Admissions & Support: support@mockattempt.com
Phone / WhatsApp: +91 95400 07917
Course Code: DSAP • **Updated:** 04 Nov 2026
https://mockattempt.com/courses/dsa-with-python-training

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

100% LIVE INDUSTRY EXPERT COHORT + INTERNSHIP LETTER

DSA with Python Training

Master product-company DSA & Design and Analysis of Algorithms (DAA) in Python 3.12: Complexity & Master Theorem, bisect Binary Search, Merge/Quick/Heap Sort, CPython Dict Hashing, Red-Black Trees, Memoization vs Tabulation DP, 0/1 Knapsack, LCS, O(N log N) LIS, Greedy Huffman, Backtracking N-Queens, Dijkstra, Bellman-Ford & KMP — with 244 Topic Mock Tests (10,440 MCQs) and 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 (Python 3.12 (Data Structures & Algorithms)): Live trainer support to crack LeetCode & HackerRank problems, optimize complexity, and handle edge cases.**

Core Technologies & Engineering Stack: Python 3.12 Type-Hinted DSA • Master Theorem & Recurrence Analysis • bisect, heapq & collections.deque • CPython Compact Dict & Red-Black Trees • Dynamic Programming (Knapsack, LCS, LIS) • Greedy, Huffman & Backtracking (N-Queens) • Dijkstra, Bellman-Ford & KMP String Match

1. PROGRAM MASTERY OUTCOMES & ELIGIBILITY

End-to-End Engineering Outcomes:

1. Solve divide-and-conquer recurrences via the Master Theorem and avoid hidden CPython `O(N)` traps (`list.pop(0)`, string `+=` loops, shallow `[[0]*M]*N` aliasing)
2. Implement optimal Binary Search (`bisect_left`), Merge/Quick/Heap Sort, CPython compact-dict hashing, and Red-Black Tree invariants
3. Conquer FAANG Dynamic Programming (`@cache` vs Rolling-Row Tabulation, 1D 0/1 Knapsack, LCS, `O(N log N)` Patience LIS), Greedy Huffman Trees, and N-Queens Backtracking
4. Clear 244 Timed Mock Exams (10,000 Python DSA MCQs) and 200 Spoken Voice + MockAttempt AI Algorithmic Viva Rounds
5. Deep understanding of CPython complexity (`list` vs `collections.deque`, `dict` hashing, `heapq`, `bisect`, `Counter`).
6. Crack LeetCode and HackerRank problems in Python 3.12 (Data Structures & Algorithms) with live trainer mentorship on algorithmic/query patterns, edge cases, and optimal complexity.
7. Pattern-driven mastery: Sliding Window, Two Pointers, Fast/Slow Pointers, Prefix Sums, Binary Search on Answer, and Monotonic Stacks.
8. Trees, Tries, Heaps, Backtracking pruning, Graph traversals/shortest paths, Union-Find, and 1D/2D Dynamic Programming.

Eligibility & Prerequisites: Basic familiarity with Python 3 syntax (lists, dicts, loops, functions) — recursion, complexity math, and stdlib DSA modules are taught step-by-step • Any Windows, macOS, or Linux laptop running Python 3.11/3.12 and VS Code or PyCharm Community • No prior DSA with Python 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: CPython Complexity Internals & Divide-and-Conquer Master Theorem

Analyze exact CPython container time/space complexities (`list`, `deque`, `set`, `dict`, slicing costs) and solve `T(n) = aT(n/b) + f(n)` divide-and-conquer recurrences instantaneously via the Master Theorem.

Objectives: ✓ Algorithm Time & Space Complexity in Python — CPython Stdlib Costs & Hidden `O(N)` Traps — Understand worst, average, and amortized complexity in CPython: why `list.append`/`pop()` are `O(1)` amortized while `list.pop(0)`/`insert(0, x)` shift pointers in `O(N)`, and why `arr[mid:]` inside recursion secretly adds `O(N)` copy cost!. • ✓ Master Theorem (`T(n) = aT(n/b) + f(n)`) — Critical Exponent `n^(log_b a)` & Cases 1, 2, 3 — Solve recurrences for Binary Search (`a=1, b=2, f=1 -> Θ(log n)`), Merge Sort (`a=2, b=2, f=n -> Θ(n log n)`), Karatsuba (`a=3, b=2, f=n -> Θ(n^1.585)`), and Strassen (`a=7, b=2, f=n^2 -> Θ(n^2.807)`) by comparing `f(n)` with watershed `n^(log_b a)`. • ✓ Build, verify, and ship Module Capstone: Automated Master Theorem Recurrence Classifier & CPython Complexity Auditor.

1.1 Algorithm Time & Space Complexity in Python — CPython Stdlib Costs & Hidden `O(N)` Traps

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

Subtopics Covered: List.pop(0) O(N) vs deque.popleft() O(1) & [[0]*m for _ in range(n)] independent rows • Core Syntax, CLI & Production APIs: collections.deque.popleft() O(1) + set / dict O(1) avg + [[0] * m for _ in range(n)] • Deep-Dive Mechanics: Slice copy cost a[l:r] = O(r - l) & 10^7 Python ops/sec TLE budget • Advanced Patterns: Sys.setrecursionlimit stack safety & "".join(parts) O(N) string assembly • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Independent 2D Grid Comprehension & `O(N)` Two-Sum Lookup • Idiomatic DSAP Implementation: PEP-8 type hints, generators, and O(1) dict/set structures • CPython 3.12 Bytecode & Heap Execution & Memory Lifecycle for Algorithm Time

Purpose: To understand and apply Algorithm Time & Space Complexity in Python — CPython Stdlib Costs & Hidden `O(N)` Traps in real DSA with Python systems — specifically: Understan...

What Topic Is Next → Next Topic 1.2: Master Theorem (`T(n) = aT(n/b) + f(n)`) — Critical Exponent `n^(log_b a)` & Cases 1, 2, 3

1.2 Master Theorem (`T(n) = aT(n/b) + f(n)`) — Critical Exponent `n^(log_b a)` & Cases 1, 2, 3

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

Subtopics Covered: Watershed exponent c_crit = log_b(a) & Case 1 leaf-dominated Theta(n^c_crit) • Core Syntax, CLI & Production APIs: c_crit = math.log(a, b); compare f(n) = Theta(n^k * log^p n) against n^(c_crit) • Deep-Dive Mechanics: Case 2 balanced Theta(n^c_crit log n) & Case 3 root-dominated + regularity • Advanced Patterns: Polynomial n^epsilon gap requirement & Karatsuba O(n^1.585) & Strassen O(n^2.81) • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Master Theorem Symbolic Complexity Solver • Idiomatic DSAP Implementation: PEP-8 type hints, generators, and O(1) dict/set structures • CPython 3.12 Bytecode & Heap Execution & Memory Lifecycle for Master Theorem (`T(n) = aT(n/b) + f(n)`)

Purpose: To understand and apply Master Theorem (`T(n) = aT(n/b) + f(n)`) — Critical Exponent `n^(log_b a)` & Cases 1, 2, 3 in real DSA with Python systems — specifically: Solve r...

What Topic Is Next → Module 1 Capstone (Automated Master Theorem Recurrence Classifier & CPython Complexity Auditor) → Next Topic 2.1: Binary Search in Python — `bisect_left` vs `bisect_right` & Parametric Search on Answer

Module 1 Hands-On Capstone Project (100 Marks): Automated Master Theorem Recurrence Classifier & CPython Complexity Auditor — Write a Python 3.12 solver classifying `T(n) = a * T(n / b) + n^k * (log n)^p` across Cases 1, 2, and 3 by comparing `c_crit = log_b(a)` against `k`, paired with a `deque` vs `list.pop(0)` o...

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

Module 2: Python `bisect` Binary Search & Guaranteed `O(N log N)` Merge Sort

Leverage C-accelerated `bisect_left` / `bisect_right` boundary searches and monotonic parametric Binary Search, and implement index-bounded `O(N log N)` Merge Sort without `O(N log N)` slice-copy overhead.

Objectives: ✓ Binary Search in Python — `bisect_left` vs `bisect_right` & Parametric Search on Answer — Master C-implemented `bisect.bisect_left` / `bisect_right` (Python 3.10+ supports `key=`), counting duplicates in `O(log N)`, and monotonic predicate search (`Koko Eating Bananas`, `Minimize Max Distance`). • ✓ Merge Sort in Python — Index-Bounded Divide & Conquer, Stability & CPython Timsort (`O(N)` Best Case) — Implement clean `Θ(N log N)` stable Merge Sort using a single shared auxiliary list, and understand how Tim Peters' **Timsort** (`sorted()`) combines natural monotonic runs with binary insertion sort + galloping merge. • ✓ Build, verify, and ship Module Capstone: `bisect` Range-Frequency Query Engine & Index-Bounded `O(N log N)` Merge Sort.

2.1 Binary Search in Python — `bisect_left` vs `bisect_right` & Parametric Search on Answer

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

Subtopics Covered: Bisect_left first >= target & Bisect_right first > target • Core Syntax, CLI & Production APIs: from bisect import bisect_left, bisect_right; mid = (lo + hi) // 2 • Deep-Dive Mechanics: O(log N) frequency = right - left & Insort O(N) list-shift pitfall • Advanced Patterns: Python arbitrary-precision mid & Monotonic predicate search on answer • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: `O(log N)` Range Frequency & Parametric Eating-Speed Search • Idiomatic DSAP Implementation: PEP-8 type hints, generators, and O(1) dict/set structures • CPython 3.12 Bytecode & Heap Execution & Memory Lifecycle for Binary Search in Python

Purpose: To understand and apply Binary Search in Python — `bisect_left` vs `bisect_right` & Parametric Search on Answer in real DSA with Python systems — specifically: Master C-i...

What Topic Is Next → Next Topic 2.2: Merge Sort in Python — Index-Bounded Divide & Conquer, Stability & CPython Timsort (`O(N)` Best Case)

2.2 Merge Sort in Python — Index-Bounded Divide & Conquer, Stability & CPython Timsort (`O(N)` Best Case) [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: Theta(N log N) worst-case guarantee & Aux[i] <= aux[j] tie-break stability • Core Syntax, CLI & Production APIs: merge_sort(a, aux, lo, hi) -> Theta(N log N) worst-case, stable (aux[i] <= aux[j]) • Deep-Dive Mechanics: Index-bounded merge vs slice copies & CPython Timsort natural run detection • Advanced Patterns: Multi-pass radix/tuple key stability & Counting Smaller Numbers After Self • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Buffer-Reusing Stable Merge Sort in Python 3.12 • Idiomatic DSAP Implementation: PEP-8 type hints, generators, and O(1) dict/set structures • CPython 3.12 Bytecode & Heap Execution & Memory Lifecycle for Merge Sort in Python

Purpose: To understand and apply Merge Sort in Python — Index-Bounded Divide & Conquer, Stability & CPython Timsort (`O(N)` Best Case) in real DSA with Python systems — specifical...

What Topic Is Next → Module 2 Capstone (`bisect` Range-Frequency Query Engine & Index-Bounded `O(N log N)` Merge Sort) → Next Topic 3.1: Quick Sort & Quickselect — Randomized Pivot, 3-Way Partitioning & Python Recursion Depth Guard

Module 2 Hands-On Capstone Project (100 Marks): `bisect` Range-Frequency Query Engine & Index-Bounded `O(N log N)` Merge Sort — Build an `O(log N)` sorted range-count query responder (`count_in_range(arr, low_val, high_val) = bisect_right(arr, high_val) - bisect_left(arr, low_val)`) paired with a stable Merge Sort co...

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

Module 3: Randomized 3-Way `Quick Sort` / Quickselect & In-Place `Binary Heap Sort` (`heapq`)

Implement duplicate-proof Randomized 3-Way Quick Sort + `O(N)` Quickselect, and master `O(1)`-space in-place Binary Heap Sort alongside Python's `heapq` (`heapify`, `heappushpop`, `nlargest`).

Objectives: ✓ Quick Sort & Quickselect — Randomized Pivot, 3-Way Partitioning & Python Recursion Depth Guard — Understand Quick Sort's `O(N log N)` expected divide-and-conquer partition mechanics, why CPython's default recursion limit (`1000`) immediately crashes naive fixed-pivot Quick Sort on sorted `N = 2000`, and iterative Quickselect. • ✓ Binary Heap Sort & Python `heapq` — `O(N)` Sift-Down `heapify`, `O(1)` Space Sort & Min-Heap Idioms — Sort any array in worst-case `O(N log N)` time and strict `O(1)` auxiliary space via in-place Max-Heapification + root swaps, and master Python's C-accelerated `heapq` module (`heapify`, `heappush`, `heappop`, `heappushpop`). • ✓ Build, verify, and ship Module Capstone: Randomized 3-Way Quickselect (`O(N)` `K`-th Element) & `O(1)`-Space In-Place Heap Sort.

3.1 Quick Sort & Quickselect — Randomized Pivot, 3-Way Partitioning & Python Recursion Depth Guard [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: Randomized pivot adversary defense & 3-Way < pivot, == pivot, > pivot • Core Syntax, CLI & Production APIs: pivot = a[random.randint(lo, hi)] + 3-Way Dutch Flag (lt, i, gt) + Iterative Quickselect • Deep-Dive Mechanics: CPython 1000 RecursionError limit & Tail-loop smaller-half stack cap • Advanced Patterns: In-place swap vs comprehension copy & O(N) iterative Quickselect K-th order • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Iterative `O(N)` Expected 3-Way Quickselect (`K`-th Smallest) • Idiomatic DSAP Implementation: PEP-8 type hints, generators, and O(1) dict/set structures • CPython 3.12 Bytecode & Heap Execution & Memory Lifecycle for Quick Sort

Purpose: To understand and apply Quick Sort & Quickselect — Randomized Pivot, 3-Way Partitioning & Python Recursion Depth Guard in real DSA with Python systems — specifically: Und...

What Topic Is Next → Next Topic 3.2: Binary Heap Sort & Python `heapq` — `O(N)` Sift-Down `heapify`, `O(1)` Space Sort & Min-Heap Idioms

3.2 Binary Heap Sort & Python `heapq` — `O(N)` Sift-Down `heapify`, `O(1)` Space Sort & Min-Heap Idioms [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: O(N) bottom-up sift-down heapify & O(N log N) worst-case O(1)-space sort • Core Syntax, CLI & Production APIs: import heapq; heapq.heapify(nums) # O(N) in-place Min-Heap + sift_down Max-Heap Sort • Deep-Dive Mechanics: Heapq is strictly a Min-Heap & -val negation Max-Heap idiom • Advanced Patterns: Tuple tie_breaker_id TypeError guard & Heappushpop faster than push+pop • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Zero-Allocation `O(1)`-Space In-Place Heap Sort • Idiomatic DSAP Implementation: PEP-8 type hints, generators, and O(1) dict/set structures • CPython 3.12 Bytecode & Heap Execution & Memory Lifecycle for Binary Heap Sort

Purpose: To understand and apply Binary Heap Sort & Python `heapq` — `O(N)` Sift-Down `heapify`, `O(1)` Space Sort & Min-Heap Idioms in real DSA with Python systems — specifically...

What Topic Is Next → Module 3 Capstone (Randomized 3-Way Quickselect (`O(N)` `K`-th Element) & `O(1)`-Space In-Place Heap Sort) → Next Topic 4.1: Hashing & CPython `dict` Internals — Compact Open Addressing, Tombstones & `frozenset`/Tuple Keys

Module 3 Hands-On Capstone Project (100 Marks): Randomized 3-Way Quickselect (`O(N)` `K`-th Element) & `O(1)`-Space In-Place Heap Sort — Build an `O(N)` expected-time Quickselect solver handling 100,000 duplicate values without `RecursionError` (iteratively narrowing `(lo, hi)` around the 3-way `(lt, gt)` pivot band), paired...

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

Module 4: CPython Open-Addressing `dict` Hashing & Self-Balancing Red-Black Trees

Explore CPython 3.6+ compact insertion-ordered `dict` internals (open addressing, perturbed probing, tombstones, SipHash DoS randomization) and master the 5 invariants of **Red-Black Trees** (`<= 2 log2(N+1)` height).

Objectives: ✓ Hashing & CPython `dict` Internals — Compact Open Addressing, Tombstones & `frozenset`/Tuple Keys — How Python's `dict` preserves insertion order while cutting memory by ~25% via Raymond Hettinger's compact index array, why open-addressed `del d[k]` writes a dummy **tombstone**, and rolling polynomial Rabin-Karp hashing. • ✓ Red-Black Tree — 5 Color Invariants, Black-Height, Recoloring vs Rotations & `O(1)` Fixup Bound — Understand why Linux `CFS` scheduler (`rbtree`), C++ `std::map`, and Java `TreeMap` choose **Red-Black Trees**: how color flips + at most **2 rotations on insert** (3 on delete) bound tree height to `<= 2 log2(N + 1)`. • ✓ Build, verify, and ship Module Capstone: Open-Addressing Linear/Perturbed Hash Table & Red-Black Tree Invariant Verifier.

4.1 Hashing & CPython `dict` Internals — Compact Open Addressing, Tombstones & `frozenset`/Tuple Keys [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: CPython compact sparse-indices + dense-entries & Open Addressing perturbed probing • Core Syntax, CLI & Production APIs: dict / set / collections.Counter / defaultdict + tuple & frozenset hashable keys • Deep-Dive Mechanics: TOMBSTONE deletion probe preservation & 2/3 load factor resize threshold • Advanced Patterns: SipHash PYTHONHASHSEED DoS defense & Tuple / frozenset canonical anagram key • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Canonical Frequency-Tuple `O(N * K)` Group Anagrams • Idiomatic DSAP Implementation: PEP-8 type hints, generators, and O(1) dict/set structures • CPython 3.12 Bytecode & Heap Execution & Memory Lifecycle for Hashing

Purpose: To understand and apply Hashing & CPython `dict` Internals — Compact Open Addressing, Tombstones & `frozenset`/Tuple Keys in real DSA with Python systems — specifically:...

What Topic Is Next → Next Topic 4.2: Red-Black Tree — 5 Color Invariants, Black-Height, Recoloring vs Rotations & `O(1)` Fixup Bound

4.2 Red-Black Tree — 5 Color Invariants, Black-Height, Recoloring vs Rotations & `O(1)` Fixup Bound [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: 5 Red-Black Tree invariants & No consecutive RED parent-child rule • Core Syntax, CLI & Production APIs: 5 RBT Rules: (1) Red/Black, (2) Black Root, (3) Black NILs, (4) No Red-Red edge, (5) Equal B... • Deep-Dive Mechanics: Uniform root-to-NIL black-height & Height <= 2 log2(N + 1) proof • Advanced Patterns: <= 2 rotations on insert (O(1) structural) & Red uncle recolor vs Black uncle rotate • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Red-Black Tree 5-Invariant & Uniform Black-Height Verifier • Idiomatic DSAP Implementation: PEP-8 type hints, generators, and O(1) dict/set structures • CPython 3.12 Bytecode & Heap Execution & Memory Lifecycle for Red-Black Tree

Purpose: To understand and apply Red-Black Tree — 5 Color Invariants, Black-Height, Recoloring vs Rotations & `O(1)` Fixup Bound in real DSA with Python systems — specifically: Un...

What Topic Is Next → Module 4 Capstone (Open-Addressing Linear/Perturbed Hash Table & Red-Black Tree Invariant Verifier) → Next Topic 5.1: Dynamic Programming — Overlapping Subproblems, Optimal Substructure & State Compression

Module 4 Hands-On Capstone Project (100 Marks): Open-Addressing Linear/Perturbed Hash Table & Red-Black Tree Invariant Verifier — Implement an open-addressed hash map handling collisions and `TOMBSTONE` dummy deletions without breaking probe chains, paired with a Red-Black Tree property verifier computing exact black-h...

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

Module 5: Dynamic Programming Foundations — Optimal Substructure & `@cache` Memoization vs Tabulation

Transition smoothly from exponential `O(2^N)` brute-force recursion to polynomial Dynamic Programming: identify **Overlapping Subproblems + Optimal Substructure**, and master Top-Down `@functools.cache` vs Space-Optimized Bottom-Up Tabulation.

Objectives: ✓ Dynamic Programming — Overlapping Subproblems, Optimal Substructure & State Compression — Understand Richard Bellman's Dynamic Programming: caching shared subproblem DAG vertices so exponential `O(2^N)` trees collapse into `O(|States| * |Transitions|)`, plus `O(1)` rolling-variable compression. • ✓ Tabulation vs Memoization — `@functools.cache` Top-Down vs Iterative Rolling-Row Bottom-Up — Compare Top-Down Memoization (`@cache` lazy DFS over reachable states) against Bottom-Up Tabulation (tight CPU loops, zero recursion stack frames, and instant `O(M)` row-swapping space reduction). • ✓ Build, verify, and ship Module Capstone: Coin Change Minimum Coins & Rolling-State House Robber DP Engine.

5.1 Dynamic Programming — Overlapping Subproblems, Optimal Substructure & State Compression [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: Overlapping Subproblems DAG & Optimal Substructure requirement • Core Syntax, CLI & Production APIs: dp[state] = min/max/sum(transition(prev_state)) + Rolling Scalar Compression • Deep-Dive Mechanics: States x Transition Cost formula & O(1) scalar rolling compression • Advanced Patterns: Longest Simple Path non-DP counterexample & Amount + 1 unreachable sentinel • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Bottom-Up Coin Change + `O(1)`-Space House Robber DP • Idiomatic DSAP Implementation: PEP-8 type hints, generators, and O(1) dict/set structures • CPython 3.12 Bytecode & Heap Execution & Memory Lifecycle for Dynamic Programming

Purpose: To understand and apply Dynamic Programming — Overlapping Subproblems, Optimal Substructure & State Compression in real DSA with Python systems — specifically: Understand...

What Topic Is Next → Next Topic 5.2: Tabulation vs Memoization — `@functools.cache` Top-Down vs Iterative Rolling-Row Bottom-Up

5.2 Tabulation vs Memoization — `@functools.cache` Top-Down vs Iterative Rolling-Row Bottom-Up [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: @functools.cache unbounded memo & Top-Down sparse reachable states • Core Syntax, CLI & Production APIs: from functools import cache (Top-Down) vs prev_row, curr_row iterative loops (Bottom-Up) • Deep-Dive Mechanics: Bottom-Up zero call-stack frames & Rolling-row O(min(N, M)) compression • Advanced Patterns: Fn.cache_clear() multi-test trap & Hashable tuple state keys • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Equivalence of `@cache` Memoization & `O(N)` Rolling-Row Grid Paths • Idiomatic DSAP Implementation: PEP-8 type hints, generators, and O(1) dict/set structures • CPython 3.12 Bytecode & Heap Execution & Memory Lifecycle for Tabulation vs Memoization

Purpose: To understand and apply Tabulation vs Memoization — `@functools.cache` Top-Down vs Iterative Rolling-Row Bottom-Up in real DSA with Python systems — specifically: Compare...

What Topic Is Next → Module 5 Capstone (Coin Change Minimum Coins & Rolling-State House Robber DP Engine) → Next Topic 6.1: 0/1 Knapsack — Reverse-Capacity `O(W)` 1D Compression, Subset Sum & Pseudo-Polynomial Time

Module 5 Hands-On Capstone Project (100 Marks): Coin Change Minimum Coins & Rolling-State House Robber DP Engine — Solve Unbounded Coin Change (`min coins to make amount`, returning `-1` if unreachable) via Bottom-Up Tabulation alongside a space-compressed `O(1)`-memory state-machine DP (`House Robber` /...

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

Module 6: Classic 2D & Space-Compressed DP — `0/1 Knapsack` & `Longest Common Subsequence (LCS)`

Master pseudo-polynomial **0/1 Knapsack** (and why iterating `w` BACKWARDS from `W` down to `wt` compresses space to `O(W)`!) and **Longest Common Subsequence (LCS)** string/diff alignment + backpointer reconstruction.

Objectives: ✓ 0/1 Knapsack — Reverse-Capacity `O(W)` 1D Compression, Subset Sum & Pseudo-Polynomial Time — Maximize value within capacity `W` where each item can be taken `0` or `1` time (no fractions — greedy value/weight ratio FAILS on 0/1 Knapsack!), and master the reverse-loop `O(W)` 1D array trick. • ✓ Longest Common Subsequence (LCS) — 2-String Alignment DP, Backtracking & `O(min(N, M))` Space — Align two sequences `s1` and `s2` via `dp[i][j] = 1 + dp[i-1][j-1]` (match) vs `max(dp[i-1][j], dp[i][j-1])` (mismatch), reconstruct the actual LCS string, and derive Shortest Common Supersequence & Longest Palindromic Subsequence. • ✓ Build, verify, and ship Module Capstone: 1D Space-Optimized 0/1 Knapsack & Git-Style LCS Diff Reconstructor.

6.1 0/1 Knapsack — Reverse-Capacity `O(W)` 1D Compression, Subset Sum & Pseudo-Polynomial Time

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

Subtopics Covered: Reverse loop W -> wt prevents item reuse & Forward loop wt -> W = Unbounded Knapsack • Core Syntax, CLI & Production APIs: for wt, val in items: for c in range(W, wt - 1, -1): dp[c] = max(dp[c], dp[c - wt] + val) • Deep-Dive Mechanics: O(N * W) pseudo-polynomial complexity & Why Greedy value/weight fails on 0/1 • Advanced Patterns: Partition Equal Subset Sum reduction & Target Sum (+/-) subset transform • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: `O(W)` Space 1D 0/1 Knapsack (Backward Capacity Sweep) • Idiomatic DSAP Implementation: PEP-8 type hints, generators, and O(1) dict/set structures • CPython 3.12 Bytecode & Heap Execution & Memory Lifecycle for 0/1 Knapsack

Purpose: To understand and apply 0/1 Knapsack — Reverse-Capacity `O(W)` 1D Compression, Subset Sum & Pseudo-Polynomial Time in real DSA with Python systems — specifically: Maximiz...

What Topic Is Next → Next Topic 6.2: Longest Common Subsequence (LCS) — 2-String Alignment DP, Backtracking & `O(min(N, M))` Space

6.2 Longest Common Subsequence (LCS) — 2-String Alignment DP, Backtracking & `O(min(N, M))` Space

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

Subtopics Covered: Match 1 + diag vs mismatch max(up, left) & Subsequence vs contiguous Substring • Core Syntax, CLI & Production APIs: if a[i] == b[j]: dp[i+1][j+1] = 1 + dp[i][j] else: max(dp[i][j+1], dp[i+1][j]) • Deep-Dive Mechanics: O(N + M) diagonal/arrow reconstruction & O(min(N, M)) rolling 2-row length • Advanced Patterns: LCS(s, reversed(s)) palindrome bridge & Diff utility / Edit Distance cousin • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: LCS Length + Exact Subsequence String Reconstruction • Idiomatic DSAP Implementation: PEP-8 type hints, generators, and O(1) dict/set structures • CPython 3.12 Bytecode & Heap Execution & Memory Lifecycle for Longest Common Subsequence (LCS)

Purpose: To understand and apply Longest Common Subsequence (LCS) — 2-String Alignment DP, Backtracking & `O(min(N, M))` Space in real DSA with Python systems — specifically: Alig...

What Topic Is Next → Module 6 Capstone (1D Space-Optimized 0/1 Knapsack & Git-Style LCS Diff Reconstructor) → Next Topic 7.1: Longest Increasing Subsequence (LIS) — `O(N^2)` DP vs `O(N log N)` Patience `tails` + `bisect_left`

Module 6 Hands-On Capstone Project (100 Marks): 1D Space-Optimized 0/1 Knapsack & Git-Style LCS Diff Reconstructor — Implement a 1D `O(W)`-space 0/1 Knapsack solver alongside Subset-Sum Partitioning, plus an LCS DP engine that reconstructs the exact longest common subsequence string (basis of `git diff`) v...

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

Module 7: `O(N log N)` Longest Increasing Subsequence (LIS) & Greedy Exchange Proofs

Accelerate Longest Increasing Subsequence from `O(N^2)` DP to `O(N log N)` **Patience Sorting + `bisect_left`**, and master **Greedy Algorithms** (Greedy Choice Property, Exchange Argument, Interval Scheduling, Jump Game).

Objectives: ✓ Longest Increasing Subsequence (LIS) — `O(N^2)` DP vs `O(N log N)` Patience `tails` + `bisect_left` — Maintain `tails[k]` = smallest possible tail value of all increasing subsequences of length `k + 1`: because `tails` stays strictly sorted, `bisect_left(tails, x)` updates or extends LIS in `O(log N)` per element!. • ✓ Greedy Algorithms — Greedy Choice Property, Exchange Argument, Interval Scheduling & Fractional Knapsack — Make locally irrevocable optimal choices backed by **Exchange Argument** proofs: Activity Selection (sort by earliest finish!), Minimum Meeting Rooms (min-heap / sweep-line), Jump Game, and Gas Station. • ✓ Build, verify, and ship Module Capstone: `O(N log N)` 2D Russian-Doll Envelope LIS & Earliest-Finish Interval Scheduler.

7.1 Longest Increasing Subsequence (LIS) — `O(N^2)` DP vs `O(N log N)` Patience `tails` + `bisect_left`

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

Subtopics Covered: Tails[k] minimal-end invariant & O(N log N) bisect_left Patience LIS • Core Syntax, CLI & Production APIs: idx = bisect_left(tails, x); if idx == len(tails): tails.append(x) else: tails[idx] = x • Deep-Dive Mechanics: Tails != actual subsequence warning & Bisect_left (<) vs bisect_right (<=) • Advanced Patterns: Russian Doll (w ASC, -h DESC) trick & Dilworth theorem minimum chains • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: `O(N log N)` Strictly Increasing Subsequence Length (`bisect_left`) • Idiomatic DSAP Implementation: PEP-8 type hints, generators, and O(1) dict/set structures • CPython 3.12 Bytecode & Heap Execution & Memory Lifecycle for Longest Increasing Subsequence (LIS)

Purpose: To understand and apply Longest Increasing Subsequence (LIS) — `O(N^2)` DP vs `O(N log N)` Patience `tails` + `bisect_left` in real DSA with Python systems — specifically...

What Topic Is Next → Next Topic 7.2: Greedy Algorithms — Greedy Choice Property, Exchange Argument, Interval Scheduling & Fractional Knapsack

7.2 Greedy Algorithms — Greedy Choice Property, Exchange Argument, Interval Scheduling & Fractional Knapsack

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

Subtopics Covered: Greedy Choice Property (no lookback) & Exchange Argument optimality proof • Core Syntax, CLI & Production APIs: intervals.sort(key=lambda iv: iv[1]) # Earliest Finish Time Greedy Choice • Deep-Dive Mechanics: Activity Selection sort by finish time & Fractional Knapsack ratio sort • Advanced Patterns: Sweep-Line / Min-Heap Meeting Rooms & O(N) Jump Game farthest-reach greedy • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Optimal Earliest-Finish-Time Activity Selection (`O(N log N)`) • Idiomatic DSAP Implementation: PEP-8 type hints, generators, and O(1) dict/set structures • CPython 3.12 Bytecode & Heap Execution & Memory Lifecycle for Greedy Algorithms

Purpose: To understand and apply Greedy Algorithms — Greedy Choice Property, Exchange Argument, Interval Scheduling & Fractional Knapsack in real DSA with Python systems — specifi...

What Topic Is Next → Module 7 Capstone (`O(N log N)` 2D Russian-Doll Envelope LIS & Earliest-Finish Interval Scheduler) → Next Topic 8.1: Huffman Coding — Optimal Prefix-Free Binary Tree, `heapq` Merging & Compression Bit Ratio

Module 7 Hands-On Capstone Project (100 Marks): `O(N log N)` 2D Russian-Doll Envelope LIS & Earliest-Finish Interval Scheduler — Solve 2D Envelope Nesting (`w1 < w2` and `h1 < h2`) in `O(N log N)` by sorting `(width ASC, -height DESC)` + Patience LIS on heights, paired with an optimal Greedy Non-Overlapping Activity S...

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

Module 8: Prefix-Free Greedy `Huffman Coding` & State-Space `Backtracking` Pruning

Build optimal lossless prefix-free binary codes in `O(K log K)` using a Min-Heap **Huffman Tree**, and master exhaustive search with aggressive constraint pruning via the **Choose -> Explore -> Unchoose** Backtracking template.

Objectives: ✓ Huffman Coding — Optimal Prefix-Free Binary Tree, `heapq` Merging & Compression Bit Ratio — Assign shorter bit codes to frequent symbols and longer codes to rare symbols by repeatedly merging the two lightest subtrees in a Min-Heap (`O(K log K)` over `K` unique symbols). • ✓ Backtracking — State-Space Tree, `Choose -> Explore -> Unchoose`, Pruning & Duplicate Skipping — Explore combinatorial search spaces systematically by mutating a single shared `path` list in place (`path.append(x)` -> recurse -> `path.pop()`), sorting candidates for early `break` pruning, and skipping duplicate branches. • ✓ Build, verify, and ship Module Capstone: Lossless Huffman Bit-Encoder/Decoder & Pruned Combination-Sum Backtracker.

8.1 Huffman Coding — Optimal Prefix-Free Binary Tree, `heapq` Merging & Compression Bit Ratio

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

Subtopics Covered: Prefix-Free leaf-only invariant & Min-Heap two-lightest greedy merge • Core Syntax, CLI & Production APIs: Counter(text) + heapq.heappop() twice -> merge (f1 + f2) -> leaf bit-code DFS • Deep-Dive Mechanics: O(N + K log K) build complexity & Single-unique-char "0" edge case • Advanced Patterns: Weighted external path length sum(f*d) & GZIP DEFLATE + JPEG entropy stage • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Huffman Total Compressed Bit-Length Calculator (`heapq`) • Idiomatic DSAP Implementation: PEP-8 type hints, generators, and O(1) dict/set structures • CPython 3.12 Bytecode & Heap Execution & Memory Lifecycle for Huffman Coding

Purpose: To understand and apply Huffman Coding — Optimal Prefix-Free Binary Tree, `heapq` Merging & Compression Bit Ratio in real DSA with Python systems — specifically: Assign s...

What Topic Is Next → Next Topic 8.2: Backtracking — State-Space Tree, `Choose -> Explore -> Unchoose`, Pruning & Duplicate Skipping

8.2 Backtracking — State-Space Tree, `Choose -> Explore -> Unchoose`, Pruning & Duplicate Skipping

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

Subtopics Covered: Choose -> Explore -> Unchoose (pop) & Ans.append(path.copy()) snapshot rule • Core Syntax, CLI & Production APIs: path.append(choice); dfs(next_idx, rem - choice); path.pop() + ans.append(path.copy()) • Deep-Dive Mechanics: Sorted candidate early break pruning & I > start duplicate branch skipping • Advanced Patterns: O(depth) single-path auxiliary space & Backtracking vs Branch-and-Bound • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Sorted Early-Pruned Combination Sum Backtracker • Idiomatic DSAP Implementation: PEP-8 type hints, generators, and O(1) dict/set structures • CPython 3.12 Bytecode & Heap Execution & Memory Lifecycle for Backtracking

Purpose: To understand and apply Backtracking — State-Space Tree, `Choose -> Explore -> Unchoose`, Pruning & Duplicate Skipping in real DSA with Python systems — specifically: Exp...

What Topic Is Next → Module 8 Capstone (Lossless Huffman Bit-Encoder/Decoder & Pruned Combination-Sum Backtracker) → Next Topic 9.1: N-Queens Problem — Row-by-Row Placement & `O(1)` Diagonal Invariants (`r - c` and `r + c`)

Module 8 Hands-On Capstone Project (100 Marks): Lossless Huffman Bit-Encoder/Decoder & Pruned Combination-Sum Backtracker — Implement a complete Huffman encoder + tree-walking bit decoder handling both multi-symbol strings and single-unique-character edge cases (`"AAAAA"`), paired with a sorted early-break Combin...

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

Module 9: Constraint-Satisfaction `N-Queens` (`O(1)` Diagonal Sets) & Weighted `Dijkstra's Algorithm`

Slash N-Queens conflict checks from `O(N)` board scans to `O(1)` diagonal invariants (`r - c` and `r + c`) and compute single-source weighted shortest paths in `O((V + E) log V)` with lazy-heap **Dijkstra's Algorithm**.

Objectives: ✓ N-Queens Problem — Row-by-Row Placement & `O(1)` Diagonal Invariants (`r - c` and `r + c`) — Place one Queen per row (`N^N -> N!` immediately) and prune illegal columns/diagonals in `O(1)` using two mathematical invariants: cells on the same `\` main diagonal share `r - c`, and cells on the same `/` anti-diagonal share `r + c`!. • ✓ Dijkstra's Algorithm — Non-Negative Edge Relaxation, Lazy `heapq` & Stale-Entry Guard — Compute single-source shortest paths across non-negative weighted graphs (`w >= 0`) in `O((V + E) log V)` using Python `heapq`, lazy duplicate entries (replacing complex `decrease-key`), and `if d > dist[u]: continue`. • ✓ Build, verify, and ship Module Capstone: Bitmask-Accelerated `N-Queens` Solver & Path-Reconstructing `Dijkstra` Router.

9.1 N-Queens Problem — Row-by-Row Placement & `O(1)` Diagonal Invariants (`r - c` and `r + c`)

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

Subtopics Covered: One Queen per row N! reduction & Main diagonal r - c constant • Core Syntax, CLI & Production APIs: cols: set[int], main_diag (r - c), anti_diag (r + c) # or bitmasks (cols | d1 | d2) • Deep-Dive Mechanics: Anti-diagonal r + c constant & O(1) set / bitmask attack check • Advanced Patterns: Symmetric unchoose set.remove() & 8-Queens = 92 canonical benchmark • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: `O(1)`-Conflict Set-Pruned N-Queens Solution Counter • Idiomatic DSAP Implementation: PEP-8 type hints, generators, and O(1) dict/set structures • CPython 3.12 Bytecode & Heap Execution & Memory Lifecycle for N-Queens Problem

Purpose: To understand and apply N-Queens Problem — Row-by-Row Placement & `O(1)` Diagonal Invariants (`r - c` and `r + c`) in real DSA with Python systems — specifically: Place o...

What Topic Is Next → Next Topic 9.2: Dijkstra's Algorithm — Non-Negative Edge Relaxation, Lazy `heapq` & Stale-Entry Guard

9.2 Dijkstra's Algorithm — Non-Negative Edge Relaxation, Lazy `heapq` & Stale-Entry Guard [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: W >= 0 non-negative weight precondition & If d > dist[u]: continue lazy skip • Core Syntax, CLI & Production APIs: heapq.heappush(pq, (nd, v)) + if d > dist[u]: continue # Mandatory lazy-heap guard! • Deep-Dive Mechanics: Optimal ONLY at heappop (not push) & O((V + E) log V) heapq complexity • Advanced Patterns: Parent[v] = u shortest-path trace & Why negative edges break greedy lock • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Lazy `heapq` Dijkstra Single-Source Shortest Paths • Idiomatic DSAP Implementation: PEP-8 type hints, generators, and O(1) dict/set structures • CPython 3.12 Bytecode & Heap Execution & Memory Lifecycle for Dijkstra's Algorithm

Purpose: To understand and apply Dijkstra's Algorithm — Non-Negative Edge Relaxation, Lazy `heapq` & Stale-Entry Guard in real DSA with Python systems — specifically: Compute sing...

What Topic Is Next → Module 9 Capstone (Bitmask-Accelerated `N-Queens` Solver & Path-Reconstructing `Dijkstra` Router) → Next Topic 10.1: Bellman-Ford Algorithm — `V - 1` Edge Relaxations, Negative Cycle Detection & `K`-Stop Flights

Module 9 Hands-On Capstone Project (100 Marks): Bitmask-Accelerated `N-Queens` Solver & Path-Reconstructing `Dijkstra` Router — Implement an `O(1)`-conflict N-Queens solver counting distinct solutions for `N = 1..10`, paired with a `heapq` Dijkstra router returning both minimum latency and the exact hop-by-hop shorte...

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

Module 10: Negative-Weight `Bellman-Ford` Cycle Detection & Linear `O(N + M)` `KMP` String Matching

Handle negative edge weights, currency-arbitrage negative cycles, and `<= K`-stop flight constraints with **Bellman-Ford (`O(V * E)`)**, and search substrings in strict worst-case `O(N + M)` linear time using **Knuth-Morris-Pratt (KMP)** `LPS` failure tables.

Objectives: ✓ Bellman-Ford Algorithm — `V - 1` Edge Relaxations, Negative Cycle Detection & `K`-Stop Flights — Why any simple shortest path in a `V`-vertex graph has at most `V - 1` edges: relax all edges `V - 1` rounds in `O(V * E)` to support negative weights safely, and detect negative cycles if any edge still relaxes on round `V`!. • ✓ Knuth-Morris-Pratt (KMP) — `LPS` Prefix-Suffix Automaton & Worst-Case `O(N + M)` Pattern Search — Eliminate naive `O(N * M)` string-matching backtracking (e.g., `"AAAAAAAAB"` in `"AAAA...AAAA"`) by precomputing `lps[k]` = length of the longest **proper** prefix of `pat[0..k]` that is also a suffix of `pat[0..k]`. • ✓ Build, verify, and ship Module Capstone: Negative-Cycle Arbitrage Detector (`Bellman-Ford`) & `O(N + M)` Multi-Match `KMP` Scanner.

10.1 Bellman-Ford Algorithm — `V - 1` Edge Relaxations, Negative Cycle Detection & `K`-Stop Flights [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: Simple path <= V - 1 edges theorem & Handles negative edge weights safely • Core Syntax, CLI & Production APIs: for _ in range(V - 1): relax all (u, v, w) + V-th pass dist[u] + w < dist[v] cycle check • Deep-Dive Mechanics: V-th pass negative-cycle detection & Early-stop when updated == False • Advanced Patterns: Dist.copy() snapshot for K-stop limit & FX log(-rate) arbitrage cycle detection • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Bellman-Ford Shortest Paths + Negative-Cycle Detector • Idiomatic DSAP Implementation: PEP-8 type hints, generators, and O(1) dict/set structures • CPython 3.12 Bytecode & Heap Execution & Memory Lifecycle for Bellman-Ford Algorithm

Purpose: To understand and apply Bellman-Ford Algorithm — `V - 1` Edge Relaxations, Negative Cycle Detection & `K`-Stop Flights in real DSA with Python systems — specifically: Why...

What Topic Is Next → Next Topic 10.2: Knuth-Morris-Pratt (KMP) — `LPS` Prefix-Suffix Automaton & Worst-Case `O(N + M)` Pattern Search

10.2 Knuth-Morris-Pratt (KMP) — `LPS` Prefix-Suffix Automaton & Worst-Case `O(N + M)` Pattern Search [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: Longest Proper Prefix = Suffix (LPS) & Text pointer i NEVER backs up • Core Syntax, CLI & Production APIs: lps = build_lps(pattern) # O(M) + KMP scan j = lps[j - 1] # O(N + M) worst-case • Deep-Dive Mechanics: Amortized potential proof <= 2N ops & Worst-case O(N + M) vs naive O(N * M) • Advanced Patterns: J = lps[j - 1] overlapping matches & Periodic string check n % (n - lps[-1]) • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Linear `O(N + M)` KMP Overlapping Pattern Matcher • Idiomatic DSAP Implementation: PEP-8 type hints, generators, and O(1) dict/set structures • CPython 3.12 Bytecode & Heap Execution & Memory Lifecycle for Knuth-Morris-Pratt (KMP)

Purpose: To understand and apply Knuth-Morris-Pratt (KMP) — `LPS` Prefix-Suffix Automaton & Worst-Case `O(N + M)` Pattern Search in real DSA with Python systems — specifically: El...

What Topic Is Next → Module 10 Capstone (Negative-Cycle Arbitrage Detector (`Bellman-Ford`) & `O(N + M)` Multi-Match `KMP` Scanner) → Next Topic 11.1: collections.deque BFS, Iterative DFS & Recursion-Limit Safety

Module 10 Hands-On Capstone Project (100 Marks): Negative-Cycle Arbitrage Detector (`Bellman-Ford`) & `O(N + M)` Multi-Match `KMP` Scanner — Build a Bellman-Ford shortest-path & negative-cycle detector supporting bounded `K`-edge relaxations via snapshot arrays, paired with a full Knuth-Morris-Pratt (`LPS` builder + streaming mat...

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

Advanced Graph Algorithms in Python: deque, heapq, Topo Sort & DSU

Production-grade DSA with Python Training engineering module covering Advanced Graph Algorithms in Python: deque, heapq, Topo Sort & DSU with hands-on labs and capstone build.

Objectives: ✓ collections.deque BFS, Iterative DFS & Recursion-Limit Safety — Purpose: Avoid Python RecursionError crashes on 10^5-node graphs using explicit stacks/deques. • ✓ Dijkstra with heapq, 0-1 BFS & Shortest Path with State Tuples — Purpose: Solve constrained shortest-path OA problems cleanly using tuple priority queues. • ✓ Topological Sort (Kahn's Algorithm) & Disjoint Set Union (Union-Find) — Purpose: Detect dependency cycles and merge dynamic connected components in O(alpha(N)). • ✓ Multi-Source Grid BFS & Implicit Word/State-Space Graphs — Purpose: Crack shortest transformation and grid wave-propagation problems rapidly.

11.1 collections.deque BFS, Iterative DFS & Recursion-Limit Safety [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of Iterative DFS • Integration with Dijkstra with heapq, 0 • Live Debugging, Telemetry & Technical Interview Trade • What Is collections.deque BFS, Iterative DFS & Recursion • In DSA with Python Training, collections.deque BFS, Iterative DFS & Recursion • Even if you come from a non • Runnable Code Implementation (PYTHON)

Purpose: To understand and apply collections.deque BFS, Iterative DFS & Recursion-Limit Safety in real DSA with Python systems — specifically: Avoid Python RecursionError crashes...

What Topic Is Next → Next Topic 11.2: Dijkstra with heapq, 0-1 BFS & Shortest Path with State Tuples

11.2 Dijkstra with heapq, 0-1 BFS & Shortest Path with State Tuples [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of 0 • Integration with Topological Sort (Kahn's Algorithm) & Disjoint Set Union (Union • Live Debugging, Telemetry & Technical Interview Trade • What Is Dijkstra with heapq, 0 • In DSA with Python Training, Dijkstra with heapq, 0 • Even if you come from a non • Runnable Code Implementation (PYTHON)

Purpose: To understand and apply Dijkstra with heapq, 0-1 BFS & Shortest Path with State Tuples in real DSA with Python systems — specifically: Solve constrained shortest-path OA...

What Topic Is Next → Next Topic 11.3: Topological Sort (Kahn's Algorithm) & Disjoint Set Union (Union-Find)

11.3 Topological Sort (Kahn's Algorithm) & Disjoint Set Union (Union-Find) [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of Disjoint Set Union (Union • Integration with Multi • Live Debugging, Telemetry & Technical Interview Trade • What Is Topological Sort (Kahn's Algorithm) & Disjoint Set Union (Union • In DSA with Python Training, Topological Sort (Kahn's Algorithm) & Disjoint Set Union (Union • Even if you come from a non • Runnable Code Implementation (PYTHON)

Purpose: To understand and apply Topological Sort (Kahn's Algorithm) & Disjoint Set Union (Union-Find) in real DSA with Python systems — specifically: Detect dependency cycles and...

What Topic Is Next → Next Topic 11.4: Multi-Source Grid BFS & Implicit Word/State-Space Graphs

11.4 Multi-Source Grid BFS & Implicit Word/State-Space Graphs [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of Implicit Word/State • Integration with collections.deque BFS, Iterative DFS & Recursion • Live Debugging, Telemetry & Technical Interview Trade • What Is Multi • In DSA with Python Training, Multi • Even if you come from a non • Runnable Code Implementation (PYTHON)

Purpose: To understand and apply Multi-Source Grid BFS & Implicit Word/State-Space Graphs in real DSA with Python systems — specifically: Crack shortest transformation and grid wa...

What Topic Is Next → Module 11 Capstone (High-Speed Network Routing & Dependency Graph Solver in Python) → Next Topic 12.1: functools.cache Top-Down -> Iterative Bottom-Up -> 1D Rolling Arrays

Module 11 Hands-On Capstone Project (100 Marks): High-Speed Network Routing & Dependency Graph Solver in Python — Design, implement, test, and document 'High-Speed Network Routing & Dependency Graph Solver in Python' applying every production concept from Module 11 (Advanced Graph Algorithms in Python:...

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

Dynamic Programming Mastery in Python: @cache, Knapsack, LIS & Trees

Production-grade DSA with Python Training engineering module covering Dynamic Programming Mastery in Python: @cache, Knapsack, LIS & Trees with hands-on labs and capstone build.

Objectives: ✓ functools.cache Top-Down -> Iterative Bottom-Up -> 1D Rolling Arrays — Purpose: Prototype recurrence relations in 3 minutes and optimize memory to pass strict OA limits. • ✓ 0/1 & Unbounded Knapsack, Partition & Target-Sum Patterns — Purpose: Identify state variables and base cases immediately under timed test pressure. • ✓ LCS, Edit Distance, bisect_left O(N log N) LIS & Interval DP — Purpose: Combine Python bisect binary search with DP for optimal subsequence runtimes. • ✓ Tree DP, Bitmask DP (N <= 20) & Stock Cooldown State Machines — Purpose: Conquer hard-tier product-company OA problems with clean bitwise transitions.

12.1 functools.cache Top-Down -> Iterative Bottom-Up -> 1D Rolling Arrays [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, Partition & Target • Live Debugging, Telemetry & Technical Interview Trade • What Is functools.cache Top • In DSA with Python Training, functools.cache Top • Even if you come from a non • Runnable Code Implementation (PYTHON)

Purpose: To understand and apply functools.cache Top-Down -> Iterative Bottom-Up -> 1D Rolling Arrays in real DSA with Python systems — specifically: Prototype recurrence relation...

What Topic Is Next → Next Topic 12.2: 0/1 & Unbounded Knapsack, Partition & Target-Sum Patterns

12.2 0/1 & Unbounded Knapsack, Partition & Target-Sum Patterns [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, bisect_left O(N log N) LIS & Interval DP • Live Debugging, Telemetry & Technical Interview Trade • What Is 0/1 & Unbounded Knapsack, Partition & Target • In DSA with Python Training, 0/1 & Unbounded Knapsack, Partition & Target • Even if you come from a non • Runnable Code Implementation (PYTHON)

Purpose: To understand and apply 0/1 & Unbounded Knapsack, Partition & Target-Sum Patterns in real DSA with Python systems — specifically: Identify state variables and base cases...

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

12.3 LCS, Edit Distance, bisect_left O(N log N) LIS & Interval 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 Tree DP, Bitmask DP (N <= 20) & Stock Cooldown State Machines • Live Debugging, Telemetry & Technical Interview Trade • Integration with Tree DP, Bitmask DP (N • What Is LCS, Edit Distance, bisect_left O(N log N) LIS & Interval DP &

Why Industry Teams Use It • In DSA with Python Training, LCS, Edit Distance, bisect_left O(N log N) LIS & Interval DP exists for one clear real • Even if you come from a non

Purpose: To understand and apply LCS, Edit Distance, bisect_left O(N log N) LIS & Interval DP in real DSA with Python systems — specifically: Combine Python bisect binary search w...

What Topic Is Next → Next Topic 12.4: Tree DP, Bitmask DP (N <= 20) & Stock Cooldown State Machines

12.4 Tree DP, Bitmask DP (N <= 20) & Stock Cooldown State Machines

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

Subtopics Covered: Plain • Production Implementation of Bitmask DP (N <= 20) • Integration with functools.cache Top • Live Debugging, Telemetry & Technical Interview Trade • Production Implementation of Bitmask DP (N • What Is Tree DP, Bitmask DP (N • In DSA with Python Training, Tree DP, Bitmask DP (N • Runnable Code Implementation (PYTHON)

Purpose: To understand and apply Tree DP, Bitmask DP (N <= 20) & Stock Cooldown State Machines in real DSA with Python systems — specifically: Conquer hard-tier product-company OA...

What Topic Is Next → Module 12 Capstone (Sequence, Knapsack & State-Machine DP Toolkit in Python) → Next Topic 13.1: Nested-Dict / Node Tries & Word-Search Pruning

Module 12 Hands-On Capstone Project (100 Marks): Sequence, Knapsack & State-Machine DP Toolkit in Python — Design, implement, test, and document 'Sequence, Knapsack & State-Machine DP Toolkit in Python' applying every production concept from Module 12 (Dynamic Programming Mastery in Python: @cach...

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

Tries, Segment/Fenwick Trees, String Hashing & Sweep-Line Intervals

Production-grade DSA with Python Training engineering module covering Tries, Segment/Fenwick Trees, String Hashing & Sweep-Line Intervals with hands-on labs and capstone build.

Objectives: ✓ Nested-Dict / Node Tries & Word-Search Pruning — Purpose: Implement fast prefix autocomplete and prune exponential board backtracking searches. • ✓ Fenwick Tree (Binary Indexed Tree) & Iterative Segment Tree — Purpose: Support O(log N) dynamic prefix sums and range queries without TLE in Python. • ✓ Polynomial Rolling Hash, KMP & Z-Array Linear String Matching — Purpose: Find repeated substrings and palindromes in guaranteed O(N) linear time. • ✓ Interval Sweep-Line, Prefix Difference Arrays & bisect Event Queries — Purpose: Solve calendar booking, CPU scheduling, and overlapping skylines cleanly.

13.1 Nested-Dict / Node Tries & Word-Search Pruning

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

Subtopics Covered: Plain • Production Implementation of Word • Integration with Fenwick Tree (Binary Indexed Tree) & Iterative Segment Tree • Live Debugging, Telemetry & Technical Interview Trade • What Is Nested • In DSA with Python Training, Nested • Even if you come from a non • Runnable Code Implementation (PYTHON)

Purpose: To understand and apply Nested-Dict / Node Tries & Word-Search Pruning in real DSA with Python systems — specifically: Implement fast prefix autocomplete and prune expone...

What Topic Is Next → Next Topic 13.2: Fenwick Tree (Binary Indexed Tree) & Iterative Segment Tree

13.2 Fenwick Tree (Binary Indexed Tree) & Iterative Segment Tree

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

Subtopics Covered: Plain • Production Implementation of Iterative Segment Tree • Integration with Polynomial Rolling Hash, KMP & Z • Live Debugging, Telemetry & Technical Interview Trade • What Is Fenwick Tree (Binary Indexed Tree) & Iterative Segment Tree & Why Industry Teams Use It • In DSA with Python Training, Fenwick Tree (Binary Indexed Tree) & Iterative Segment Tree exists for one clear real • Even if you come from a non • Runnable Code Implementation (PYTHON)

Purpose: To understand and apply Fenwick Tree (Binary Indexed Tree) & Iterative Segment Tree in real DSA with Python systems — specifically: Support O(log N) dynamic prefix sums a...

What Topic Is Next → Next Topic 13.3: Polynomial Rolling Hash, KMP & Z-Array Linear String Matching

13.3 Polynomial Rolling Hash, KMP & Z-Array Linear String Matching

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

Subtopics Covered: Plain • Production Implementation of KMP • Integration with Interval Sweep • Live Debugging, Telemetry & Technical Interview Trade • What Is Polynomial Rolling Hash, KMP & Z • In DSA with Python Training, Polynomial Rolling Hash, KMP & Z • Even if you come from a non • Runnable Code Implementation (PYTHON)

Purpose: To understand and apply Polynomial Rolling Hash, KMP & Z-Array Linear String Matching in real DSA with Python systems — specifically: Find repeated substrings and palindr...

What Topic Is Next → Next Topic 13.4: Interval Sweep-Line, Prefix Difference Arrays & bisect Event Queries

13.4 Interval Sweep-Line, Prefix Difference Arrays & bisect Event Queries

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

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

Purpose: To understand and apply Interval Sweep-Line, Prefix Difference Arrays & bisect Event Queries in real DSA with Python systems — specifically: Solve calendar booking, CPU s...

What Topic Is Next → Module 13 Capstone (Prefix Search & Streaming Range-Query Engine in Python) → Next Topic 14.1: Monotonic Stack & collections.deque Sliding Window Maximum

Module 13 Hands-On Capstone Project (100 Marks): Prefix Search & Streaming Range-Query Engine in Python — Design, implement, test, and document 'Prefix Search & Streaming Range-Query Engine in Python' applying every production concept from Module 13 (Tries, Segment/Fenwick Trees, String Hashing...

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

Monotonic Deques, Multi-Heap Streams & Python LLD Machine Coding

Production-grade DSA with Python Training engineering module covering Monotonic Deques, Multi-Heap Streams & Python LLD Machine Coding with hands-on labs and capstone build.

Objectives: ✓ Monotonic Stack & collections.deque Sliding Window Maximum — Purpose: Achieve strict O(N) amortized window scans where naive slicing O(N*K) times out. • ✓ Two-Heap Streaming Median, K-Way Merge & Counter/heapq Top-K — Purpose: Maintain real-time percentiles and merge sorted log streams effortlessly. • ✓ OrderedDict / Custom Doubly-Linked O(1) LRU/LFU & Time-Based KV Store — Purpose: Build clean, extensible classes for 60-minute machine-coding interviews. • ✓ Python Constant-Factor Speedups & Interview Verbalization Framework — Purpose: Avoid list.pop(0) O(N) traps, prove Big-O clearly, and test corner cases first.

14.1 Monotonic Stack & collections.deque Sliding Window Maximum [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of collections.deque Sliding Window Maximum • Integration with Two • Live Debugging, Telemetry & Technical Interview Trade • What Is Monotonic Stack & collections.deque Sliding Window Maximum & Why Industry Teams Use It • In DSA with Python Training, Monotonic Stack & collections.deque Sliding Window Maximum exists for one clear real • Even if you come from a non • Runnable Code Implementation (PYTHON)

Purpose: To understand and apply Monotonic Stack & collections.deque Sliding Window Maximum in real DSA with Python systems — specifically: Achieve strict O(N) amortized window sc...

What Topic Is Next → Next Topic 14.2: Two-Heap Streaming Median, K-Way Merge & Counter/heapq Top-K

14.2 Two-Heap Streaming Median, K-Way Merge & Counter/heapq Top-K [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of K • Integration with OrderedDict / Custom Doubly • Live Debugging, Telemetry & Technical Interview Trade • What Is Two • In DSA with Python Training, Two • Even if you come from a non • Runnable Code Implementation (PYTHON)

Purpose: To understand and apply Two-Heap Streaming Median, K-Way Merge & Counter/heapq Top-K in real DSA with Python systems — specifically: Maintain real-time percentiles and me...

What Topic Is Next → Next Topic 14.3: OrderedDict / Custom Doubly-Linked O(1) LRU/LFU & Time-Based KV Store

14.3 OrderedDict / Custom Doubly-Linked O(1) LRU/LFU & Time-Based KV Store [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of Time • Integration with Python Constant • Live Debugging, Telemetry & Technical Interview Trade • What Is OrderedDict / Custom Doubly • In DSA with Python Training, OrderedDict / Custom Doubly • Even if you come from a non • Runnable Code Implementation (PYTHON)

Purpose: To understand and apply OrderedDict / Custom Doubly-Linked O(1) LRU/LFU & Time-Based KV Store in real DSA with Python systems — specifically: Build clean, extensible clas...

What Topic Is Next → Next Topic 14.4: Python Constant-Factor Speedups & Interview Verbalization Framework

14.4 Python Constant-Factor Speedups & Interview Verbalization Framework [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of Interview Verbalization Framework • Integration with Monotonic Stack & collections.deque Sliding Window Maximum • Live Debugging, Telemetry & Technical Interview Trade • What Is Python Constant • In DSA with Python Training, Python Constant • Even if you come from a non • Runnable Code Implementation (PYTHON)

Purpose: To understand and apply Python Constant-Factor Speedups & Interview Verbalization Framework in real DSA with Python systems — specifically: Avoid list.pop(0) O(N) traps,...

What Topic Is Next → Module 14 Capstone (In-Memory LRU Cache, Rate Limiter & Job Scheduler in Python) → Next Topic 15.1: Trie Prefix Trees, Rolling Hash / KMP & Counter/Deque Sliding-Window Idioms

Module 14 Hands-On Capstone Project (100 Marks): In-Memory LRU Cache, Rate Limiter & Job Scheduler in Python — Design, implement, test, and document 'In-Memory LRU Cache, Rate Limiter & Job Scheduler in Python' applying every production concept from Module 14 (Monotonic Deques, Multi-Heap Streams & P...

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

Advanced Pythonic Graph, Trie, Union-Find & Range-Query Algorithms

Self-explanatory list-and-bullet production mastery module covering Advanced Pythonic Graph, Trie, Union-Find & Range-Query Algorithms in DSA with Python Training, paired with hands-on labs, capstone repository, Topic Mock Tests, and Spoken MockAttempt AI Viva.

Objectives: ✓ Trie Prefix Trees, Rolling Hash / KMP & Counter/Deque Sliding-Window Idioms — Purpose: Write concise, bug-free Python string and stream algorithms using collections.deque and defaultdict. • ✓ Union-Find (Rank + Path Compression), Shortest Paths (0-1 BFS, Dijkstra) & DAG DP — Purpose: Solve multi-source grid, network latency, and scheduling dependency problems optimally. • ✓ Binary Search on Answer, Monotonic Stacks & Bitmask Dynamic Programming — Purpose: Crack hard optimization and subset-state problems using functools.cache and clean state transitions. • ✓ Build, verify, and document 'High-Speed Python Autocomplete Trie, Union-Find & Fenwick Tree Library' with clean Git commits and edge-case protection. • ✓ Clear 100% of Topic Timed Mock Tests and Spoken Voice + MockAttempt AI Technical Interviews.

15.1 Trie Prefix Trees, Rolling Hash / KMP & Counter/Deque Sliding-Window Idioms [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Security Hygiene & Data Safety Rules for Trie Prefix Trees, Rolling Hash / KMP & Counter/Deque Sliding • Performance Optimization, Complexity & Latency Impact of Trie Prefix Trees, Rolling Hash / KMP & Counter/Deque Slid... • Live Debugging, Stack • Runnable Code Implementation (PYTHON) • Hands • Idiomatic DSAP Implementation: PEP-8 type hints, generators, and O(1) dict/set structures • CPython 3.12 Bytecode & Heap Execution & Memory Lifecycle for Trie

Prefix Trees • Production Hardening & Security: input sanitization, safe deserialization & exception hierarchy

Purpose: To understand and apply Trie Prefix Trees, Rolling Hash / KMP & Counter/Deque Sliding-Window Idioms in real DSA with Python systems — specifically: Write concise, bug-fre...

What Topic Is Next → Next Topic 15.2: Union-Find (Rank + Path Compression), Shortest Paths (0-1 BFS, Dijkstra) & DAG DP

15.2 Union-Find (Rank + Path Compression), Shortest Paths (0-1 BFS, Dijkstra) & DAG DP [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Security Hygiene & Data Safety Rules for Union • Performance Optimization, Complexity & Latency Impact of Union • Live Debugging, Stack • Runnable Code Implementation (PYTHON) • Hands • Idiomatic DSAP Implementation: PEP-8 type hints, generators, and O(1) dict/set structures • CPython 3.12 Bytecode & Heap Execution & Memory Lifecycle for Union-Find (Rank + Path Compression) • Production Hardening & Security: input sanitization, safe deserialization & exception hierarchy

Purpose: To understand and apply Union-Find (Rank + Path Compression), Shortest Paths (0-1 BFS, Dijkstra) & DAG DP in real DSA with Python systems — specifically: Solve multi-sour...

What Topic Is Next → Next Topic 15.3: Binary Search on Answer, Monotonic Stacks & Bitmask Dynamic Programming

15.3 Binary Search on Answer, Monotonic Stacks & Bitmask Dynamic Programming [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Security Hygiene & Data Safety Rules for Binary Search on Answer, Monotonic Stacks & Bitmask Dynamic Programming • Performance Optimization, Complexity & Latency Impact of Binary Search on Answer, Monotonic Stacks & Bitmask Dynami... • Live Debugging, Stack • Runnable Code Implementation (PYTHON) • Hands • Idiomatic DSAP Implementation: PEP-8 type hints, generators, and O(1) dict/set structures • CPython 3.12 Bytecode & Heap Execution & Memory Lifecycle for Binary Search on Answer • Production Hardening & Security: input sanitization, safe deserialization & exception hierarchy

Purpose: To understand and apply Binary Search on Answer, Monotonic Stacks & Bitmask Dynamic Programming in real DSA with Python systems — specifically: Crack hard optimization an...

What Topic Is Next → Module 15 Capstone (High-Speed Python Autocomplete Trie, Union-Find & Fenwick Tree Library) → Next Topic 16.1: Python Performance Rules: Avoiding O(N) list.pop(0), String Concatenation & Recursion Limits

Module 15 Hands-On Capstone Project (100 Marks): High-Speed Python Autocomplete Trie, Union-Find & Fenwick Tree Library — Architect, implement, test, and document 'High-Speed Python Autocomplete Trie, Union-Find & Fenwick Tree Library' applying every bullet-wise engineering concept from Module 15 (Advanced Pyth...

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

Python Runtime Pitfalls, Stress Testing & Live Coding Interview Execution

Self-explanatory list-and-bullet production mastery module covering Python Runtime Pitfalls, Stress Testing & Live Coding Interview Execution in DSA with Python Training, paired with hands-on labs, capstone repository, Topic Mock Tests, and Spoken MockAttempt AI Viva.

Objectives: ✓ Python Performance Rules: Avoiding O(N) list.pop(0), String Concatenation & Recursion Limits — Purpose: Prevent Time Limit Exceeded (TLE) verdicts caused by hidden quadratic operations in Python. • ✓ Property/Stress Testing: Generating Random Counterexamples to Catch Edge-Case Bugs — Purpose: Prove your optimal algorithm matches brute-force output across 1,000+ tricky boundary inputs. • ✓ FAANG / Product-Company Live Coding Framework & Spoken Complexity Walkthroughs — Purpose: Clarify constraints, dry-run examples aloud, and write clean typed Python under 35-minute timers. • ✓ Build, verify, and document 'Randomized Brute-Force vs Optimal Stress-Test Harness in Python 3.12' with clean Git commits and edge-case protection. • ✓ Clear 100% of Topic Timed Mock Tests and Spoken Voice + MockAttempt AI Technical Interviews.

16.1 Python Performance Rules: Avoiding O(N) list.pop(0), String Concatenation & Recursion Limits [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Security Hygiene & Data Safety Rules for Python Performance Rules • Performance Optimization, Complexity & Latency Impact of Python Performance Rules • Live Debugging, Stack • Runnable Code Implementation (PYTHON) • Hands • Idiomatic DSAP Implementation: PEP-8 type hints, generators, and O(1) dict/set structures • CPython 3.12 Bytecode & Heap Execution & Memory Lifecycle for Python Performance Rules • Production Hardening & Security: input sanitization, safe deserialization & exception hierarchy

Purpose: To understand and apply Python Performance Rules: Avoiding O(N) list.pop(0), String Concatenation & Recursion Limits in real DSA with Python systems — specifically: Preve...

What Topic Is Next → Next Topic 16.2: Property/Stress Testing: Generating Random Counterexamples to Catch Edge-Case Bugs

16.2 Property/Stress Testing: Generating Random Counterexamples to Catch Edge-Case Bugs [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Security Hygiene & Data Safety Rules for Property/Stress Testing • Performance Optimization, Complexity & Latency Impact of Property/Stress Testing • Live Debugging, Stack • Runnable Code Implementation (PYTHON) • Hands • Idiomatic DSAP Implementation: PEP-8 type hints, generators, and O(1) dict/set structures • CPython 3.12 Bytecode & Heap Execution & Memory Lifecycle for Property/Stress Testing • Production Hardening & Security: input sanitization, safe deserialization & exception hierarchy

Purpose: To understand and apply Property/Stress Testing: Generating Random Counterexamples to Catch Edge-Case Bugs in real DSA with Python systems — specifically: Prove your opti...

What Topic Is Next → Next Topic 16.3: FAANG / Product-Company Live Coding Framework & Spoken Complexity Walkthroughs

16.3 FAANG / Product-Company Live Coding Framework & Spoken Complexity Walkthroughs [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Security Hygiene & Data Safety Rules for FAANG / Product • Performance Optimization, Complexity & Latency Impact of FAANG / Product • Live Debugging, Stack • Runnable Code Implementation (PYTHON) • Hands • Idiomatic DSAP Implementation: PEP-8 type hints, generators, and O(1) dict/set structures • CPython 3.12 Bytecode & Heap Execution & Memory Lifecycle for FAANG / Product-Company Live Coding Framework • Production Hardening & Security: input sanitization, safe deserialization & exception hierarchy

Purpose: To understand and apply FAANG / Product-Company Live Coding Framework & Spoken Complexity Walkthroughs in real DSA with Python systems — specifically: Clarify constraints...

What Topic Is Next → Module 16 Capstone → End-of-Course Live Client Project: PyGraph Fraud Ring Detector, Sliding-Window Rate Limiter & Priority Scheduler

Module 16 Hands-On Capstone Project (100 Marks): Randomized Brute-Force vs Optimal Stress-Test Harness in Python 3.12 — Architect, implement, test, and document 'Randomized Brute-Force vs Optimal Stress-Test Harness in Python 3.12' applying every bullet-wise engineering concept from Module 16 (Python Runtime...

3. LIVE CLIENT PRODUCTION PROJECT & CAREER SCOPE

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

Project: PyGraph Fraud Ring Detector, Sliding-Window Rate Limiter & Priority Scheduler • **Confirmed Role:** Python Data Structures, Algorithms & Problem-Solving Intern

Domain & Production Infra: FinTech Risk & Real-Time Transaction Screening Team (Python 3.12 + Custom Graph/Union-Find/Heap/Deque Engines + Pytest Complexity Suite)

- Built Disjoint-Set (Union-Find) and BFS/DFS cycle-detection algorithms to flag connected fraud rings in real time.
- Implemented O(1) deque sliding-window rate limiters, monotonic queues, and heapq priority job schedulers.
- Optimized recursive backtracking and dynamic programming solvers using memoization and bottom-up tabulation.
- Validated asymptotic runtime and worst-case adversarial inputs with automated pytest benchmark suites.

Target Job Roles: Software Development Engineer — SDE-1 (Python), Backend & Systems Product Engineer, AI / ML Algorithms Engineer, Platform Software Engineer

Realistic Annual CTC Scope: **Product Fresher / SDE-1 (0–1 Yr):** ₹8.0 LPA – ₹18.0 LPA **Product SDE-2 / AI Engineer (1–3 Yrs):** ₹16.0 LPA – ₹32.0 LPA **Senior SDE / Principal Python Architect (3–6+ Yrs):** ₹30.0 LPA – ₹58.0+ LPA

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

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