Admissions & Support: support@mockattempt.com
Phone / WhatsApp: +91 95400 07917
Course Code: DSAJ • **Updated:** 05 Nov 2026
https://mockattempt.com/courses/dsa-with-java-training

DSAJ • DATA STRUCTURES, ALGORITHMS & FAANG CODING PREP | **BEGINNER TO COMPETITIVE & PRODUCT-COMPANY DSA (JAVA)**

100% LIVE INDUSTRY EXPERT COHORT + INTERNSHIP LETTER

DSA with Java Training

Crack FAANG & top product-company coding rounds with Java 21 DSA: Big-O Asymptotic Analysis, Two-Pointer Arrays, Singly/Doubly Linked Lists, ArrayDeque Monotonic Stacks, Circular/Deque Windows, HashMap Internals, Binary Search, Merge/Quick Sort, BST/AVL Trees, PriorityQueue Heaps, BFS/DFS & Kruskal/Prim MST — with 244 Topic Mock Tests (10,440 MCQs) and Spoken Voice + MockAttempt AI Interviews at ₹3,499 (Incl. 18% GST).

Duration: 15–17 Weeks • Live + Hands-On Projects (222 Hrs) • **Modules:** 16 • **Live Topics:** 42 • **Subtopics:** 336+ • **Labs & Tasks:** 84 • **Mocks / AI Vivas:** 244 / 244 • **Projects:** 16 Module + 1 Live Client Prod

All-Inclusive Direct Fee (Incl. 18% GST • Zero Hidden Charges): Total Payable: ₹3,499 (Net Tuition ₹2,965 + Included 18% GST ₹534) • **Split Option:** 2 × ₹1,750 Split • **Effective ROI:** ~₹16/Guided Hr (~₹83/Live Topic)

Faculty: 100% Active Working IT Professionals • **Live Batch Tracks (7:00 AM – 8:00 PM IST Flexible Slots):** Weekday Live Online Batch (Mon – Fri • 1 Hr/Day • 7:00 AM – 8:00 PM) (Monday to Friday (1 Hour Live Session / Day) • 07:00 AM – 08:00 PM IST (Students Can Choose Any 1-Hour Slot Based on Trainer Availability) • 19–21 Weeks Fast-Track) | Weekend Live Online Batch (Sat – Sun • 2 Hrs/Day • 7:00 AM – 8:00 PM) (Saturday & Sunday (2 Hours Live Session / Day) • 07:00 AM – 08:00 PM IST (Students Can Choose Any 2-Hour Slot Based on Trainer Availability) • 21–23 Weeks Weekend Track)

✓ **Trainer LeetCode & HackerRank Mentorship (Java 21 LTS (Data Structures & Algorithms)): Live trainer support to crack LeetCode & HackerRank problems, optimize complexity, and handle edge cases.**

Core Technologies & Engineering Stack: Java 21 LTS (Records & Generics) • Big-O, Amortized & Space Complexity • Arrays, Two Pointers & Linked Lists • ArrayDeque Stacks, Circular Queues & Deques • HashMap (TreeBin), Binary Search & Sorting • Binary Trees, BST, AVL & Binary Heaps • Graph BFS/DFS, Union-Find & Kruskal/Prim MST

1. PROGRAM MASTERY OUTCOMES & ELIGIBILITY

End-to-End Engineering Outcomes:

1. Derive tight Big-O, Big-Omega, Big-Theta, and amortized bounds while understanding JVM CPU cache locality vs pointer-chasing heap overhead
2. Solve product-company interview patterns across Two Pointers, Fast/Slow Linked Lists, Monotonic Stacks, Sliding Window Deques, and Binary Search on Answer
3. Implement self-balancing AVL rotations, O(N) Floyd Heapify / Top-K PriorityQueues, iterative/recursive Graph BFS/DFS, and Union-Find / Prim Minimum Spanning Trees
4. Clear 244 Timed Mock Exams (10,000 Java DSA MCQs) and 200 Spoken Voice + MockAttempt AI Coding & Algorithmic Viva Rounds
5. Asymptotic Time & Auxiliary Space analysis (`Big-O`, amortized complexity, and recursion stack trade-offs).
6. Crack LeetCode and HackerRank problems in Java 21 LTS (Data Structures & Algorithms) with live trainer mentorship on algorithmic/query patterns, edge cases, and optimal complexity.
7. Core linear & hierarchical structures: Arrays, Two Pointers, Sliding Window, Linked Lists, Stacks, Monotonic Queues, Trees, Tries, and Heaps.
8. Graph & Optimization mastery: BFS/DFS, Topological Sort, Union-Find, Dijkstra, Greedy proofs, and Dynamic Programming.

Eligibility & Prerequisites: Basic familiarity with Java syntax (primitive types, loops, methods, classes) — generics, collections, and recursion are built up systematically • Any laptop/PC with JDK 21 installed and IntelliJ IDEA Community / VS Code • No prior DSA with Java industry experience required — Module 1 starts from environment setup, core mental models, and plain-English fundamentals. • Ideal for B.Tech / BCA / MCA / B.Sc IT students, non-CS beginners, career switchers, and working developers upgrading to 2026 production standards. • Commitment to attend live Weekday or Weekend classes, complete daily hands-on lab checklists, and practice topic-wise Mock Tests & AI Interviews.

2. DETAILED ZERO-TO-EXPERT MODULE, TOPIC & SUBTOPIC SYLLABUS (16 MODULES • 42 LIVE TOPICS • 336+ SUBTOPICS)

Progressive Zero-to-Expert Mastery Stages: Modules 1–4: Stage 1: Day-1 Foundations & Core Mental Models (8 Topics / 64 Subtopics) • Modules 5–8: Stage 2: Applied Engineering, Data & API Workflows (8 Topics / 64 Subtopics) • Modules 9–12: Stage 3: Internals, Concurrency, Security & Performance (12 Topics / 96 Subtopics) • Modules 13–16: Stage 4: Production Architecture, Testing, CI/CD & Expert Mastery (14 Topics / 112 Subtopics)

MODULE 1 OF 16 • STAGE 1 • 13 LIVE + LAB HOURS • 2 TOPICS (16 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

Module 1: Algorithmic Thinking, JVM Memory Layout & Asymptotic Complexity (`O`, `Ω`, `Θ`)

Understand Abstract Data Types (ADTs) vs concrete Java data structures, primitive contiguous arrays vs boxed heap objects, and rigorous Big-O, Big-Omega, Big-Theta, and Amortized complexity analysis.

Objectives: ✓ DSA Foundations in Java — ADTs vs Implementations, Primitive Cache Locality & Judge Constraints — Why choosing the right data structure turns a 10-second Time-Limit-Exceeded (`TLE`) brute-force loop into a 4ms optimal pass, and why `int[]` beats `ArrayList<Integer>` by 4–5x in memory and cache speed. • ✓ Asymptotic Notation — Big-O (`O`), Big-Omega (`Ω`), Big-Theta (`Θ`) & Amortized Analysis — Formal growth-rate bounds (`O(1) < O(log N) < O(N) < O(N log N) < O(N^2) < O(2^N)`), dropping constants and lower-order terms, harmonic series `O(N log N)` loops, and `ArrayList` `O(1)` amortized doubling. • ✓ Build, verify, and ship Module Capstone: Constraint-to-Complexity Feasibility Estimator & Amortized Dynamic Array Profiler.

1.1 DSA Foundations in Java — ADTs vs Implementations, Primitive Cache Locality & Judge Constraints [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: Abstract Data Type vs implementation & CPU cache locality (int[] vs Integer) • Core Syntax, CLI & Production APIs: ADT Interfaces (List, Queue, Deque, Map) + Primitive int[] / long[] + Kadane / Prefix Patter... • Deep-Dive Mechanics: 10^8 ops/sec TLE judge rule & StringBuilder O(N) vs String + O(N^2) • Advanced Patterns: Time vs auxiliary space trade-off & 32-bit int overflow -> long cast • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Kadane's O(N) Maximum Subarray Sum (vs O(N^2) Brute Force) • Idiomatic DSAJ Implementation: immutable Records, Generics, Streams & Virtual Threads • JDK 21 JVM Stack/Heap & JIT Execution & Memory Lifecycle for DSA Foundations in Java

Purpose: To understand and apply DSA Foundations in Java — ADTs vs Implementations, Primitive Cache Locality & Judge Constraints in real DSA with Java systems — specifically: Why...

What Topic Is Next → Next Topic 1.2: Asymptotic Notation — Big-O (`O`), Big-Omega (`Ω`), Big-Theta (`Θ`) & Amortized Analysis

1.2 Asymptotic Notation — Big-O (`O`), Big-Omega (`Ω`), Big-Theta (`Θ`) & Amortized Analysis [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: Big-O asymptotic upper bound & Big-Omega lower & Big-Theta tight • Core Syntax, CLI & Production APIs: Big-O Upper Bound, Big-Omega Lower Bound, Big-Theta Tight Bound, Amortized Accounting Method • Deep-Dive Mechanics: Amortized O(1) ArrayList doubling & Harmonic series O(N log N) trap • Advanced Patterns: Auxiliary vs input space complexity & Call-stack recursion space O(depth) • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Harmonic Sieve Loop Counter & Amortized Doubling Verifier • Idiomatic DSAJ Implementation: immutable Records, Generics, Streams & Virtual Threads • JDK 21 JVM Stack/Heap & JIT Execution & Memory Lifecycle for Asymptotic Notation

Purpose: To understand and apply Asymptotic Notation — Big-O (`O`), Big-Omega (`Ω`), Big-Theta (`Θ`) & Amortized Analysis in real DSA with Java systems — specifically: Formal grow...

What Topic Is Next → Module 1 Capstone (Constraint-to-Complexity Feasibility Estimator & Amortized Dynamic Array Profiler) → Next Topic 2.1: Arrays in Java — `base + i * stride` Random Access, 3-Step Rotation & Two-Pointer Technique

Module 1 Hands-On Capstone Project (100 Marks): Constraint-to-Complexity Feasibility Estimator & Amortized Dynamic Array Profiler — Build a Java 21 benchmark & constraint analyzer that maps input size `N` (`10`, `20`, `10^3`, `10^5`, `10^7`) to maximum non-TLE complexity classes under a `10^8` operations/sec judge budget...

MODULE 2 OF 16 • STAGE 1 • 13 LIVE + LAB HOURS • 2 TOPICS (16 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

Module 2: Contiguous Arrays (Two Pointers, Prefix Sums) & Singly Linked Lists

Exploit `O(1)` random-access contiguous arrays with in-place reversal, Prefix Sums, and Two Pointers, and master pointer manipulation in Singly Linked Lists (Dummy Head, Fast/Slow Tortoise-Hare, In-Place Reversal).

Objectives: ✓ Arrays in Java — `base + i * stride` Random Access, 3-Step Rotation & Two-Pointer Technique — Understand why array indexing `arr[i]` is strictly `O(1)` while middle insertion/deletion costs `O(N)` `System.arraycopy` shifts, and solve classic in-place array problems with Two Pointers. • ✓ Singly Linked List — Pointer Rewiring, Sentinel Dummy Head, Floyd's Cycle & In-Place Reversal — Master dynamic node chains (`val`, `next`): `O(1)` head prepend, why random access is `O(N)`, iterative `prev/curr/next` pointer reversal, and Floyd's Tortoise & Hare cycle detection. • ✓ Build, verify, and ship Module Capstone: In-Place Array Rotation Engine & `O(1)` Space Singly Linked List Palindrome Checker.

2.1 Arrays in Java — `base + i * stride` Random Access, 3-Step Rotation & Two-Pointer Technique [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: O(1) base + index * size addressing & O(N) middle insert/delete shift • Core Syntax, CLI & Production APIs: int[] arr + Arrays.copyOf() + System.arraycopy() + Two Pointers (left, right) • Deep-Dive Mechanics: 3-reversal O(1)-space k-rotation & Converging Two Pointers pattern • Advanced Patterns: Prefix sum O(1) range query & HotSpot JIT bounds-check hoisting • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: O(N) Time, O(1) Space Right Array Rotation (3-Reversal Algorithm) • Idiomatic DSAJ Implementation: immutable Records, Generics, Streams & Virtual Threads • JDK 21 JVM Stack/Heap & JIT Execution & Memory Lifecycle for Arrays in Java

Purpose: To understand and apply Arrays in Java — `base + i * stride` Random Access, 3-Step Rotation & Two-Pointer Technique in real DSA with Java systems — specifically: Understa...

What Topic Is Next → Next Topic 2.2: Singly Linked List — Pointer Rewiring, Sentinel Dummy Head, Floyd's Cycle & In-Place Reversal

2.2 Singly Linked List — Pointer Rewiring, Sentinel Dummy Head, Floyd's Cycle & In-Place Reversal [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: O(1) head vs O(N) tail-without-ptr & 3-pointer iterative reversal • Core Syntax, CLI & Production APIs: class Node { int val; Node next; } + Sentinel Dummy Node + Fast/Slow Pointers • Deep-Dive Mechanics: Floyd Tortoise & Hare cycle math & Sentinel dummy head edge-case shield • Advanced Patterns: Fast != null && fast.next != null & Pointer-chasing cache miss cost • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Iterative O(1)-Space Singly Linked List Reversal • Idiomatic DSAJ Implementation: immutable Records, Generics, Streams & Virtual Threads • JDK 21 JVM Stack/Heap & JIT Execution & Memory Lifecycle for Singly Linked List

Purpose: To understand and apply Singly Linked List — Pointer Rewiring, Sentinel Dummy Head, Floyd's Cycle & In-Place Reversal in real DSA with Java systems — specifically: Master...

What Topic Is Next → Module 2 Capstone (In-Place Array Rotation Engine & `O(1)` Space Singly Linked List Palindrome Checker) → Next Topic 3.1: Doubly Linked List — `prev`/`next` Sentinel Splicing, `O(1)` Node Deletion & LRU Cache Design

Module 2 Hands-On Capstone Project (100 Marks): In-Place Array Rotation Engine & `O(1)` Space Singly Linked List Palindrome Checker — Implement an `O(N)` time, `O(1)` auxiliary-space `k`-step right array rotator alongside an `O(N)` time, `O(1)` space Singly Linked List palindrome verifier using Fast/Slow midpoint discovery...

MODULE 3 OF 16 • STAGE 1 • 13 LIVE + LAB HOURS • 2 TOPICS (16 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

Module 3: Doubly Linked Lists (`O(1)` LRU Eviction) & Monotonic Stacks (`ArrayDeque`)

Perform `O(1)` arbitrary node splicing with bidirectional sentinel Doubly Linked Lists (powering LRU Caches) and solve Next Greater Element, Balanced Parentheses, and Histogram problems using `ArrayDeque` Stacks.

Objectives: ✓ Doubly Linked List — `prev`/`next` Sentinel Splicing, `O(1)` Node Deletion & LRU Cache Design — Why storing both `prev` and `next` unlocks `O(1)` deletion given a node reference (impossible in `O(1)` on tail nodes of a Singly Linked List!), making Doubly Linked Lists + `HashMap` the backbone of `O(1)` LRU Caches. • ✓ Stack in Java — LIFO `ArrayDeque` vs Legacy `Stack`, Balanced Brackets & `O(N)` Monotonic Stack — Implement LIFO stacks with `Deque<Integer> st = new ArrayDeque<>()`, understand why `java.util.Stack` (extending synchronized `Vector`) is an anti-pattern, and master `O(N)` Monotonic Decreasing/Increasing Stacks. • ✓ Build, verify, and ship Module Capstone: `O(1)` LRU Cache (HashMap + Sentinel Doubly Linked List) & Monotonic Next-Greater Engine.

3.1 Doubly Linked List — `prev`/`next` Sentinel Splicing, `O(1)` Node Deletion & LRU Cache Design [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: O(1) arbitrary DNode unlink & Dummy head + tail sentinels • Core Syntax, CLI & Production APIs: class DNode { int key, val; DNode prev, next; } + Head/Tail Sentinels + LinkedHashMap • Deep-Dive Mechanics: 4-pointer insert wire order & LRU Cache HashMap + DLL pattern • Advanced Patterns: Bidirectional O(1) tail pop & Memory cost: 2 refs per node • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Sentinel-Guarded `O(1)` LRU Cache Core (`get` / `put`) • Idiomatic DSAJ Implementation: immutable Records, Generics, Streams & Virtual Threads • JDK 21 JVM Stack/Heap & JIT Execution & Memory Lifecycle for Doubly Linked List

Purpose: To understand and apply Doubly Linked List — `prev`/`next` Sentinel Splicing, `O(1)` Node Deletion & LRU Cache Design in real DSA with Java systems — specifically: Why st...

What Topic Is Next → Next Topic 3.2: Stack in Java — LIFO `ArrayDeque` vs Legacy `Stack`, Balanced Brackets & `O(N)` Monotonic Stack

3.2 Stack in Java — LIFO `ArrayDeque` vs Legacy `Stack`, Balanced Brackets & `O(N)` Monotonic Stack [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: LIFO ArrayDeque vs legacy Vector Stack & O(N) Monotonic Stack amortized proof • Core Syntax, CLI & Production APIs: Deque<Integer> stack = new ArrayDeque<>() (push, pop, peek, isEmpty) • Deep-Dive Mechanics: Next Greater Element index pushing & Balanced parentheses matching • Advanced Patterns: Infix-to-postfix / RPN evaluation & Explicit stack simulating recursion • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: `O(N)` Next Greater Element Using `ArrayDeque` Monotonic Stack • Idiomatic DSAJ Implementation: immutable Records, Generics, Streams & Virtual Threads • JDK 21 JVM Stack/Heap & JIT Execution & Memory Lifecycle for Stack in Java

Purpose: To understand and apply Stack in Java — LIFO `ArrayDeque` vs Legacy `Stack`, Balanced Brackets & `O(N)` Monotonic Stack in real DSA with Java systems — specifically: Impl...

What Topic Is Next → Module 3 Capstone (`O(1)` LRU Cache (HashMap + Sentinel Doubly Linked List) & Monotonic Next-Greater Engine) → Next Topic 4.1: Circular Queue (Ring Buffer) — Modulo Wrap-Around `(idx + 1) % k` & Full vs Empty Disambiguation

Module 3 Hands-On Capstone Project (100 Marks): `O(1)` LRU Cache (HashMap + Sentinel Doubly Linked List) & Monotonic Next-Greater Engine — Implement an interview-standard `LRUCache(capacity)` supporting `get(key)` and `put(key, val)` in strict `O(1)` time by pairing a `HashMap<Integer, DNode>` with a sentinel-guarded Doubly Lin...

MODULE 4 OF 16 • STAGE 1 • 13 LIVE + LAB HOURS • 2 TOPICS (16 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

Module 4: Ring Buffers (`CircularQueue`) & Double-Ended Queues (`Deque` Sliding Window Max)

Eliminate wasted front capacity in array queues via modulo wrap-around (`CircularQueue`), and master Double-Ended Queues (`Deque`) to solve `O(N)` Sliding Window Maximum and 0-1 BFS.

Objectives: ✓ Circular Queue (Ring Buffer) — Modulo Wrap-Around `(idx + 1) % k` & Full vs Empty Disambiguation — Why a naive array queue runs out of space after `N` dequeues even when empty, and how a **Circular Queue (Ring Buffer)** reuses freed front slots in `O(1)` time for producer-consumer buffers and networking rings. • ✓ Double-Ended Queue (`Deque`) — `offerFirst`/`offerLast`, 0-1 BFS & `O(N)` Sliding Window Maximum — Push and pop from BOTH ends in `O(1)` time with `ArrayDeque`: unify Stack + Queue interfaces, solve 0-1 shortest paths without Dijkstra's log factor, and compute Sliding Window Maximum in `O(N)`. • ✓ Build, verify, and ship Module Capstone: Lock-Free Style Ring Buffer Queue & `O(N)` Sliding Window Maximum Analytics Engine.

4.1 Circular Queue (Ring Buffer) — Modulo Wrap-Around `(idx + 1) % k` & Full vs Empty Disambiguation [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: Ring Buffer modulo wrap-around & Head + size full/empty disambiguation • Core Syntax, CLI & Production APIs: head = (head + 1) % capacity; tailIdx = (head + size - 1) % capacity; size == capacity • Deep-Dive Mechanics: Zero O(N) left-shift overhead & Preallocated zero-GC int[k] array • Advanced Patterns: Power-of-two & (len - 1) bitmask & Producer-consumer bounded backpressure • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Bounded `O(1)` Circular Ring Buffer Implementation • Idiomatic DSAJ Implementation: immutable Records, Generics, Streams & Virtual Threads • JDK 21 JVM Stack/Heap & JIT Execution & Memory Lifecycle for Circular Queue (Ring Buffer)

Purpose: To understand and apply Circular Queue (Ring Buffer) — Modulo Wrap-Around `(idx + 1) % k` & Full vs Empty Disambiguation in real DSA with Java systems — specifically: Why...

What Topic Is Next → Next Topic 4.2: Double-Ended Queue (`Deque`) — `offerFirst`/`offerLast`, 0-1 BFS & `O(N)` Sliding Window Maximum

4.2 Double-Ended Queue (`Deque`) — `offerFirst`/`offerLast`, 0-1 BFS & `O(N)` Sliding Window Maximum

[50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: O(1) both-ends offer/poll/peek & Monotonic Decreasing Deque window • Core Syntax, CLI & Production APIs: Deque<Integer> dq = new ArrayDeque<>() (offerFirst, offerLast, pollFirst, pollLast, peekFirs... • Deep-Dive Mechanics: 0-1 BFS edge-weight 0 front push & ArrayDeque forbids null rule • Advanced Patterns: O(N) vs O(N log K) heap comparison & Window index expiration i - k • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: `O(N)` Sliding Window Maximum via Monotonic Decreasing `ArrayDeque` • Idiomatic DSAJ Implementation: immutable Records, Generics, Streams & Virtual Threads • JDK 21 JVM Stack/Heap & JIT Execution & Memory Lifecycle for Double-Ended Queue (`Deque`)

Purpose: To understand and apply Double-Ended Queue (`Deque`) — `offerFirst`/`offerLast`, 0-1 BFS & `O(N)` Sliding Window Maximum in real DSA with Java systems — specifically: Pus...

What Topic Is Next → Module 4 Capstone (Lock-Free Style Ring Buffer Queue & `O(N)` Sliding Window Maximum Analytics Engine) → Next Topic 5.1: Hash Table & Java `HashMap` — `(h ^ (h >>> 16))`, Load Factor `0.75` & Red-Black `TreeNode`

Module 4 Hands-On Capstone Project (100 Marks): Lock-Free Style Ring Buffer Queue & `O(N)` Sliding Window Maximum Analytics Engine — Build a bounded `CircularQueue` supporting `enQueue`, `deQueue`, `Front`, `Rear`, `isEmpty`, and `isFull` in strict `O(1)` time using a `size` counter, alongside an `O(N)` Monotonic `Deque`...

MODULE 5 OF 16 • STAGE 2 • 13 LIVE + LAB HOURS • 2 TOPICS (16 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

Module 5: Java `HashMap` Internals (TreeBins, Load Factor) & Binary Search on Answer

Master Java 8+ `HashMap` internals (XOR hash spreading, load factor `0.75`, collision chaining -> Red-Black `TreeNode` at threshold 8) and write bug-free lower/upper-bound Binary Search & Binary Search on Answer.

Objectives: ✓ Hash Table & Java `HashMap` — `(h ^ (h >>> 16))`, Load Factor `0.75` & Red-Black `TreeNode` — How `HashMap` achieves `O(1)` average lookup and `O(log N)` worst-case bucket lookup (since Java 8's `TREEIFY_THRESHOLD = 8` when `capacity >= 64`), Open Addressing vs Separate Chaining, and prefix-hash patterns. • ✓ Binary Search — Overflow-Safe Midpoint, Lower/Upper Bound & Binary Search on Answer — Go far beyond simple array lookup: master invariant-driven `lowerBound` (first index `>= target`), rotated sorted arrays, and **Binary Search on Answer** over any monotonic boolean predicate `canShip(capacity)`. • ✓ Build, verify, and ship Module Capstone: Prefix-Sum Subarray Target Counter (`HashMap`) & Monotonic Binary Search Capacity Solver.

5.1 Hash Table & Java `HashMap` — `(h ^ (h >>> 16))`, Load Factor `0.75` & Red-Black `TreeNode`

[50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: (h ^ (h >>> 16)) upper-bit spreading & Load factor 0.75 resize doubling • Core Syntax, CLI & Production APIs: Map<K, V> map = new HashMap<>() (getOrDefault, merge, computeIfAbsent) + Java record keys • Deep-Dive Mechanics: TREEIFY_THRESHOLD = 8 Red-Black tree & Equals() & hashCode() immutable key rule • Advanced Patterns: Separate Chaining vs Linear Probing & Prefix Sum + HashMap O(N) pattern • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Subarray Sum Equals K in `O(N)` (Prefix Sum + `HashMap`) • Idiomatic DSAJ Implementation: immutable Records, Generics, Streams & Virtual Threads • JDK 21 JVM Stack/Heap & JIT Execution & Memory Lifecycle for Hash Table

Purpose: To understand and apply Hash Table & Java `HashMap` — `(h ^ (h >>> 16))`, Load Factor `0.75` & Red-Black `TreeNode` in real DSA with Java systems — specifically: How `Has...

What Topic Is Next → Next Topic 5.2: Binary Search — Overflow-Safe Midpoint, Lower/Upper Bound & Binary Search on Answer

5.2 Binary Search — Overflow-Safe Midpoint, Lower/Upper Bound & Binary Search on Answer

[50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: (low + high) >>> 1 overflow-free mid & LowerBound first >= target invariant • Core Syntax, CLI & Production APIs: int mid = (low + high) >>> 1; // or low + (high - low) / 2 • Deep-Dive Mechanics: UpperBound first > target invariant & Binary Search on Answer predicate • Advanced Patterns: Rotated sorted array half-order check & Infinite-loop floor vs ceil mid rule • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Binary Search on Answer — Minimum Daily Ship Capacity in `D` Days • Idiomatic DSAJ Implementation: immutable Records, Generics, Streams & Virtual Threads • JDK 21 JVM Stack/Heap & JIT Execution & Memory Lifecycle for Binary Search

Purpose: To understand and apply Binary Search — Overflow-Safe Midpoint, Lower/Upper Bound & Binary Search on Answer in real DSA with Java systems — specifically: Go far beyond si...

What Topic Is Next → Module 5 Capstone (Prefix-Sum Subarray Target Counter (`HashMap`) & Monotonic Binary Search Capacity Solver) → Next Topic 6.1: Merge Sort — Guaranteed `O(N log N)` Divide & Conquer, Stability (`<=`) & Inversion Counting

Module 5 Hands-On Capstone Project (100 Marks): Prefix-Sum Subarray Target Counter (`HashMap`) & Monotonic Binary Search Capacity Solver — Solve Subarray Sum Equals `K` (including negative integers where two-pointer sliding window fails!) in `O(N)` using a prefix-sum frequency `HashMap`, paired with an `O(N log(sum))` Binary Se...

MODULE 6 OF 16 • STAGE 2 • 13 LIVE + LAB HOURS • 2 TOPICS (16 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

Module 6: Divide-and-Conquer Sorting — Stable `Merge Sort` & Partitioned `Quick Sort`

Master guaranteed `O(N log N)` stable Merge Sort (plus Inversion Counting) and cache-friendly in-place Quick Sort (Lomuto, Hoare, 3-Way Dutch National Flag) alongside why JDK uses Dual-Pivot Quicksort vs TimSort.

Objectives: ✓ Merge Sort — Guaranteed `O(N log N)` Divide & Conquer, Stability (`<=`) & Inversion Counting — Recursively split arrays to base size 1 and merge sorted halves in linear time: understand `T(N) = 2T(N/2) + O(N) => Θ(N log N)`, single-buffer allocation, linked-list `O(1)`-space merge sort, and inversion counting. • ✓ Quick Sort — Lomuto vs Hoare vs 3-Way Dutch Flag Partitioning & Randomized Pivot Safety — Understand why in-place Quick Sort beats Merge Sort on primitives via superior L1 cache locality, why naive `a[high]` pivots degrade to `O(N^2)` on sorted arrays, and how 3-Way partitioning crushes duplicates in `O(N)`. • ✓ Build, verify, and ship Module Capstone: `O(N log N)` Inversion Counter (Merge Sort) & 3-Way Quickselect `K`-th Largest Engine.

6.1 Merge Sort — Guaranteed `O(N log N)` Divide & Conquer, Stability (`<=`) & Inversion Counting [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: Guaranteed Theta(N log N) worst case & Left <= right stability invariant • Core Syntax, CLI & Production APIs: T(N) = 2T(N/2) + O(N) -> O(N log N) time, O(N) auxiliary array, stable (left <= right) • Deep-Dive Mechanics: Single preallocated aux[] buffer & O(N log N) inversion counting • Advanced Patterns: External merge sort on disk runs & TimSort natural run exploitation • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Merge Sort `O(N log N)` Inversion Counter • Idiomatic DSAJ Implementation: immutable Records, Generics, Streams & Virtual Threads • JDK 21 JVM Stack/Heap & JIT Execution & Memory Lifecycle for Merge Sort

Purpose: To understand and apply Merge Sort — Guaranteed `O(N log N)` Divide & Conquer, Stability (`<=`) & Inversion Counting in real DSA with Java systems — specifically: Recursi...

What Topic Is Next → Next Topic 6.2: Quick Sort — Lomuto vs Hoare vs 3-Way Dutch Flag Partitioning & Randomized Pivot Safety

6.2 Quick Sort — Lomuto vs Hoare vs 3-Way Dutch Flag Partitioning & Randomized Pivot Safety [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: In-place cache-friendly partitioning & O(N^2) fixed-pivot sorted/equal trap • Core Syntax, CLI & Production APIs: Randomized Pivot + 3-Way Partition (lt, i, gt) + Dual-Pivot Quicksort (Arrays.sort(int[])) • Deep-Dive Mechanics: Randomized / median-of-three pivot & Dijkstra 3-Way Dutch National Flag • Advanced Patterns: Quickselect O(N) expected K-th order & Smaller-partition-first O(log N) stack • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: 3-Way Dutch National Flag Quick Sort (Duplicate-Resilient) • Idiomatic DSAJ Implementation: immutable Records, Generics, Streams & Virtual Threads • JDK 21 JVM Stack/Heap & JIT Execution & Memory Lifecycle for Quick Sort

Purpose: To understand and apply Quick Sort — Lomuto vs Hoare vs 3-Way Dutch Flag Partitioning & Randomized Pivot Safety in real DSA with Java systems — specifically: Understand w...

What Topic Is Next → Module 6 Capstone (`O(N log N)` Inversion Counter (Merge Sort) & 3-Way Quickselect `K`-th Largest Engine) → Next Topic 7.1: Binary Tree — DFS Traversals, Level-Order BFS, `O(N)` Postorder Diameter & Lowest Common Ancestor

Module 6 Hands-On Capstone Project (100 Marks): `O(N log N)` Inversion Counter (Merge Sort) & 3-Way Quickselect `K`-th Largest Engine — Implement a stable Merge Sort pass counting exact out-of-order pairs `(i < j && a[i] > a[j])` in `O(N log N)`, paired with a randomized 3-Way Dutch National Flag Quickselect finding the `k`-...

MODULE 7 OF 16 • STAGE 2 • 13 LIVE + LAB HOURS • 2 TOPICS (16 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

Module 7: Hierarchical Trees — Binary Tree Traversals, Diameter/LCA & Binary Search Trees (BST)

Master recursive and iterative DFS/BFS Binary Tree algorithms (Preorder, Inorder, Postorder, Level-Order, Diameter, Lowest Common Ancestor) and Binary Search Tree invariants (`(min, max)` validation, Hibbard deletion, `k`-th smallest).

Objectives: ✓ Binary Tree — DFS Traversals, Level-Order BFS, `O(N)` Postorder Diameter & Lowest Common Ancestor — Master Binary Tree properties (Full, Complete, Perfect, Balanced; `2^h` leaves), Pre/In/Postorder + BFS Level-Order (`q.size()` snapshot per level), and single-pass `O(N)` postorder aggregation. • ✓ Binary Search Tree (BST) — Global Subtree Bounds, Sorted Inorder, Hibbard Delete & `TreeMap` — Exploit the global BST ordering invariant (EVERY node in `left` `< node.val <` EVERY node in `right`), inorder sorted streaming, Hibbard inorder-successor deletion, and why unguarded BSTs degenerate to `O(N)` sticks. • ✓ Build, verify, and ship Module Capstone: Single-Pass Binary Tree Diameter/LCA Solver & Range-Bounded BST Validator.

7.1 Binary Tree — DFS Traversals, Level-Order BFS, `O(N)` Postorder Diameter & Lowest Common Ancestor [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: Preorder / Inorder / Postorder DFS & Level-Order BFS levelSize snapshot • Core Syntax, CLI & Production APIs: class TreeNode { int val; TreeNode left, right; } + Level-Order Queue + Postorder Bottom-Up... • Deep-Dive Mechanics: O(N) bottom-up postorder Diameter & Lowest Common Ancestor (LCA) • Advanced Patterns: O(H) stack: log N balanced vs N skewed & Morris O(1)-space threaded inorder • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Single-Pass `O(N)` Binary Tree Diameter Calculator • Idiomatic DSAJ Implementation: immutable Records, Generics, Streams & Virtual Threads • JDK 21 JVM Stack/Heap & JIT Execution & Memory Lifecycle for Binary Tree

Purpose: To understand and apply Binary Tree — DFS Traversals, Level-Order BFS, `O(N)` Postorder Diameter & Lowest Common Ancestor in real DSA with Java systems — specifically: Ma...

What Topic Is Next → Next Topic 7.2: Binary Search Tree (BST) — Global Subtree Bounds, Sorted Inorder, Hibbard Delete & `TreeMap`

7.2 Binary Search Tree (BST) — Global Subtree Bounds, Sorted Inorder, Hibbard Delete & `TreeMap` [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: Global (min, max) Long BST invariant & Inorder = strictly sorted sequence • Core Syntax, CLI & Production APIs: isValidBST(node, Long.MIN_VALUE, Long.MAX_VALUE) + Hibbard Successor Delete + NavigableMap (... • Deep-Dive Mechanics: Hibbard 2-child inorder successor delete & O(N) sorted-insertion degeneration • Advanced Patterns: Floor() / ceiling() via TreeSet/TreeMap & O(H + K) iterative K-th smallest • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Global `(Long min, Long max)` Overflow-Safe BST Validator • Idiomatic DSAJ Implementation: immutable Records, Generics, Streams & Virtual Threads • JDK 21 JVM Stack/Heap & JIT Execution & Memory Lifecycle for Binary Search Tree (BST)

Purpose: To understand and apply Binary Search Tree (BST) — Global Subtree Bounds, Sorted Inorder, Hibbard Delete & `TreeMap` in real DSA with Java systems — specifically: Exploit...

What Topic Is Next → Module 7 Capstone (Single-Pass Binary Tree Diameter/LCA Solver & Range-Bounded BST Validator) → Next Topic 8.1: AVL Tree — Balance Factor `{-1, 0, +1}`, LL/RR/LR/RL `O(1)` Rotations & `1.44 log2(N)` Height Bound

Module 7 Hands-On Capstone Project (100 Marks): Single-Pass Binary Tree Diameter/LCA Solver & Range-Bounded BST Validator — Implement a Binary Tree analyzer computing exact Diameter + Lowest Common Ancestor in `O(N)` time, paired with a strict `(long min, long max)` BST validator that handles `Integer.MIN_VALUE`...

MODULE 8 OF 16 • STAGE 2 • 13 LIVE + LAB HOURS • 2 TOPICS (16 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

Module 8: Self-Balancing `AVL Trees` (LL/RR/LR/RL Rotations) & Binary `Heaps` (`PriorityQueue`)

Prevent `O(N)` skewed-BST degeneration permanently with height-balanced **AVL Trees** (`|bf| <= 1`, 4 rotation cases) and master array-backed **Binary Heaps** (`O(N)` Floyd `buildHeap`, Java `PriorityQueue`, Top-`K`, Two-Heap Median).

Objectives: ✓ AVL Tree — Balance Factor `{-1, 0, +1}`, LL/RR/LR/RL `O(1)` Rotations & `1.44 log2(N)` Height Bound — How Adelson-Velsky & Landis (AVL) trees enforce `|height(L) - height(R)| <= 1` at every node via `O(1)` pointer rotations, guaranteeing `O(log N)` worst-case search even under adversarial sorted insertions. • ✓ Binary Heap & Java `PriorityQueue` — `O(N)` Floyd Heapify, Top-`K` Min-Heap & `Integer.compare` — Understand Complete Binary Tree array indexing, why bottom-up Floyd `buildHeap` runs in linear `O(N)` (not `O(N log N)`!), and why finding the **`K` Largest** elements uses a **Min-Heap** of size `K` in `O(N log K)`. • ✓ Build, verify, and ship Module Capstone: Self-Balancing AVL Tree Dictionary & Two-Heap Streaming Median Finder.

8.1 AVL Tree — Balance Factor `{-1, 0, +1}`, LL/RR/LR/RL `O(1)` Rotations & `1.44 log2(N)` Height Bound

[50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: Balance Factor bf in {-1, 0, +1} & LL -> single Right Rotation • Core Syntax, CLI & Production APIs: bf = height(left) - height(right) + rotateRight (LL) + rotateLeft (RR) + LR / RL Double Rota... • Deep-Dive Mechanics: RR -> single Left Rotation & LR & RL double rotations • Advanced Patterns: Strict 1.44 log2(N) height ceiling & AVL read-speed vs Red-Black write-speed • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: AVL Self-Balancing Insert on Sorted Keys `10..50` • Idiomatic DSAJ Implementation: immutable Records, Generics, Streams & Virtual Threads • JDK 21 JVM Stack/Heap & JIT Execution & Memory Lifecycle for AVL Tree

Purpose: To understand and apply AVL Tree — Balance Factor `{-1, 0, +1}`, LL/RR/LR/RL `O(1)` Rotations & `1.44 log2(N)` Height Bound in real DSA with Java systems — specifically:...

What Topic Is Next → Next Topic 8.2: Binary Heap & Java `PriorityQueue` — `O(N)` Floyd Heapify, Top-`K` Min-Heap & `Integer.compare`

8.2 Binary Heap & Java `PriorityQueue` — `O(N)` Floyd Heapify, Top-`K` Min-Heap & `Integer.compare`

[50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: Complete Binary Tree 2*i+1, 2*i+2 & Floyd bottom-up O(N) buildHeap • Core Syntax, CLI & Production APIs: PriorityQueue<Integer> minH = new PriorityQueue<>(); PriorityQueue<Integer> maxH = new Prior... • Deep-Dive Mechanics: Top-K Largest = size-K Min-Heap & Integer.compare vs b - a overflow bug • Advanced Patterns: Two-Heap O(log N) streaming median & K-Way Merge sorted lists O(N log K) • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: `O(N log K)` K-th Largest Element via Bounded Min-Heap • Idiomatic DSAJ Implementation: immutable Records, Generics, Streams & Virtual Threads • JDK 21 JVM Stack/Heap & JIT Execution & Memory Lifecycle for Binary Heap

Purpose: To understand and apply Binary Heap & Java `PriorityQueue` — `O(N)` Floyd Heapify, Top-`K` Min-Heap & `Integer.compare` in real DSA with Java systems — specifically: Unde...

What Topic Is Next → Module 8 Capstone (Self-Balancing AVL Tree Dictionary & Two-Heap Streaming Median Finder) → Next Topic 9.1: Breadth-First Search (BFS) — Unweighted Shortest Path, Multi-Source BFS & Enqueue-Mark Rule

Module 8 Hands-On Capstone Project (100 Marks): Self-Balancing AVL Tree Dictionary & Two-Heap Streaming Median Finder — Implement an AVL Tree supporting sequential sorted insertions `1..15` while maintaining strict `|bf| <= 1` height `<= 4`, paired with a `MedianFinder` maintaining a Max-Heap (lower half) and...

MODULE 9 OF 16 • STAGE 3 • 13 LIVE + LAB HOURS • 2 TOPICS (16 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

Module 9: Graph Traversals — Breadth-First Search (`BFS`) & Depth-First Search (`DFS` / Topological Sort)

Represent sparse graphs cleanly with Adjacency Lists (`List<List<Integer>>`), compute unweighted shortest paths & multi-source wave propagation via **BFS**, and detect cycles, connected components & Topological Order via **DFS**.

Objectives: ✓ Breadth-First Search (BFS) — Unweighted Shortest Path, Multi-Source BFS & Enqueue-Mark Rule — Explore graphs ring-by-ring in monotonic distance order (`dist = 0, 1, 2, ...`) using `ArrayDeque`: unweighted shortest paths, Bipartite 2-coloring, Word Ladder, and Multi-Source Grid BFS. • ✓ Depth-First Search (DFS) — Connected Components, 3-Color Directed Cycle Detection & Topological Sort — Plunge along graph paths recursively or iteratively: flood-fill islands, classify back-edges via 3-color states (`0/1/2`) to detect DAG cycles, and emit reverse-postorder Topological Sorts. • ✓ Build, verify, and ship Module Capstone: Multi-Source Grid Shortest-Wave BFS & Directed Course-Prerequisite Cycle Detector.

9.1 Breadth-First Search (BFS) — Unweighted Shortest Path, Multi-Source BFS & Enqueue-Mark Rule

[50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: O(V + E) Adjacency List BFS & Mark visited ON ENQUEUE rule • Core Syntax, CLI & Production APIs: List<List<Integer>> adj + Deque<Integer> q = new ArrayDeque<>() + boolean[] visited (mark on... • Deep-Dive Mechanics: Unweighted shortest-path guarantee & Multi-Source simultaneous seed BFS • Advanced Patterns: Bipartite 2-color odd-cycle check & 4-directional grid dr/dc bounds • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Unweighted Shortest Path Length via Adjacency-List BFS • Idiomatic DSAJ Implementation: immutable Records, Generics, Streams & Virtual Threads • JDK 21 JVM Stack/Heap & JIT Execution & Memory Lifecycle for Breadth-First Search (BFS)

Purpose: To understand and apply Breadth-First Search (BFS) — Unweighted Shortest Path, Multi-Source BFS & Enqueue-Mark Rule in real DSA with Java systems — specifically: Explore...

What Topic Is Next → Next Topic 9.2: Depth-First Search (DFS) — Connected Components, 3-Color Directed Cycle Detection & Topological Sort

9.2 Depth-First Search (DFS) — Connected Components, 3-Color Directed Cycle Detection & Topological Sort

[50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: O(V + E) recursive & iterative DFS & 3-color GRAY back-edge DAG cycle test • Core Syntax, CLI & Production APIs: int[] color (0=WHITE, 1=GRAY on-stack, 2=BLACK done) + Postorder Topological Reverse • Deep-Dive Mechanics: Diamond DAG boolean[] false-positive trap & Reverse postorder Topological Sort • Advanced Patterns: Connected components / Number of Islands & JVM StackOverflowError at depth 10^5 • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: 3-Color Directed Graph Cycle Detector & Course Schedule Validator • Idiomatic DSAJ Implementation: immutable Records, Generics, Streams & Virtual Threads • JDK 21 JVM Stack/Heap & JIT Execution & Memory Lifecycle for Depth-First Search (DFS)

Purpose: To understand and apply Depth-First Search (DFS) — Connected Components, 3-Color Directed Cycle Detection & Topological Sort in real DSA with Java systems — specifically:...

What Topic Is Next → Module 9 Capstone (Multi-Source Grid Shortest-Wave BFS & Directed Course-Prerequisite Cycle Detector) → Next Topic 10.1: Kruskal's Algorithm & Disjoint Set Union — Cut Property, Path Compression & `O(E log E)` MST

Module 9 Hands-On Capstone Project (100 Marks): Multi-Source Grid Shortest-Wave BFS & Directed Course-Prerequisite Cycle Detector — Implement a Multi-Source BFS grid solver (`Rotting Oranges` / `01 Matrix` wave distance) marking cells visited on enqueue, paired with a 3-color Directed Graph DFS cycle detector and Topolog...

MODULE 10 OF 16 • STAGE 3 • 13 LIVE + LAB HOURS • 2 TOPICS (16 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

Module 10: Minimum Spanning Trees — `Kruskal's` (Disjoint Set Union) & `Prim's` (`PriorityQueue`)

Connect all `V` vertices of a weighted undirected graph at minimum total edge cost (`V - 1` edges, zero cycles) using edge-greedy **Kruskal's Algorithm + Union-Find (`O(α(V))`)** and vertex-cut **Prim's Algorithm**.

Objectives: ✓ Kruskal's Algorithm & Disjoint Set Union — Cut Property, Path Compression & `O(E log E)` MST — Sort all edges by ascending weight and greedily accept each edge `(u, v, w)` whose endpoints belong to distinct connected components via near-constant `O(α(V))` **Disjoint Set Union (Union-Find)**. • ✓ Prim's Algorithm — Vertex-Cut Min-Heap Growth, Lazy `inMst` Filtering & Dense vs Sparse Choice — Grow a single connected tree from an arbitrary start vertex `0` by repeatedly pulling the lightest frontier edge `(w, v)` from a `PriorityQueue` where `v` is not yet in the MST. • ✓ Build, verify, and ship Module Capstone: Minimum-Cost Fiber Network Planner Comparing Kruskal (DSU) & Prim (Min-Heap).

10.1 Kruskal's Algorithm & Disjoint Set Union — Cut Property, Path Compression & `O(E log E)` MST

[50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: MST V - 1 edges & Cut Property & Disjoint Set Union (Union-Find) • Core Syntax, CLI & Production APIs: Arrays.sort(edges, Comparator.comparingInt(e -> e[2])) + DSU (find with Path Compression + u... • Deep-Dive Mechanics: Path Compression + Union by Size & Inverse Ackermann O(alpha(V)) <= 4 • Advanced Patterns: O(E log E) sparse-graph sweet spot & Picked == V - 1 connectivity check • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Kruskal's MST with Path-Compressed Union-Find • Idiomatic DSAJ Implementation: immutable Records, Generics, Streams & Virtual Threads • JDK 21 JVM Stack/Heap & JIT Execution & Memory Lifecycle for Kruskal's Algorithm

Purpose: To understand and apply Kruskal's Algorithm & Disjoint Set Union — Cut Property, Path Compression & `O(E log E)` MST in real DSA with Java systems — specifically: Sort al...

What Topic Is Next → Next Topic 10.2: Prim's Algorithm — Vertex-Cut Min-Heap Growth, Lazy `inMst` Filtering & Dense vs Sparse Choice

10.2 Prim's Algorithm — Vertex-Cut Min-Heap Growth, Lazy `inMst` Filtering & Dense vs Sparse Choice

[50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: Single-tree frontier growth & Prim edge weight vs Dijkstra dist[u]+w • Core Syntax, CLI & Production APIs: PriorityQueue<int[]> pq = new PriorityQueue<>(Comparator.comparingInt(a -> a[0])) + boolean[... • Deep-Dive Mechanics: Mark inMst[u] ON POP rule & O(E log V) heap vs O(V^2) dense scan • Advanced Patterns: 2D point cloud Manhattan MST & Works cleanly with negative weights • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Lazy Min-Heap Prim's Minimum Spanning Tree • Idiomatic DSAJ Implementation: immutable Records, Generics, Streams & Virtual Threads • JDK 21 JVM Stack/Heap & JIT Execution & Memory Lifecycle for Prim's Algorithm

Purpose: To understand and apply Prim's Algorithm — Vertex-Cut Min-Heap Growth, Lazy `inMst` Filtering & Dense vs Sparse Choice in real DSA with Java systems — specifically: Grow...

What Topic Is Next → Module 10 Capstone (Minimum-Cost Fiber Network Planner Comparing Kruskal (DSU) & Prim (Min-Heap)) → Next Topic 11.1: BFS/DFS State Modeling, Cycle Detection & Kahn's Topological Sort

Module 10 Hands-On Capstone Project (100 Marks): Minimum-Cost Fiber Network Planner Comparing Kruskal (DSU) & Prim (Min-Heap) — Build both Kruskal's (Union-Find with Path Compression + Union by Size) and Prim's (`PriorityQueue` cut expansion) MST solvers verifying identical minimum total cable cost and returning `-1`...

MODULE 11 OF 16 • STAGE 3 • 16 LIVE + LAB HOURS • 4 TOPICS (32 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

Advanced Graph Algorithms: Shortest Paths, Topo Sort, MST & Union-Find

Production-grade DSA with Java Training engineering module covering Advanced Graph Algorithms: Shortest Paths, Topo Sort, MST & Union-Find with hands-on labs and capstone build.

Objectives: ✓ BFS/DFS State Modeling, Cycle Detection & Kahn's Topological Sort — Purpose: Solve course-schedule, build-system order, and grid shortest-path interview problems. • ✓ Dijkstra (PriorityQueue), 0-1 BFS & Bellman-Ford Weighted Paths — Purpose: Compute minimum-cost network and delivery routes with optimal O((V+E) log V) Java code. • ✓ Disjoint Set Union (Path Compression + Union by Rank) & Kruskal MST — Purpose: Answer dynamic connectivity and clustering queries in near-constant O(alpha(N)) time. • ✓ Multi-Source BFS & Bipartite Graph Coloring — Purpose: Tackle rotten-oranges, flood-spread, and conflict-partitioning FAANG graph rounds.

11.1 BFS/DFS State Modeling, Cycle Detection & Kahn's Topological Sort

[65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of Cycle Detection • Integration with Dijkstra (PriorityQueue), 0 • Live Debugging, Telemetry & Technical Interview Trade • What Is BFS/DFS State Modeling, Cycle Detection & Kahn's Topological Sort & Why Industry Teams Use It • In DSA with Java Training, BFS/DFS State

Modeling, Cycle Detection & Kahn’s Topological Sort exists for one clear r... • Even if you come from a non • Runnable Code Implementation (JAVA)

Purpose: To understand and apply BFS/DFS State Modeling, Cycle Detection & Kahn’s Topological Sort in real DSA with Java systems — specifically: Solve course-schedule, build-syste...

What Topic Is Next → Next Topic 11.2: Dijkstra (PriorityQueue), 0-1 BFS & Bellman-Ford Weighted Paths

11.2 Dijkstra (PriorityQueue), 0-1 BFS & Bellman-Ford Weighted Paths [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of 0 • Integration with Disjoint Set Union (Path Compression + Union by Rank) & Kruskal MST • Live Debugging, Telemetry & Technical Interview Trade • What Is Dijkstra (PriorityQueue), 0 • In DSA with Java Training, Dijkstra (PriorityQueue), 0 • Even if you come from a non • Runnable Code Implementation (JAVA)

Purpose: To understand and apply Dijkstra (PriorityQueue), 0-1 BFS & Bellman-Ford Weighted Paths in real DSA with Java systems — specifically: Compute minimum-cost network and del...

What Topic Is Next → Next Topic 11.3: Disjoint Set Union (Path Compression + Union by Rank) & Kruskal MST

11.3 Disjoint Set Union (Path Compression + Union by Rank) & Kruskal MST [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of Kruskal MST • Integration with Multi • Live Debugging, Telemetry & Technical Interview Trade • What Is Disjoint Set Union (Path Compression + Union by Rank) & Kruskal MST & Why Industry Teams Use It • In DSA with Java Training, Disjoint Set Union (Path Compression + Union by Rank) & Kruskal MST exists for one clear... • Even if you come from a non • Runnable Code Implementation (JAVA)

Purpose: To understand and apply Disjoint Set Union (Path Compression + Union by Rank) & Kruskal MST in real DSA with Java systems — specifically: Answer dynamic connectivity and...

What Topic Is Next → Next Topic 11.4: Multi-Source BFS & Bipartite Graph Coloring

11.4 Multi-Source BFS & Bipartite Graph Coloring [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of Bipartite Graph Coloring • Integration with BFS/DFS State Modeling, Cycle Detection & Kahn’s Topological Sort • Live Debugging, Telemetry & Technical Interview Trade • What Is Multi • In DSA with Java Training, Multi • Even if you come from a non • Runnable Code Implementation (JAVA)

Purpose: To understand and apply Multi-Source BFS & Bipartite Graph Coloring in real DSA with Java systems — specifically: Tackle rotten-oranges, flood-spread, and conflict-partit...

What Topic Is Next → Module 11 Capstone (Low-Latency Route Planner & Dependency Cycle Detector in Java) → Next Topic 12.1: Top-Down Memoization -> Bottom-Up Tabulation -> Space Optimization

Module 11 Hands-On Capstone Project (100 Marks): Low-Latency Route Planner & Dependency Cycle Detector in Java — Design, implement, test, and document 'Low-Latency Route Planner & Dependency Cycle Detector in Java' applying every production concept from Module 11 (Advanced Graph Algorithms: Shortest Pa...

MODULE 12 OF 16 • STAGE 3 • 16 LIVE + LAB HOURS • 4 TOPICS (32 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

Dynamic Programming Mastery: 1D/2D, Knapsack, Intervals, Trees & Bitmask

Production-grade DSA with Java Training engineering module covering Dynamic Programming Mastery: 1D/2D, Knapsack, Intervals, Trees & Bitmask with hands-on labs and capstone build.

Objectives: ✓ Top-Down Memoization -> Bottom-Up Tabulation -> Space Optimization — Purpose: Convert exponential O(2^N) recursion into crisp O(N*W) DP systematically every time. • ✓ 0/1 & Unbounded Knapsack, Subset Sum & Coin-Change Pattern Family — Purpose: Recognize capacity-allocation problems immediately in OA coding assessments. • ✓ LCS, Edit Distance, LIS (O(N log N) Patience) & Palindromic DP — Purpose: Master string/subsequence DP asked across Amazon, Google, and Microsoft interviews. • ✓ Interval DP, DP on Trees & State-Machine Stock Trading Problems — Purpose: Handle cooldown/transaction-capped trading and tree diameter/robber rounds confidently.

12.1 Top-Down Memoization -> Bottom-Up Tabulation -> Space Optimization [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of Edge • Integration with 0/1 & Unbounded Knapsack, Subset Sum & Coin • Live Debugging, Telemetry & Technical Interview Trade • What Is Top • In DSA with Java Training, Top • Even if you come from a non • Runnable Code Implementation (JAVA)

Purpose: To understand and apply Top-Down Memoization -> Bottom-Up Tabulation -> Space Optimization in real DSA with Java systems — specifically: Convert exponential O(2^N) recurs...

What Topic Is Next → Next Topic 12.2: 0/1 & Unbounded Knapsack, Subset Sum & Coin-Change Pattern Family

12.2 0/1 & Unbounded Knapsack, Subset Sum & Coin-Change Pattern Family [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of Unbounded Knapsack • Integration with LCS, Edit Distance, LIS (O(N log N) Patience) & Palindromic DP • Live Debugging, Telemetry & Technical Interview Trade • What Is 0/1 & Unbounded Knapsack, Subset Sum & Coin • In DSA with Java Training, 0/1 & Unbounded Knapsack, Subset Sum & Coin • Even if you come from a non • Runnable Code Implementation (JAVA)

Purpose: To understand and apply 0/1 & Unbounded Knapsack, Subset Sum & Coin-Change Pattern Family in real DSA with Java systems — specifically: Recognize capacity-allocation prob...

What Topic Is Next → Next Topic 12.3: LCS, Edit Distance, LIS (O(N log N) Patience) & Palindromic DP

12.3 LCS, Edit Distance, LIS (O(N log N) Patience) & Palindromic DP [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of Edit Distance • Integration with Interval DP, DP on Trees & State • Live Debugging, Telemetry & Technical Interview Trade • What Is LCS, Edit Distance, LIS (O(N log N) Patience) & Palindromic DP & Why Industry Teams Use It • In DSA with Java Training, LCS, Edit Distance, LIS (O(N log N) Patience) & Palindromic DP exists for one clear real • Even if you come from a non • Runnable Code Implementation (JAVA)

Purpose: To understand and apply LCS, Edit Distance, LIS (O(N log N) Patience) & Palindromic DP in real DSA with Java systems — specifically: Master string/subsequence DP asked ac...

What Topic Is Next → Next Topic 12.4: Interval DP, DP on Trees & State-Machine Stock Trading Problems

12.4 Interval DP, DP on Trees & State-Machine Stock Trading Problems [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of DP on Trees • Integration with Top • Live Debugging, Telemetry & Technical Interview Trade • What Is Interval DP, DP on Trees & State • In DSA with Java Training, Interval DP, DP on Trees & State • Even if you come from a non • Runnable Code Implementation (JAVA)

Purpose: To understand and apply Interval DP, DP on Trees & State-Machine Stock Trading Problems in real DSA with Java systems — specifically: Handle cooldown/transaction-capped t...

What Topic Is Next → Module 12 Capstone (Algorithmic Portfolio Optimizer & Sequence Alignment Suite) → Next Topic 13.1: Prefix Tries & Bitwise XOR Tries

Module 12 Hands-On Capstone Project (100 Marks): Algorithmic Portfolio Optimizer & Sequence Alignment Suite — Design, implement, test, and document 'Algorithmic Portfolio Optimizer & Sequence Alignment Suite' applying every production concept from Module 12 (Dynamic Programming Mastery: 1D/2D, Knaps...

MODULE 13 OF 16 • STAGE 4 • 16 LIVE + LAB HOURS • 4 TOPICS (32 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

Tries, Segment Trees, Fenwick Trees (BIT) & String Pattern Matching

Production-grade DSA with Java Training engineering module covering Tries, Segment Trees, Fenwick Trees (BIT) & String Pattern Matching with hands-on labs and capstone build.

Objectives: ✓ Prefix Tries & Bitwise XOR Tries — Purpose: Power O(L) search autocomplete, spellcheck, and maximum-XOR subarray queries. • ✓ Fenwick Tree (BIT) & Segment Tree with Point Update / Range Query — Purpose: Maintain live prefix sums and range minimums across millions of streaming updates. • ✓ Rolling Hash (Rabin-Karp), KMP Prefix Function & Z-Algorithm — Purpose: Match substrings in strict linear O(N + M) time without worst-case quadratic TLEs. • ✓ Sweep-Line Intervals, Difference Arrays & Coordinate Compression — Purpose: Count peak concurrent meetings/servers cleanly over large timestamp ranges.

13.1 Prefix Tries & Bitwise XOR Tries [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of Bitwise XOR Tries • Integration with Fenwick Tree (BIT) & Segment Tree with Point Update / Range Query • Live Debugging, Telemetry & Technical Interview Trade • What Is Prefix Tries & Bitwise XOR Tries & Why Industry Teams Use It • In DSA with Java Training, Prefix Tries & Bitwise XOR Tries exists for one clear real • Even if you come from a non • Runnable Code Implementation (JAVA)

Purpose: To understand and apply Prefix Tries & Bitwise XOR Tries in real DSA with Java systems — specifically: Power O(L) search autocomplete, spellcheck, and maximum-XOR subarra...

What Topic Is Next → Next Topic 13.2: Fenwick Tree (BIT) & Segment Tree with Point Update / Range Query

13.2 Fenwick Tree (BIT) & Segment Tree with Point Update / Range Query [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of Segment Tree with Point Update / Range Query • Integration with Rolling Hash (Rabin • Live Debugging, Telemetry & Technical Interview Trade • What Is Fenwick Tree (BIT) & Segment Tree with Point Update / Range Query & Why Industry Teams Use It • In DSA with Java Training, Fenwick Tree (BIT) & Segment Tree with Point Update / Range Query exists for one clear r... • Even if you come from a non • Runnable Code Implementation (JAVA)

Purpose: To understand and apply Fenwick Tree (BIT) & Segment Tree with Point Update / Range Query in real DSA with Java systems — specifically: Maintain live prefix sums and rang...

What Topic Is Next → Next Topic 13.3: Rolling Hash (Rabin-Karp), KMP Prefix Function & Z-Algorithm

13.3 Rolling Hash (Rabin-Karp), KMP Prefix Function & Z-Algorithm [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of KMP Prefix Function • Integration with Sweep • Live Debugging, Telemetry & Technical Interview Trade • What Is Rolling Hash (Rabin • In DSA with Java Training, Rolling Hash (Rabin • Even if you come from a non • Runnable Code Implementation (JAVA)

Purpose: To understand and apply Rolling Hash (Rabin-Karp), KMP Prefix Function & Z-Algorithm in real DSA with Java systems — specifically: Match substrings in strict linear O(N +...

What Topic Is Next → Next Topic 13.4: Sweep-Line Intervals, Difference Arrays & Coordinate Compression

13.4 Sweep-Line Intervals, Difference Arrays & Coordinate Compression [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of Difference Arrays • Integration with Prefix Tries & Bitwise XOR Tries • Live Debugging, Telemetry & Technical Interview Trade • What Is Sweep • In DSA with Java Training, Sweep • Even if you come from a non • Runnable Code Implementation (JAVA)

Purpose: To understand and apply Sweep-Line Intervals, Difference Arrays & Coordinate Compression in real DSA with Java systems — specifically: Count peak concurrent meetings/serv...

What Topic Is Next → Module 13 Capstone (Autocomplete Prefix Engine & Live Range-Query Analytics in Java) → Next Topic 14.1: Monotonic Stack & Deque (Sliding Window Max, Largest Histogram)

Module 13 Hands-On Capstone Project (100 Marks): Autocomplete Prefix Engine & Live Range-Query Analytics in Java — Design, implement, test, and document 'Autocomplete Prefix Engine & Live Range-Query Analytics in Java' applying every production concept from Module 13 (Tries, Segment Trees, Fenwick Trees...

MODULE 14 OF 16 • STAGE 4 • 16 LIVE + LAB HOURS • 4 TOPICS (32 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

Greedy Proofs, Monotonic Queues, Concurrency DSA & Machine-Coding LLD

Production-grade DSA with Java Training engineering module covering Greedy Proofs, Monotonic Queues, Concurrency DSA & Machine-Coding LLD with hands-on labs and capstone build.

Objectives: ✓ Monotonic Stack & Deque (Sliding Window Max, Largest Histogram) — Purpose: Turn brute-force O(N*K) window/span problems into single-pass O(N) solutions. • ✓ Heap Design Patterns: Two-Heaps Median, K-Way Merge & Top-K Frequent — Purpose: Stream live medians and merge sorted shards using Java PriorityQueue. • ✓ Doubly-Linked + HashMap O(1) LRU/LFU Cache & Token-Bucket Rate Limiter — Purpose: Ace the most frequent SDE-1/SDE-2 low-level machine-coding interview rounds. • ✓ Complexity Trade-Off Communication & Edge-Case Dry-Run Protocol — Purpose: Verbalize invariants, Big-O bounds, and integer-overflow guards like a top 5% candidate.

14.1 Monotonic Stack & Deque (Sliding Window Max, Largest Histogram) [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of Deque (Sliding Window Max • Integration with Heap Design Patterns • Live Debugging, Telemetry & Technical Interview Trade • What Is Monotonic Stack & Deque (Sliding Window Max, Largest Histogram) & Why Industry Teams Use It • In DSA with Java Training, Monotonic Stack & Deque (Sliding Window Max, Largest Histogram) exists for one clear rea... • Even if you come from a non • Runnable Code Implementation (JAVA)

Purpose: To understand and apply Monotonic Stack & Deque (Sliding Window Max, Largest Histogram) in real DSA with Java systems — specifically: Turn brute-force O(N*K) window/span...

What Topic Is Next → Next Topic 14.2: Heap Design Patterns: Two-Heaps Median, K-Way Merge & Top-K Frequent

14.2 Heap Design Patterns: Two-Heaps Median, K-Way Merge & Top-K Frequent [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of K • Integration with Doubly • Live Debugging, Telemetry & Technical Interview Trade • What Is Heap Design Patterns • In DSA with Java Training, Heap Design Patterns • Even if you come from a non • Runnable Code Implementation (JAVA)

Purpose: To understand and apply Heap Design Patterns: Two-Heaps Median, K-Way Merge & Top-K Frequent in real DSA with Java systems — specifically: Stream live medians and merge s...

What Topic Is Next → Next Topic 14.3: Doubly-Linked + HashMap O(1) LRU/LFU Cache & Token-Bucket Rate Limiter

14.3 Doubly-Linked + HashMap O(1) LRU/LFU Cache & Token-Bucket Rate Limiter [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of Token • Integration with Complexity Trade • Live Debugging, Telemetry & Technical Interview Trade • What Is Doubly • In DSA with Java Training, Doubly • Even if you come from a non • Runnable Code Implementation (JAVA)

Purpose: To understand and apply Doubly-Linked + HashMap O(1) LRU/LFU Cache & Token-Bucket Rate Limiter in real DSA with Java systems — specifically: Ace the most frequent SDE-1/S...

What Topic Is Next → Next Topic 14.4: Complexity Trade-Off Communication & Edge-Case Dry-Run Protocol

14.4 Complexity Trade-Off Communication & Edge-Case Dry-Run Protocol [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of Edge • Integration with Monotonic Stack & Deque (Sliding Window Max, Largest Histogram) • Live Debugging, Telemetry & Technical Interview Trade • What Is Complexity Trade • In DSA with Java Training, Complexity Trade • Even if you come from a non • Runnable Code Implementation (JAVA)

Purpose: To understand and apply Complexity Trade-Off Communication & Edge-Case Dry-Run Protocol in real DSA with Java systems — specifically: Verbalize invariants, Big-O bounds,...

What Topic Is Next → Module 14 Capstone (Thread-Safe LRU/Rate-Limiter Machine-Coding Round in Java) → Next Topic 15.1: Prefix Tries, Bitwise XOR Tries & String Rolling-Hash / KMP Pattern Matching

Module 14 Hands-On Capstone Project (100 Marks): Thread-Safe LRU/Rate-Limiter Machine-Coding Round in Java — Design, implement, test, and document 'Thread-Safe LRU/Rate-Limiter Machine-Coding Round in Java' applying every production concept from Module 14 (Greedy Proofs, Monotonic Queues, Concurren...

MODULE 15 OF 16 • STAGE 4 • 14 LIVE + LAB HOURS • 3 TOPICS (24 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

Advanced Graph, Trie, Segment Tree & Bit-Manipulation Patterns in Java

Self-explanatory list-and-bullet production mastery module covering Advanced Graph, Trie, Segment Tree & Bit-Manipulation Patterns in Java in DSA with Java Training, paired with hands-on labs, capstone repository, Topic Mock Tests, and Spoken MockAttempt AI Viva.

Objectives: ✓ Prefix Tries, Bitwise XOR Tries & String Rolling-Hash / KMP Pattern Matching — Purpose: Solve dictionary search, IP routing, and linear-time substring problems in O(L) / O(N) time. • ✓ Disjoint Set Union (Union-Find with Path Compression), Bridges & Topological Cycles — Purpose: Detect dynamic network connectivity, dependency deadlocks, and minimum spanning trees rapidly. • ✓ Segment Trees, Fenwick (BIT) Trees & Monotonic Queue/Stack Optimization — Purpose: Drop range-sum/min updates from O(N) to O(log N) and sliding-window extrema to O(N). • ✓ Build, verify, and document 'Autocomplete Trie + Range-Query Segment Tree Benchmark Engine in Java 21' with clean Git commits and edge-case protection. • ✓ Clear 100% of Topic Timed Mock Tests and Spoken Voice + MockAttempt AI Technical Interviews.

15.1 Prefix Tries, Bitwise XOR Tries & String Rolling-Hash / KMP Pattern Matching [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Security Hygiene & Data Safety Rules for Prefix Tries, Bitwise XOR Tries & String Rolling • Performance Optimization, Complexity & Latency Impact of Prefix Tries, Bitwise XOR Tries & String Rolling • Live Debugging, Stack • Runnable Code Implementation (JAVA) • Hands • Idiomatic DSAJ Implementation: immutable Records, Generics, Streams & Virtual Threads • JDK 21 JVM Stack/Heap & JIT Execution & Memory Lifecycle for Prefix Tries • Production Hardening & Security: defensive immutability, null-safety & thread-confinement

Purpose: To understand and apply Prefix Tries, Bitwise XOR Tries & String Rolling-Hash / KMP Pattern Matching in real DSA with Java systems — specifically: Solve dictionary search...

What Topic Is Next → Next Topic 15.2: Disjoint Set Union (Union-Find with Path Compression), Bridges & Topological Cycles

15.2 Disjoint Set Union (Union-Find with Path Compression), Bridges & Topological Cycles [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Security Hygiene & Data Safety Rules for Disjoint Set Union (Union • Performance Optimization, Complexity & Latency Impact of Disjoint Set Union (Union • Live Debugging, Stack • Runnable Code Implementation (JAVA) • Hands • Idiomatic DSAJ Implementation: immutable Records, Generics, Streams & Virtual Threads • JDK 21 JVM Stack/Heap & JIT Execution & Memory Lifecycle for Disjoint Set Union (Union-Find with Path Compression... • Production Hardening & Security: defensive immutability, null-safety & thread-confinement

Purpose: To understand and apply Disjoint Set Union (Union-Find with Path Compression), Bridges & Topological Cycles in real DSA with Java systems — specifically: Detect dynamic n...

What Topic Is Next → Next Topic 15.3: Segment Trees, Fenwick (BIT) Trees & Monotonic Queue/Stack Optimization

15.3 Segment Trees, Fenwick (BIT) Trees & Monotonic Queue/Stack Optimization [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Security Hygiene & Data Safety Rules for Segment Trees, Fenwick (BIT) Trees & Monotonic Queue/Stack Optimization • Performance Optimization, Complexity & Latency Impact of Segment Trees, Fenwick (BIT) Trees & Monotonic Queue/Stack... • Live Debugging, Stack • Runnable Code Implementation (JAVA) • Hands • Idiomatic DSAJ Implementation: immutable Records, Generics, Streams & Virtual Threads • JDK 21 JVM Stack/Heap & JIT Execution & Memory Lifecycle for Segment Trees • Production Hardening & Security: defensive immutability, null-safety & thread-confinement

Purpose: To understand and apply Segment Trees, Fenwick (BIT) Trees & Monotonic Queue/Stack Optimization in real DSA with Java systems — specifically: Drop range-sum/min updates f...

What Topic Is Next → Module 15 Capstone (Autocomplete Trie + Range-Query Segment Tree Benchmark Engine in Java 21) → Next Topic 16.1: Recognizing Patterns Instantly: Mapping Problem Constraints (N <= 10^5) to Exact Big-O Targets

Module 15 Hands-On Capstone Project (100 Marks): Autocomplete Trie + Range-Query Segment Tree Benchmark Engine in Java 21 — Architect, implement, test, and document 'Autocomplete Trie + Range-Query Segment Tree Benchmark Engine in Java 21' applying every bullet-wise engineering concept from Module 15 (Advanced Gr...

MODULE 16 OF 16 • STAGE 4 • 14 LIVE + LAB HOURS • 3 TOPICS (24 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

Competitive Edge-Case Mastery, Complexity Proofs & FAANG Java Coding Rounds

Self-explanatory list-and-bullet production mastery module covering Competitive Edge-Case Mastery, Complexity Proofs & FAANG Java Coding Rounds in DSA with Java Training, paired with hands-on labs, capstone repository, Topic Mock Tests, and Spoken MockAttempt AI Viva.

Objectives: ✓ Recognizing Patterns Instantly: Mapping Problem Constraints (N <= 10^5) to Exact Big-O Targets — Purpose: Choose between DP, Greedy, Binary Search on Answer, or Two Pointers within 2 minutes of reading a prompt. • ✓ Integer Overflow Guards, Off-by-One Invariants & Iterative Stack Conversions in Java — Purpose: Eliminate StackOverflowError and long-overflow WA failures on adversarial hidden test cases. • ✓ Communicating Brute-Force -> Optimal Trade-Offs Aloud in Live SDE Interviews — Purpose: Structure dry runs, complexity proofs, and clean modular Java helper methods under time pressure. • ✓ Build, verify, and document 'Timed 45-Minute Multi-Constraint Problem Solver & JUnit Stress-Tester' with clean Git commits and edge-case protection. • ✓ Clear 100% of Topic Timed Mock Tests and Spoken Voice + MockAttempt AI Technical Interviews.

16.1 Recognizing Patterns Instantly: Mapping Problem Constraints (N <= 10^5) to Exact Big-O Targets [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Security Hygiene & Data Safety Rules for Recognizing Patterns Instantly • Performance Optimization, Complexity & Latency Impact of Recognizing Patterns Instantly • Live Debugging, Stack • Runnable Code Implementation (JAVA) • Hands • Idiomatic DSAJ Implementation: immutable Records, Generics, Streams & Virtual Threads • JDK 21 JVM Stack/Heap & JIT Execution & Memory Lifecycle for Recognizing Patterns Instantly • Production Hardening & Security: defensive immutability, null-safety & thread-confinement

Purpose: To understand and apply Recognizing Patterns Instantly: Mapping Problem Constraints (N <= 10^5) to Exact Big-O Targets in real DSA with Java systems — specifically: Choos...

What Topic Is Next → Next Topic 16.2: Integer Overflow Guards, Off-by-One Invariants & Iterative Stack Conversions in Java

16.2 Integer Overflow Guards, Off-by-One Invariants & Iterative Stack Conversions in Java [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Security Hygiene & Data Safety Rules for Integer Overflow Guards, Off • Performance Optimization, Complexity & Latency Impact of Integer Overflow Guards, Off • Live Debugging, Stack • Runnable Code Implementation (JAVA) • Hands • Idiomatic DSAJ Implementation: immutable Records, Generics, Streams & Virtual Threads • JDK 21 JVM Stack/Heap & JIT Execution & Memory Lifecycle for Integer Overflow Guards • Production Hardening & Security: defensive immutability, null-safety & thread-confinement

Purpose: To understand and apply Integer Overflow Guards, Off-by-One Invariants & Iterative Stack Conversions in Java in real DSA with Java systems — specifically: Eliminate Stack...

What Topic Is Next → Next Topic 16.3: Communicating Brute-Force -> Optimal Trade-Offs Aloud in Live SDE Interviews

16.3 Communicating Brute-Force -> Optimal Trade-Offs Aloud in Live SDE Interviews [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Security Hygiene & Data Safety Rules for Communicating Brute • Performance Optimization, Complexity & Latency Impact of Communicating Brute • Live Debugging, Stack • Runnable Code Implementation (JAVA) • Hands • Idiomatic DSAJ Implementation: immutable Records, Generics, Streams & Virtual Threads • JDK 21 JVM Stack/Heap & JIT Execution & Memory Lifecycle for Communicating Brute-Force -> Optimal Trade-Offs Alou... • Production Hardening & Security: defensive immutability, null-safety & thread-confinement

Purpose: To understand and apply Communicating Brute-Force -> Optimal Trade-Offs Aloud in Live SDE Interviews in real DSA with Java systems — specifically: Structure dry runs, com...

What Topic Is Next → Module 16 Capstone → End-of-Course Live Client Project: AlgoSearch High-Speed Autocomplete, Graph Route Planner & LRU Cache Engine in Java

Module 16 Hands-On Capstone Project (100 Marks): Timed 45-Minute Multi-Constraint Problem Solver & JUnit Stress-Tester — Architect, implement, test, and document 'Timed 45-Minute Multi-Constraint Problem Solver & JUnit Stress-Tester' applying every bullet-wise engineering concept from Module 16 (Competitive Ed...

3. LIVE CLIENT PRODUCTION PROJECT & CAREER SCOPE

End-of-Course Live Client Production Project + Official Internship Confirmation Letter

Project: AlgoSearch High-Speed Autocomplete, Graph Route Planner & LRU Cache Engine in Java • **Confirmed Role:** Java Algorithms, Data Structures & Low-Latency Systems Intern

Domain & Production Infra: High-Scale Search & Routing Core Platform Team (Java 21 LTS + Custom Trie/Heap/Graph/LRU Engines + JMH Benchmarks + JUnit 5)

- Implemented O(L) prefix autocomplete using compressed Tries and top-K min-heap frequency ranking in Java 21.
- Engineered weighted shortest-path routing with Dijkstra/A* priority queues and thread-safe O(1) LRU caching.
- Benchmarked time and heap-memory complexity across 1M+ synthetic inputs, eliminating GC bottlenecks.
- Documented Big-O trade-offs and passed rigorous edge-case stress tests modeled on FAANG/product interviews.

Target Job Roles: Software Development Engineer — SDE-1 / SDE-2, Product Backend Engineer (Java), Algorithms & Core Platform Engineer, Member of Technical Staff (MTS)

Realistic Annual CTC Scope: **Product Fresher / SDE-1 (0–1 Yr):** ₹8.0 LPA – ₹18.0 LPA **Product SDE-2 (1–3 Yrs):** ₹16.0 LPA – ₹32.0 LPA **Senior SDE / Staff Product Architect (3–6+ Yrs):** ₹30.0 LPA – ₹60.0+ LPA

MockAttempt Academy Pvt. Ltd. • Email: support@mockattempt.com • Phone/WhatsApp: +91 95400 07917 • Web: https://mockattempt.com/courses/dsa-with-java-training

Try 2 Free Topic-Wise Timed Topic Mock Tests & Spoken MockAttempt AI Interviews daily on every topic above before enrolling.