

DSA, one step at a time.

Data Structures and Algorithms | Beginner to interview practice

**Stop collecting roadmaps.
Start building understanding.**

09

PHASES. ONE CLEAR PATH.

Foundations > patterns > structures > algorithms

Learn. Attempt. Explain. Revisit.

Choose	One language and one main learning path.
Practise	Understand the pattern, correctness and complexity.
Track	Use phase checkpoints and the final practice log.

Redesigned edition

Reformatted and editorially condensed from the supplied 17-page roadmap. All nine phases and the main topic groups are retained; explanations are shortened, absolute claims softened, and a reusable tracker added. Resource names come from the source. Links may change.

Use the sequence

Consistency matters more than how many resources you bookmark.

Three commitments

One language	Keep it stable while learning core DSA. Familiarity usually matters more than theoretical fit.
One primary path	Use another source to clear up a specific doubt, rather than taking several full courses at once.
One practice set	Work topic by topic. Measure independent reasoning, not only accepted submissions.

Planning your time

The source suggests roughly 8-12 months for core interview preparation at two focused hours a day, and longer for deeper coverage. These are planning estimates, not hiring promises. Your starting point and target roles can change the pace substantially.

The phase map

00-02	Programming foundations; arrays and strings; searching and sorting.
03-05	Recursion and backtracking; linear structures; trees, heaps and tries.
06-08	Graphs; dynamic programming; advanced topics selected for your goal.

A better progress check

Can you clarify the input, explain a sound approach, implement it, test edge cases and state its costs?

Pattern recognition helps, but interviews can assess more than recognition alone.

Pick your language

Choose one you can use confidently and that your target platform accepts.

Python	Concise and approachable. Watch runtime costs, recursion depth and the complexity of built-in operations.
Java	Explicit types and a strong collections library. More verbose; learn fast input if you enter contests.
C++	STL containers and strong contest support. Expect more syntax, memory concepts and tricky error messages.

Practical choice

Already comfortable with a language? Stay with it. A complete beginner can start with Python; C++ is a common contest choice. You can reassess later without repeatedly switching during core practice.

Know these operations before Phase 1

Containers	Dynamic arrays/lists, hash maps and sets.
Sorting	Built-in sort, custom comparison or key functions.
Strings	Split, join, substrings and character/number conversion.
Functions	Parameters, return values and mutation versus copying.
Recursion + I/O	Base case, recursive step; basic input/output and contest input when relevant.

Learn enough language to solve small problems first. Advanced language features and full object-oriented design can be studied alongside other goals.

Programming foundations

3-6 weeks | Suggested pace from the source, not a deadline

Variables and types	Integer limits, overflow and how your language represents numbers.
Conditionals and loops	Nested loops, boundaries and how often operations run.
Functions	Parameters, returns, copies and modifying original data.
Arrays and strings	Indexing, iteration, in-place changes and off-by-one errors.
Input and output	Multiple values, end-of-input and formatting results.
Debugging	Read errors, print intermediate values and test small cases.
Big O basics	Constant, logarithmic, linear, $n \log n$ and quadratic growth; include space costs.

LEARN + PRACTISE

Official language tutorials, learncpp.com, CS50 or [freeCodeCamp](https://freeCodeCamp.org). Practise simple HackerRank tasks and introductory Codeforces A problems.

Checkpoint

Find the second-largest distinct number, handling duplicates and inputs with fewer than two distinct values. Explain your chosen behavior.

Arrays, strings & patterns

5-7 weeks | Suggested pace from the source, not a deadline

Traversal	Running totals, prefix maxima, reversals and rotations.
Hashing	Frequency maps, membership checks and duplicate detection.
Two pointers	Sorted pair sums, palindromes, partitioning and in-place removal.
Sliding window	Fixed and variable windows; identify when the window invariant is valid.
Prefix sums	Range sums; later extend to 2D and prefix XOR.
String basics	Anagrams, parsing, frequency maps and transformations.
Sorting as a tool	Ask whether sorting simplifies the problem before inventing a complex approach.

LEARN + PRACTISE

Choose Striver A2Z or NeetCode as the main path. Practise LeetCode by topic and the CSES introductory/sorting sections; use GeeksforGeeks for targeted reference.

Checkpoint

Solve an unfamiliar problem in a studied pattern, explain why it works and revisit it from a blank editor after a week. The source suggests 90-120 focused problems; treat that as optional guidance.

Searching & sorting

3-4 weeks | Suggested pace from the source, not a deadline

Elementary sorting	Bubble, selection and insertion sort: learn the mechanisms and quadratic costs.
Merge sort	Divide and conquer, merging and counting inversions.
Quicksort + selection	Partitioning, average versus worst case and quickselect.
Counting + radix sort	Useful when key ranges and representations suit the algorithm.
Classic binary search	State the invariant and decide how bounds change.
Lower / upper bounds	First value $\geq$ target and first value $>$ target.
Rotated arrays	Use the sorted-part invariant carefully.
Search on an answer	A monotone feasibility check over an ordered answer range.
Real-valued search	Use a precision goal or bounded iteration count.

LEARN + PRACTISE

Codeforces EDU binary search course, CSES Sorting and Searching, or LeetCode binary search problems.

Checkpoint

Recognise a monotone minimum/maximum feasibility problem. Explain the predicate and boundaries, then test absent targets and one-element inputs.

Recursion & backtracking

4-6 weeks | Suggested pace from the source, not a deadline

Basic recursion	Factorial, Fibonacci, digit sums and string reversal; state a base case.
Arrays + decision trees	Subsets and subsequences using include/exclude choices.
Permutations	Generate arrangements; handle duplicate values correctly.
Backtracking	N-queens, sudoku, word search and maze exploration.
Choose, explore, undo	Restore mutable state when a branch finishes.
Pruning	Stop branches that cannot lead to valid solutions.
Explicit stacks	Understand the call stack and convert a recursive traversal to iteration.

LEARN + PRACTISE

Striver recursion material, Abdul Bari for analysis, and LeetCode backtracking problems.

Checkpoint

Generate unique subsets of an array containing duplicates. Explain what each recursive call promises and how state is restored.

Linear data structures

4-5 weeks | Suggested pace from the source, not a deadline

Linked lists	Singly and doubly linked: traverse, insert and delete.
Reversal + merging	Iterative/recursive reversal, sorted merge and list merge sort.
Fast / slow pointers	Middle node, cycle detection and cycle entry.
Stacks	Array/list implementations; balanced parentheses and expression evaluation.
Expressions	Infix, postfix and prefix representations.
Monotonic stack	Next greater element, stock span and histogram area.
Min stack	Track the minimum with constant-time queries.
Queues + dequeues	Circular queues and wraparound; insertion/removal at both ends.
Monotonic deque	Sliding-window maximum in linear time.
Conversions	Queue using stacks; stack using queues.

LEARN + PRACTISE

William Fiset for structures; LeetCode linked-list, stack and queue tags; selected CSES problems.

Checkpoint

Build each structure once and explain operation costs. Then prefer standard library containers unless the problem asks for an implementation.

Trees, heaps & tries

6-8 weeks | Suggested pace from the source, not a deadline

Tree traversals	Inorder, preorder, postorder; recursive and iterative. Level-order with a queue.
Tree properties	Height, diameter, balance and node counts.
Views + construction	Left/right/top views, boundary and vertical order; build from traversal pairs.
Paths + LCA	Root-to-leaf paths, maximum path sum and lowest common ancestor.
BST operations	Search, insert, delete; validate using range bounds. Inorder is sorted.
Balanced BST concepts	Know the guarantees of AVL and red-black trees; implementation is goal-dependent.
Heaps	Insert, extract and heapify; build one once to understand it.
Heap patterns	Top K, running median with two heaps and merging K sorted structures.
Tries	Insert, exact/prefix search, autocomplete and word-break applications.

LEARN + PRACTISE

Striver tree series; LeetCode tree, BST, heap and trie tags; CSES Tree Algorithms after the basics.

Checkpoint

Choose an appropriate traversal or heap strategy and justify it. Handle empty trees, skewed trees, duplicate values and K boundaries.

Graphs

6-8 weeks | Suggested pace from the source, not a deadline

Representation	Adjacency lists versus matrices; know the storage tradeoff.
BFS	Unweighted shortest paths and multi-source BFS.
DFS + grids	Components, paths, islands and flood fill; recursive or iterative.
Cycles + bipartite checks	Distinguish directed and undirected cases; colour a graph for bipartiteness.
Topological sort	DAG dependencies; Kahn's algorithm and DFS ordering.
Dijkstra	Shortest paths with non-negative edge weights and a priority queue.
Bellman-Ford	Negative edges; detect relevant reachable negative cycles.
Floyd-Warshall	All-pairs paths with $O(V^3)$ time on smaller graphs.
Union-find + MST	Path compression, union by size/rank; Kruskal and Prim.
Optional depth	Strongly connected components (Tarjan/Kosaraju), bridges, articulation points and maximum flow.

LEARN + PRACTISE

William Fiset graph series; LeetCode graph/union-find tags; CSES Graph Algorithms; CP-Algorithms for reference.

Checkpoint

Model an unstated graph: dependencies, word transformations or puzzle states. Explain nodes, edges, traversal costs and when your algorithm applies.

Dynamic programming

8-10 weeks | Suggested pace from the source, not a deadline

1D + grids	Climbing stairs, house robber, unique paths and obstacles.
0/1 knapsack	Subset sum, equal partition and target sum.
Unbounded knapsack	Coin change and rod cutting; explain reuse of an item.
Subsequences	LCS, edit distance and LIS.
Intervals	Matrix-chain multiplication, burst balloons and palindrome partitioning.
Stocks	Track the states, transactions and allowed actions.
Trees	Combine recursive tree reasoning with stored subproblem results.
Bitmask DP	Small-set state spaces and travelling-salesperson variants; optional for many interviews.

LEARN + PRACTISE

Striver DP series, AtCoder Educational DP Contest and Errichto for more contest-oriented explanations.

Checkpoint

State the meaning of a state, transition, base case and final answer before coding. Derive a recursive solution, memoise repeated states, then consider tabulation and space optimisation.

Advanced, by goal

Ongoing / selective | Suggested pace from the source, not a deadline

Range queries	Segment trees, lazy propagation, Fenwick trees and sparse tables.
String algorithms	KMP, Z algorithm, Rabin-Karp/rolling hash and Manacher.
Specialised strings	Suffix arrays and suffix automata, mainly for deeper contest work.
Number theory	GCD/LCM, sieve, modular arithmetic/inverses and fast exponentiation.
Combinatorics	Permutations, combinations, Pascal's triangle and inclusion-exclusion.
Probability	Expected value when your target problem set calls for it.
Bit manipulation	XOR, set/clear bits, popcount and subset enumeration; basics can be learned much earlier.
Greedy	Interval scheduling, activity selection and correctness arguments.
Other techniques	Divide and conquer, meet in the middle, square-root decomposition and Mo's algorithm.

LEARN + PRACTISE

Use USACO Guide or CP-Algorithms for targeted depth. Select advanced practice to match interviews, contests or a specific role.

Checkpoint

Do not wait until this phase to learn basic greedy reasoning or bit operations. Study advanced structures when a clear goal requires them, after the core phases are comfortable.

Complexity, with context

Read the constraints first. Use these as rough filters, not runtime guarantees.

n around 10	Factorial / exhaustive permutations may be plausible.
n around 20	Subset enumeration or $O(2^n)$ approaches may fit.
n around 100	Some $O(n^3)$ algorithms may be feasible.
n in the low thousands	$O(n^2)$ may fit, depending on time, language and constants.
n around 100,000	$O(n \log n)$ or better is often a useful starting target.
Very large n	Seek linear, logarithmic or mathematical approaches as the input representation permits.

Common operation costs

Dynamic array	Index $O(1)$; append amortised $O(1)$; middle insertion/deletion $O(n)$.
Hash map / set	Typical average $O(1)$ lookup/insert/delete; worst-case behavior depends on implementation and keys.
Balanced BST	Search, insert and delete $O(\log n)$.
Heap	Top $O(1)$; push/pop $O(\log n)$; standard bottom-up heap build $O(n)$.
Linked list	Search $O(n)$; local splice $O(1)$ when needed node/predecessor links are already available.
Trie / DSU	Trie operations $O(\text{key length})$; DSU almost constant amortised with compression and union heuristics.

Also count auxiliary arrays, stored states and recursion depth. Raising a recursion limit alone does not remove stack or memory risks.

Pick a main path

Names preserved from the supplied roadmap. Core access and paid extras can change.

[Striver / takeUforward](#) - Ordered DSA topics and practice material.

[NeetCode](#) - Pattern-based interview practice; some offerings are paid.

[CSES Problem Set](#) - Curated introductory, sorting, graph, tree and DP problems.

[USACO Guide](#) - Written progression with a competitive-programming focus.

[Codeforces EDU](#) - Courses such as binary search; account/access conditions may apply.

[LeetCode](#) - Interview-style practice; some content requires payment.

[AtCoder](#) - Beginner contests and Educational DP practice.

[HackerRank](#) - Introductory programming and problem-solving tasks.

[Kattis](#) - An ICPC-oriented problem archive.

[Codeforces](#) - Rated contests and upsolving practice.

Keep the choice small

Choose one main course and one practice set for the current phase. Add a second explanation only when it solves a specific gap.

Learn, then look things up

Resources support practice; they do not replace it.

Foundations + reference

[Python tutorial](#) - Language foundations.

[Java learning](#) - Language and platform concepts.

[LearnCpp](#) - C++ foundations.

[CS50](#) - Computer science foundations.

[freeCodeCamp](#) - Practical tutorials.

[CP-Algorithms](#) - Algorithm reference and implementations.

[GeeksforGeeks](#) - Broad topic reference; cross-check important details.

Video teachers named in the source

William Fiset: data structures and graph theory.

Abdul Bari: classical algorithm analysis.

Errichto: contest techniques and DP.

NeetCode: interview problem walkthroughs.

Books

Competitive Programmer's Handbook	Antti Laaksonen. Free PDF at cses.fi/book/book.pdf .
The Algorithm Design Manual	Steven Skiena. Paid book; practical algorithm selection.
Introduction to Algorithms	Cormen and coauthors. Paid reference; consult by topic.

Interviews or contests?

The material overlaps. The practice style and depth can differ.

Interview focus	Explain a correct approach, test edge cases, communicate tradeoffs and implement clearly.
Contest focus	Solve unfamiliar algorithmic problems under stricter time pressure; practise speed and accuracy.
Shared foundation	Phases 0-7, complexity analysis, greedy reasoning and basic bit operations.
Optional depth	Advanced mathematics, specialised structures and hard contests when they support your goals.

If you enjoy competitive programming

Begin	Try Codeforces Div 3/4 or AtCoder beginner tasks. Start at a level you can engage with.
After each contest	Upsolve an unfinished problem. Try independently before consulting an editorial.
Alongside contests	Study the missing topic through CSES, USACO Guide or a focused reference.
Over time	Increase difficulty gradually and treat rating as limited feedback, not a measure of your worth.

When to keep it optional

If placement preparation is urgent, prioritise the core gaps and mock interviews. Also make room for projects and role-specific engineering skills. DSA alone is not the whole job.

Two focused hours

A sample routine you can resize to fit your day.

20 minutes	Learn one concept from the current phase.
70 minutes	Attempt one or a few relevant problems. Quality beats forcing a count.
15 minutes	Compare approaches and write the pattern, correctness idea and complexity.
15 minutes	Revisit an older problem from a blank editor.

Attempt > hint > explain > revisit

Give a problem a genuine attempt; the source suggests about 25 minutes before a hint. Adjust to the difficulty. Read a hint before solution code when possible. Afterward, close the solution and reconstruct the reasoning. Revisit after roughly a week and again later.

Weekly rhythm

Five topic-practice days, one timed or mock-interview session, and one rest day is a workable starting plan. In the final preparation stretch, explain aloud, ask clarifying questions and code without relying entirely on autocomplete.

Watch for these traps

Collecting / random solving	Too many courses and random tags hide the gaps. Keep a sequence.
Copying / chasing counts	Accepted code is not the same as retained reasoning.
Skipping / never reviewing	Weak recursion or missed fundamentals can surface later. Revisit the prerequisite.
Silent practice	Rehearse explaining assumptions, correctness, edge cases and complexity.

Your weekly practice log

Print or duplicate this page. Track learning rather than only totals.

Week of: _____ Current phase: _____

Session 1

Problem / source: _____

Pattern + key insight: _____

Independent / hint / reviewed: _____ Revisit date: _____

Session 2

Problem / source: _____

Pattern + key insight: _____

Independent / hint / reviewed: _____ Revisit date: _____

Session 3

Problem / source: _____

Pattern + key insight: _____

Independent / hint / reviewed: _____ Revisit date: _____

Session 4

Problem / source: _____

Pattern + key insight: _____

Independent / hint / reviewed: _____ Revisit date: _____

Session 5

Problem / source: _____

Pattern + key insight: _____

Independent / hint / reviewed: _____ Revisit date: _____

Weekly reflection

What can I now explain without notes? Which error repeated? What is one prerequisite to revisit next week?