

THE COMPLETE PRACTICAL GUIDE

75 Coding Interview Patterns

Problem-Solving Patterns in Python, Java, and JavaScript

Python · Java · JavaScript

28 CHAPTERS · 4 APPENDICES · MOCK INTERVIEWS

THENEXBOOKS

Coding Interview Patterns

75 Problem-Solving Patterns in Python, Java, and JavaScript

Recognize the Pattern, Explain the Idea, Write the Code

First Edition

THENEXBOOKS

Coding Interview Patterns
75 Problem-Solving Patterns in Python, Java, and JavaScript

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples may be used freely for your own study and projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. This book does not guarantee any interview outcome or job offer. Interview formats vary between companies and change over time.

Python is a trademark of the Python Software Foundation. Java is a trademark of Oracle and/or its affiliates. JavaScript is a trademark of Oracle and/or its affiliates. Node.js is a trademark of the OpenJS Foundation. Other product names are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement.

Kestrel Transit, the running example in this book, is fictional. All problem statements in this book are original. Any resemblance to real companies or products is coincidental.

Preface

Most people who struggle in coding interviews do not lack intelligence or programming skill. They struggle because a new problem looks like a blank page. Under time pressure, with someone watching, it is hard to invent an algorithm from nothing.

Experienced interviewees rarely invent from nothing. They *recognize*. A question about the longest stretch of ride logs without a repeated station is a sliding window. A question about the cheapest way to cover a set of trips is dynamic programming. A question about which routes must be built before others is a topological sort. Once you see the pattern, the blank page turns into a familiar template that you adapt, explain, and test.

This book teaches **75 problem-solving patterns** that show up again and again in algorithmic interview problems. For every pattern, we show you how to recognize it from the wording of a problem, why it works, and how to implement it cleanly in **Python, Java, and JavaScript** — so you can practice in the language you will actually use on interview day.

Who This Book Is For

- **Job seekers** preparing for software engineering interviews, from internships and new-grad roles to senior positions.
- **Self-taught and bootcamp developers** who build real software but never studied algorithms formally.
- **Experienced engineers** returning to the job market who need a structured, efficient refresher.
- **Students** who want a practical companion to an algorithms course.

What You Will Learn

- How coding interviews typically work and what interviewers usually look for beyond a correct answer.
- Complexity analysis you can do quickly and explain confidently.
- A repeatable framework for attacking any unfamiliar problem.
- 75 patterns across arrays, strings, hashing, linked lists, stacks, queues, heaps, trees, tries, graphs, backtracking, dynamic programming, greedy algorithms, bit manipulation, math, and advanced data structures.
- Clean, idiomatic solutions in three languages, with hand traces, complexity analysis, common bugs, and follow-up questions.
- Five complete mock interviews and a study plan that turns the patterns into interview-day fluency.

How This Book Is Organized

Part I — Interview Foundations (Chapters 1–4) covers the interview process, complexity analysis, a problem-solving framework, and language toolkits. **Part II — Arrays, Strings, and Hashing** (Chapters 5–10) teaches Patterns 1–20. **Part III — Linear Data Structures** (Chapters 11–14) covers Patterns 21–32. **Part IV — Trees and Graphs** (Chapters 15–20) covers Patterns 33–50. **Part V — Recursion, Dynamic Programming, and Advanced Techniques** (Chapters 21–26) covers Patterns 51–75. **Part VI — Putting It All Together** (Chapters 27–28) provides mock interviews and a study plan. The appendices provide a pattern cheat sheet, language reference and templates, an interview-day checklist, and a glossary.

Each chapter ends with a **Key Takeaways** box for quick review.

The Running Example: Kestrel Transit

Throughout the book, problems come from **Kestrel Transit**, a fictional city mobility platform with buses, trams, a bike-share network, and on-demand rides. Its engineers keep running into questions — about station networks, trip schedules, rider logs, fares, and dispatch queues — that turn out to be classic interview patterns. Each pattern starts with a short Kestrel Transit scenario, then generalizes it into the kind of problem you will see in an interview.

Conventions Used in This Book

> **TIP**
> A technique, shortcut, or interview habit that makes your work easier.

> **NOTE**
> Background information, language details, or a useful aside.

> **WARNING**
> A common mistake or edge case that often breaks otherwise correct solutions.

Code appears in `monospace`. Solutions are labeled **Python**, **Java**, and **JavaScript**. Python examples target recent Python 3 versions, Java examples target Java 17 or later, and JavaScript examples use modern ECMAScript and run in Node.js.

Contents

Preface

PART I — INTERVIEW FOUNDATIONS

1 How Coding Interviews Work — and How to Use This Book

2 Complexity Analysis: Big-O, Space, and Choosing Data Structures

3 A Repeatable Problem-Solving Framework

4 Language Toolkits: Python, Java, and JavaScript for Interviews

PART II — ARRAYS, STRINGS, AND HASHING

5 Hashing Patterns (Patterns 1–4)

6 Two Pointers (Patterns 5–8)

7 Sliding Window (Patterns 9–11)

8 Prefix Sums and Difference Arrays (Patterns 12–14)

9 Binary Search (Patterns 15–17)

10 Intervals, Sorting, and Sweep Line (Patterns 18–20)

PART III — LINEAR DATA STRUCTURES

11 Linked List Patterns (Patterns 21–24)

12 Stacks and Monotonic Stacks (Patterns 25–27)

13 Queues, Deques, and Monotonic Queues (Patterns 28–29)

14 Heaps: Top-K, K-Way Merge, and Two Heaps (Patterns 30–32)

PART IV — TREES AND GRAPHS

15 Tree Traversal Patterns (Patterns 33–36)

16 Binary Search Trees and Tree Construction (Patterns 37–39)

17 Tries and String Search (Patterns 40–41)

18 Graph Traversal: BFS, DFS, and Grids (Patterns 42–45)

19 Topological Sort and Union-Find (Patterns 46–47)

20 Shortest Paths and Minimum Spanning Trees (Patterns 48–50)

PART V — RECURSION, DYNAMIC PROGRAMMING, AND ADVANCED TECHNIQUES

21 Backtracking (Patterns 51–54)

22 Dynamic Programming I: 1D and Knapsack (Patterns 55–58)

23 Dynamic Programming II: Grids, Strings, Intervals, and States (Patterns 59–62)

24 Greedy Algorithms (Patterns 63–65)

25 Bit Manipulation and Math (Patterns 66–69)

26 Advanced Patterns: Design, Range Queries, and String Matching (Patterns 70–75)

PART VI — PUTTING IT ALL TOGETHER

27 Mock Interviews: Five Full Walkthroughs Across the Patterns

28 Your Study Plan: The 75-Pattern Map, Practice Schedules, and Interview Day

APPENDICES

A Pattern Cheat Sheet

B Language Reference and Code Templates

C Interview Day Checklist

D Glossary

PART I

Interview Foundations

Before the patterns, the fundamentals: how interviews work, how to analyze complexity quickly, a repeatable framework for unfamiliar problems, and the language features you will lean on in Python, Java, and JavaScript.

1 · How Coding Interviews Work
2 · Complexity Analysis
3 · A Repeatable Problem-Solving Framework
4 · Language Toolkits

CHAPTER 1 · PART I — INTERVIEW FOUNDATIONS

How Coding Interviews Work — and How to Use This Book

Coding interviews reward a specific, learnable skill: turning an unfamiliar problem into a familiar one, then explaining and implementing the solution under time pressure. This chapter explains what that process usually looks like, what interviewers tend to evaluate, and how the 75 patterns in this book are organized so that you can study them efficiently.

Why Coding Interviews Look the Way They Do

If you have ever wondered why a job that involves building web services, mobile apps, or data pipelines is screened with puzzles about arrays, trees, and graphs, you are not alone. The algorithmic interview is an imperfect tool, but it persists because it is cheap to run, reasonably consistent across candidates, and it exposes several things that are hard to see any other way in an hour: how you reason about an unfamiliar problem, how you communicate while you think, how carefully you write code, and how you react when your first idea does not work.

Notice what is *not* on that list. A well-run interview is not primarily testing whether you have seen the exact problem before. Interviewers often choose problems that can be solved in several ways, starting with a slow but obviously correct approach and ending with something faster. The conversation in between — the moment when you notice that a sorted input suggests two pointers, or that "the longest stretch with at most k of something" suggests a sliding window — is where most of the signal lives.

That is the core bet of this book. The number of distinct interview problems is large and keeps growing, but the number of underlying ideas is surprisingly small. We have collected those ideas into 75 named patterns. Each pattern comes with the signals that tell you when to use it, the invariant that explains why it works, a clean implementation in Python, Java, and JavaScript, and the common variations you are likely to meet. If you can recognize the pattern, the problem stops being a puzzle and becomes an engineering task.

The Shape of a Typical Process

Hiring processes vary a great deal between organizations, teams, and seniority levels, so treat the following as a common shape rather than a promise. Many processes include some combination of these stages:

Stage	What usually happens	What to focus on

Recruiter screen	A short call about background, role, and logistics	Clear summary of your experience
Online assessment	Timed problems solved alone, graded by tests	Correctness, edge cases, speed
Technical phone screen	One or two problems in a shared editor	Talking through your approach
Onsite or virtual loop	Several interviews in one day or week	Stamina and consistency
System design	Designing a larger service (often senior roles)	Trade-offs, scaling, clarity
Behavioral	Questions about past projects and teamwork	Specific, honest stories

This book concentrates on the coding stages: online assessments, technical screens, and the coding rounds of a loop. System design and behavioral interviews deserve their own preparation, and we mention them only where they touch coding rounds — for example, when a coding problem turns into a small design question, as the LRU cache in Chapter 26 often does.

Online Assessments

An online assessment usually gives you a fixed amount of time and one or more problems in a browser-based editor. Hidden test cases grade your submission, and there may be no human watching. The priorities shift slightly here: nobody hears your explanation, so correctness and handling of edge cases matter even more. The patterns in this book are just as useful, but you should practice reading constraints carefully, because the input size often tells you exactly which complexity is expected. Chapter 2 shows how to turn constraints into a target running time.

Live Coding Rounds

In a live round, an interviewer presents a problem, you discuss it, and you write code in a shared editor, a whiteboard tool, or occasionally on a real whiteboard. Some environments let you run code; many do not. The interviewer may offer hints, ask follow-up questions, or change the constraints after you finish ("what if the input does not fit in memory?"). A typical round leaves time for one medium-sized problem or two smaller ones, plus a few minutes of questions at the end.

Live rounds reward the ability to think out loud. Silence is not a failure, but an interviewer cannot give you credit for reasoning they never heard. Chapter 3 gives you a repeatable framework that also doubles as a script for what to say at each step.

What Interviewers Tend to Evaluate

Rubrics differ, but most of them circle around the same four or five areas. Knowing them helps you spend your limited time on the things that actually produce a positive signal.

Problem Understanding

Before any code, can you restate the problem, ask the questions that matter, and pin down the input and output precisely? Candidates who start typing immediately often solve a slightly different problem than the one asked. A good clarifying question — "Can the list of fares be empty?" or "Are station IDs unique?" — is cheap and shows care.

Problem Solving and Algorithmic Thinking

Can you find a correct approach, estimate its cost, and then improve it? Interviewers usually care more about the path than about jumping straight to the optimal answer. Stating a brute force solution, analyzing it, and then explaining which repeated work you want to eliminate is a strong, visible line of reasoning.

Coding

Is the code correct, readable, and idiomatic in the language you chose? Good variable names, small helper functions, and sensible use of the standard library all count. You are not expected to remember every method signature, but you should be fluent with the basics of your language: maps, sets, sorting with a custom order, queues, and string handling. Chapter 4 is a compact toolkit for exactly this.

Verification

Do you test your own code? Walking through a small example by hand, checking an empty input, and looking for off-by-one errors before the interviewer points them out is one of the clearest signals of professional habits. Many candidates lose points not because their idea was wrong but because they declared victory on code they never traced.

Communication and Collaboration

Can the interviewer follow you? Do you take hints gracefully and use them? Do you explain trade-offs rather than just announcing a final answer? An interview is a small simulation of working together on a hard problem, and the way you collaborate is part of the evaluation.

> **NOTE**
>
> Nothing in this book depends on a particular organization's process, and we make no claims about what any specific company asks. Formats change over time. The durable skills — recognizing patterns, analyzing complexity, writing clean code, and testing it — transfer to almost any technical interview.

Why Patterns Beat Memorization

A common way to prepare is to grind through hundreds of problems, hoping that the interview will contain one you have already solved. This works a little, but it has two weaknesses. First, you cannot memorize enough problems to cover the space, and small wording changes can make a familiar

problem unrecognizable. Second, memorized solutions are fragile: if the interviewer adds a twist, a memorized answer offers no path forward.

Patterns solve both problems. A pattern is a reusable idea with three parts:

- **Signals** — features of the problem statement that suggest the idea. "Contiguous subarray," "at most k distinct," and "longest or shortest stretch" together point strongly toward a sliding window.
- **An invariant** — a property the algorithm maintains at every step, which is the reason it is correct. For a sliding window, the invariant might be "the window always satisfies the constraint after we shrink it."
- **A template** — a skeleton of code that you adapt rather than invent from scratch. Templates free your attention for the parts of the problem that are genuinely new.

When you learn a pattern this way, one hour of study covers dozens of problems. More importantly, patterns compose. Many harder problems are two patterns glued together: a binary search on the answer whose check function is a greedy scan, or a graph traversal whose visited state is a bitmask. Once you see problems as combinations, the hard ones become approachable.

Patterns also give you language for the interview itself. Saying "the input is sorted, so I want to try two pointers from both ends; the invariant is that every pair outside the pointers has already been ruled out" tells the interviewer a lot in one sentence. It shows that you know an idea, why it applies, and why it is correct.

A First Taste: From Brute Force to a Pattern

Let us make this concrete with a small problem from our running example. Kestrel Transit sells single-ride tickets at many different prices. A promotion lets a rider combine exactly two tickets whose prices add up to a voucher value. Given a list of ticket prices and the voucher value, the team wants the positions of any two distinct tickets whose prices sum to the voucher value, or a signal that no such pair exists.

Stated as an interview problem: given an array of integers `prices` and an integer `target`, return the indices of two different elements whose sum equals `target`, or an empty result if none exist.

The brute force approach checks every pair. It is obviously correct, easy to write, and a perfectly good first thing to say out loud. With n prices it examines about $n^2/2$ pairs, so it runs in $O(n^2)$ time and $O(1)$ extra space.

Python

```python
def pair_for_voucher_brute(prices, target):
    n = len(prices)
    for i in range(n):
        for j in range(i + 1, n):
            if prices[i] + prices[j] == target:
                return [i, j]
    return []
```

```
print(pair_for_voucher_brute([4, 11, 7, 2, 9], 9))  # [2, 3]
```

Now ask the question that drives almost every optimization in this book: *what work is being repeated?* For each price `p`, the inner loop searches the rest of the array for the value `target - p`, its complement. Searching for a value is exactly what a hash map does in expected constant time. If we remember every price we have already seen along with its index, then for each new price we can ask "have I already seen its complement?" in one step. This is Pattern 2, Complement Lookup, which Chapter 5 covers in depth.

Python

```
def pair_for_voucher(prices, target):
    seen = {}  # price -> index where we saw it
    for i, p in enumerate(prices):
        need = target - p
        if need in seen:
            return [seen[need], i]
        seen[p] = i
    return []


print(pair_for_voucher([4, 11, 7, 2, 9], 9))  # [2, 3]
print(pair_for_voucher([5, 5], 10))           # [0, 1]
print(pair_for_voucher([1, 2], 7))            # []
```

Java

```
import java.util.*;

public class Solution {
    public static int[] pairForVoucher(int[] prices, int target) {
        Map<Integer, Integer> seen = new HashMap<>();
        for (int i = 0; i < prices.length; i++) {
            int need = target - prices[i];
            Integer j = seen.get(need);
            if (j != null) {
                return new int[] {j, i};
            }
            seen.put(prices[i], i);
        }
        return new int[0];
    }

    public static void main(String[] args) {
        int[] prices = {4, 11, 7, 2, 9};
        System.out.println(Arrays.toString(pairForVoucher(prices, 9))); // [2, 3]
    }
}
```

JavaScript

```
function pairForVoucher(prices, target) {
    const seen = new Map(); // price -> index
    for (let i = 0; i < prices.length; i++) {
        const need = target - prices[i];
        if (seen.has(need)) {
            return [seen.get(need), i];
        }
        seen.set(prices[i], i);
    }
    return [];
}

console.log(pairForVoucher([4, 11, 7, 2, 9], 9)); // [ 2, 3 ]
```

The optimized version makes a single pass, so it runs in O(n) expected time and uses O(n) extra space for the map. Notice the subtle ordering: we check for the complement *before* inserting the current price. That ordering is what prevents a single ticket from being paired with itself when its price is exactly half of the target, and it is also what makes `[5, 5]` with target 10 work correctly. Small details like this are where interviews are often won or lost, and every pattern section in this book calls them out explicitly.

The whole journey — state the brute force, identify the repeated work, reach for a data structure or pattern that removes it, then check the edge cases — is the process you will practice 75 times in this book. By the end, it should feel routine.

How This Book Is Organized

The book has six parts. They are designed to be read roughly in order, but after Part I you can jump to any pattern you need.

Part	Chapters	What you get
I. Interview Foundations	1–4	Process, complexity, framework, language tools
II. Arrays, Strings, Hashing	5–10	Patterns 1–20
III. Linear Data Structures	11–14	Patterns 21–32
IV. Trees and Graphs	15–20	Patterns 33–50
V. Recursion, DP, Advanced	21–26	Patterns 51–75
VI. Putting It Together	27–28	Mock interviews and a study plan

Part I builds the foundation. Chapter 2 teaches complexity analysis well enough that you can predict the running time of your own code and choose data structures deliberately. Chapter 3 gives you a seven-step problem-solving framework that you will use in every chapter after it. Chapter 4 is a

language toolkit for Python, Java, and JavaScript, including the gotchas that trip people up under pressure and a small reusable heap class for JavaScript, which lacks a built-in priority queue.

Part II covers the patterns that appear most often on arrays and strings: hashing, two pointers, sliding windows, prefix sums, binary search, and interval techniques. These are the workhorses. If you have very limited time, this part gives the highest return.

Part III moves to linked lists, stacks, queues, and heaps. Each of these data structures has a small number of signature tricks — fast and slow pointers, monotonic stacks, monotonic deques, top-k with a heap — that show up again and again.

Part IV covers trees and graphs: traversals, binary search trees, tries, breadth-first and depth-first search on grids and graphs, topological sort, union-find, shortest paths, and minimum spanning trees. Kestrel Transit's network of stations and routes is a natural home for these ideas.

Part V tackles the topics many candidates find hardest: backtracking, dynamic programming, greedy algorithms, bit manipulation, and a set of advanced patterns such as segment trees, Fenwick trees, and string matching. We take these slowly and emphasize how to derive a solution rather than recall one.

Part VI puts everything together. Chapter 27 contains five complete mock interviews, written as dialogues, that combine several patterns each. Chapter 28 is a map of all 75 patterns with their signals, plus practice schedules of different lengths and advice for the day of the interview.

Four appendices support quick review: a one-page-per-part pattern cheat sheet, a language reference with code templates, an interview day checklist, and a glossary of the terms we use.

How Each Pattern Section Works

From Chapter 5 onward, every pattern has the same structure, so that you always know where to look.

1. **Scenario and problem.** A short Kestrel Transit story introduces the problem, followed by a precise, interview-style statement of it.
2. **Recognize it.** A short list of signals in the wording of a problem that should make you think of this pattern.
3. **Intuition.** A brute force approach, the repeated work it performs, and the invariant that lets the optimized approach skip that work.
4. **Solutions.** Complete implementations in Python, then Java, then JavaScript. Each is runnable on its own.
5. **Complexity and trace.** Time and space costs with a justification, plus a hand trace on a small input, usually as a table or ASCII diagram.
6. **Variations and bugs.** Follow-up questions an interviewer might ask, related problems that use the same idea, and the mistakes people most commonly make.

The "Recognize it" lists are the most important part for interview day. When you review, try covering the solution and reading only the signals; ask yourself what the approach would be. That habit of going from wording to pattern is exactly the skill an interview tests.

> **TIP**
> Keep a personal "signals notebook." Every time a problem surprises you — you did not recognize a pattern you already knew — write down the exact phrase in the problem that should have tipped you off. Over a few weeks this notebook becomes the most valuable review material you own, because it targets your specific blind spots.

Choosing Your Interview Language

Every solution in this book is given in Python, Java, and JavaScript. You do not need to read all three. Pick one primary language for interviews and read its version first; glance at the others when you want to see a different idiom.

If you are free to choose, the most important factor is fluency. The best interview language is the one in which you can write a hash map lookup, a sort with a custom comparator, and a queue-based traversal without thinking about syntax. That said, each of our three languages has a distinct profile:

- **Python** is concise, has a rich standard library (`collections`, `heapq`, `bisect`), and integers never overflow. Its main pitfalls in interviews are the low default recursion limit and occasional performance surprises from hidden copying, such as slicing inside a loop.
- **Java** is verbose but explicit, with strong typing that catches some errors early and a comprehensive collections framework including a built-in `PriorityQueue`. Watch for `int` overflow and for comparing boxed `Integer` objects with `==`.
- **JavaScript** is familiar to many web developers and has `Map` and `Set`, but no built-in heap or deque, and its default array sort compares elements as strings. You will need a small heap class, which Chapter 4 provides and later chapters reuse.

If your target role uses a particular language, interviewing in it can be a small advantage, since interviewers may be more familiar with its idioms. But fluency matters more than fashion. Switching languages a few weeks before an interview is rarely a good trade.

Meet Kestrel Transit

Throughout the book we return to a single fictional organization, Kestrel Transit, a city mobility platform that runs buses and trams, operates a bike-share network, and dispatches on-demand rides. Its engineering team keeps running into problems that turn out to be classic interview patterns. A running example has two benefits. First, it gives each pattern a concrete motivation, which makes the idea easier to remember. Second, it shows how the same domain produces very different algorithmic questions depending on what you ask.

Here is a preview of the kinds of objects you will meet and the patterns they tend to lead to:

Kestrel object	Typical shape	Patterns it leads to
Stations and stops	Graph nodes	BFS, DFS, shortest paths
Routes and track segments	Weighted edges	Dijkstra, MST, union-find
Trip schedules	Intervals	Merge, sweep line, greedy
Tap-in and tap-out logs	Arrays of timestamps	Two pointers, binary search
Vehicle GPS pings	Streams	Sliding windows, deques
Fares and passes	Costs and choices	Dynamic programming, greedy
Bike docks	Capacities	Prefix sums, heaps
Station name search	Strings	Tries, hashing
Dispatch queues	Priorities	Heaps, two heaps
City grid maps	2D grids	Flood fill, grid DP
Route codes like "B12"	Short strings	Parsing, stacks, hashing

The scenarios are deliberately light. They are a hook for memory, not a simulation of a real transit system, and every problem is stated precisely in general terms after the story. If a scenario ever seems to conflict with the formal statement, the formal statement wins.

How to Study with This Book

Reading about patterns is necessary but not sufficient. Interview skill comes from retrieving ideas under pressure, and retrieval is trained by practice, not by rereading. Here is a study method that works well with the structure of this book.

The Study Loop for One Pattern

1. **Read the scenario and problem statement only.** Before reading further, spend a few minutes sketching an approach on your own, even if it is just the brute force.
2. **Read the signals and the intuition.** Compare them with your own attempt. Where did your thinking diverge?
3. **Close the book and write the solution** in your interview language from memory. Run it on the example. Only then compare it with the book's version.
4. **Hand-trace a small input** the way the book does, writing down the key variables at each step.
5. **Work through one variation** from the end of the section without looking at a solution.
6. **Schedule a review.** Revisit the pattern after a few days and again after a week or two, starting from the signals list alone.

This loop is slower than reading straight through, and that is the point. Struggling briefly before seeing the answer makes the answer stick. Spacing reviews out over time is far more effective than cramming the same material in one sitting.

Three Suggested Paths

Chapter 28 contains detailed schedules, but here is a quick overview so you can plan from the start.

Time available	Suggested focus
About two weeks	Part I skim, Parts II and III, trees and BFS
About six weeks	Parts I–IV fully, core DP, mock interviews
Three months or more	Entire book, all variations, repeated mocks

In every path, Chapters 2 and 3 are worth reading carefully even if you skim the rest of Part I. Complexity analysis and a problem-solving framework are used in literally every interview, whereas any single pattern might not come up at all.

Practice Out Loud and on a Clock

Once a pattern feels comfortable on paper, practice it the way you will be tested. Set a timer for a realistic duration, explain your reasoning out loud as though someone were listening, and write code without running it until you believe it is correct. Recording yourself or practicing with a friend is uncomfortable at first but reveals habits you cannot see otherwise: long silences, vague explanations, or skipping the testing step.

If your interviews will allow code execution, practice both with and without it. Without execution, you must trace carefully by hand; with execution, you must still reason about edge cases, because a passing example does not prove correctness.

> **WARNING**
> Do not measure progress by the number of problems you have "done." Solving a problem once after reading the answer builds very little skill. A better measure is how many patterns you can recognize and implement from a blank page, on a problem you have not seen, within a time limit.

What to Do When You Get Stuck

Every candidate gets stuck at some point, in practice and in real interviews. Being stuck is not the problem; how you respond to it is. Here are strategies that work in both settings, and Chapter 3 expands on them.

- **Go back to a brute force.** If you cannot see an efficient solution, a correct slow one is a solid foundation. Analyzing its cost often reveals exactly what to optimize.

- **Work a tiny example by hand.** Solving a five-element case manually frequently exposes the structure that an abstract description hides.
- **Ask what property of the input you have not used.** Sorted order, small value ranges, distinct elements, and non-negative numbers are all clues that usually exist for a reason.
- **Run through the pattern families.** Could this be hashing? Two pointers? A window? Binary search on the answer? A graph in disguise? Chapter 3 includes a table that maps common signals to candidate patterns.
- **Say what you are thinking.** In a live interview, describing where you are stuck gives the interviewer a chance to offer a well-aimed hint. A hint that you use well is not a failure; it is part of the collaboration.

In practice sessions, set a limit for how long you will stay stuck before looking at a hint — perhaps fifteen to twenty minutes for a medium problem. Beyond that point, the marginal learning drops, and it is better to read the idea, then come back and solve the problem again from scratch a few days later.

Mindset for the Long Haul

Interview preparation can feel like an endless list. It helps to remember three things.

First, the material is finite. Seventy-five patterns sounds like a lot, but many of them are small refinements of each other, and you will notice them clustering into families: pointer techniques, window techniques, search techniques, traversal techniques, and optimization techniques. By the middle of the book, new patterns will increasingly feel like variations on ideas you already own.

Second, interview outcomes have noise. A great candidate can have a bad day, meet an unusual problem, or simply click poorly with a particular interviewer. One result does not define your ability. The goal of preparation is to raise the floor of your performance so that even an average day is a good one.

Third, the skills are real. Complexity analysis, careful testing, clear explanation, and the ability to decompose a problem are useful long after the interview ends. The patterns in this book are not just tricks for a test; they are the vocabulary that experienced engineers use when they reason about performance and correctness in production code.

Conventions Used in This Book

A few conventions will make the rest of the book easier to read.

- **Pattern numbers are global.** "Pattern 28" always means the Monotonic Deque in Chapter 13, wherever it is mentioned.
- **Complexity uses n** for the main input size unless stated otherwise. For graphs we use V for vertices and E for edges; for grids, R rows and C columns.

- **Indices start at zero** in all three languages, and ranges are half-open (the start is included, the end is not) unless we say otherwise.
- **Code is complete.** Each block is runnable on its own in a recent version of its language: Python 3, Java 17 or later, and a current Node.js release. Java examples use a `Solution` class with static methods for easy testing.
- **Callout boxes** mark tips (practical shortcuts), notes (useful background), and warnings (common mistakes that cost points).
- **Each chapter ends with Key Takeaways,** a short list you can use for quick review before an interview.

With that in place, we are ready to build the foundation. Chapter 2 starts with the one skill that every other chapter relies on: predicting how fast your code will run and how much memory it will use.

Key Takeaways

- Coding interviews mainly evaluate how you understand, solve, implement, test, and communicate — not whether you have memorized a specific problem.
- Processes vary widely; common stages include online assessments, live coding rounds, system design, and behavioral interviews.
- A pattern combines signals, an invariant, and a code template, which lets one idea cover many problems.
- The standard path is brute force, identify repeated work, remove it with the right pattern, then check edge cases.
- Every pattern section follows the same structure: scenario, signals, intuition, three-language solutions, complexity and trace, variations and bugs.
- Choose one interview language based on fluency, and learn its specific pitfalls from Chapter 4.
- Study actively: attempt first, write from memory, trace by hand, and space out reviews over time.
- When stuck, fall back to brute force, work a tiny example, and look for unused properties of the input.