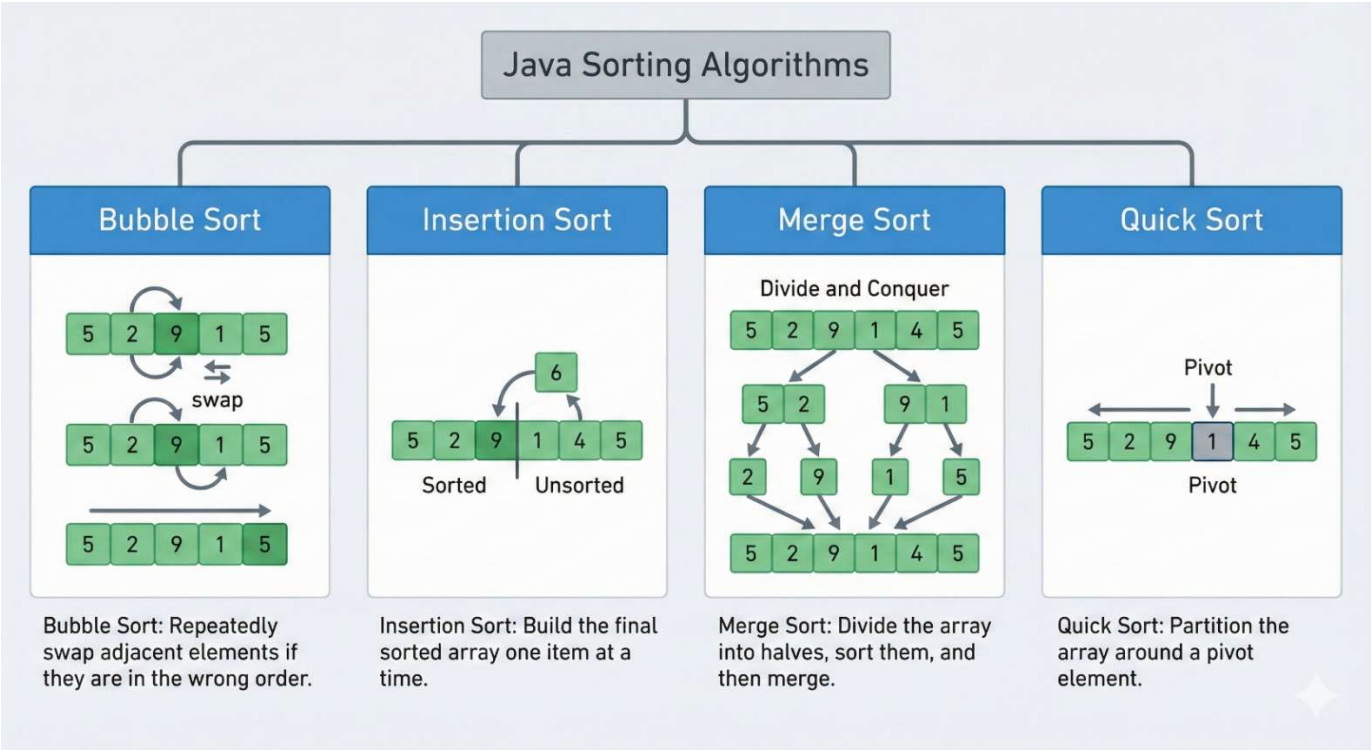
Algorithm Arena: Java 21 Performance Matrix

Architected by Steven Lopez





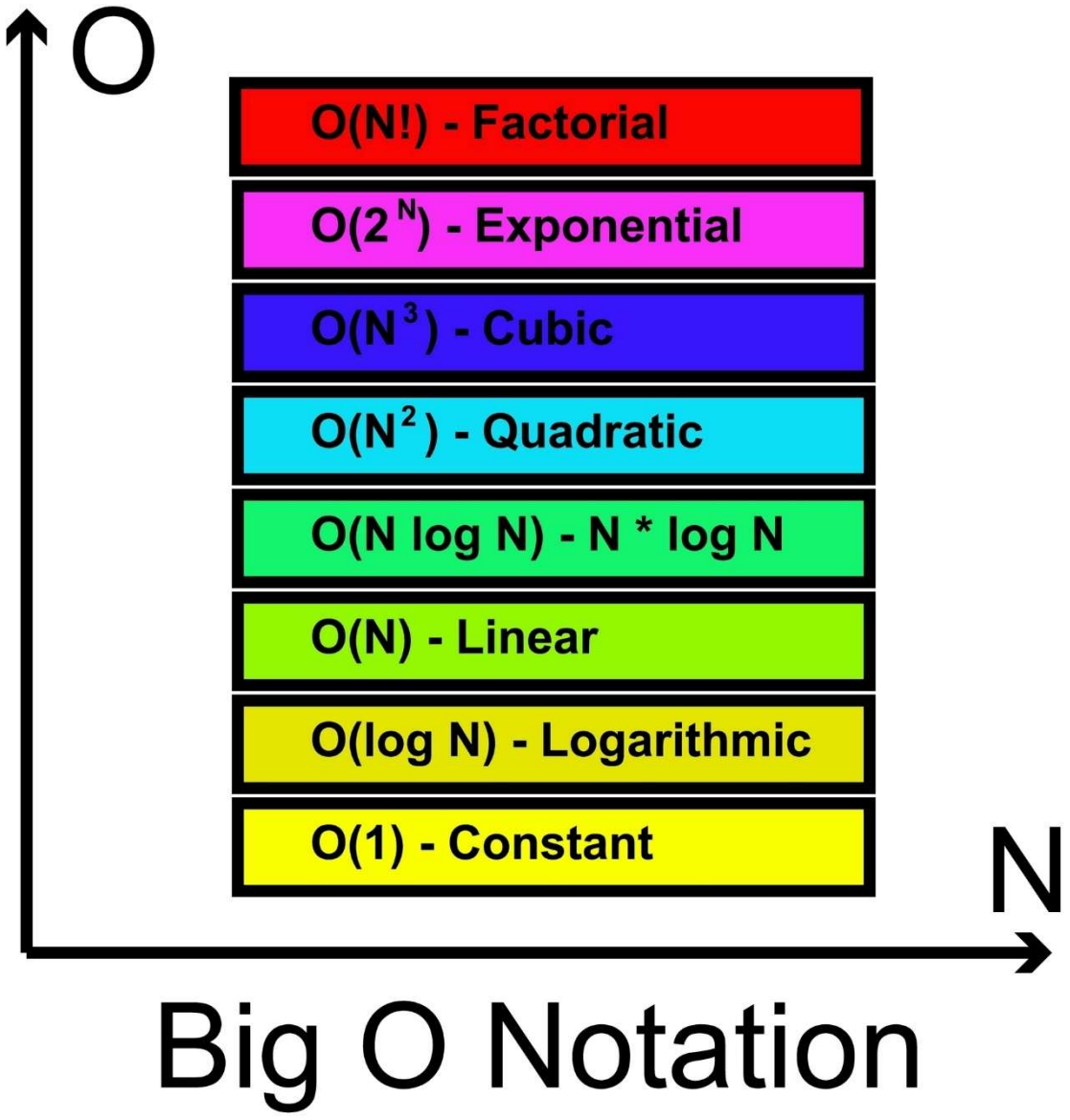
https://github.com/IntuitiveDesigns/AlgorithmArena

Introduction

Big O Notation measures **Scalability**, not speed.

It answers the specific question: *"As the input data (N) grows, how does the number of operations grow?"*

It allows us to predict if code will crash when moving from a test file (10 lines) to production data (10 million lines).



The 4 Tiers You Will See in Interviews

1. **O(1) - Constant Time (Excellent)**
    - **The Rule:** It takes the same time regardless of input size.
    - **Example:** Accessing an array index arr[5], checking a HashSet.

- **Analogy:** Knocking on a specific door number. It takes the same effort whether the hotel has 10 rooms or 10,000 rooms.

2. **O(log N) - Logarithmic Time (Great)**
    - **The Rule:** The input gets cut in half repeatedly.
    - **Example:** Binary Search, looking up a key in a Database Index (B-Tree).
    - **Analogy:** Finding a word in a dictionary. You flip to the middle, ignore the wrong half, and repeat. You find the word in huge books very quickly.
3. **O(N) - Linear Time (Okay / Standard)**
    - **The Rule:** If you double the input, the time doubles.
    - **Example:** Reading a file line-by-line, looping through a List to find a value.
    - **Analogy:** Reading a book. Reading 200 pages takes twice as long as reading 100 pages.
4. **O(N^2) - Quadratic Time (The Danger Zone)**
    - **The Rule:** If you double the input, the time quadruples (2^2 = 4).
    - **Example:** Nested loops (Bubble Sort). Comparing every user to every other user.
    - **Analogy:** A handshake line. If 100 people enter a room and have to shake hands with everyone else, the number of handshakes explodes.

Part 1: Data Structures Big O Matrix

This table summarizes the time complexity for standard operations and the space complexity required to store N elements.

Data Structure	Access (Index)	Search (Value)	Insertion	Deletion	Space (Worst)	Notes
Static Array (int[])	**O**(1)	**O**(N)	N/A (Fixed)	N/A (Fixed)	**O**(N)	Super fast access, but fixed size.
Dynamic Array (ArrayList)	**O**(1)	**O**(N)	**O**(N)*	**O**(N)*	**O**(N)	*Insertion/Deletion at the end is **O**(1) (amortized). Middle is **O**(N) due to shifting.
LinkedList	**O**(N)	**O**(N)	**O**(1)**	**O**(1)**	**O**(N)	**Insertion/Deletion at start/end is **O**(1) if you have the pointer. Finding the middle is still **O**(N).
Stack/Queue (ArrayDeque)	N/A	**O**(N)	**O**(1) (push/offer)	**O**(1) (pop/poll)	**O**(N)	Highly optimized for end-based operations.
HashMap / HashSet	N/A	**O**(1) avg	**O**(1) avg	**O**(1) avg	**O**(N)	Worst case is **O**(log N) or **O**(N) depending on Java version and collision handling.
TreeMap / TreeSet (BST)	N/A	**O**(log N)	**O**(log N)	**O**(log N)	**O**(N)	Keeps data sorted. Uses Red-Black Tree internally.
PriorityQueue (Heap)	**O**(1) (peek)	**O**(N)	**O**(log N)	**O**(log N)	**O**(N)	**O**(1) access only applies to the top (min/max) element.
Graph (Adj List)	**O**(V+E)	**O**(V+E)	**O(1)**	**O(E)**	**O(V+E)**	Used for BFS/DFS. V = Vertices, E = Edges.
2D Matrix	**O(1)**	**O(R+C)**	**O(1)**	**O(1)**	**O(R+C)**	Standard for grid-based dynamic programming.

Part 2: Sorting Algorithms Big O Matrix

This table summarizes the execution time depending on the input state and the extra memory required during the sort.

Algorithm	Type	Best Time	Average Time	Worst Time	Space (Worst)	Key Interview Characteristic
Bubble Sort	Comparison	**O**(N)*	**O**(N^2)	**O**(N^2)	**O**(1)	***O**(N) only if optimized with a "swapped" flag. Very slow.
Selection Sort	Comparison	**O**(N^2)	**O**(N^2)	**O**(N^2)	**O**(1)	Always performs **O**(N^2) comparisons but only **O**(N) swaps.
Insertion Sort	Comparison	**O**(N)	**O**(N^2)	**O**(N^2)	**O**(1)	Excellent for very small (N < 50) or nearly sorted arrays.
Heap Sort	Comparison	**O**(N log N)	**O**(N log N)	**O**(N log N)	**O**(1)	Great worst-case guarantee with **O**(1) space, but rarely used in practice due to poor cache locality.
Merge Sort	D&C	**O**(N log N)	**O**(N log N)	**O**(N log N)	**O**(N)	Stable, predictable performance. Costs extra memory.
Quick Sort	D&C	**O**(N log N)	**O**(N log N)	**O**(N^2)*	**O**(log N)	*Worst case happens with bad pivots, but it is the fastest in practice generally.
TimSort	Hybrid	**O**(N)	**O**(N log N)	**O**(N log N)	**O**(N)	The standard Java/Python sort. Highly optimized for real-world data.
Tree Sort	BST	**O**(N log N)	**O**(N log N)	**O**(N^2)*	**O**(N)	*Worst case if the tree becomes unbalanced (like a linked list).

Algorithm	Type	Best Time	Average Time	Worst Time	Space (Worst)	Key Interview Characteristic
Counting Sort	Integer	**O**(N+K)	**O**(N+K)	**O**(N+K)	**O**(K)	K is the range of input values. Fast if K is small; terrible if K is huge.
Radix Sort	Digit	**O**(NK)	**O**(NK)	**O**(NK)	**O**(N+K)	K is the number of digits (or bits). Good for strings or fixed-length integers.
Bucket Sort	Distribution	**O**(N+K)	**O**(N+K)	**O**(N^2)	**O**(N)	Great for uniformly distributed data (e.g., floats between 0.0 and 1.0).
Shell Sort	Comparison	**O**(N log N)	**O**(N log^2 N)	**O**(N log^2 N)	**O**(1)	

Legend:

- **D&C:** Divide and Conquer strategy.
- **N:** The number of elements.
- **K:** A secondary factor specific to non-comparison sorts (e.g., range of numbers, number of digits, or number of buckets).